PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Tolerância a Falhas em Elementos de Processamento de MPSoCs

Francisco Favorino da Silva Barreto

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Dr. Alexandre Morais Amory

Porto Alegre
2015

Dados Internacionais de Catalogação na Publicação (CIP)

B273t Barreto, Francisco Favorino da Silva
Tolerância a falhas em elementos de processamento de MPSoCs / Francisco Favorino da Silva Barreto. – Porto Alegre, 2015.
74 p.

Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Alexandre de Morais Amory.

1. Informática. 2. Multiprocessadores. 4. Tolerância a Falhas (Informática). I. Amory, Alexandre de Morais. II. Título.

CDD 004.35

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "Tolerância a Falhas em Elementos de Processamento de MPSoCs" apresentada por Francisco Favorino da Silva Barreto como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, aprovada em 13/03/2015 pela Comissão Examinadora:

Prof. Dr. Alexandre de Morais Amory – Orientador — PPGCC/PUCRS

Prof. Dr. Fernando Gehn Moraes – PPGCC/PUCRS

Profa. Dra. Fernanda Gusmão de Lima Kastensmidt – UFRGS

Homologada em......./......./........., conforme Ata No. .... pela Comissão Coordenadora.

Prof. Dr. Luiz Gustavo Leão Fernandes
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P32– sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320–3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

AGRADECIMENTOS

Agradeço a todos os que contribuíram de alguma forma para a realização deste trabalho e me apoiaram incondicionalmente nos momentos complicados de incertezas, cansaço e dificuldade.

Minha família que é uma grande incentivadora de todas as minhas realizações. Meu pai e minha mãe que estão na torcida e me apoiam em todas as minhas empreitadas. Ao meu cãozinho Bart, que esteve presente no meu segundo ano do mestrado e foi muito importante em nossas vidas. Em especial a minha esposa Cristina que está sempre ao meu lado em todas as passagens boas e ruins e nunca deixa de me apoiar e levantar o meu astral nos momentos mais cansativos. A ela eu dedico todo o meu amor e cumplicidade. Um beijo amor!

Agradeço aos meus amigos que acompanharam minha saga de estudos e sempre estiveram na torcida aguardando o termino de mais uma fase para comemorarmos. Aos meus colegas de mestrado e professores que tiveram significante participação nesta jornada.

À empresa onde trabalho, Hewlett Packard, que possibilitou minha matrícula no curso de mestrado e apostou em minha capacidade provendo o suporte financeiro para a realização deste curso.

Ao meu orientador Prof. Dr. Alexandre Amory, que foi um excelente conselheiro e parceiro de trabalho. Ao demais professores da FACIN que de alguma forma tiveram contribuições em meus estudos nesses dois anos, seja em disciplinas cursadas, seja em conversas do laboratório e corredores da PUC.

Aos professores avaliadores Dr. Fernando Gehm Moraes e Dra. Fernanda Gusmão de Lima Kastensmidt que aceitaram avaliar este trabalho.

E por fim a instituição Pontifícia Universidade Católica do Rio Grande do Sul que proporcionou todo o ambiente para o aprendizado necessário para realização deste curso.

FAULT TOLERANCE IN MPSOC PROCESSING ELEMENTS

ABSTRACT

The need of more processing capacity for embedded systems nowadays is pushing the research of MPSoCs with tens or hundreds of processors. These characteristics bring design challenges such as scalability and dependability. Such complex systems must have fault tolerant methods to ensure acceptable reliability and availability. This way, the user is not exposed to significant data losses, malfunctioning and even the total system failure.

Considering this technology trend, the present work proposes a fault tolerance method with focus in fault recovery. The method uses concepts largely explored in distributed systems to solve the problem of permanent failures in the processing elements of MPSoCs. The implementation is exclusively in software, and recovers the system exposed to a permanent failure on processing elements, reallocating all tasks that were executing in the faulty element to a healthy processing element. The failed application tasks restart their executions since there is no context saving, enabling a lightweight method.

The experiments are performed in the HeMPS platform, evaluating the most relevant parameters as recovery time, communication bandwidth impact, scalability and others. In the absence of faults, the proposed protocol has 21 Kbytes of memory area (20% more compared to the original kernel) and no overhead in terms of execution time. In the presence of faults, the results demonstrate total recovery times from 0.2ms to 1ms, depending on the number of reallocated tasks (1 to 7). The biggest impact in the protocol time is related with the reallocation task phase.

Keywords: task reallocation protocol; fault tolerance; MPSoCs.

TOLERÂNCIA A FALHAS EM ELEMENTOS DE PROCESSAMENTO DE MPSOCS

RESUMO

A pesquisa em MPSoCs (do inglês, *Multiprocessor System on Chip*) tem sido motivada pela necessidade crescente de maior capacidade de processamento das aplicações de sistemas embarcados. Devido à esta tendência, os MPSoCs tornam-se cada vez mais complexos e miniaturizados. Estas características trazem associados desafios como escalabilidade e dependabilidade. O sistema que tem a necessidade de ser confiável e estar disponível em todo o seu tempo operação precisa ser tolerante a falhas a ponto de recuperar-se automaticamente. Dessa forma o usuário não será exposto a perdas de informação, execução malsucedida ou até mesmo a falha total do sistema.

Este trabalho propõe um método de tolerância a falhas com foco na recuperação de falhas. O método utiliza conceitos utilizados em computação distribuída para solucionar o problema de falhas permanentes em elementos de processamento de um MPSoCs. O método proposto, implementado exclusivamente em software, recupera um sistema exposto a uma falha permanente de um elemento de processamento, realizando uma realocação das tarefas que estavam sendo executadas pelo elemento que falhou para um elemento de processamento saudável do sistema. As tarefas da aplicação que falharam devem reiniciar suas execuções do ponto de partida dado que o contexto da execução não é salvo, mantendo assim um baixo *overhead* no sistema, como demonstrado nos resultados obtidos.

Os experimentos foram realizados na plataforma HeMPS com uma avaliação dos parâmetros mais relevantes como tempo de recuperação, impacto em banda de comunicação, escala e outros, que justificam a viabilidade e as vantagens do método proposto. Na ausência de falhas, o protocolo proposto não altera o tempo de execução, porém aumenta o tamanho de memória do kernel para 21 Kbytes, 20% de acréscimo comparado com o kernel original. Os resultados obtidos na presença de falhas mostram que o tempo total de recuperação de falhas do método é de 0,2ms a 1ms, dependendo do número de tarefas realocadas devido ao PE defeituoso. O maior impacto de tempo no protocolo se dá com a etapa de realocação de tarefas.

Palavras chave: protocolo de realocação de tarefas; tolerância a falhas; MPSoCs.

LISTA DE FIGURAS

Figura 1 - Camadas de software do projeto MADNESS [DER13] 20
Figura 2 - Processo de recuperação a falha [MUS13] 21
Figura 3 - Técnicas de detecção de erro [GIZ11] 22
Figura 4 - *Loan Process* em cluster dinâmicos [CAS13A]..26
Figura 5 - Desempenho relativo ao tamanho do sistema [KOB11] 27
Figura 6 - Passos da migração de tarefas [ALM10] 29
Figura 7 - HeMPS gerência distribuída [CAS12A] 33
Figura 8 - Estrutura de um slot de Pipe 35
Figura 9 - Task control block 35
Figura 10 - Estrutura de uma aplicação no kernel master / local 36
Figura 11 - Funções *Send* e *Receive* 38
Figura 12 - Exemplo de um arquivo de configuração da HeMPS 39
Figura 13 - Protocolo de recuperação na HeMPS 44
Figura 14 - Pseudocódigo da recuperação de falhas no kernel master ou local. 44
Figura 15 - Pseudocódigo da recuperação de falhas no kernel slave 45
Figura 16 - Pseudocódigo - limpeza do pipe 47
Figura 17 - Diagrama de comunicação do protocolo de recuperação de falhas da HeMPS 48
Figura 18 - Grafos de Comunicação das Aplicações 51
Figura 19 - Configuração do experimento 1 – uma tarefa realocada 52
Figura 20 - Tempo de recuperação relativo ao número de tarefas realocadas 52
Figura 21 - Tempo de recuperação decomposto em etapas 53
Figura 22 - Configuração do experimento 2 – número total de tarefas 4 54
Figura 23 - Tempo de recuperação relativo ao tamanho da aplicação 55
Figura 25 - Consumo relativo de banda com as mensagens do protocolo de recuperação de falhas 56
Figura 24 - Configuração do experimento 3 - 1 tarefa sendo realocada 56
Figura 26 - Configuração do experimento 4 – primeiro teste com uma falha 57
Figura 27 - Tempo gasto nas etapas da recuperação em relação com o número de falhas 58
Figura 29 - Tempo de recuperação em relação ao tamanho da NoC (Gerência centralizada)...59
Figura 28 - Configuração do experimento 5 - dimensão 4x4 cluster de 2x2 59

Figura 30 - Tempo de recuperação em relação ao tamanho da NoC (Gerência distribuída) 60
Figura 31 - Configuração do experimento 6 - alocação inicial das tarefas....................61
Figura 32 - Impacto do número de tarefas da aplicação 62
Figura 33 - Impacto do número de bytes transferidos na realocação 63
Figura 34 – detalhamento do tempo gasto na realização do protocolo de recuperação de falhas. 67

LISTA DE TABELAS

Tabela 1 - Avaliação de trabalhos relacionados 31
Tabela 2 - Status da TCB 36
Tabela 3 - Serviços de gerenciamento de kernel da HeMPS 38
Tabela 4 - Estados da TCB 45
Tabela 5 - Novos serviços da HeMPS 46

SUMÁRIO

ABSTRACT 8

RESUMO 9

1. INTRODUÇÃO 15

1.1. Objetivos 16

1.2. Contribuição 17

1.3. Estrutura do documento 18

2. ESTADO DA ARTE 19

2.1. Tolerância a falhas 19

2.2. Gerência distribuída 25

2.3. Migração de tarefas 28

2.4. Considerações finais 30

3. DESENVOLVIMENTO 33

3.1. Estudo de caso – HeMPS 33

3.2. Estudo de alternativas de projeto 40

3.3. Detalhamento da solução desenvolvida 41

4. RESULTADOS 50

4.1. Experimento 1 – Tempo de recuperação x número de tarefas realocadas 52

4.2. Experimento 2 – Tempo de recuperação x tamanho das aplicações 54

4.3. Experimento 3 – Largura de banda utilizada na NoC proporcional aos principais serviços executados 55

4.4. Experimento 4 – Tempo de recuperação detalhado em etapas x número de falhas em diferentes clusters 57

4.5. Experimento 5 – Tempo de recuperação em relação ao tamanho da NoC 58

4.6. Experimento 6 – Tempo de recuperação em diferentes aplicações 61

4.7. Comparação com outras abordagens 63

4.8. Avaliação geral do protocolo proposto 64

4.9. Análise geral dos resultados de desempenho do protocolo 65

4.10. Limitações....67
5. CONCLUSÕES E TRABALHOS FUTUROS....70
5.1. Contribuições....71
5.2. Trabalhos futuros....71
REFERÊNCIAS....72

1. INTRODUÇÃO

Os sistemas computacionais de hoje escalam em tamanha proporção de tal forma que as aplicações atuais requerem um número de tarefas cada vez mais abundante, requisitando mais funcionalidades e velocidade dos circuitos integrados. Estes circuitos tornaram-se capazes de comportar sistemas complexos inteiros devido à capacidade de miniaturização de transistores.

Um dos principais recursos que permitiu o aumento massivo da integração de núcleos de processamento em um chip foi o emprego de redes de interconexões em chip, também conhecidas como *Network on-Chip* (NoC) [PIR04] [RAD13]. Utilizando redes como arquitetura de comunicação para os elementos de processamento, em inglês *Processing Elements* (PEs), obtém-se uma alta escalabilidade, grande vazão de dados e paralelismo conveniente para a execução massiva de múltiplas tarefas. Este processo de integração permite inclusive a criação de sistemas com múltiplos processadores programáveis, também chamados de *MultiProcessor Systems-on-Chip* (MPSoC) [JER05].

Por outro lado, essa miniaturização que reduz a área e consumo de energia do circuito também o torna suscetível a falhas causadas por interferências, por desgastes devido ao envelhecimento do circuito ou falhas no processo de fabricação [JER05]. Devido a capacidade de escalabilidade e paralelismo dos MPSoCs com arquitetura baseada em NoC, aliada a grande quantidade de tarefas a serem processadas, uma questão como a dependabilidade do sistema torna-se crucial para que o processamento realizado neste sistema seja confiável [AVI04].

Dependabilidade possui diversos atributos tais como disponibilidade (em inglês, *availability*), confiabilidade (em inglês, *reliability*), segurança (em inglês, *safety*), confidencialidade (em inglês, *confidentiality*), entre outros atributos [AVI04] [WEB01]. No contexto deste trabalho o foco será dado à **disponibilidade** e **confiabilidade** pois estes atributos atenuam o efeito da suscetibilidade a falhas mencionado anteriormente. Além disso, as ameaças típicas a dependabilidade são definidas como defeito de serviço[1] (em inglês, *failure*), erro (em inglês, *error*) e falha (em inglês, *fault*). Um defeito de serviço é caracterizado por um desvio do comportamento esperado do sistema. Um defeito de

---

[1] A área de teste de hardware costuma utilizar o termo 'defeito' como falha de hardware como, por exemplo, curto, fio aberto, etc. Assim, utilizamos o termo 'defeito de serviço' para não confundir com o termo 'defeito de hardware'.

serviço ocorre quando um erro se manifesta para o mundo externo. Um erro é definido como a manifestação ou efeito de uma falha. Uma falha é a causa física ou algorítmica do erro.

A forma típica de atribuir disponibilidade e confiabilidade em microeletrônica é através do desenvolvimento de técnicas de **tolerância a falhas**, que será o foco deste trabalho. De acordo com [MUS11], as técnicas de tolerância a falhas são definidas como meio de evitar o surgimento de defeitos de serviço. As técnicas para tolerância a falhas podem ser divididas em: *detecção de erros* (em inglês, *error detection*), onde se identifica a presença de erros devido a mensagens de erro ou sinais de erro; e *recuperação* (em inglês, *recovery*), que significa transformar um estado do sistema que contém um ou mais erros para um estado sem erros detectados [AVI04].

Dado o contexto de MPSoCs com os PEs interligados através de uma NoC, as tarefas que serão processadas em paralelo necessitam de uma gerência eficaz de recursos em tempo de execução para que o sistema opere satisfatoriamente [SIN13]. A gerência de recursos tem como exemplo de funções o mapeamento das tarefas em tempo de execução, escalonamento das tarefas, controle dos recursos, controle das configurações e migração de tarefas. A gerência desses recursos pode ser centralizada em um PE, distribuída em mais de um PE ou uma mistura de ambas [SIN13]. A gerência centralizada significa que um PE é responsável pela gerência dos recursos enquanto na gerência distribuída, a plataforma é dividida em regiões chamadas de *clusters*, e um PE de cada cluster gerencia as tarefas que o cluster executa. A gerência distribuída é recomendada em sistemas críticos para ganho de desempenho e para que não haja um ponto único de falhas que pode comprometer todo o sistema MPSoC.

1.1. Objetivos

Os objetivos estratégicos deste trabalho são:

- Domínio de técnicas de projeto e desenvolvimento de programação paralela e distribuída aplicadas a sistemas embarcados.
- Domínio de Sistemas operacionais embarcados (micro-kernel) executados em processadores de MPSoCs.
- Domínio de técnicas de tolerância a falhas, migração de tarefas e gerência distribuída de processadores.

O objetivo específico é propor métodos distribuídos de *recuperação de erros* para MPSoCs baseados em NoCs. As falhas consideradas neste trabalho estão localizadas no elemento de processamento uma vez que o problema de recuperação de erros na rede está bem estabelecido [RAD13] [WAC13]. Assume-se que os elementos de processamento possuem um método de teste que detecta existência de falhas permanentes. Desta forma, as tarefas executadas em um PE com falhas devem ser transferidas para um PE saudável. O MPSoC acadêmico HeMPS [CAR09] [CAS12b] é utilizado como estudo de caso para este trabalho.

1.2. Contribuição

O trabalho vai descrever no capítulo 3 com mais detalhes a contribuição que pode ser caracterizada como projeto e implementação de algoritmos e estruturas modulares no nível de *kernel* para um MPSoC que permitam a execução de uma técnica com baixo custo de desenvolvimento de recuperação a falhas permanentes de elementos de processamento. As vantagens do método proposto são:

- Baixo custo de desenvolvimento, dado que se trata de uma solução exclusivamente de software, sem a necessidade de adição de hardware.
- Baixo impacto no desempenho do sistema, dado que não há a necessidade de redundância de software em PEs secundários e nenhum contexto de execução da aplicação é recuperado tornando a solução leve e menos intrusiva comparada a estratégias de recuperação de falhas comumente empregadas em computação paralela e distribuída.
- Para muitas aplicações no modelo de fluxo de dados (em inglês, *dataflow*), o custo de recuperação do sistema pode apresentar-se imperceptível na visão do usuário. Fato que se deve ao curto tempo de inatividade da aplicação no momento do processo de recuperação de falhas.
- Não está limitada a uma configuração de memória compartilhada, como ocorrido com os artigos revisados apresentados posteriormente. A proposta está baseada em um MPSoC que realiza comunicação via troca de mensagens.
- A solução de recuperação está desacoplada de uma técnica de detecção de falhas, trazendo a capacidade de integrar com diferentes abordagens de teste e detecção.
- Não possui a necessidade de alteração nas aplicações em nível de usuário.

1.3. Estrutura do documento

O capítulo 2 apresenta uma revisão dos trabalhos que compõem o estado da arte em áreas de conhecimento relacionadas ao presente trabalho. O capítulo 3 caracteriza o desenvolvimento da solução proposta ao problema apresentado. O capítulo 4 apresenta os resultados obtidos. O capítulo 5 apresenta as limitações da proposta, as conclusões e trabalhos futuros. Por fim, o capítulo final de referências.

2. ESTADO DA ARTE

Neste capítulo é feita uma revisão em trabalhos relacionados a tolerância a falhas em MPSoCs, gerência de recursos distribuída e migração de tarefas, que são tópicos chave do presente trabalho.

2.1. Tolerância a falhas

2.1.1. A system-level approach to adaptivity and fault-tolerance in NoC-based MPSoCs: The MADNESS project [DER13]

O projeto MADNESS tem o objetivo de prover uma solução adaptativa para tolerância a falhas em MPSoCs baseados em NoC. Utiliza como recurso de recuperação a falhas a migração de tarefas que executavam em um elemento de processamento que apresentou falha permanente para um processador livre de falhas. O método proposto em [DER13], envolve diferentes camadas do projeto de sistemas MPSoC. No nível de aplicação foi proposta uma infraestrutura de software que permite a execução das aplicações em um modelo computacional chamado de *Polyhedral Process Network* (PPN), que consiste em processos autônomos e concorrentes que comunicam entre si por intermédio de canais FIFO. A camada de middleware implementa o suporte à comunicação entre os processos, bem como módulos de migração de tarefas para tolerância a falhas e o módulo chamando Run-time Manager, responsável por tomar decisões acerca dos recursos a serem migrados na presença de uma falha permanente em um elemento de processamento. A Figura 1 ilustra os elementos de software para o suporte a tolerância a falhas no MADNESS. Os níveis descritos são:

- O nível de aplicação, onde executam os processos comunicantes;
- O nível chamado de middleware, onde opera a camada de comunicação de processos, a migração de processos e o gerenciador de processos;
- O nível de sistema operacional local responsável pelas chamadas de sistema.

