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

RONNISON REGES VIDAL

ESCALONAMENTO PARA SERVIÇOS DE COMUNICAÇÃO DE TEMPO REAL EM REDES EM CHIP

MOSSORÓ – RN
Agosto, 2013

RONNISON REGES VIDAL

ESCALONAMENTO PARA SERVIÇOS DE COMUNICAÇÃO DE TEMPO REAL EM REDES EM CHIP

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

Orientador: Prof. Dra. Karla Darlene Nepomuceno Ramos – UERN.

MOSSORÓ – RN

2013

Catalogação da Publicação na Fonte.

Universidade do Estado do Rio Grande do Norte.

Vidal, Ronnison Reges

I.
Escalonamento para serviços de comunicação de tempo real em redes em chip. / Ronnison Reges Vidal. – Mossoró, RN, 2013.

72f.

Orientador: Prof. Dra. Karla Darlene Nepomuceno Ramos

Tese (Mestrado em Ciência da Computação). Universidade do Estado do Rio Grande do Norte. Universidade Federal Rural do Semiárido. Programa de Pós-Graduação em Ciência da Computação

Bibliotecária: Jocelania Marinho Maia de Oliveira CRB 15 / 319

RONNISON REGES VIDAL

ESCALONAMENTO PARA SERVIÇOS DE COMUNICAÇÃO DE TEMPO REAL EM REDES EM CHIP

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

APROVADA EM: ___ / ___ / _____.

BANCA EXAMINADORA

_____

Prof[a]. Dr[a]. Cláudia Maria Fernandes Araújo Ribeiro (Membro Interno)
Presidente

_____

Prof. Dr. Ivan Saraiva Silva (Membro Externo)
Universidade Federal do Piauí - UFPI

_____

Prof. Dr. Adilson Barbosa Lopes (Membro Externo)
Universidade Federal do Rio Grande do Norte - UFRN

Dedico este trabalho a quem mais me ajudou nessa caminhada, minha mãe, Rosa Maria Maia Reges, e minha orientadora, Karla Darlene Nepomuceno Ramos.

AGRADECIMENTOS

Agradeço a Deus, em primeiro lugar, por todas as oportunidades que ele me proporcionou e por estar vivo para apresentar esse trabalho. Em seguida à minha mãe por todo o apoio e paciência que tem tido por mim e pela liberdade que me deu.

A Professora Karla, minha orientadora, por ter me acompanhado durante o mestrado e pelo incentivo que tem me dado para continuar na vida acadêmica. E aos Professores da banca por estarem avaliando minha pesquisa.

Agradeço ao professor Pedro Fernandes e aos professores do Laboratório de Engenharia de Software da UERN pelos seus incentivos e por me acolherem durante a graduação e a pós-graduação.

Agradeço a João Phellipe pelos conselhos e acompanhamento nos momentos difíceis nesses dois anos. Bem como a todos os colegas da turma de 2011 do mestrado e os companheiros e amigos do Laboratório e da graduação da UERN e da UFERSA.

Agradeço a Rosita, secretária do metrado, por estar sempre me acompanhando e ajudando quando necessário. A Denilson companheiro de estudos pelos dois anos de parceria. E finalmente a todos que estiveram comigo nesses dois anos acompanhando minha jornada, estando perto ou longe fisicamente.

RESUMO

A redução do tamanho dos transistores favoreceu o surgimento de arquiteturas de computadores diferenciadas como os Multiprocessadores em um único chip (*Multiprocessor System on Chip* – MPSoC), que podem utilizar barramentos ou redes em chip como estrutura de interconexão e comunicação. O barramento é o elemento de comunicação mais utilizado, porém as redes em chip são apontadas como a solução para suportar a diversidade funcional e a estrutura complexa dos MPSoC. A utilização dessas arquiteturas em sistemas críticos acarretou em novos desafios para os projetistas de hardware e desenvolvedores de software. Saber lidar com restrições de tempo é um dos desafios, sobretudo quando o objetivo é implementar sistemas de tempo real, seja de tempo real *hard* ou *soft*. Uma das abordagens conhecidas, foco desse estudo, é a análise de escalonabilidade das tarefas de um sistema. A análise de escalonabilidade verifica os parâmetros temporais das tarefas a fim de certificar que os prazos sejam cumpridos mesmo nas situações adversas. Por situações adversas pode-se compreender preempção e concorrência pelos recursos da rede, como canais virtuais nos links físicos e buffers de entrada nos roteadores, nos cenários de pior caso de latência de comunicação da rede em chip. Além da questão de verificação, a análise de escalonabilidade serve como ferramenta juntamente com um algoritmo de atribuição de prioridades para validar uma ordenação de prioridades específica das tarefas. Dessa forma, esta pesquisa investigou a política de escalonamento por prioridade guiada por um método de análise de escalonabilidade baseada nas interferências dos fluxos em uma rede em chip, visando suporte a serviços de tempo real *hard