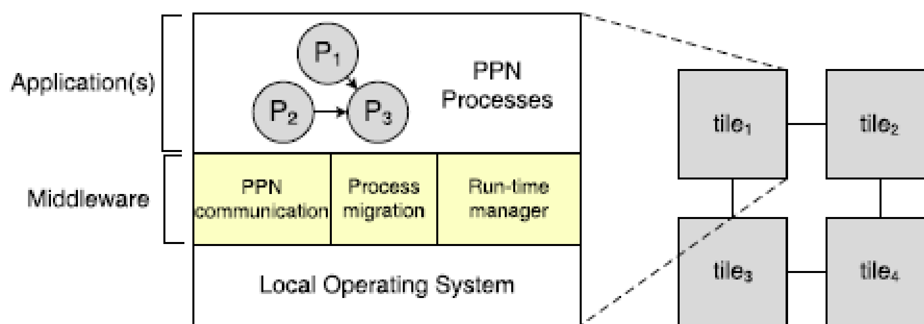


Figura 1 - Camadas de software do projeto MADNESS [DER13]

Uma última camada de apoio a solução foi desenvolvida no nível de hardware que é composta de um módulo de auto-teste utilizado na detecção de falhas e um módulo de migração de tarefas. Este módulo de migração tem como principais funções: isolar o processador que apresentou falha permanente; notificar um módulo run-time manager que esteja executando em um processador livre de falhas para que a migração de tarefas seja executada; receber todas as mensagens pendentes e FIFO *tokens* relativos aos processadores predecessores e sucessores (derivados do modelo PPN); e por fim enviar o contexto das tarefas e dos canais FIFO para o run-time manager acionado.

A migração feita pela infraestrutura de software se limita a migrar apenas o contexto das tarefas que estavam no processador falhado, dado que os códigos das tarefas migráveis são carregados em todos os processadores do sistema.

2.1.2. Efficient software-based fault tolerance approach on multicore platforms [MUS13]

Mustaq et al. [MUS13] apresentam uma técnica de tolerância a falhas baseada em memória compartilhada para sistemas multiprocessados. O estudo desenvolvido mostra resultados para aplicações com apenas um processo principal e processos auxiliares redundantes. Desta forma, a comunicação entre os processos é feita por memória compartilhada, no entanto a técnica limita-se a um processo único e suas cópias sem abordar diferentes processos comunicantes que poderiam trazer desafios a serem tratados como condições de corrida (*race condition*) na disputa de recursos compartilhados.

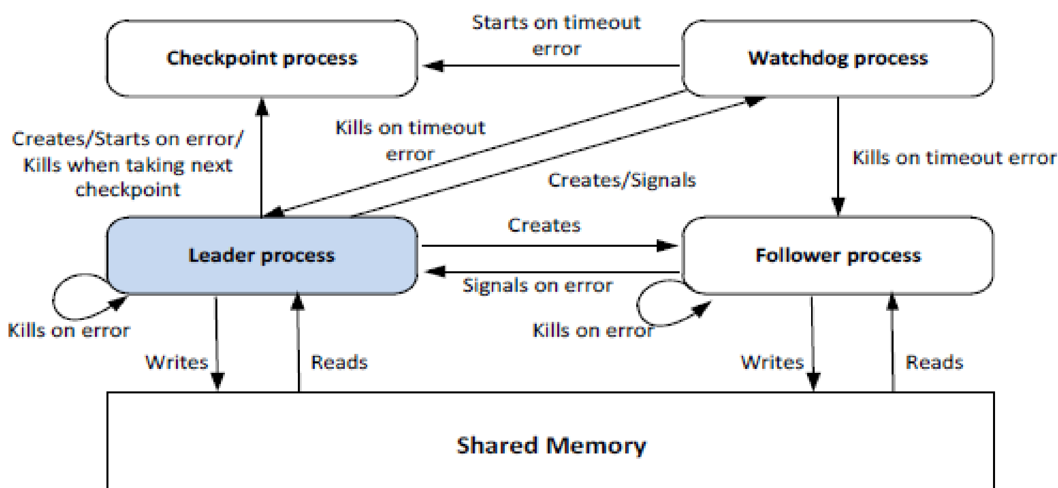


Figura 2 - Processo de recuperação a falha [MUS13]

O esquema de tolerância a falhas ilustrado na Figura 2 consiste dos seguintes elementos: Um processo chamado de líder que representa o processo origem; um processo chamado *checkpoint* que vai controlar a recuperação no caso de falhas; um processo chamado de *watchdog*, também responsável pela recuperação; e um processo chamado de seguidor que é uma réplica do processo líder.

Neste esquema a execução do processo líder é dividida em fatias de tempo que são chamadas de períodos (em inglês, *epoch*) e que são limitados por barreiras no programa de nível de usuário. Ao final de cada período as memórias do processo líder e do processo seguidor são comparadas. No caso de ausência de divergências nos valores o processo de checkpoint marca este ponto de execução como um ponto estável e confirma a saída da execução como válida. Caso contrário, o processo seguidor sinaliza um erro para o processo líder. O processo líder sinaliza para o processo de checkpoint iniciar de um ponto estável e logo após termina a sua execução simultaneamente com o processo seguidor. Quando o processo de checkpoint inicia ele se torna o novo processo líder criando o seu próprio processo seguidor. Nos casos em que o processo líder ou o processo seguidor não alcancem o ponto de final de um período, existe um quarto processo chamado de *watchdog* que possui temporizadores que quando expiram o seu tempo de contagem indicam uma situação de tempo excedido (em inglês *timeout).* O processo de *watchdog* com tempo excedido vai sinalizar ao processo de checkpoint que inicie a sua execução de recuperação de falhas.

2.1.3. Architectures for online error detection and recovery in multicore processors [GIZ11]

Gizopoulous et al. [GIZ11] apresentam uma análise feita sobre algumas das técnicas utilizadas para detecção de erros e recuperação de falhas online apresentadas em trabalhos de outros autores. Ele aprofunda em seus capítulos finais em três métodos para as tarefas propostas que são a *software-anomaly detection technique* (SWAT), uma técnica de verificação dinâmica chamada Argus, e uma metodologia chamada de *core salvaging methodology*.

O artigo mostra em resumo técnicas de detecção de erros ilustradas na Figura 3, cada uma classificada em uma determinada categoria:

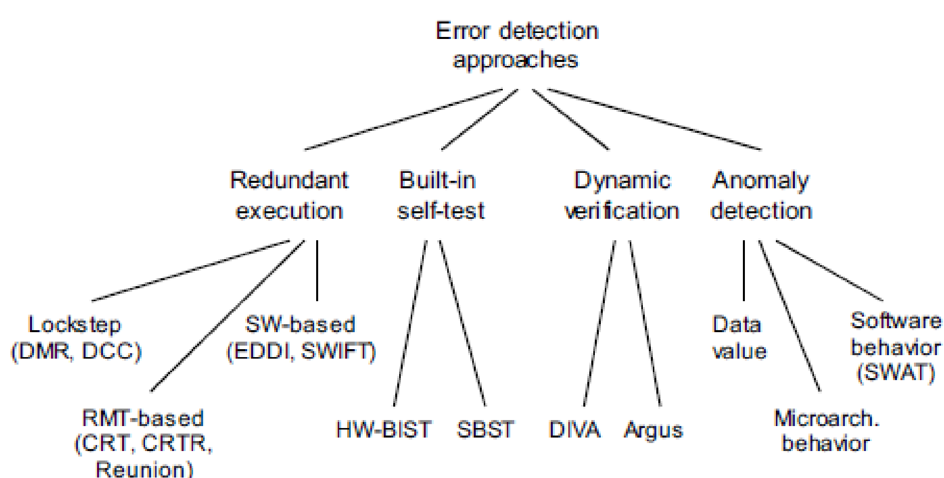


Figura 3 - Técnicas de detecção de erro [GIZ11]

A primeira categoria, execução redundante, significa ter duas threads independentes executando o mesmo programa com os mesmos resultados e comparando estes resultados para detectar anomalias. A segunda categoria é auto teste interno periódico, que compreende uma técnica de avaliação dos componentes internos periodicamente para detectar algum componente com falhas. Outra categoria é a de verificação dinâmica que significa ter hardware dedicado para verificação em tempo de execução. Por fim, a detecção de anomalia que utiliza hardware de baixo custo e monitores de software para detectar anomalias de execução dos processos ou sintomas decorrentes de falhas internas.

De acordo com os autores, as técnicas de recuperação de falhas podem ser classificadas em dois grupos: recuperação de erros para estado posterior, em inglês, *forward error recovery* (FER) e recuperação de erros para estado anterior, em inglês, *backward error recovery* (BER). Recuperação do tipo FER necessita de redundância

enquanto para implementação de técnicas BER existem soluções baseadas no uso de *checkpoints* e *rollback* para a tolerância a falhas.

Por fim, são apresentadas as soluções relevantes de detecção de erros e recuperação de falhas. A SWAT que compreende em uma solução para detectar, diagnosticar e recuperar falhas de hardware com baixo custo. SWAT detecta falhas de hardware na avaliação do software que está sendo executado, passando para um estágio de diagnóstico para isolar a fonte do problema e evitar a propagação da falha para outros componentes.

A segunda solução de detecção de erros se chama Argus e está baseada em verificação dinâmica, também chamada de teste online de software. Argus tem como princípio determinar parâmetros invariantes na execução do programa, desenvolver e executar verificadores para estes parâmetros em tempo de execução. Para identificar tais invariantes Argus utiliza-se da metodologia de divisão e conquistas subdividindo o sistema em subcomponentes aos quais vai identificar as invariantes. Para a definição dos verificadores, Argus parte da premissa que arquiteturas von Neumann executam três principais atividades: controle de fluxo (escolha das instruções a executar), computação das instruções e fluxo de dados onde os resultados são passados de um produtor para um consumidor. Argus implementa verificadores para cada uma destas atividades e os seus resultados experimentais confirmam que o Argus detecta erros com baixo custo em desempenho.

Finalmente o método de *core salvaging* tem como princípio isolar apenas instruções ou pipelines de instruções que apresentam defeitos em um determinado elemento de processamento desviando as duas sub sequentes execuções para outros PEs. Com isso o PE que apresentou o defeito ainda será utilizado para instruções de outra natureza que ainda possam funcionar. A proposta também enfatiza que pode ser utilizada em conjunto com a redundância completa de PEs sendo assim uma alternativa viável de maximização do uso dos recursos da microarquitetura.

2.1.4. A survey of rollback-recovery protocols in message-passing systems [ELN02]

Em [ELN02], foi apresentado um trabalho de avaliação de protocolos de *rollback and recovery* (RR) para sistemas com modelo de comunicação de troca de mensagens. Inicialmente os autores classificam os protocolos de RR em duas categorias chamando de baseados em checkpoint e baseados em log. São apresentados primeiramente os principais problemas e desafios da implementação de protocolos de RR em sistemas com

comunicação por troca de mensagens. Em seguida aprofunda nas principais categorias de protocolos.

O protocolo baseado em checkpoint usa os checkpoints que consistem em registros do estado do sistema no processador e das tarefas que estão executando no processador. Estes checkpoints são utilizados no processo de RR para reconfigurar aplicações em um processador garantindo coerência com o contexto estável de execução antes da falha que disparou o protocolo de RR. O autor ainda classifica as técnicas de RR com checkpoint em três categorias: *uncoordinated checkpoint*, que provê ao processador o máximo de autonomia para decidir quando o checkpoint é salvo; *coordinated checkpoint,* que consiste em ter um processador central para orquestrar os checkpoints dos processadores para garantir uma consistência global no sistema; e *communication-induced checkpoints,* que consiste em ter processadores gerando dois tipos de checkpoints: *local checkpoints*, que são criados de forma independente nos processadores e *forced checkpoints,* que são criados para garantir consistência do estado global do sistema sobrepondo-se ao Local quando este não tem valia para a possível execução de um protocolo de recuperação.

Na categoria de protocolos baseados em log o autor mostra que estes protocolos fazem uso do fato que a execução de processos pode ser modelada como sequência de intervalos de estados determinísticos. Estes intervalos são chamados de *determinants*. Os limites de cada *determinant* são dados pela ocorrência de eventos não determinísticos, como por exemplo o tratamento de interrupções. Cada processador também executa checkpoints como forma de evitar sobrecarga do *rollback* no processo de recuperação. A recuperação nestes casos dá-se com o uso dos dados dos *determinants* e *checkpoints* para reproduzir um evento não determinístico ocorrido antes da falha. Quando não é possível reproduzir um processo que tem a dependência de eventos não determinísticos com determinants e checkpoints é dito que este processo é órfão. Estes protocolos também são classificados em três categorias: *Pessimistic*, *Optimistic* e *Casual*. Protocolos *Pessimistic* garantem que processos órfãos não são criados. Os protocolos pessimistic simplificam a recuperação, mas tem um custo alto em desempenho. Protocolos *Optimistic* reduzem o custo em desempenho, mas permitem a criação de processos órfãos. Por fim, os protocolos Casual combinam os dois anteriores, mas possuem uma alta complexidade no processo de recuperação.

2.2. Gerência distribuída

2.2.1. Distributed resource management in NoC-based MPSoCs with dynamic cluster sizes [CAS13A]

Em [CAS13A], é apresentado um método para redimensionamento de clusters de um MPSoC. Este trabalho descreve um sistema multiprocessado com um número variável de processadores interligados por uma NoC de comunicação por troca de mensagens. Este MPSoC é dividido em clusters que possuem um PE em cada cluster que é designado para gerência de processos que vão ser executados. Neste ambiente serão executadas aplicações compostas por *n* tarefas que vão ser alocadas no MPSoC.

Quando um PE responsável pela gerência e mapeamento de tarefas no cluster requisita um novo PE no cluster para alocação de uma nova tarefa e este não encontra espaço disponível no cluster, procura então um nodo em clusters vizinhos. Após a execução de heurísticas de busca e negociação com os outros clusters um novo PE é alocado para esta aplicação alterando o tamanho inicial do cluster dinamicamente.

A Figura 4 demonstra os passos de aquisição temporária de um nodo em clusters vizinhos. O passo 1 mostra o mestre local LMP requisitando aos demais mestres da NoC um PE adicional para mapeamento de uma tarefa. O passo 2 mostra os mestres dos demais clusters enviando a resposta para que o LMP que fez a requisição. Estas respostas serão analisadas por um algoritmo de mapeamento definindo o PE que receberá o mapeamento dinâmico. O passo 3 mostra o LMP origem executando o comando *Loan Release* para abortar o processo de “empréstimo” nos clusters que não foram escolhidos. O passo 4 finaliza o processo enviando o pedido ao mestre global GMP que mapeie a tarefa no PE emprestado pelo cluster vizinho.

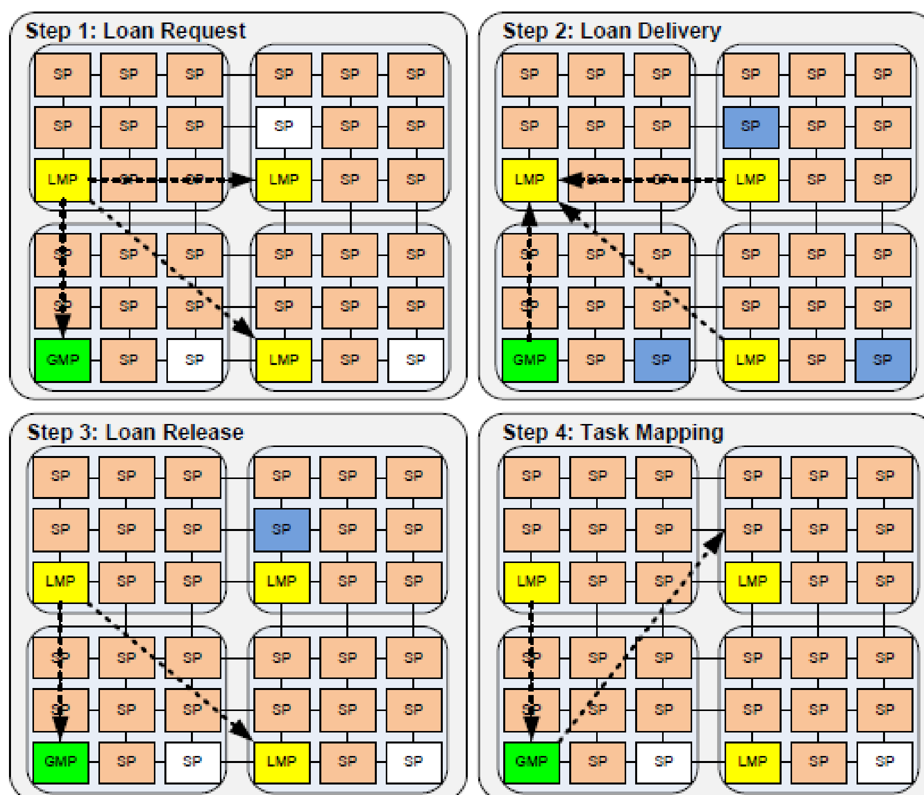


Figura 4 - ***Loan Process*** **em cluster dinâmicos [CAS13A]**

2.2.2. DistRM: Distributed resource management for on-chip many-core systems [KOB11]

Kobbe et al. [KOB11] descrevem um modelo de gerência descentralizada baseado em agentes de gerenciamento dos recursos que vão executar determinada aplicação no MPSoC baseado em NoCs. Esses agentes são aplicações de gerenciamento que executam em um PE próximo aos PEs que executam as tarefas de uma determinada aplicação. Seu domínio de conhecimento é o da aplicação e de outros agentes próximos. O modelo prevê a distribuição de um agente por aplicação, tendo estes agentes como objetivo acelerar a execução de uma aplicação e para isso avaliam o desempenho de uma aplicação, executam heurísticas de busca de novos PEs que estejam disponíveis para executar tarefas desta mesma aplicação. Seu objetivo é aumentar o paralelismo das tarefas para acelerar a execução total da aplicação. Um benefício adquirido com este modelo são a redução de pontos únicos de falha que os sistemas de gerenciamento de recursos centralizados possuem. Outro benefício está na redução de comunicação em longas distâncias pela NoC. Isto ocorre devido ao gerenciamento estar localizado próximo em relação aos PEs que executam determinada aplicação.

O método tem como característica executar de forma contínua as heurísticas de busca de novos PEs para acelerar a execução da aplicação e não apenas na inicialização

da aplicação. Este modelo conta com um agente especial chamado de agente de gerenciamento dos PEs livres que é responsável pelos PEs que não estão alocados por nenhuma aplicação. Suas responsabilidades são de fornecer os PEs livres para os agentes de aplicação e receber os PEs livres após a execução ter sido finalizada por uma aplicação.

Para avaliação deste método foram realizados testes em um ambiente de simulação em nível de sistema capaz de executar aplicações em diversas configurações arbitrárias coletando como métrica a aceleração da aplicação em função do tamanho do sistema em número de PEs. Os valores coletados no sistema rodando DistRM são comparados a um sistema de gerência centralizada. Os resultados mostram que o esquema centralizado ainda possui melhores resultados em termos de aceleração da aplicação comparado ao DistRM. No entanto, conforme o sistema escala, a diferença entre os métodos diminui tornando o método descentralizado viável com melhores heurísticas de mapeamento e melhorias no sistema de migração de tarefas.

A Figura 5 mostra o DistRM mostrou-se com melhor desempenho devido ao gerenciamento estar próximo aos PEs da aplicação dado que o número de cálculos necessários para realizar a aceleração da aplicação ficou abaixo do centralizado. Em um cenário de 32x32 PEs, com 32 aplicações, o esquema centralizado realizou aproximadamente 777.000 cálculos de aceleração enquanto o esquema do DistRM precisou apenas de 27.000 em média. Este valor distribuído entre os agentes ficou em torno de 850 cálculos por agente.

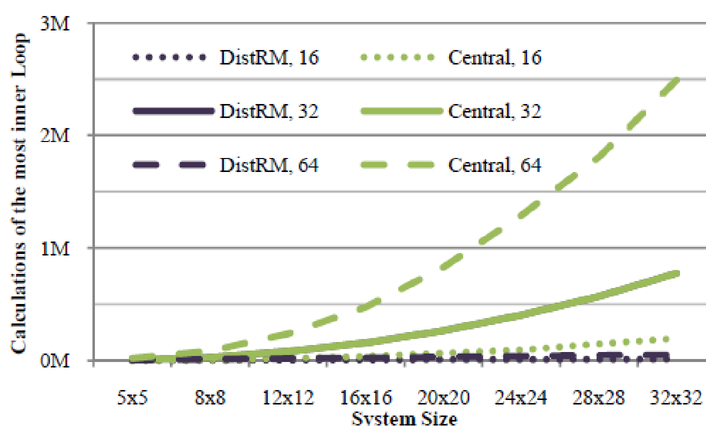


Figura 5 - Desempenho relativo ao tamanho do sistema [KOB11]

2.3. Migração de tarefas

2.3.1. Evaluating the impact of task migration in multi-processor systems-on-chip [ALM10]

Almeida et al. [ALM10] descrevem um trabalho de avaliação de impacto para migração de tarefas em MPSoCs baseados em NoC com o objetivo de balanceamento de carga em tempo de execução para ganho de desempenho.

Nos trabalhos relacionados estão citadas duas técnicas importantes para migração de tarefas: uma baseada em memória compartilhada e a outra em memória distribuída. Para a técnica de memória compartilhada observa-se que a migração de tarefas é facilitada considerando o fato de que tanto código, quanto a memória de dados, já estão acessíveis pelo PE de destino da migração. Esta característica garante menos tráfego de comunicação de dados pela rede e possui consumo reduzido de energia. Para o caso de memória distribuída, os desafios são maiores devido a questão de coerência de dados. Devido ao fato de cada PE possuir sua memória privada, ambos código e dados precisam ser migrados da memória do PE de origem para a memória do PE destino utilizando troca de mensagens em uma NoC.

A descrição da arquitetura utilizada para a proposta de migração de tarefas é descrita como um conjunto de PEs comunicando através de troca de mensagens em uma NoC. Neste contexto os PEs são chamados de NPU (*Network Processing Unit*) e cada NPU possui a capacidade de processamento de múltiplas tarefas. As NPUs possuem um pequeno sistema operacional responsável pela carga de tarefas. Estas tarefas possuem código em nível de aplicação para suportar os mecanismos de migração das tarefas que estão sendo propostos.

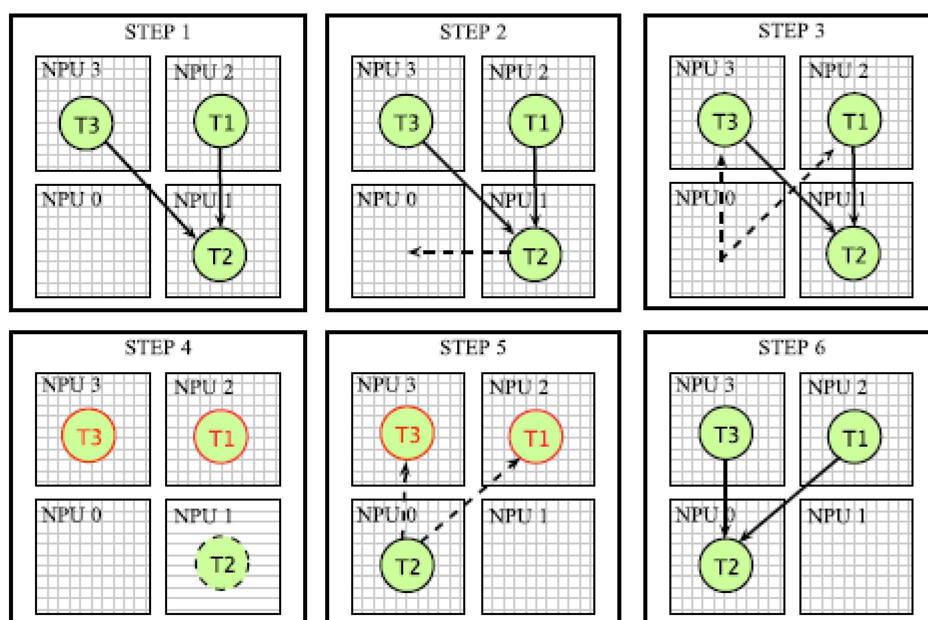


Figura 6 - Passos da migração de tarefas [ALM10]

Na validação dos cenários foi utilizada uma NoC com topologia mesh 2x2. O cenário descrito na Figura 6 compreende uma aplicação de multimídia dividida em três tarefas (T1, T2 e T3) que são distribuídas estaticamente por três NPUs. Quando a NPU1 decide migrar a tarefa T2 para a NPU0, NPU1 envia um pacote de controle para o PE mestre do cluster pedindo autorização para a migração. O cenário possui um nodo mestre localizado em NPU0 devido a este PE possuir uma tabela de roteamento global do cluster. O nodo mestre consulta sua tabela de roteamento para descobrir quais são as tarefas comunicantes com a tarefa a ser migrada. A seguir, o PE mestre envia um pacote de controle as tarefas comunicantes para informar que elas devem parar de enviar pacotes a tarefa a ser migrada. Assim sendo, a migração é realizada de NPU1 para NPU0. Após registrar a nova tarefa na memória do NPU destino o mestre envia um novo pacote para as tarefas que haviam sido congeladas para retomarem a enviar pacotes para o novo NPU em que T2 foi migrado.

2.3.2. Task Migration in Mesh NoCs over Virtual Point-to-Point Connections [GOO11]

Goodarzi et al. [GOO11] apresentam um método para migração de tarefas baseados em canais dedicados chamados de *virtual point-to-point connections* abreviados como VIP. A migração proposta tem como objetivo diminuir o custo de comunicação entre tarefas nos PEs do MPSoC obtendo diminuição na latência e consumo de energia. Além

disso, os canais virtuais dedicados para a migração mantêm os custos de latência e consumo de energia baixos.

A migração de tarefas impõe penalidades de latência para os tráfegos normais da rede. Devido a isso, [GOO11] propõe que os caminhos de comunicação para a migração sejam configurados com base no algoritmo *Gathering-Routing-Scattering* (GRS), no entanto a segunda fase de roteamento do algoritmo é substituída neste método por VIPs. Este algoritmo configura os PEs a serem migrados em diagonais adjacentes em um cluster origem migrando para um cluster destino com roteamento XY através dos canais VIP criados para o tráfego dedicado. A migração do método mostrado em [GOO11] ocorre pelos caminhos virtuais criados entre o cluster origem e o cluster destino e durante este processo de migração, os PEs que recebem os dados no cluster destino mantêm seu funcionamento normal.

Na análise de resultados [GOO11] compara o seu método de VIPs com dois métodos de roteamento em migração de tarefas: GRS sem VIPs e o método de Diagonais. A avaliação foi feita em uma simulação de NoC 16x6 migrando tarefas de uma sub malha 5x5 de origem para uma sub malha 5x5 de destino. Os resultados do método proposto foram extraídos em uma medição de taxa de geração de mensagens entre nodos em relação a três parâmetros: latência média, latência média de migração e consumo de energia. Nos três cenários, G-VIP-S obteve melhor desempenho com redução de 13% em latência média, 14% de latência média de migração e 10% de energia consumida em relação ao segundo melhor resultado que foi do algoritmo GRS.

2.4. Considerações finais

Este capítulo apresenta brevemente os trabalhos pesquisados nas principais áreas de conhecimento relevantes a este trabalho, tolerância a falhas, gerência distribuída e migração de tarefas. Foi possível comprovar a utilização de diversas técnicas que mostram bons resultados para superar os desafios dessas áreas de conhecimento. Algumas destas técnicas inspiram este estudo para alcançar o objetivo de garantir disponibilidade e confiabilidade almejadas para os sistemas MPSoC.

Tabela 1 - Avaliação de trabalhos relacionados

Trabalho	Explora tolerância a falhas	Explora gerência distribuída	Explora migração de tarefas	Organização da memória	Implementação
MUS13	Sim	Não	Não	Compartilhada	Software
DER13	Sim	Não	Sim	Distribuída	Hardware e Software
GIZ11	Sim	Não	Sim	Revisão de várias técnicas	Revisão de várias técnicas
ELN02	Sim	Não	Não	Revisão de várias técnicas	Revisão de várias técnicas
CAS12A	Não	Sim	Não	Distribuída	Software
ALM10	Não	Não	Sim	Distribuída	Software
KOB11	Não	Sim	Sim	Distribuída	Software
GOO11	Não	Não	Sim	Distribuída	Hardware e Software
Trabalho Proposto	Sim	Sim	Sim	Distribuída	Software

A Tabela 1 mostra que na área de tolerância a falhas, [MUS13] descreve um sistema de checkpoint e rollback para garantir a recuperação do sistema. No entanto a proposta se limita a configurações de sistema com memória compartilhada e necessita de adaptação da aplicação para a inserção de código específico para uso da técnica de tolerância a falhas. [DER13] apresenta uma solução para tolerância a falhas com checkpoint e rollback. A solução está implementada em hardware e software. No entanto tem a premissa de que todos os processadores do sistema já possuem o código das tarefas que estarão sendo executadas no sistema para simplificar a migração de contexto. O presente trabalho diferencia-se no aspecto de comunicação entre processadores em relação a [MUS13] que usa memória compartilhada, porque a técnica proposta é específica para troca de mensagens. Em relação a [DER13] que implementa software e hardware, a implementação apenas em software é o diferencial do atual trabalho, dado que esta técnica pode ser adaptada mais facilmente em diferentes ambientes de MPSoC.

Ainda na área de tolerância a falhas, [GIZ11] faz uma revisão geral de algumas técnicas e apresenta 3 métodos com foco em detecção de erros e recuperação, porém o trabalho apenas avalia as técnicas de forma genérica e traça um comparativo de suas utilizações sem especificar o tipo ou configuração de MPSoC que obteria melhores resultados para as técnicas. Por fim, foi apresentado uma revisão sobre recuperação com

checkpoint e rollback em [ELN02]. Ambos trabalhos têm relevância nesta revisão dado que as técnicas apresentadas foram importantes na avaliação de viabilidade de implementação dos métodos utilizados neste trabalho.

Em gerência distribuída são apresentados métodos em [KOB11] e [CAS12a] que exploram mapeamento distribuído em tempo de execução. O uso de gerência distribuída permite que o sistema reduza pontos únicos de falha e aumenta o desempenho do gerenciamento com a redução do custo de comunicação na NoC devido à redução de distância do gerente com o PE escravo. O método utilizado em [CAS12a] tem relação mais próxima com os métodos e implementação desenvolvida neste trabalho dado que o ambiente de desenvolvimento utilizado foi o mesmo, HeMPS [CAR09]. Estes métodos descritos em [CAS12a] e [KOB11] vão contribuir no estudo e desenvolvimento da proposta deste trabalho dado que a técnica de recuperação proposta vai ser orquestrada por PEs gerenciadores de clusters de aplicações.

Em migração de tarefas, os métodos apresentados por [ALM10] e [GOO11] estão focados em balanceamento de carga redução de latência e consumo de energia. Entretanto, para este trabalho as propostas de migração são adaptadas para tolerância a falhas.

3. DESENVOLVIMENTO

Considerando o cenário revisado onde os futuros MPSoCs vão escalar a um número considerável de processadores, é correto afirmar que haverá muitos desafios para manter disponibilidade e a confiabilidade da operação do chip. Os autores revisados apresentam importantes métodos como tolerância a falhas com redundância das tarefas, *checkpoint* e *rollback* para recuperação de contexto e migração de tarefas para um PE saudável. Estas técnicas inspiram o presente trabalho no desenvolvimento de um modelo de recuperação de falhas a partir da detecção de erros em PEs de um MPSoC configurado em uma NoC.

Considerando a revisão realizada, a *originalidade* do presente trabalho está na integração de conceitos utilizados em programação paralela e distribuída (como migração de tarefas para recuperação de falhas) no contexto de MPSoCs baseados em comunicação por troca de mensagens e gerência distribuída.

3.1. Estudo de caso – HeMPS

A plataforma HeMPS, sigla para Hermes Multi Processor System-on-Chip [CAR09], é o ambiente usado para o estudo de caso neste trabalho. Consiste em uma plataforma para estudo de MPSoCs e tem como configuração, processadores homogêneos interconectados por uma Network on Chip (NoC) chamada Hermes [MOR04].

3.1.1. Plataforma de Hardware

A Figura 7 ilustra a HeMPS composta por um conjunto variável de processadores plasma [PLA14] conectados a NoC Hermes e a um repositório de tarefas.

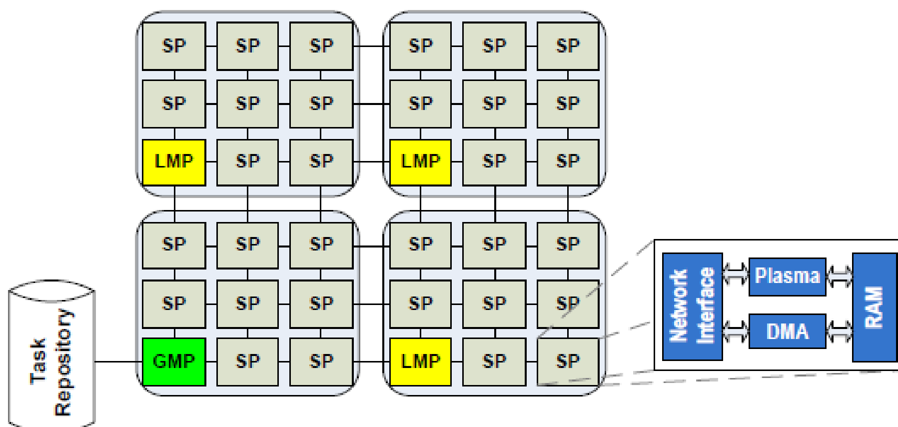


Figura 7 - HeMPS gerência distribuída [CAS12A]

O processador plasma possui uma interface de rede que conecta-se a NoC, um módulo de DMA (do inglês, *Direct Memory Access*) para transferir e receber os códigos fonte das tarefas que irão executar nos PEs escravos. E por fim um módulo de memória RAM onde executa o *microkernel* do PE. Existem 3 tipos de processadores na HeMPS quanto a sua função: o *Global Master Processor* (GMP), o *Local Master Processor* (LMP) e o *Slave Processor* (SP). GMP é o único PE conectado diretamente ao repositório de tarefas e tem a função da gerência global da HeMPS. LMP são configurados na quantidade de 1 por cluster. Sua função é de gerenciar os PEs de seu cluster e comunicar com o mestre global GMP. Por fim, o SP que atua como o PE responsável pela execução de tarefas da HeMPS. O repositório de tarefas é uma memória que armazena o cabeçalho de informações e o código fonte de todas as tarefas que vão executar no sistema.

3.1.2. Plataforma de software

A plataforma de software é composta basicamente por código que executa em nível de sistema, chamado de *microkernel,* e por código que executa em nível de usuário, chamado de aplicação. O *microkernel* executa funções de gerenciamento, escalonamento, comunicação e execução de tarefas de aplicações do repositório. Dado que a HeMPS pode ser configurada com dois modelos de gerenciamento, centralizado e distribuído, existem três tipos de *microkernel* que diferenciam a função de cada elemento de processamento. A aplicação é composta por várias tarefas que são executadas paralelamente em um ou mais PEs.

3.1.2.1. Kernel slave

Este kernel, executado no SP, tem o objetivo de executar as tarefas do nível de aplicação e gerenciar a comunicação entre as mesmas.

A comunicação entre os SPs é feita por meio de estruturas de dados que são enviadas e recebidas por mecanismo de DMA chamadas de “*Pipe*”. A Figura 8 mostra a definição da estrutura de um slot de *pipe* do kernel slave da HeMPS.

```c
typedef struct {
    unsigned int remote_addr;        /* Remote processor address */
    unsigned int pkt_size;           /* NoC packet size (flits) */
    unsigned int service;            /* Service identifier */
    unsigned int local_addr;         /* Local processor address */
    unsigned int target;             /* Target task */
    unsigned int source;             /* Source task */
    unsigned int length;             /* Message lenght (32 bits words)*/
    unsigned int message[MSG_SIZE];
    enum PipeSlotStatus status;
    unsigned int order;
} PipeSlot;
```

Figura 8 - Estrutura de um slot de Pipe

Os *PipeSlots*, são utilizados para identificação da mensagem. Possuem um campo para identificação do alvo da mensagem chamado de *remote_addr*. Um campo de indicação do tamanho do pacote: *pkt_size*. O terceiro campo chamado *service* é o identificador do comando que está sendo enviado. O próximo campo chamado *local_addr* é o campo de identificação do PE origem da mensagem. Os campos seguintes são de identificação das tarefas origem e destino que estão comunicando: *target* e *source*. A mensagem é armazenada no campo *message* que contém tamanho parametrizável em tempo de compilação. O pipe ainda possui um campo de indicação se o slot está sendo ou não utilizado *status* e por fim, a ordem da mensagem é indicada no campo *order*. A estrutura de *Pipe* está implementada no kernel slave como um vetor de *PipeSlots* de tamanho parametrizável.

A execução das tarefas da aplicação é possibilitada por um escalonador implementado com o algoritmo "*Round-Robin*". As tarefas são modeladas e gerenciadas com uma estrutura de dados chamada *Task Control Block* (TCB). A Figura 9 mostra como está definida a TCB.

```c
typedef struct {
    unsigned int reg[30];       //30 registers (Vn,An,Tn,Sn,RA)
    unsigned int pc;            //program counter
    unsigned int offset;        //initial address of the code
    unsigned int id;            //identifier
    unsigned int status;        //status (READY, WAITING, TERMINATED, RUNNING, FREE, ALLOCATING)
} TCB;
```

Figura 9 - Task control block

A definição mostra os campos *reg* que armazena os valores dos registradores no contexto de execução da tarefa. O campo chamado *pc* armazena o endereço de memória da instrução atual que está sendo executada. *Offset* armazena o endereço inicial do código binário da tarefa. O campo seguinte é o identificador da tarefa: *id*. O campo *status* das TCB no sistema é responsável por manter o estado de execução da tarefa no escalonador. Uma tarefa pode passar pelos seis estados definidos na Tabela 2:

Tabela 2 - Status da TCB

STATUS	**DESCRIÇÃO**
READY	A tarefa está pronta para executar.
RUNNING	Indica que esta tarefa está executando na CPU.
TERMINATED	Indica que esta tarefa terminou a sua execução.
WAITING	A tarefa requisitou uma mensagem e está esperando pela resposta.
FREE	Indica que a TCB está livre e uma tarefa pode ser alocada.
ALLOCATING	Indica que uma TCB está sendo alocada.

Por fim, o kernel slave possui uma função (HandlerNI) responsável por tratar mensagens de gerenciamento enviadas pelos demais kernels: kernel master e kernel local. Os serviços recebidos nesta interface são detalhados na subseção seguinte de kernel master.

3.1.2.2. Kernel master e kernel local

Kernel master executa no GMP e faz a gerência dos recursos do MPSoC, bem como acessa diretamente o repositório de tarefas para alocar as mesmas nos outros PEs da HeMPS. Entre suas principais tarefas pode-se enumerar: Inicialização de clusters, inicialização de SPs, mapeamento e alocação de tarefas. Em configurações de gerência centralizada o GMP é o único PE gerente do sistema. Em uma configuração de gerência distribuída o kernel master vai atuar como o gerente global e também como gerente de um dos clusters criados. As aplicações que executam no sistema são descritas no kernel master em uma estrutura chamada *ApplicationPackage* como mostrado na Figura 10.

```
typedef struct {
    int task;
    int flits;
} DependencePackage;

typedef struct {
    int id;
    int code_size;
    int initial_address;
    int proc;
    int static_position;
    int dependences_number;
    DependencePackage dependences[10];
} TaskPackage;

typedef struct {
    int size;
    TaskPackage tasks[MAX_APP_SIZE];
} ApplicationPackage;
```

Figura 10 - Estrutura de uma aplicação no kernel master / local

Esta estrutura armazena as informações básicas para alocação das aplicações que serão executadas em uma simulação da HeMPS. Inicialmente a estrutura principal possui uma lista de estruturas que armazenam tarefas: *TaskPackage*. O campo de *size* indica quantas tarefas estão descritas em *TaskPackage*. A estrutura TaskPackage possui os seguintes campos: um identificador da tarefa chamado id; um campo que indica o tamanho do código binário chamado *code_size*; o campo *initial_address* que indica o endereço inicial da tarefa no repositório de tarefas; o campo *proc* que indica o processador onde executa esta tarefa; um campo chamado de *static_position* que tem o objetivo de sinalizar ao algoritmo de mapeamento que esta tarefa precisa ser alocada em um PE específico. Os dois últimos campos do descritor da tarefa se chamam *dependencies_number* e *dependences*. Estes campos são utilizados pelo algoritmo de mapeamento para identificar as tarefas que são dependências das tarefas que está descrita na estrutura. As dependências têm peso no mapeamento para garantir que as tarefas sejam alocadas próximas na NoC, evitando altos custos de tráfego de pacotes devido à distância.

O kernel local executa no LMP e também tem funções gerenciamento semelhantes ao kernel master, entretanto o seu escopo de atuação é dentro do cluster em que está encarregado de gerenciar. No entanto para serviços de alocação de tarefas nos SPs, o kernel local ainda precisa requisitar ao kernel master pois o acesso ao repositório está apenas a cargo do kernel master que executa no GMP.

A comunicação entre kernel master, local e slave é feita pelos chamados serviços da HeMPS. Os serviços são utilizados para enviar comandos de gerenciamento das aplicações do sistema e de gerência do MPSoC. Estes serviços são enviados como uma sequência de bytes pela network interface. Estes bytes serão recebidos por um PE destino que vai decodifica-los em um determinado comando que precisa ser executado. A Tabela 3 descreve os principais serviços da HeMPS.

Tabela 3 - Serviços de gerenciamento de kernel da HeMPS

Serviço	**Descrição**
MESSAGE_REQUEST	Requisição de mensagem.
MESSAGE_DELIVERY	Entrega de mensagem.
TASK_ALLOCATION	Alocação de uma tarefa em um SP.
TASK_ALLOCATED	Sinaliza que uma tarefa foi alocada e indica a posição de alocação.
TASK_REQUEST	Requisita uma tarefa para o LMP ou GMP.
TASK_TERMINATED	Sinaliza o término de uma tarefa.
TASK_DEALLOCATED	Sinaliza que uma tarefa foi desalocada.
LOAN_PROCESSOR_RELEASE	Libera um PE de um empréstimo a um cluster vizinho.
NEW_TASK	Pedido de uma nova tarefa do repositório.
APP_TERMINATED	Sinaliza o término de uma aplicação.
NEW_APP	Pedido de uma nova aplicação do repositório.
INITIALIZE_CLUSTER	Configura e inicia um cluster.
INITIALIZE_SLAVE	Configura e inicia um SP.
TASK_TERMINATED_OTHER_CLUSTER	Sinaliza que uma tarefa que executava em um PE emprestado em cluster vizinho terminou sua execução.
LOAN_PROCESSOR_REQUEST	Requisita um PE emprestado a um cluster vizinho.
LOAN_PROCESSOR_DELIVERY	Sinaliza um pedido de sucesso de empréstimo de PE a cluster vizinho.

3.1.3. Aplicações

As aplicações são programas compostos por 1 ou mais tarefas que são carregados para a memória RAM dos PEs na HeMPS para que sejam executados na simulação. As aplicações utilizam comandos de *send* e *receive* para efetuar comunicação inter tarefa.

As assinaturas dos comandos *Send* e *Receive* de troca de mensagens entre as tarefas são mostradas na Figura 11.

```
Send(unsigned int *message, unsigned int * target);
Receive(unsigned int *message, unsigned int * source);
```

Figura 11 - Funções *Send* e *Receive*

O parâmetro message armazena a informação a ser transmitida e os parâmetros target e source são ponteiros para as tarefas comunicantes. Essas funções executam leituras e escritas nos Pipes de comunicação que vão transmitir as mensagens configuradas para transmissão e recepção.

3.1.4. Simulação na HeMPS

A operação HeMPS dá-se a partir da configuração de um arquivo de parâmetros com extensão "*.hmp*" que indicam as características básicas do MPSoC e dos kernels que são instalados em cada PE. A Figura 12 demostra um exemplo de um arquivo de configurações da HeMPS.

```
[project name]
SumVecApp
[tasks per pe]
3
[processor description]
sc
[noc buffer size]
8
[noc routing algorithm]
west
[dimensions]
4
4
[cluster size]
2
2
[masters location]
LB
[global master]
0
[application]
mpeg
```

Figura 12 - Exemplo de um arquivo de configuração da HeMPS

O arquivo possui como primeiro parâmetro *[project name]*, o nome do folder de projeto em que todos os artefatos gerados na simulação vão ser armazenados (ex. logs, configurações, códigos binários, etc). O segundo parâmetro, *[tasks per pe]*, indica o número de tarefas que cada SP pode executar. *[processor description]* é o parâmetro que indica se o hardware está descrito em linguagem SystemC ou VHDL. Os dois próximos parâmetros indicam características da NoC, *[noc buffe size]* e *[noc routing algorithm]* e significam respectivamente tamanho do buffer da NoC e qual algoritmo de roteamento está sendo usado. *[dimensions]* indica as dimensões da rede de PEs. *[cluster size]* indica as dimensões dos clusters em configuração de gerência distribuída. *[master location]* indica a posição dos mestres GMP e LMP em cada cluster, (ex. LB – em inglês *left-bottom* – canto de baixo com o lado esquerdo de cada cluster). O parâmetro *[global master]* indica o número identificador do mestre global. Por fim, a lista de todas as aplicações que serão executadas na simulação. Cada aplicação é precedida da tag *[application]*.

A simulação de funcionamento do MPSoC parte da carga de aplicações nos processadores SP. As aplicações são requisitadas pelo kernel master ao repositório de tarefas e carregadas via um dispositivo de DMA no SP. Cada SP tem a capacidade de processar as tarefas e sinalizar o término de execução de cada tarefa. Ao receber tais

sinalizações, o kernel master identifica que todas as tarefas de todas as aplicações finalizaram sua execução e sinaliza o término da simulação.

As aplicações e os kernels podem executar escritas em arquivos texto com comandos de "*puts(char *)".* Estas escritas podem ser informações sobre a simulação, debug (número de *ticks* do processador, conteúdo de variáveis, etc) e escritas de aplicações. Todas as escritas são salvas em arquivos ".txt" respectivos a cada PE do MPSoC.

3.2. Estudo de alternativas de projeto

Esta seção apresenta um estudo preliminar que ajudou a determinar o método proposto de recuperação de tarefas afetadas por PEs com falhas. A proposta inicial deste trabalho era a implementação de um método de checkpoint e rollback. Entretanto, após um estudo mais aprofundado de artigo anteriores [ELN02], algumas dúvidas em relação ao custo-benefício da implementação de checkpoints foram levantadas. Na técnica de checkpoint, o sistema salva o estado atual do processador em momentos estáveis de execução armazenando em uma memória protegida de falhas um conjunto de checkpoints. Quando o rollback é acionado, será solicitado o checkpoint mais recente da memória para que o sistema de rollback carregue, em um PE saudável, o contexto de execução estável da tarefa falhada. Esta abordagem levantou algumas preocupações relacionadas a sua implementação em sistemas embarcados:

- *Checkpoints* foram propostos inicialmente para aplicações com longos tempos de execução (ex. horas). Esta característica de execução não é comum na execução de MPSoCs, especialmente no domínio de aplicações de sistemas embarcados;
- Muitas das abordagens requerem um gerenciamento de memória complexo incluindo por exemplo, "*garbage collection"* para deletar entradas de contextos obsoletos;
- O local de armazenamento dos checkpoints para cada tarefa no sistema que, sem uma memória compartilhada, implica em um armazenamento distribuído, fazendo com que o uso de memória de contexto escale com o tamanho do sistema o que pode ser inviável para MPSoCs;
- Alto custo em desempenho, largura de banda na NoC e energia para executar checkpoints periódicos e manter consistência dos checkpoints distribuídos;

- Algumas abordagens de checkpoints possuem a necessidade de instrumentação do código de aplicação para executar checkpoints;
- Algumas abordagens de checkpoint não tratam eventos assíncronos e não determinísticos como tratamento de interrupções.

Diferente das aplicações de grande porte como, computação em clusters, atualmente as aplicações em MPSoCs atingem o mercado consumidor de eletrônicos que demandam maior poder de processamento e capacidades multimídia. A tolerância a falhas é necessária para atingir a disponibilidade e confiabilidade possibilitando o funcionamento do sistema que pode ser afetado pelos problemas de fabricação e envelhecimento.

Uma abordagem simplificada e eficiente de tolerância a falhas sem salvamento e recuperação de contexto pode sofrer perda de dados já executados no momento da detecção de uma falha permanente. Entretanto, ela deve garantir que dados futuros sejam executados normalmente. Apesar da desvantagem de perda de dados, uma abordagem simplificada de tolerância a falhas ainda é eficaz no âmbito de aplicações de MPSoC, pois permite que o sistema continue funcionando imediatamente depois que o local da falha esteja isolado. Além disto, não teria o custo de desempenho, complexidade de software, e energia consumida de uma abordagem com checkpoint e rollback convencional. Por fim, aplicações multimídia, muito comuns no contexto de MPSoC, são naturalmente resilientes a perdas momentâneas de dados. Por exemplo, a perda de um frame de vídeo vai ser imperceptível ao usuário final.

Avaliando este cenário e os desafios inerentes a implementação do ckeckpoint na HeMPS, decidiu-se neste primeiro momento executar um rollback sem checkpoint, ou seja, sem recuperação de contexto. Neste sentido, a aplicação que possui um processo afetado por uma falha vai ser reiniciada automaticamente, sendo que somente a tarefa afetada sofrerá o remapeamento para um PE saudável. No futuro, caso esta primeira abordagem mostre resultados satisfatórios, será possível pesquisar formas eficientes de executar checkpoint em MPSoCs, bem como o estudo de técnicas alternativas com suporte a hardware como o proposto em [DER13].

3.3. Detalhamento da solução desenvolvida

A solução para implementar a tolerância a falhas na HeMPS, parte do princípio de que as falhas são permanentes no PE. O protocolo de recuperação vai ser disparado por

um evento de detecção de uma falha permanente. A seção 3.3.1 detalhada o tratamento da detecção de falhas no método de tolerância a falhas proposto. A seção 3.3.2 descreve as etapas do protocolo de recuperação e como elas foram projetadas, os novos serviços criados, e o algoritmo de recuperação a falhas desenvolvido para a plataforma HeMPS.

3.3.1. Teste e detecção de falhas

Várias abordagens de teste podem ser utilizadas para testar e detectar falhas em elementos de processamento [GIZ11]. Teste é, ao mesmo tempo, uma das etapas de maior custo de um método de tolerância a falhas e principalmente a etapa mais dependente das restrições de confiabilidade e disponibilidade de uma aplicação. Desta forma, o método proposto está concentrado na etapa de recuperação de falhas, proporcionando a escolha de qualquer método desejável para teste e detecção de falhas de acordo com a necessidade da aplicação. Algumas abordagens de teste que podem ser citadas como exemplo estão relacionadas a seguir:

- BIST periódico transparente (implementado tanto em software quanto em hardware) na memória local, dado que esta abordagem BIST não apaga conteúdos de memória [WAN06];
- Uso de BIST baseado em lógica STUMPS para bus interno, interfaces de rede e lógicas de interface [WAN06];
- Auto teste estrutural/funcional de núcleo de processamento periódico baseado em software [PSA10];
- Verificação dinâmica e detecção de anomalia [GIZ11];
- Monitores de prognóstico podem ser usados para avaliar o envelhecimento do elemento de processamento e disparar o protocolo de recuperação de falhas antes da falha ser detectada [KER14].

Considerando que o presente trabalho concentra-se apenas no protocolo de recuperação de falhas, assume-se a existência de um método de teste que vai detectar falhas permanentes nos elementos de processamento escravos (SP). Para que a execução do protocolo de recuperação a falhas seja iniciada foi implementado um serviço adicional de simulação de falhas. Este serviço vai ser detalhado nas seções seguintes.

3.3.2. Protocolo de recuperação

O protocolo de recuperação possui 6 passos que compreendem as fases de detecção da falha, migração das tarefas falhadas e reinício das tarefas da aplicação. Para que as ações do protocolo sejam executadas foram criados novos serviços do sistema que são explorados na seção seguinte deste capítulo.

A partir das seguintes definições:

- PE_M - PE manager representa GMP ou LMP. O GMP apenas acessa o repositório de tarefas para realizar novas realocações e o gerenciamento do cluster fica a cargo do LMP.
- PE_F – PE que apresenta falha permanente.
- TF = $\{t_1, t_2, ..., t_n\}$ – Representa um conjunto de tarefas afetadas pelo PE_F.
- TA = $\{A_1, A_2, ..., A_m\}$ – Conjunto de aplicações afetadas que precisam ser reconfiguradas.

Os passos do protocolo de recuperação em um cenário de rede 4x4 mostradas na Figura 13 são descritos a seguir:

1. PE_M recebe uma notificação de falha em um determinado PE_F. PE_F é marcado como “defeituoso” ou “excluído” e não será mais considerado em futuros mapeamentos de tarefas.
2. Considerando que o PE_M sabe que tarefa está mapeada em cada PE, conforme descrito na seção kernel master, o PE_M constrói um conjunto TF das tarefas afetadas. PE_M identifica para cada elemento de TF, as aplicações que foram afetadas e constrói o conjunto TA.
3. PE_M envia uma mensagem de *freeze* para todas as tarefas quem compõem as aplicações em TA.
4. PE_M remapeia as tarefas de TF em PEs saudáveis conforme algoritmo de mapeamento proposto por [MAN11a] [MAN11b]. Todas as outras tarefas não afetadas pela falha permanecem nas mesmas localizações, minimizando tráfego de código na rede já que apenas as tarefas falhas serão realocadas. Uma sinalização é enviada para as tarefas não afetadas para indicar as novas localizações das tarefas remapeadas.
5. PE_M envia o comando de *task_allocation* que carrega as tarefas nos PEs saudáveis.
6. PE_M envia mensagens de *unfreeze* que irão reiniciar todas as tarefas de cada aplicação de TA.

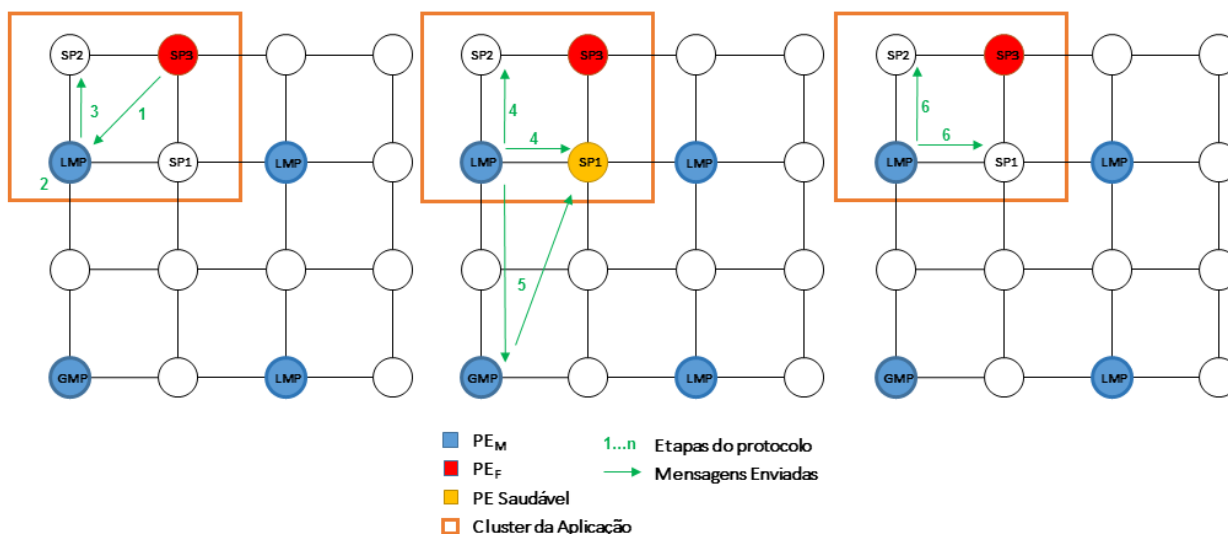


Figura 13 - Protocolo de recuperação na HeMPS

Para ilustrar a implementação do protocolo descrito acima, a Figura 14 representa o algoritmo que executa no kernel master ou local para recuperação de falhas.

```
1.  TF=getTasksFromPE(PEf)
2.  TA= ø
3.  // send freeze message
4.  for each t in TF
5.     TA= TA U getApp(t)
6.  for each app in TA
7.    for each pe in (getPE(app)-PEf)
8.      send(pe,freeze,t)
9.  // perform task reallocations
10. for each t in TF
11.   new_pe = dynamic_mapping(t)
12.   setPE(t,new_pe)
13.   send(new_pe,task_allocation,t)
14. for each app in TA
15.   for each pe in getPE(app)
16.     send(pe,reallocated)
17. // send unfreeze message
18. for each app in TA
19.   for each pe in getPE(app)
20.     send(pe,unfreeze,t)
```

Figura 14 - Pseudocódigo da recuperação de falhas no kernel master ou local.

O código das linhas 1 até 5 constroem o conjunto TA de aplicações afetadas. Entre as linhas 6 e 8 está a execução da procura por tarefas que não foram afetadas pela falha e envia a mensagem de *freeze* para todos eles. Nas linhas 9 até 13, todas as tarefas afetadas em TF pela falha são remapeadas para novos PEs saudáveis transferindo o código binário das tarefas para o novo destino. Finalmente, no último laço das linhas 14 a 20, o PE$_M$ informa as localizações das tarefas realocadas e envia as mensagens de *unfreeze* para todas as aplicações de TA.

```
1.  msg = recv()
2.  app_id = msg.app_id
3.  switch (msg.service)
4.    case freeze:
5.      for each task in task_list
6.         if task.app_id == app_id
7.            task.state = freeze
8.    case unfreeze:
9.      for each task in task_list
10.        if task.app_id == app_id
11.          clean_task_tcb(task.id)
12.          task.state = ready
13.     for each pipe_slot in pipe
14.        if pipe_slot.app_id == app_id
15.           pipe_slot.status = empty
```

Figura 15 - Pseudocódigo da recuperação de falhas no kernel slave

O algoritmo que executa no kernel slave é representado na Figura 15. O kernel slave recebe mensagens de gerenciamento como está representado na linha 1. Assim que a mensagem é decodificada o kernel vai identificar qual serviço está sendo recebido. Em caso de *freeze* (linha 4 até 7), o kernel vai buscar na lista de TCB pelo identificador da aplicação as tarefas que precisam ser configuradas com o status de *FREEZE*. Em caso de *unfreeze* (linha 8), o kernel reinicia a TCB e configurar o status da TCB para *READY* nas linhas 11 e 12. Por fim executa a limpeza do *pipe* entre as linhas 13 e 15.

3.3.3. Novo estado da TCB

O estado de *FREEZE* como demonstra a Tabela 4, é o estado novo adicionado na TCB do kernel slave na HeMPS para o protocolo de recuperação de falhas. Uma tarefa com estado *FREEZE* não é escalonada. Desta forma todo o processamento e comunicação inter tarefa são suspensos.

Tabela 4 - Estados da TCB

STATUS	**DESCRIÇÃO**
READY	A tarefa está pronta para executar.
RUNNING	Indica que esta tarefa está executando na CPU.
TERMINATED	Indica que esta tarefa terminou a sua execução.
WAITING	A tarefa requisitou uma mensagem e está esperando pela resposta.
FREE	Indica que a TCB está livre e uma tarefa pode ser alocada.
ALLOCATING	Indica que uma TCB está sendo alocada.
FREEZE	**Indica que uma TCB está congelada e a tarefa aguarda o término do processo de recuperação de falhas.**

3.3.4. Novos serviços do protocolo

Como parte do desenvolvimento do módulo de tolerância a falhas da HeMPS foi necessária a criação de novos serviços. Estes serviços têm o objetivo de executar as etapas do protocolo de tolerância a falhas descritas anteriormente. A Tabela 5 apresenta estes serviços.

Tabela 5 - Novos serviços da HeMPS

Serviço	**Descrição**
FORCE_FAIL_PROCESSOR	Simula um serviço de detecção de falhas. Recebido pelo kernel master.
FREEZE_APPLICATION	Congela todas as tarefas de uma aplicação em um determinado SP.
UNFREEZE_APPLICATION	Reinicia todas as tarefas que estão em estado de *FREEZE* em um determinado SP.
TASK_ALLOCATION *	Alocação de uma tarefa em um SP. Foi alterado adicionando mais um parâmetro para indicar que é uma realocação de tarefa.
TASK_REALLOCATED	Sinaliza a nova localização de uma tarefa que sofreu realocação.

*** - Adaptação de um serviço já existente antes da implementação do protocolo de recuperação a falhas.**

O primeiro é o serviço de *FORCE_FAIL_PROCESSOR* que é executado no kernel master e tem como objetivo simular o recebimento de uma notificação de falha permanente em um PE. Este serviço tem o propósito único de simulação para avaliação do protocolo de recuperação de falhas. Em um sistema real, o método de teste deve enviar uma notificação de falha permanente para o kernel master ou kernel local.

O serviço de *FREEZE_APPLICATION* coloca todas as tarefas de um PE no estado *FREEZE*. Este estado impede que a tarefa seja escalonada. O comando de *FREEZE_APPLICATION* deve ser enviado a todas as tarefas pertencentes a uma aplicação afetada por uma falha. Cada tarefa vai ter a sua TCB configurada com o status *FREEZE*. Desta forma, todas as comunicações inter tarefa são suspensas evitando acúmulo de mensagens no *pipe* de comunicação.

UNFREEZE_APPLICATION muda o status das TCBs das tarefas que estiverem com estado de FREEZE para o estado de READY. *UNFREEZE_APPLICATION* possui uma funcionalidade adicional que é a de reiniciar todas as tarefas da aplicação que foi afetada pela falha. Reiniciar a aplicação significa que todas as suas tarefas vão retornar ao sistema a partir de seu estado inicial. Dessa forma não haverá problemas de sincronismo e incoerência de comunicação inter tarefas dado que mensagens pertencentes a instância anterior da aplicação não vão afetar a nova instância da

aplicação. Para reiniciar uma aplicação o serviço de UNFREEZE_APPLICATION vai reconfigurar as estruturas TCB de cada tarefa da aplicação para representar o estado inicial dessas tarefas. O atributo pc é atribuído com o valor ‘0’ e o campo status é atribuído com o estado READY. O kernel deve também remover todas as mensagens em espera no pipe de comunicação com o identificador da aplicação que esta TCB pertence. Após percorrer o vetor de pipe slots o algoritmo procura em cada slot o identificador da aplicação que falhou. Caso encontre o campo “status” do pipe slot vai ser configurado para o valor “EMPTY” que significa slot livre. Sendo assim, esta mensagem está removida do pipe e não pode ser mais executada em uma nova instância da aplicação. O pseudocódigo que representa a limpeza dos pipeSlots está demonstrado na Figura 16.

```
1.     for each pipe_slot in pipe
2.        if pipe_slot.app_id == app_id
3.           pipe_slot.status = empty
```

Figura 16 - Pseudocódigo - limpeza do pipe

O serviço de *TASK_ALLOCATION* foi alterado para diferenciar seu uso em uma alocação inicial de uma realocação devido ao protocolo de recuperação de falhas. Foi adicionado um novo parâmetro ao comando indicando se a alocação trata-se de uma realocação ou não. Esta alteração foi necessária pois a TCB criada no SP por este serviço automaticamente é configurada com o estado de *READY.* Entretanto na situação de realocação, a TCB precisa se manter com o estado de *FREEZE*. Somente o recebimento do serviço de *UNFREEZE_APPLICATION* vai alterar o estado da tarefa realocada para o estado *READY*. Dessa forma fica garantido o reinício das tarefas que foram realocadas simultaneamente com as outras tarefas da aplicação.

O último serviço chama-se de *TASK_REALLOCATED* e tem a função de sinalizar a todas as tarefas de uma aplicação afetada por uma falha permanente a nova localização das tarefas realocadas. Este serviço é executado após o mapeamento das tarefas falhadas e vai ser enviado a todas as tarefas da aplicação que já estiverem alocadas no sistema.

3.3.5. Sequência de comunicação dos serviços do protocolo de recuperação na HeMPS

Na Figura 17 está representado o diagrama de comunicação do protocolo sendo executado em uma configuração de gerência distribuída. O diagrama mostra a ordem em que os novos serviços implementados na HeMPS são executados a partir do início do processo de recuperação. Neste cenário as tarefas da aplicação estão alocadas e

executando inicialmente nos PEs SP2 e SP3. A falha ocorre no PE SP3 e a tarefa que apresenta falha vai ser realocada no PE SP1.

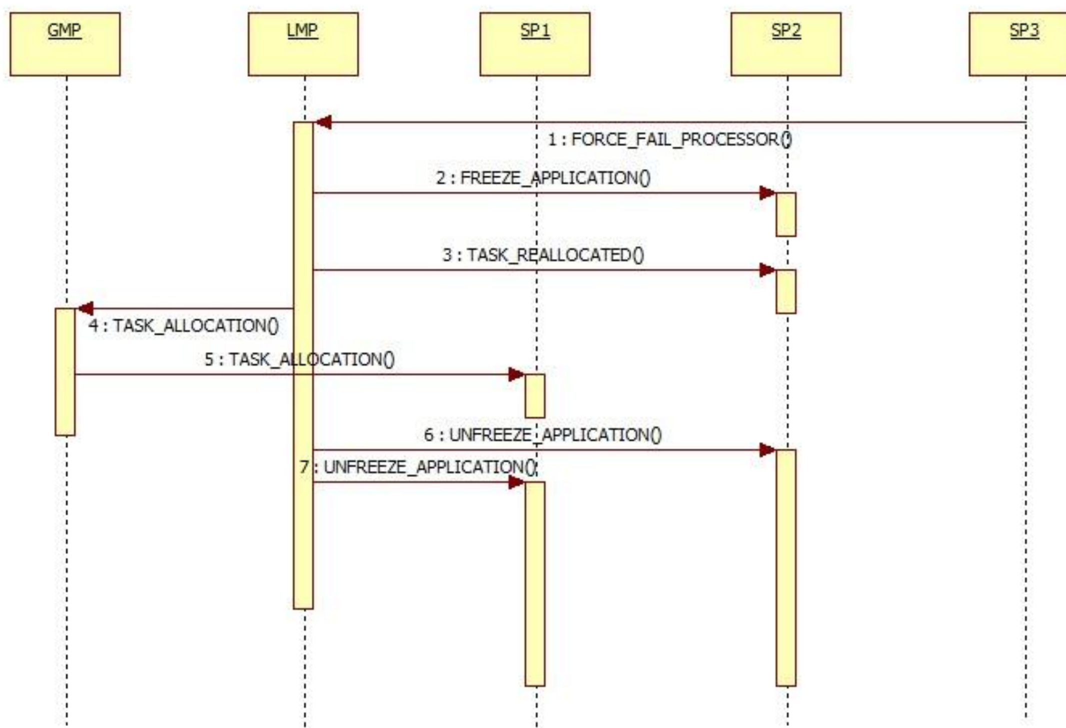


Figura 17 - Diagrama de comunicação do protocolo de recuperação de falhas da HeMPS

Considerando uma simulação em andamento, o primeiro serviço executado como mostra o diagrama é a mensagem de 1: *FORCE_FAIL_PROCESSOR* enviada pelo PE SP3 que representa o PE com falha.

Assim que o LMP recebe tal mensagem, deve identificar que uma falha permanente em seu cluster ocorreu e deve iniciar o protocolo de recuperação. A mensagem executada a seguir é 2: *FREEZE_APPLICATION*, que vai congelar as tarefas relacionadas a aplicação que falhou colocando-as em estado de *FREEZE* em seus respectivos escalonadores.

Após o LMP recalcular o novo mapeamento das tarefas que falharam, ele vai enviar a mensagem de 3: *TASK_REALLOCATED*. Esta mensagem é enviada para cada PE com tarefas da aplicação que falhou para informar a nova posição de alocação das tarefas que falharam.

O próximo passo é requisitar a alocação da tarefa falhada em um PE saudável que na Figura 17 é representado pelo elemento SP1. LMP vai enviar 4: *TASK_ALLOCATION* para GMP que possui acesso ao repositório de tarefas. Assim que GMP encontrar a tarefa

vai enviar a mensagem de 5: *TASK_ALLOCATION* para SP1. SP1 vai receber este serviço e vai identificar que se trata de uma realocação de tarefas devido a falha e vai manter esta tarefa em estado de *FREEZE*.

Finalizando o protocolo, LMP vai enviar as mensagens de 6: *UNFREEZE_APPLICATION* para SP1 e 7: *UNFREEZE_APPLICATION* para SP2 reiniciando a execução de suas respectivas tarefas.

4. RESULTADOS

Todos os experimentos foram realizados na plataforma HeMPS, configurada com 16 PEs comunicando por troca de mensagens em uma NoC de topologia mesh 4x4 com 4 clusters 2x2, exceto pelo experimento da seção 4.5 que foi realizado com tamanho variado de cluster. O *clock* global está configurado para 100MHz. Um arquivo de configuração *.hmp* foi ajustado para cada tipo de experimento. Neste arquivo são configurados os valores de formato da NoC, número de tarefas máximo nos PEs, o número de aplicações que serão carregadas e a ordem de carregamento das aplicações.

As aplicações utilizadas foram uma soma paralela de vetores, um decodificador de frames MPEG, um programa de comunicação produtor/consumidor e uma aplicação de *Dynamic time warping* (DTW). A aplicação de soma tem seu número de tarefas variável enquanto que as demais aplicações possui o número de tarefas fixa. Para os experimentos que são alterados o número de tarefas total da aplicação, foi utilizada a aplicação de soma de vetores devido a esta aplicação possuir o número de tarefas a serem executadas variável em tempo de compilação.

A soma de vetores consiste em uma quantidade variável de tarefas geradoras de números que vão preencher vetores. Estes vetores são enviados a uma única tarefa somadora que soma os valores entre esses vetores pela posição ordinal e apresenta os valores somados em um vetor resultado. A segunda aplicação é um decodificador MPEG que decodifica frames em etapas sequenciais (modelo *dataflow*). A terceira aplicação é uma representação de duas tarefas comunicantes no modelo produtor / consumidor que enviam números inteiros uma para a outra. A quarta aplicação é um programa de análise de padrões numéricos chamado DTW. Esse algoritmo calcula alinhamento de séries temporais, utilizado por exemplo em algoritmos de reconhecimento de voz. Os grafos de comunicação que representam as aplicações estão representados na Figura 18.

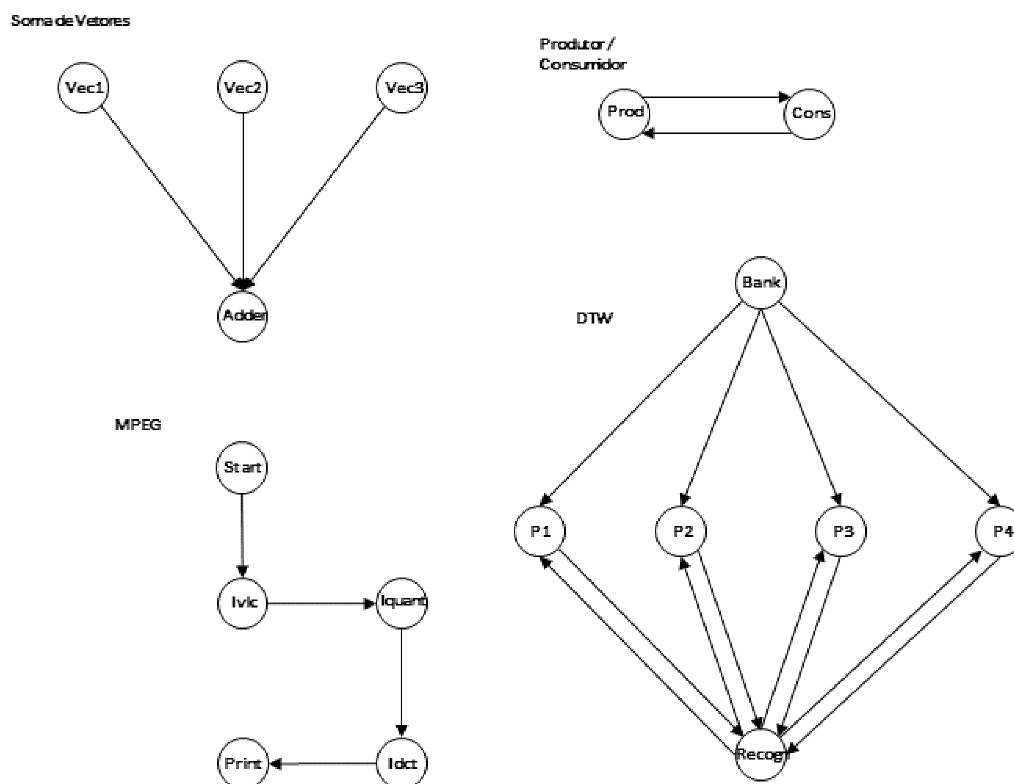


Figura 18 - Grafos de Comunicação das Aplicações

Os experimentos seguem as seguintes etapas:

1. Configura-se o arquivo de parâmetros do MPSoC e dá-se início a simulação.
2. Inicialmente o GMP vai requisitar ao repositório de tarefas a primeira aplicação. O GMP vai executar um mapeamento inicial para o carregamento desta aplicação em um cluster que estiver disponível.
3. Encontrado o cluster, o LMP deste cluster vai mapear em seus SPs cada tarefa enviada pelo GMP de acordo com a necessidade da aplicação.
4. Assim que a aplicação estiver com suas tarefas executando no cluster escolhido, em um determinado tempo de execução, o GMP vai injetar uma falha em um determinado PE deste cluster que esteja executando tarefas.
5. O protocolo de recuperação de falhas inicia no LMP deste cluster registrando o tempo de início a partir da detecção desta falha.
6. Ao término do protocolo de recuperação é registrado o tempo final do processo, marcando o momento de reinício das tarefas da aplicação que falhou.

4.1. Experimento 1 – Tempo de recuperação x número de tarefas realocadas

O primeiro experimento, demonstra o impacto no tempo de recuperação de uma falha permanente quando o número de tarefas realocadas no PE falho aumenta. Relembrando que o *microkernel* da HeMPs é multi-tarefa com o número de tarefas por PE configurável em tempo de projeto.

Todos os cenários executam com 8 tarefas no total. A diferença entre os cenários é o número de tarefas que estão alocadas no PE que vai falhar. Este experimento varia de 1 tarefa no primeiro cenário até 7 tarefas no último cenário. A Figura 19 ilustra a configuração do cenário e mostra o cluster que o PE SP3 apresenta a falha.

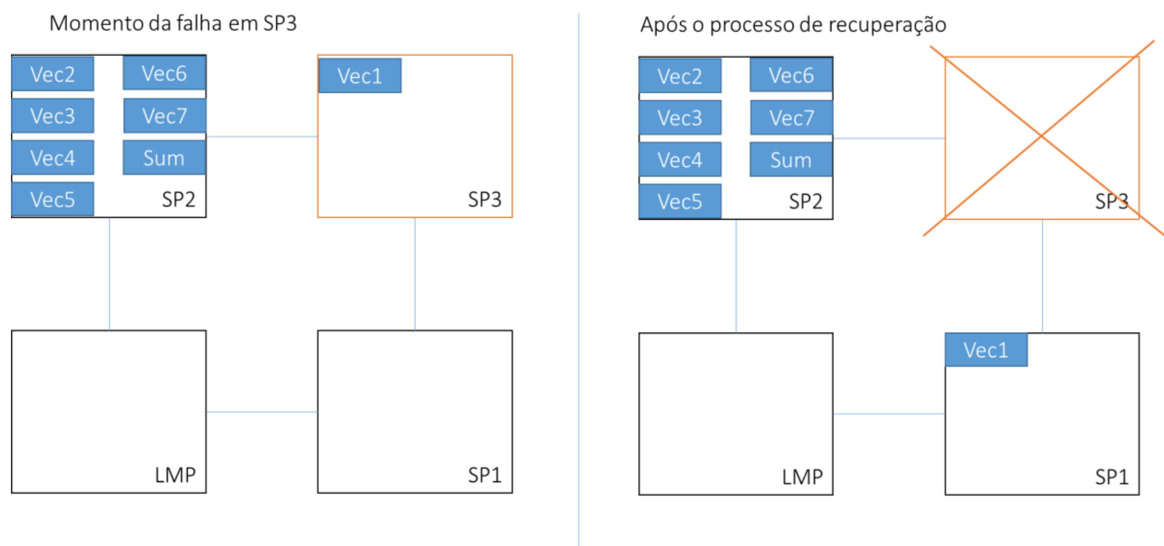


Figura 19 - Configuração do experimento 1 – uma tarefa realocada

todas as tarefas já estão carregadas e a aplicação em execução. A Figura 20 ilustra o resultado do experimento.

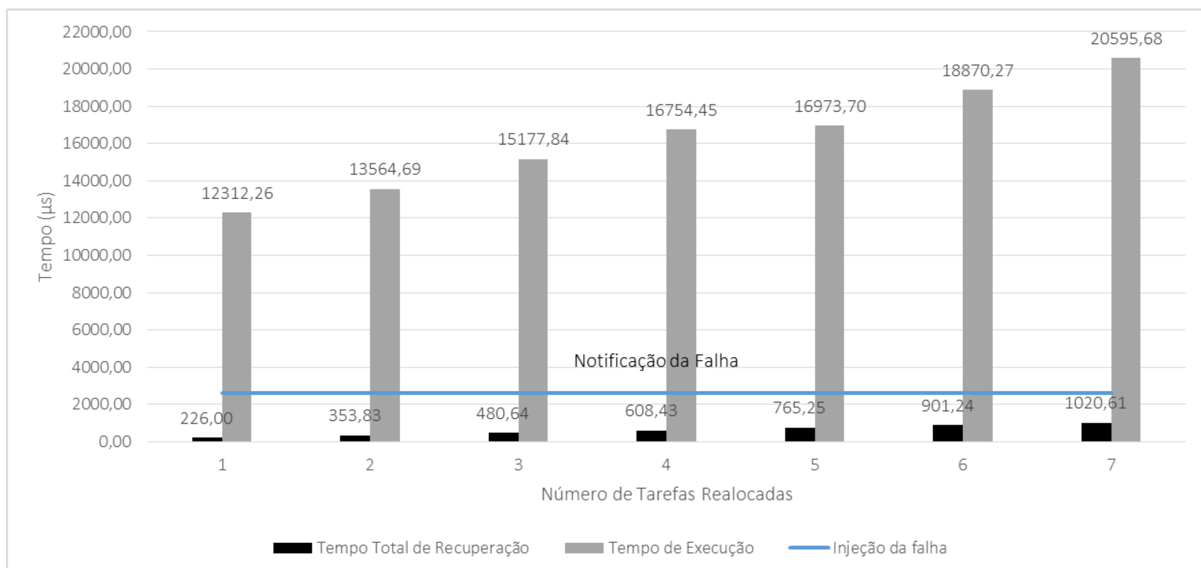


Figura 20 - Tempo de recuperação relativo ao número de tarefas realocadas

Neste experimento foi verificado que o tempo de recuperação mostrou-se como uma pequena porção do tempo total de execução da aplicação. Em seu pior caso, no cenário com 7 tarefas sendo realocadas, o tempo de recuperação foi de 1020,61 µs, representando 4,96% do tempo total de execução.

O próximo gráfico na Figura 21 demonstra o crescimento do tempo de recuperação mostrando a decomposição do protocolo em três etapas: *freeze, task reallocation e unfeeze*.

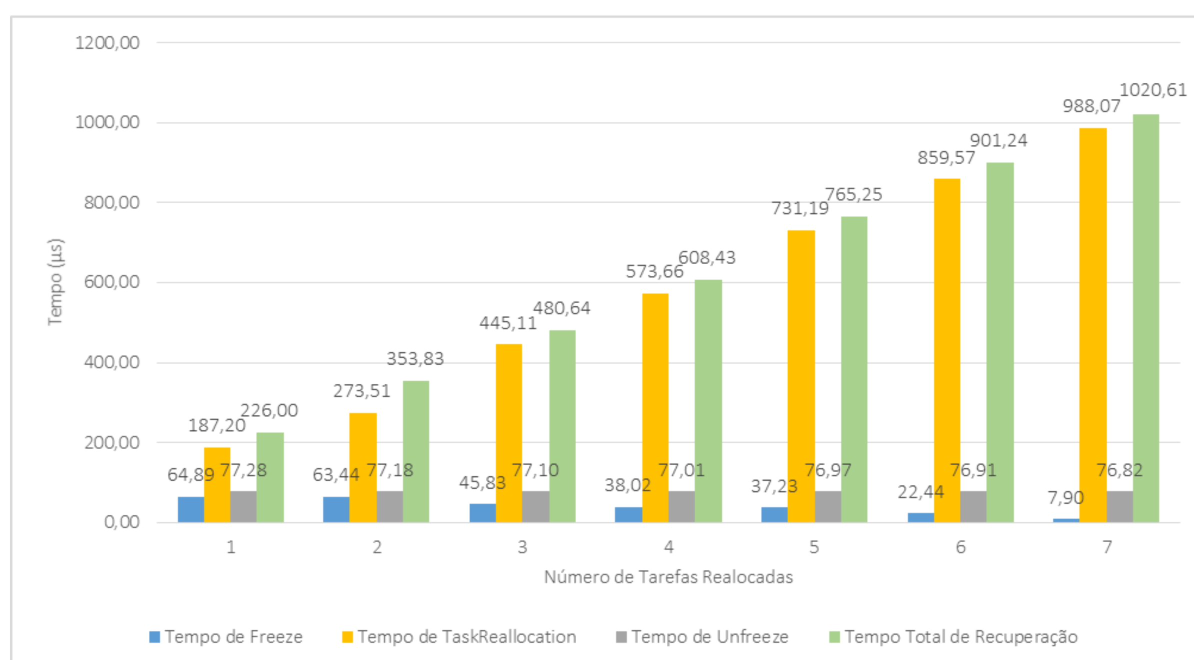


Figura 21 - Tempo de recuperação decomposto em etapas

Na decomposição das etapas deve-se ressaltar que estas têm sobreposição no tempo total do protocolo. Isto significa que partes das etapas de *freeze* e *unfreeze* acontecem ao mesmo tempo que em a etapa de realocação de tarefas já está executando. Um exemplo desta sobreposição é apresentado posteriormente na Seção 4.9. Devido a isso, a soma dos tempos das etapas não será igual ao tempo total do protocolo de recuperação. Observa-se que na realocação de 1 tarefa, as etapas têm pesos significativos, mas o tempo da etapa de realocação é o maior de todos. À medida que o número de tarefas realocadas cresce, a dominância do tempo de realocação cresce. Os tempos de freeze reduzem devido ao número de tarefas a receberem freeze que vai diminuindo. Por fim os tempos de unfreeze se mantêm, indicando que a etapa de unfreeze é dependente do número total de tarefas.

Este experimento mostra que a abordagem proposta apresenta a vantagem de crescer linearmente com o número de tarefas afetadas e demonstra números baixos (4,96% no pior caso medido) de tempo de recuperação proporcionais ao tempo total de execução.

4.2. Experimento 2 – Tempo de recuperação x tamanho das aplicações

O experimento 2 ilustrado na Figura 22 tem o objetivo de testar se o tamanho da aplicação influencia no tempo de recuperação, caso o número de tarefas realocadas seja constante. Em todos os cenários, 3 tarefas foram realocadas no PE alvo de injeção de falha permanente. O número total de tarefas da aplicação foi variado de 4 a 12 tarefas da aplicação de soma de vetores. A injeção de falha neste experimento ocorre em 2500 µs. A Figura 22 ilustra o experimento 2 onde o PE SP3 é o PE que apresenta falha.

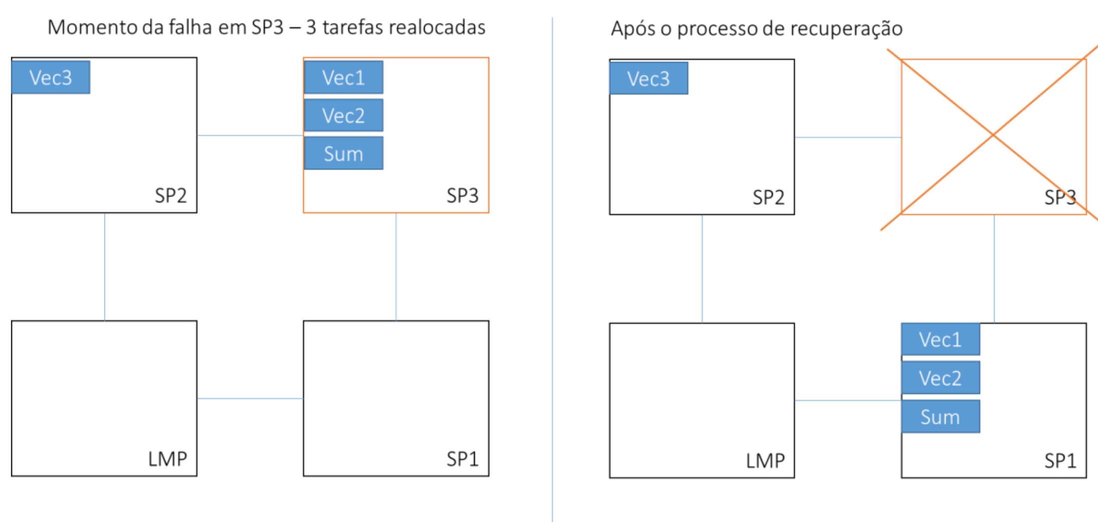


Figura 22 - Configuração do experimento 2 – número total de tarefas 4

Os tempos de recuperação mostraram-se similares nos 3 cenários. A Figura 23 mostra o tempo de recuperação relativo ao tempo total de execução na linha pontilhada. Este tempo relativo manteve-se abaixo de 4%. Apenas um aumento de tempo de execução foi observado na terceira amostra de uma aplicação com 12 tarefas que teve o tempo de execução degradado devido ao mapeamento inicial das tarefas.

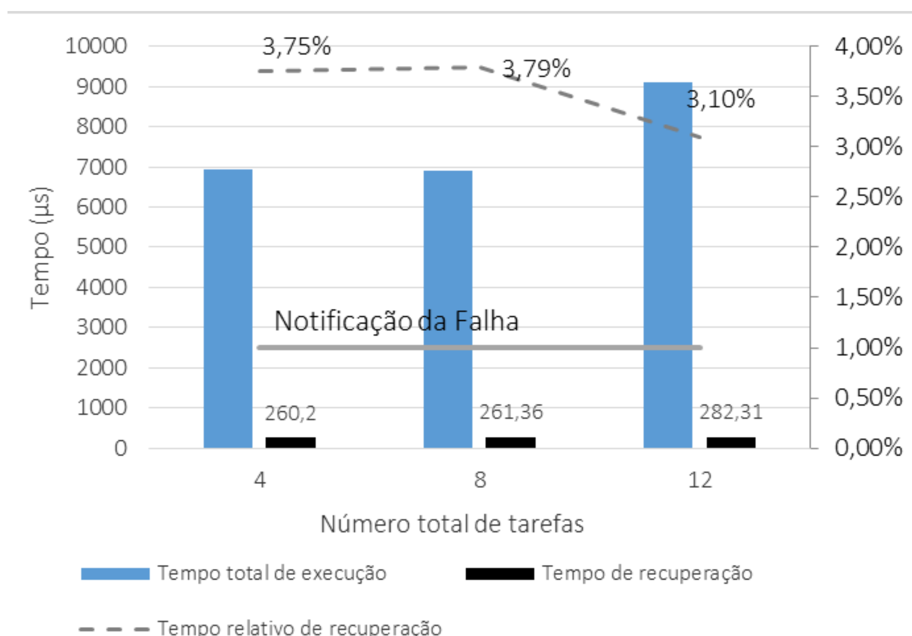


Figura 23 - Tempo de recuperação relativo ao tamanho da aplicação

Neste caso os PEs escolhidos para mapeamento ficaram mais espalhados entre os PEs causando maior latência na comunicação entre as tarefas. Conclui-se com este experimento que o tamanho da aplicação não tem impacto significativo no tempo de recuperação de uma falha permanente. Isso ocorre por que o número total de tarefas apenas influencia os tempos de *freeze* e *unfreeze*. Como demonstrado no experimento anterior, o maior tempo gasto com protocolo foi observado apenas na etapa de realocação de tarefas.

4.3. Experimento 3 – Largura de banda utilizada na NoC proporcional aos principais serviços executados

O terceiro experimento foi realizado com o objetivo de medir o *overhead* de tráfego na NoC dado que novas mensagens pertencentes ao protocolo de recuperação foram adicionadas no sistema. A Figura 24 ilustra a configuração do experimento.

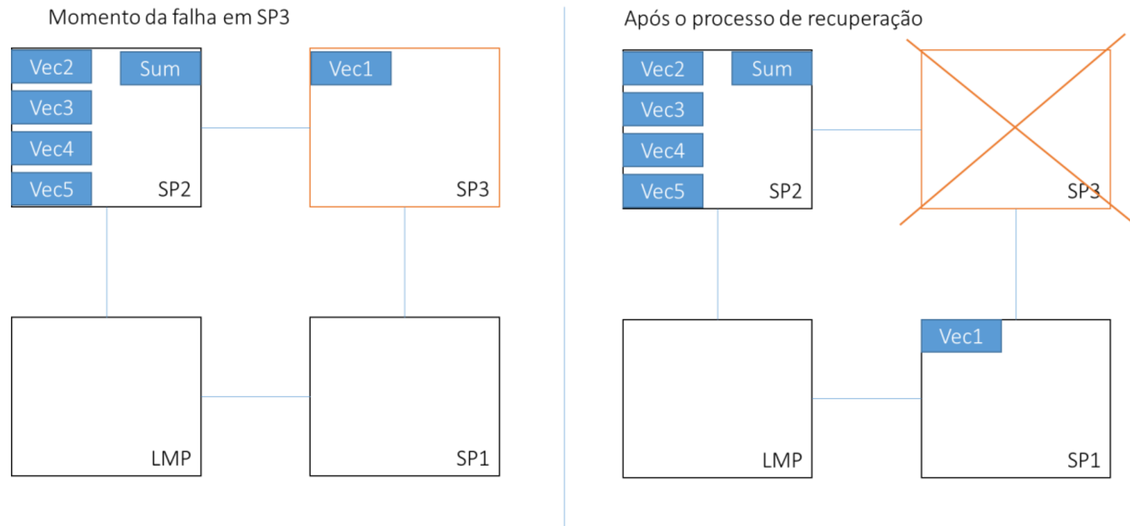
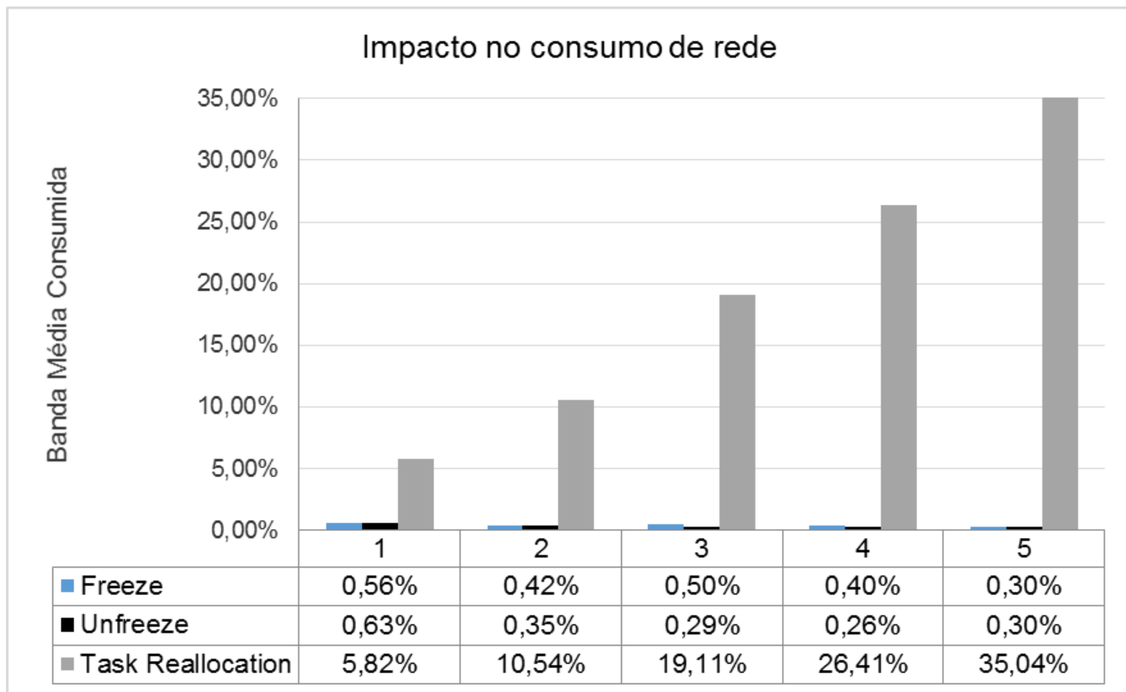


Figura 24 - Configuração do experimento 3 - 1 tarefa sendo realocada

Neste experimento, os resultados ilustrados na Figura 25, o consumo de banda relativo ao consumo total da NoC foi medido nos cenários em que o número de tarefas realocadas no sistema aumenta de 1 a 5 tarefas.



	1	2	3	4	5
Freeze	0,56%	0,42%	0,50%	0,40%	0,30%
Unfreeze	0,63%	0,35%	0,29%	0,26%	0,30%
Task Reallocation	5,82%	10,54%	19,11%	26,41%	35,04%

Figura 25 - Consumo relativo de banda com as mensagens do protocolo de recuperação de falhas

As mensagens foram separadas em três etapas do processo de recuperação de falhas para que possa-se observar o impacto de cada etapa no que diz respeito a consumo de banda na NoC. As etapas são: *Freeze*, *Unfreeze* e *Task Reallocation*. O experimento mostra a dominância dos serviços executados na etapa de realocação de tarefas. Esta etapa tem a maior representação de consumo no experimento porque é nesta fase que o código binário das tarefas a serem realocadas é transferido do

repositório de tarefas para os PEs saudáveis. Os resultados mostram que o pior caso de *overhead* de consumo de banda ficou em 35,04% com cinco tarefas realocadas no processo de recuperação. Este consumo representa o pico de tráfego gerado no processo de recuperação proposto. Importante mencionar que ao término deste processo de recuperação o consumo de banda na NoC retorna aos valores normais de operação e que a banda da NoC só é afetada durante a recuperação.

4.4. Experimento 4 – Tempo de recuperação detalhado em etapas x número de falhas em diferentes clusters

O objetivo deste experimento é avaliar o impacto nos tempos de cada etapa do protocolo de recuperação ao ter que tratar falhas subsequentes no sistema. Para isso essa simulação foi configurada com a aplicação de MPEG sendo carregada em todos os clusters do MPSoC aumentando a carga de mensagens do sistema. A falha é injetada em 2600 μs e os PEs configurados para falhar possuem 2 tarefas para serem realocadas em todas as medições como mostrado na Figura 26.

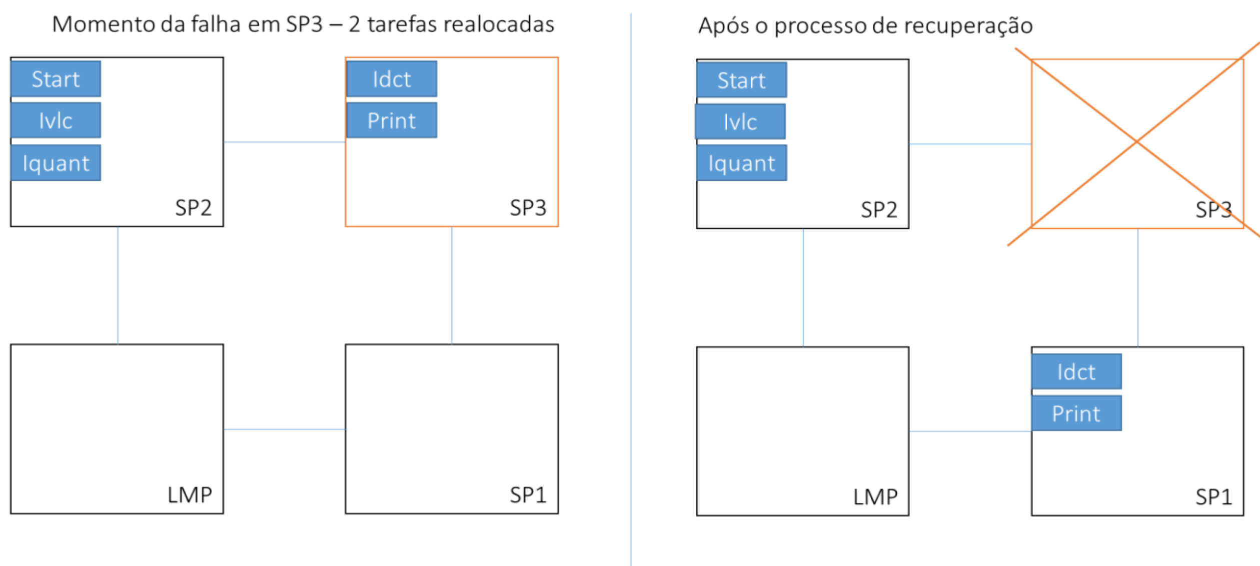


Figura 26 - Configuração do experimento 4 – primeiro teste com uma falha

A Figura 27 mostra a soma dos tempos das principais etapas do processo de recuperação: *freeze*, *task reallocation* e *unfreeze*. As falhas foram programadas para acontecer de 1 a 4 ocorrências, sempre em diferentes clusters. Apenas as etapas de *freeze* e *unfreeze* tiveram a possibilidade de acontecer ao mesmo tempo em mais de um *cluster* (sobreposição de processos de recuperação). Isso ocorre porque a etapa de *task reallocation* utiliza o mecanismo de DMA que é acionado pelo GMP e que só trata uma

requisição de realocação por vez. No pior caso, as falhas ocorrem uma por vez em cada cluster do sistema.

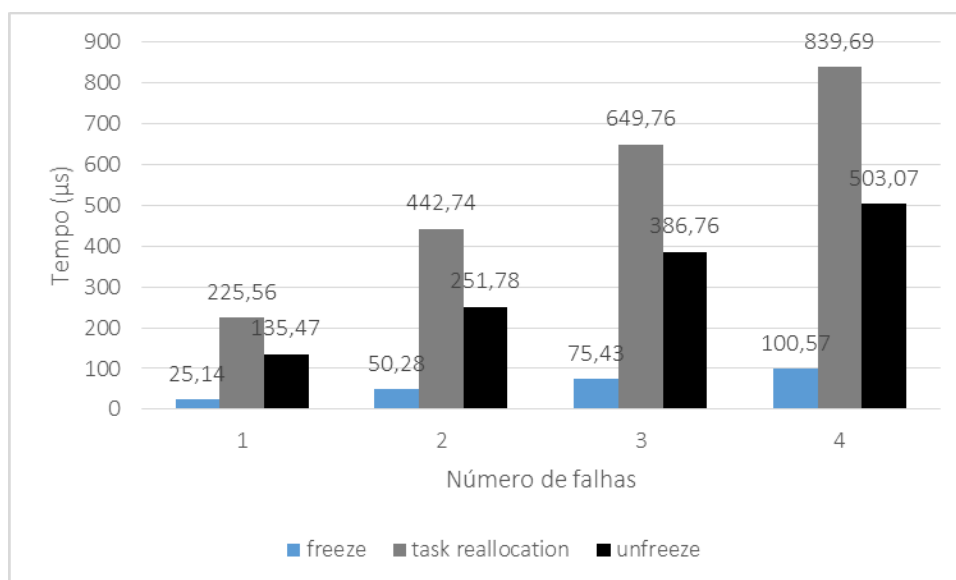


Figura 27 - Tempo gasto nas etapas da recuperação em relação com o número de falhas

Foi observado que o crescimento nos tempos de recuperação das 3 etapas é praticamente linear. A etapa de *freeze* mostrou um aumento de aproximadamente 25µs a cada medição. A etapa de *task reallocation* apresentou um aumento de aproximadamente 200µs. A etapa de *unfreeze* mostrou um aumento linear de aproximadamente de 125µs em média. O número de tarefas que receberam o serviço de *unfreeze* foi de 3 tarefas. O que significa que mesmo com a comunicação total da NoC sofrendo uma carga maior devido a todos os clusters estarem ocupados, os tempos de recuperação mostram–se estáveis e não são impactados por falhas em sequência já que estão sendo tratados por clusters diferentes. Apenas os tempos de *freeze* e *unfreeze* foram penalizados chegando próximo dos tempos de *task reallocation* dado que o número de tarefas realocadas é de 2 tarefas implicando em etapas de *task reallocation* mais rápidas. Entretanto a etapa de *task reallocation* ainda consome mais tempo do total do tempo de recuperação.

4.5. Experimento 5 – Tempo de recuperação em relação ao tamanho da NoC

O experimento 5 foi realizado para avaliar o protocolo em termos de escala o quanto o protocolo de recuperação impacta com o crescimento das dimensões da NoC. As simulações foram realizadas nas duas configurações de gerência da HeMPS: centralizada e distribuída.

Neste cenário a aplicação utilizada foi a de soma de vetores com a falha injetada em 2600 µs. Em todas as simulações apenas 2 tarefas são realocadas a partir do PE com falha permanente. A Figura 28 mostra a configuração da simulação.

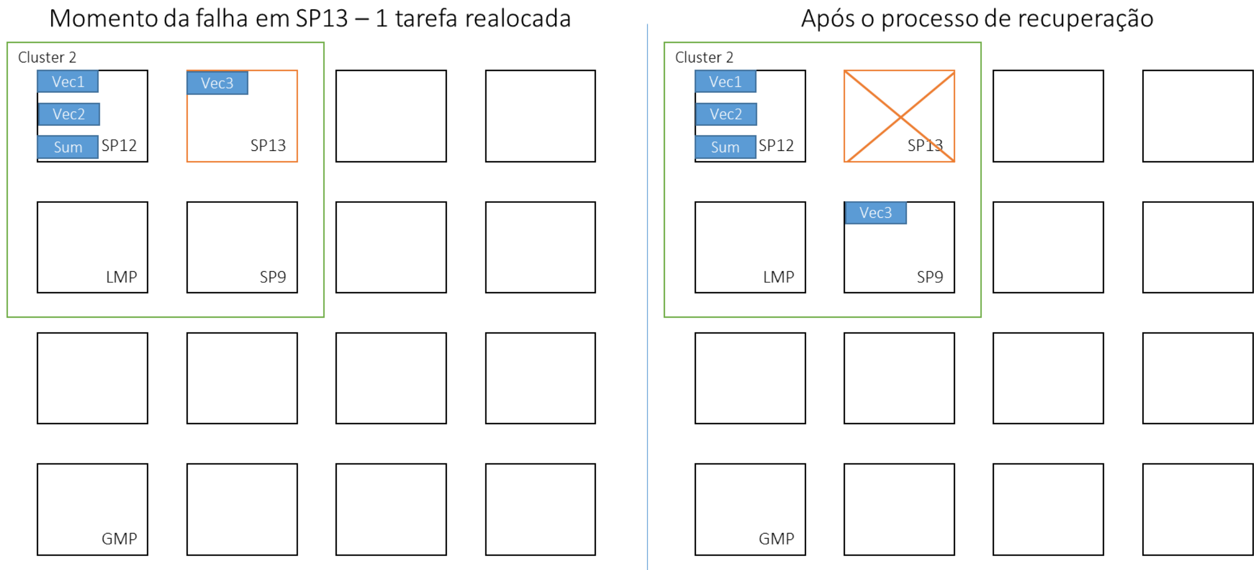


Figura 28 - Configuração do experimento 5 - dimensão 4x4 cluster de 2x2

A Figura 29 mostra a medição dos tempos de recuperação em relação ao tempo total de execução. A linha amarela no gráfico representa a relação entre Tempo de recuperação e o número total de PEs da NoC. O objetivo deste cálculo é avaliar o impacto do tempo de recuperação conforme a NoC aumenta de tamanho. Quanto menor é o valor do fator calculado maior é o impacto do protocolo com o crescimento da NoC. A Figura 29 mostra que o fator diminui com o crescimento da NoC. Entrento a redução do valor se mostra mais suave para os tamanhos maiores indicando que o protocolo é escalável.

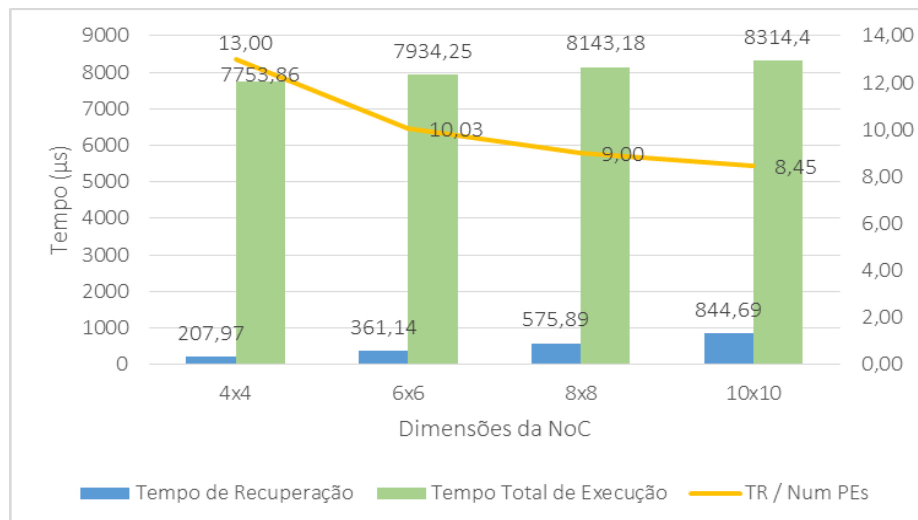


Figura 29 - Tempo de recuperação em relação ao tamanho da NoC (Gerência centralizada)

A HeMPS de gerência centralizada mostrou um aumento nos tempos de recuperação, no entanto comparados aos tempos totais de execução a porção é mínima e não afetou de forma significativa o crescimento desses totais. A relação calculada mostra que da HeMPS 4x4 para a HeMPS 6x6 o valor do fator tem uma queda brusca de 13 para 10,03. Indicando que houve impacto do protocolo. No entanto, para os tamanhos seguintes os decrescimo do fator da relação foi diminuindo fazendo a curva de descrescimo estabilizar.

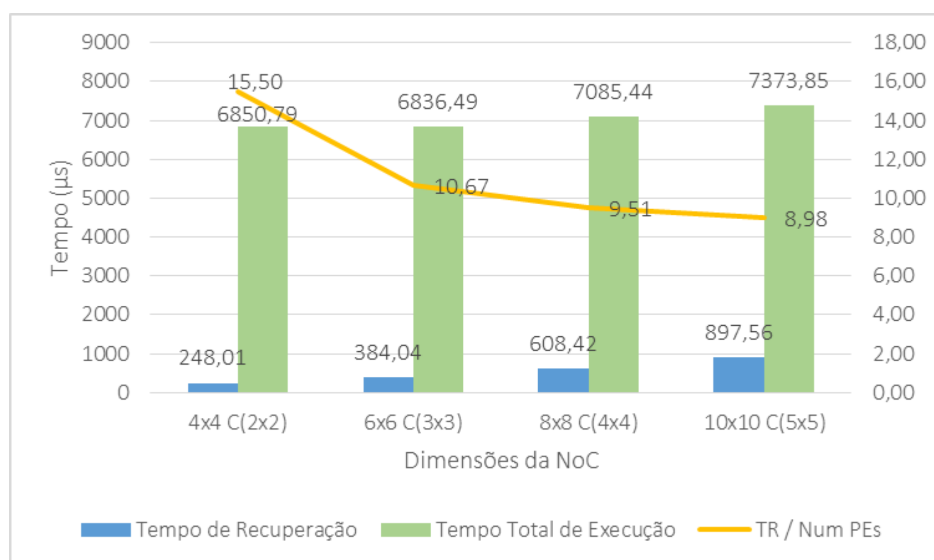


Figura 30 - Tempo de recuperação em relação ao tamanho da NoC (Gerência distribuída)

A Figura 30 apresenta a mesma simulação, porém executando em uma HeMPS de gerência distribuída. Observa-se que os tempos totais de recuperação comparados aos tempos da HeMPS centralizada são menores, dado que neste cenário existe mais gerenciamento feito em paralelo no sistema. A queda acentuada da curva da relação Tempo de recuperação com Número de PEs da NoC, também é observada no crescimento da HeMPS 4x4 para a HeMPS 6x6. Entretanto, o comportamento de escala da HeMPS distribuída mostra-se similar a HeMPS centralizada. A curva estabiliza a partir das dimensões 6x6.

Considerando que o tempo de execução da NoC sem falhas tem a tendência de crescimento linear e o acréscimo do tempo de recuperação possui um tempo menor comparado ao tempo total de execução. Conclui-se que mesmo com o aumento dos tempos de recuperação devido ao crescimento da NoC, o protocolo se mostra viável para ambientes de maior escala, dado que em dimensões maiores de NoC, os tempos de recuperação não crescem significativamente comparado ao tempo total de execução.

4.6. Experimento 6 – Tempo de recuperação em diferentes aplicações

O experimento 6 foi realizado com o objetivo de analisar de que forma diferentes aplicações têm impacto nos tempos de recuperação devido a seu tamanho e formatação do grafo de tarefas. Quatro aplicações foram avaliadas e estas têm diferenças em número total de tarefas e tamanho de código binário transferido na realocação de tarefas. As aplicações são: DTW, MPEG, soma de vetores e produtor/consumidor. O grafo destas aplicações foi apresentado na Figura 18, página 51. A Figura 31 ilustra a alocação de tarefas inicial do experimento onde SP3 é o PE que apresenta falhas e SP1 é o PE destino da tarefa que será realocada.

Figura 31 - Configuração do experimento 6 - alocação inicial das tarefas

Neste cenário, a falha acontece a 2600µs e em todos os casos apenas uma tarefa está sendo realocada com o objetivo de aproximar os tempos de cada etapa do protocolo. Como foi demonstrado nos experimentos anteriores, a diferença de tempo entre as etapas diminui quando o número de tarefas realocadas é menor. Considerando que cada aplicação possui tamanhos de código binário diferentes, é esperado que os tempos da etapa de *task reallocation* sejam diferenciados.

As figuras a seguir ilustram com detalhes dois fatores de influência para cada aplicação, mostrando os tempos de cada etapa junto ao tempo total de recuperação do protocolo.

A Figura 32 ilustra em azul a etapa de *freeze* e cinza a etapa de *unfreeze* que são as etapas afetadas pela quantidade de tarefas da aplicação representada pela linha amarela. O tempo total do protocolo de recuperação neste caso é ilustrado pela linha verde claro. A etapa de *task reallocation* está representada pela barra laranja enquanto a linha azul escuto representa o númeto total de tarefas da aplicação.

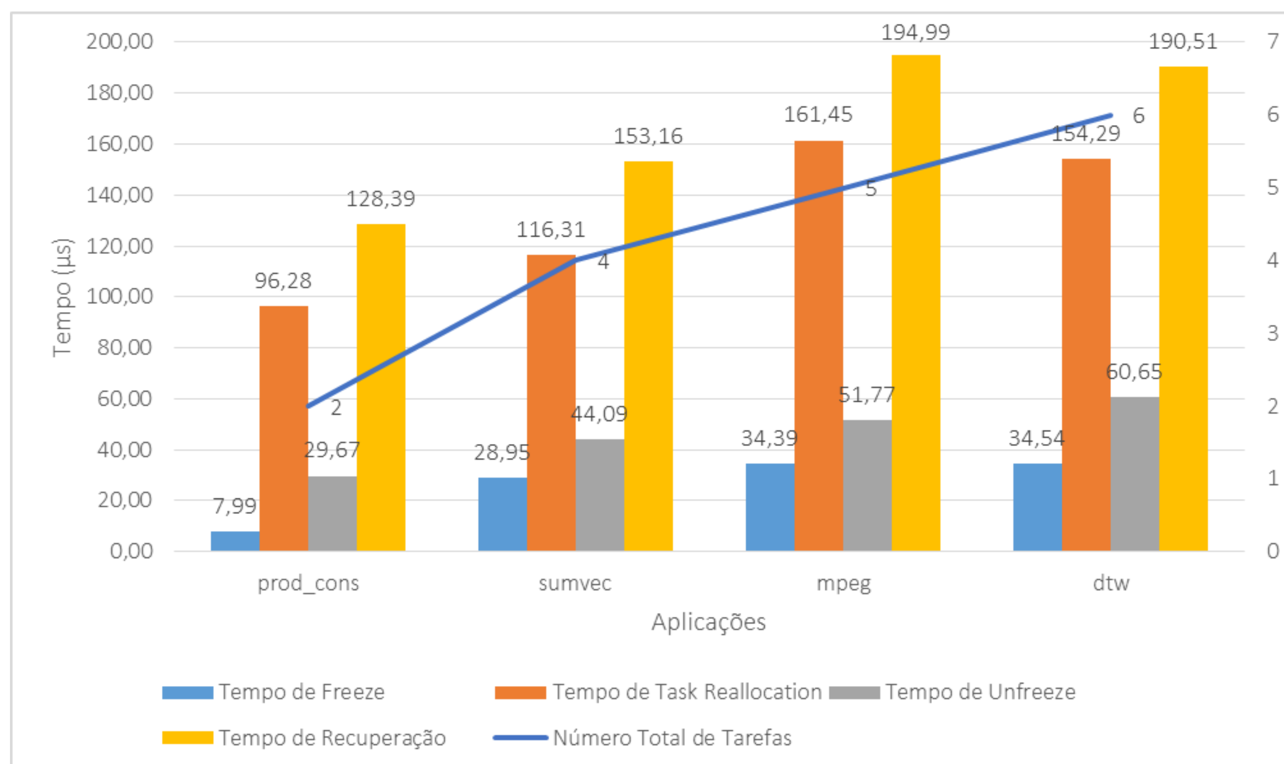


Figura 32 - Impacto do número de tarefas da aplicação

Esta visão do experimento confirma que a primeira variável que afeta o tempo da recuperação de falhas é o número de tarefas da aplicação, no entanto esta influência é baixa e apenas tem peso significativo quando poucas tarefas pequenas são realocadas, como é o caso da aplicação produtor-consumidor.

Observa-se que o crescimento da quantidade de tarefas da aplicação influi diretamente nas etapas de *freeze* e *unfreeze*, pois o gráfico demonstra que o padrão de crescimento do tempo de execução destas etapas acompanha a curva de crescimento do número de tarefas.

Outra visão do experimento está representada na Figura 33. O número de bytes transferidos na etapa de realocação interfere na etapa que tem mais impacto no processo de recuperação de falhas. A figura mostra o tempo de total de recuperação da falha em

amarelo. A etapa de *task reallocation* é representada pela barra laranja, enquanto a linha roxa demostra o tamanho em bytes da tarefa que foi realocada em cada aplicação. As demais são a representação das etapas de freeze e unfreeze.

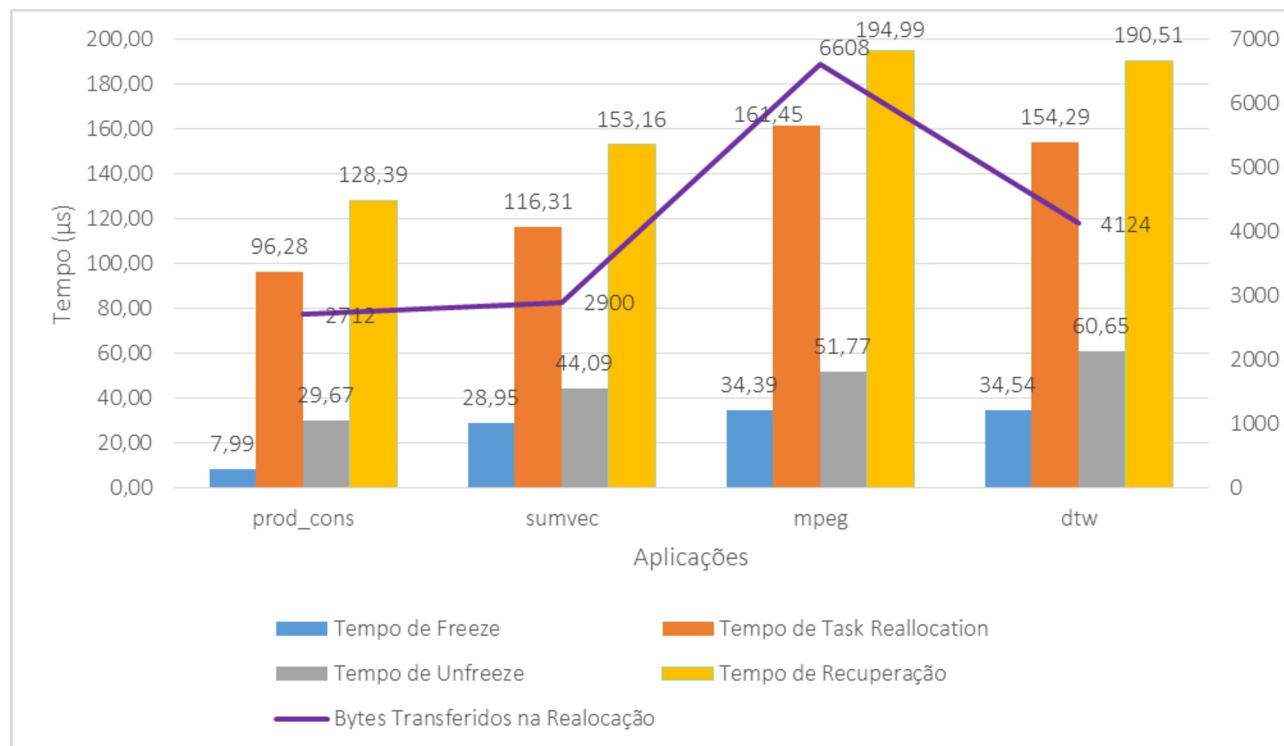


Figura 33 - Impacto do número de bytes transferidos na realocação

O experimento mostra que o tempo total do protocolo e o tempo da etapa de realocação de tarefas são semelhantes. O padrão de crescimento do tempo de realocação segue o padrão da curva de bytes transferidos, indicando qual etapa é mais afetadas por esta variável. Observa-se também que a curva de tempo de realocação está próxima do tempo total do protocolo, o que confirma os resultados anteriores que indicam que esta etapa é a mais dominante.

4.7. Comparação com outras abordagens

Diferente do presente trabalho que suporta uma configuração de troca de mensagens e não salva o contexto de execução, a solução proposta em [MUS13] tem como princípio o uso de memória compartilhada e executa o salvamento de checkpoints. Em [MUS13] são apresentados resultados que mostram uma adição de overhead de tempo de execução em 18% no seu pior caso com 4 threads em uma execução sem falhas. Este overhead se deve ao salvamento de checkpoints e sua implementação de detecção de erros de execução. O presente trabalho não possui este tipo de overhead de

execução, dado que seu foco é na recuperação de falhas e não está associado diretamente a nenhum método de detecção de erros.

A solução de tolerância a falhas mostrada em [DER13] foi projetada com hardware e software para implementar a detecção de erros e recuperação de falhas. Este método apresenta uma adição de 60% de área em chip enquanto a abordagem proposta não tem custo em hardware. No que diz respeito à migração de tarefas, apenas o contexto de execução da tarefa é migrado, dado que os códigos binários das aplicações já estão previamente carregados em todos os PEs. Apenas o contexto é migrado e possui suporte a hardware para a migração de tarefas. [DER13] mostra em seus resultados tempo de recuperação em torno de 28934 ciclos de *clock* e com isso representa 0,88% do tempo total da aplicação, contudo possui *overhead* de execução devido a detecção de falhas e checkpoints. Enquanto o presente trabalho com apenas 1 tarefa sendo migrada, consome 22600 ciclos de *clock*. Isso representa 1,84% do tempo total de execução com uma implementação puramente de software com migração completa da tarefa. O presente trabalho também não tem *overhead* na execução normal. O custo está apenas no processo de recuperação.

Outras abordagens com migração de tarefas avaliadas, não tem como objetivo a tolerância a falhas e sim o balanceamento de carga de execução nos PEs. Entretanto uma comparação do tempo de migração foi feita com base em [ALM10] que mostra um tempo de migração de 1ms em um ambiente que executa a 600MHz para uma única tarefa. O presente trabalho apresenta como tempo de migração em torno de 0,2ms a 1ms em 100MHz, dependendo no número de tarefas migradas que foi variado de 1 a 7 tarefas. Diferente do método de [ALM10], nesta abordagem não há migração de contexto de execução dado que uma falha permanente ocorreu e as tarefas serão reiniciadas do seu estado inicial de execução.

4.8. Avaliação geral do protocolo proposto

Esta seção apresenta informações a respeito do impacto das mudanças que a implementação do protocolo de recuperação causa no ambiente de desenvolvimento da HeMPS, comparada com a versão original do kernel da HeMPS.

O protocolo proposto não altera o tempo de execução das aplicações sem falhas injetadas. Caso não aja falhas permanentes nos PEs o algoritmo de recuperação permanece dormente até que uma técnica de detecção de falhas o ative.

Comparando os **tamanhos de código objeto do kernel** com e sem protocolo foram observados os seguintes números:

- Kernel Master
    - Sem o protocolo de recuperação: 17228 bytes;
    - Com o protocolo de recuperação: 21720 bytes;
    - Aumento de 20,68%.
- Kernel Local
    - Sem o protocolo de recuperação: 11224 bytes;
    - Com o protocolo de recuperação: 11352 bytes;
    - Aumento de 0,89%.
- Kernel Slave
    - Sem o protocolo de recuperação: 9096 bytes;
    - Com o protocolo de recuperação: 11448 bytes;
    - Aumento de 20,54%.

4.9. Análise geral dos resultados de desempenho do protocolo

Esta seção apresenta uma visão sistêmica do tempo gasto pelo protocolo proposto e também relaciona os resultados obtidos nas seções 4.1 a 4.6 com o pseudocódigo do protocolo proposto apresentado na página 44. O tempo total de recuperação de falhas pode ser dividido em três etapas principais após o recebimento de notificação de PE com falha: tempo para determinar PEs que devem receber mensagens de freeze, chamado de tempo de freeze; tempo de realocação das tarefas afetadas pela falha, chamado de tempo de realocação; mais o tempo de remoção da aplicação do estado de freeze, chamado de tempo de unfreeze.

O tempo de freeze, como mostrado no pseudocódigo da Figura 14, página 44, linhas 6 a 8, realiza uma busca em memória local no kernel master e envia mensagens de pequeno porte (cerca de 10 flits) proporcional ao número total de tarefas das aplicações afetadas pelo PE com falhas. Lembrando que o PE é multitarefa com o número de tarefas configurável em tempo de projeto. Mesmo se considerarmos uma situação pessimista com o suporte a várias tarefas em um PE e cada tarefa pertencente a uma aplicação diferente, o tempo de execução desta etapa ainda é rápido em relação as demais etapas e as mensagens enviadas são pequenas. Do lado do kernel slave que recebe esta mensagem de freeze, como mostrado no pseudocódigo da Figura 15, página 45, linhas 4

a 7, o kernel só precisa fazer uma busca na lista de tarefas existentes no PE por aquelas que pertencem a aplicação indicada na mensagem e atribuir o estado freeze para esta tarefa. Ou seja, o processamento do lado do kernel slave é ainda menor que do lado do kernel master.

O tempo de realocação consiste em três atividades principais. A primeira é a execução do algoritmo de mapeamento, que gera a nova posição da tarefa, que é uma atividade CPU intensiva sem comunicação pela rede. Este tempo pode crescer devido ao tamanho do sistema. A segunda é o envio do código binário para a nova posição que é IO intensiva, proporcional ao tamanho das tarefas. A terceira consiste no envio da nova posição das tarefas realocadas para as tarefas que se comunicam com ela (pseudocódigo da Figura 14, página 44, linhas 14 a 16). Esta atividade exige pouca computação e pouco IO devido ao tamanho reduzido das mensagens. O kernel slave se envolve no recebimento da nova tarefa e no recebimento das novas posições das tarefas, ou seja, do lado do slave a realocação é predominantemente IO intensiva e realizada distribuidamente.

O tempo de unfreeze, como mostrado no pseudocódigo da Figura 14, página 44, linhas 18 a 20, requer do kernel master baixa computação e IO, proporcional ao número de tarefas de todas aplicações afetadas pela falha. Por outro lado, quando o kernel slave recebe ele deve realizar buscas nas listas de tarefas do PE, limpar TCB para cada tarefa da aplicação, e remover do pipe mensagens destas mesmas tarefas. Ou seja, esta parte do protocolo é CPU intensiva, sem IO exceto a mensagem que disparou o processo.

A Figura 34 ilustra os principais momentos onde o tempo é gasto no protocolo. Esta figura está dividida em 4 etapas representadas por cores diferentes. A primeira etapa é referente ao freeze, a 2ª é referente ao remapeamento (linha master CPU) e realocação (linha master DMA), a 3ª é referente ao envio da nova posição das tarefas, a 4ª é referente ao unfreeze. Esta figura representa a passagem do tempo no eixo horizontal e cada linha representa os principais atores do protocolo, i.e. mestres (GMP e LMP) e escravos. As flechas representam as dependências dos eventos.

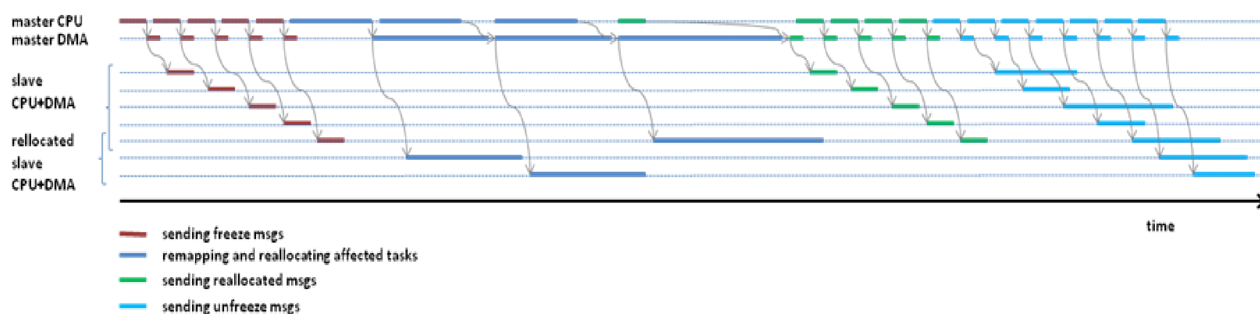


Figura 34 – detalhamento do tempo gasto na realização do protocolo de recuperação de falhas.

Nota-se, em azul escuro na Figura 34, que o maior gargalo é a etapa de envio do código binário das tarefas do repositório para os PEs. Além disso este processo é serializado pelo fato de atualmente haver somente um repositório de tarefas. O tempo total desta etapa é basicamente o somatório do tempo de transmissão de todas as tarefas afetadas mais o tempo da execução do mapeamento da primeira tarefa. As tarefas seguintes podem ser remapeadas em paralelo com a transmissão da tarefa anterior. Os escravos gastam tempo basicamente só para gravar a tarefa recebida em memória, utilizando o DMA.

Depois que os novos endereços das tarefas afetadas são conhecidos, estes novos endereços são disseminados entre o restante das tarefas das aplicações afetadas. Esta etapa é ilustrada na parte verde da Figura 34.

A etapa de *unfreeze* percebe-se que possui execução rápida do lado dos mestres, porém nos escravos este processamento pode ser maior e variável dependendo do número de tarefas em um mesmo PE que devem ser reinicializadas.

A análise da Figura 34 confirma os dados apresentados nos experimentos 4.1, 4.3 e 4.4 onde a dominância da etapa de realocação de tarefas fica evidente. Devido a transferência do código binário das tarefas do repositório para os PEs ser sequencial, o gargalo observado nesta etapa faz com que o protocolo apenas suporte processar a recuperação de uma falha por vez. O que é aceitável em se tratando de falhas permanentes.

4.10. Limitações

Algumas restrições do protocolo de recuperação de falhas na HeMPS são descritas nesta seção.

Falhas nos mestres GMP ou LMP ficaram fora do escopo de trabalho devido à complexidade e restrições de tempo para estudo em implementação. Alguns desafios desta implementação foram identificados como: recuperação de LMP precisa de

salvamento do contexto de mapeamento das tarefas em execução no sistema e outros PEs do cluster precisam ter a capacidade de serem promovidos a novos LMP dinamicamente. A recuperação do GMP necessita de redundância no acesso ao repositório de tarefas pois atualmente este é o único módulo que acessa esta memória. Suporte a múltiplos repositórios de tarefas para remover pontos únicos de falhas também não está no escopo deste trabalho, dado que se trata de uma implementação de hardware e a solução foi definida puramente em software.

Suporte à migração de tarefas com contexto de execução também ficou fora do escopo por complexidade de implementação. Esta escolha de projeto é discutida em detalhes na seção **Estudo de alternativas de projeto,** página 40.

Neste trabalho as falhas só foram injetadas depois que todas as tarefas das aplicações já estavam em execução, simulando uma situação mais provável de ocorrência de falhas. Por outro lado, pode ser que uma falha seja detectada durante a carga de uma tarefa ou, dependendo do método de teste, quando um PE slave ainda não tem nenhuma tarefa alocada. Estes tipos de situações não foram testados por falta de tempo, mas elas são relevantes para um trabalho futuro.

As tarefas da aplicação não podem ter inicialização estática de variáveis globais. Esta restrição ocorre, pois, como as tarefas que são realocadas no protocolo de recuperação são reiniciadas, a área de memória de inicialização estática não é alterada neste processo de reinicialização. Caso as variáveis globais tenham sido alteradas na instância anterior de execução elas vão manter seus valores na nova instância pós realocação. Este comportamento pode causar inconsistência na nova execução da aplicação. Devido a isso, variáveis globais devem ser inicializadas em uma rotina própria que é chamada toda vez que o código é reexecutado.

O sistema não suporta falhas simultâneas. Isso ocorre devido a uma limitação do próprio ambiente da HeMPS que possui um único repositório de tarefas. A etapa de realocação de tarefas executa o serviço de TASK_ALLOCATION que acessa o repositório de tarefas para obter o código binário de uma tarefa e transferi-lo via mecanismo de DMA. Estas operações ocorrem uma por vez na HeMPS, fazendo com que o GMP enfileire requisições simultâneas de TASK_ALLOCATION. Em certos casos como o processo de recuperação de falhas, as filas de requisições para alocação exaurem seus slots causando descarte de requisições interferindo no processo de recuperação.

A integração com métodos de detecção de falhas no PE também ficou fora do escopo deste trabalho. É necessário um estudo mais aprofundado considerando a

arquitetura da HeMPS para que a integração com o protocolo não sobrecarregue o desempenho do sistema.

Futuramente também pode ser considerada uma integração com o método de teste e recuperação da NoC, como proposto em [WAC13] e [FOC15].

5. CONCLUSÕES E TRABALHOS FUTUROS

O presente trabalho tem como objetivo recuperar tarefas de um MPSoC atingido por falhas permanentes nos elementos de processamentos. Para isto foi proposto um protocolo de recuperação das tarefas afetadas pela falha. Este método, desenvolvido exclusivamente em software, é baseado em migração de tarefas para garantir que as tarefas afetadas de um PE que apresenta falha permanente sejam realocadas em PEs saudáveis. A *originalidade* do presente trabalho está na integração de conceitos utilizados em programação paralela e distribuída (como migração de tarefas para recuperação de falhas) no contexto de MPSoCs baseados em comunicação por troca de mensagens e gerência distribuída.

Os resultados mostram que a abordagem proposta apresenta vantagens importantes considerando um protocolo leve baseado em software, implementado na camada do sistema operacional (kernel). Os resultados demonstram que, na ausência de falhas, o kernel proposto não apresenta nenhum acréscimo de tempo de execução comparado com o kernel original. O código objeto do kernel proposto é de aproximadamente 21 Kbytes, tendo um acréscimo de área de memória de 20% comparado com o kernel original.

Na presença de falhas, foi demonstrado que o tempo de recuperação tem a tendência de crescer linearmente com o número de tarefas afetadas. O consumo de banda na rede apresenta um pico de tráfego gerado no processo de recuperação, entretanto, a rede normaliza após o término do protocolo. O número de tarefas de uma aplicação não tem impacto significante no tempo de recuperação de uma falha permanente. Esta característica é interessante em MPSoCs pois o custo de recuperação torna-se independente do número de tarefas que compõe a aplicação. Falhas subsequentes em clusters diferentes do MPSoC mostraram que o impacto é a soma de cada processo de recuperação executado sem nenhum *overhead* adicional no tempo de recuperação. Observa-se que a etapa de realocação de tarefas é a etapa com maior consumo de banda na rede e maior processamento em relação ao tempo de recuperação total do protocolo. O protocolo proposto mostra-se viável em implementações para redes de maior dimensão devido ao crescimento do tempo de recuperação linear. Assim, o crescimento do tempo de recuperação pelo número de PEs possui um crescimento sublinear, possivelmente logarítmico.

5.1. Contribuições

As contribuições desta dissertação foram compiladas em artigo e submetidas para a conferência ISCAS - *IEEE International Symposium on Circuits and Systems* (Qualis A1). O artigo foi aprovado para apresentação oral.

[BAR15] Barreto, F. F. S; Moraes, F. G.; Amory, A. M., Fault Recovery Protocol for Distributed Memory MPSoCs. Em: ISCAS 2015, Lisboa, Portugal. IEEE International Symposium on Circuits and Systems, 2015. (QUALIS A1).

5.2. Trabalhos futuros

Como trabalhos futuros as seguintes atividades podem ser propostas:

- Implementação de um protocolo integrado que contemple os mestres GMP e LMP.
- Implementar o suporte à migração de tarefas com o contexto de execução.
- Resolver a inicialização estática de variáveis para que não haja restrições na implementação de aplicações de usuário.
- Integração com métodos de detecção a falhas nos PEs.
- Suportar múltiplas interfaces de memória para permitir a carga de múltiplas tarefas em paralelo, reduzindo o tempo de inicialização do sistema e o tempo de recuperação de falhas.
- Realizar mais testes do protocolo proposto, principalmente em situações menos usuais, como durante a carga de uma nova tarefa.

REFERÊNCIAS

[ALF08] Al Faruque, Mohammad Abdullah; Krist, Rudolf; Henkel, Jorg, Henkel. "ADAM: Run time Agent-based Distributed Application Mapping for on-chip Communication". Em: DAC, 2008, pp. 760-765.

[ALM10] Almeida, G. M.; Varyani, S.; Busseuil, R.; Sassatelli, G.; Benoit, P.; Torres, L. "Evaluating the Impact of Task Migration in Multi-Processor Systems-on-Chip". Em: SBCCI, 2010, pp. 73-78.

[ANA12] Anagnostopoulos, I; Bartzas, A.; Kathareios, G.; Soudris, D. "A Divide and Conquer Based Distributed Run-time Mapping Methodology for Many-core Platforms". Em: DATE, 2012, pp. 111-116.

[AVI04] Avizienis, A.; Laprie, J.C.; Randell, B.; Landwehr, C. "Basic Concepts and Taxonomy of Dependable and Secure Computing". Em: IEEE Transactions on Dependable and Secure Computing, vol. 1(1), 2004, pp. 11-33.

[CAR09] Carara, E. A.; Oliveira, R. P.; Calazans, N. L. V.; Moraes, F. G. "HeMPS - A Framework for Noc-Based MPSoC Generation". Em: ISCAS, 2009, pp. 1345-1348.

[CAS13A] Castilhos, G. M.; Mandelli, M.; Madalozzo, G.; Moraes, F. G. "Distributed Resource Management in NoC-based MPSoCs with Dynamic Cluster Sizes" Em: 2013 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Natal, 2013, pp 153-158.

[CAS13B] Castilhos, G. M. "Gerência Distribuída de Recursos em MPSoCs – Mapeamento e Migração de Tarefas". Em: Dissertação de Mestrado, Programa de Pós-Graduação em Ciências da Computação, Porto Alegre, PUCRS, 2013.

[CUI12] Cui, Y; Zhang, W; Yu, H. "Decentralized Agent Based Re-Clustering for Task Mapping of Tera-Scale Network-on-Chip System". Em: ISCAS, 2012, pp. 2437-2440.

[DER13] Derin, O.; Cannella, E.; Tuveri, G.; Meloni, P.; Stefanov, T.; Fiorin, L.; Raffo, L.; Sami, M. "A System-level Approach to Adaptivity and Fault-tolerance in NoC-based MPSoCs: The MADNESS project". Em: Microprocessors and Microsystems Journal, vol. 37(6), 2013, pp. 515-529.

[ELN02] Elnozahy, E. N.; Alvisi, L.; Wang, Y.-M.; Johnson, D. B. "A Survey of Rollback-recovery Protocols in Message-passing Systems". Em: ACM Computing Surveys Journal, vol. 34(3), 2002, pp. 375-408.

[FOC15] Fochi, V; Amory, A.M.; Moraes, F.G. "An Integrated Method for Implementing Online Fault Detection in NoC-Based MPSoCs". Em: ISCAS, 2015, 4p.

[GIZ11] Gizopoulos, D.; Psarakis, M.; Adve, S. V.; Ramachandran, P.; Hari, S. K. S.; Sorin, D.; Meixner, A.; Biswas, A.; Vera, X. "Architectures for Online Error Detection and Recovery in Multicore Processors". Em: DATE, 2011, pp. 533-538.

[GOO11] Goodarzi, B.; Sarbazi-Azad, H. "Task Migration in Mesh NoCs over Virtual Point-to-Point Connections". Em: Euromicro, 2011, pp. 463-469.

[JER05] Jerraya, A. A.; Wolf, W. "Multiprocessor Systems-on-Chips". Em: Morgan Kaufmann Publishers Inc, 2005, 602p.

[KER14] Kerkhoff, H.G.; et al. “Linking Aging Measurements of Health-monitors and Specifications for Multi-processor SoCs”. Em: DTIS, 2014. pp. 1-6.

[KOB11] Kobbe, Sebastian; Bauer, Lars; Lohmann, Daniel; Schroder Preikschat, Wolfgang; Henkel, Jorg. "DistRM: Distributed Resource Management for On-Chip Many Core Systems". Em: CODES+ISSS, 2011, pp. 119-128.

[MOR04] Moraes, F.; Calazans, N.; Mello, A.; Möller, L.; Ost, L. "HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip". Em: Integration, the VLSI Journal, vol. 38(1), 2004, pp. 69-93.

[MUS11] Mushtaq, H.; Al-Ars, Z.; Bertels, K. "Survey of Fault Tolerance Techniques for Shared Memory Multicore/Multiprocessor Systems". Em: IEEE 6th International Design and Test Workshop (IDT), 2011, pp. 12-17.

[MUS13] Mushtaq, H.; Al-Ars, Z.; Bertels, K. "Efficient Software-based Fault Tolerance Approach on Multicore Platforms". Em DATE, 2013, pp. 921-926.

[PIR04] Pirretti, M.; Link, G.M.; Brooks, R.R.; Vijaykrishnan, N.; Kandemir, M.; Irwin, M.J. "Fault Tolerant Algorithms for Network-on-chip Interconnect". Em: IEEE Computer Society Annual Symposium on VLSI, 2004, pp. 46-51.

[PLA14] Processador PLASMA. Disponível em: http://opencores.org/project,plasma, 2014.

[PSA10] Psarakis, M.; et al. “Microprocessors Software-based Self-testing,” IEEE Design and Test of Computers, vol. 27(3), 2010, pp. 4-19.

[RAD13] Radetzki, M.; Feng, C.; Zhao, X.; Jantsch, A. "Methods for Fault Tolerance in Networks on Chip". Em: ACM Surveys, vol. 46 (1), 2013, pp. 1-36.

[SIN13] Singh, A.; Shafique, M.; Kumar, A.; Henkel, J. "Mapping on Multi/Many-core Systems: Survey of Current and Emerging Trends". Em: DAC, 2013, pp. 1-10.

[WAC13] Wachter, W. E.; Juracy, L. R.; Neto, W.; Amory, A.; Moraes, F. G. "Runtime Fault Recovery Protocol for NoC-based MPSoCs". Em: ISQED, 2013, pp. 132-139.

[WAN06] Wang, L-T; Wu, C-W; Wen, X. “VLSI Test Principles and Architectures: Design for Testability”. Academic Press, 2006.

[WEB01] Weber, T. S.; “Tolerância a Falhas: Conceitos e Exemplos”. Disponível em http://www.inf.ufrgs.br/~taisy/disciplinas/textos/.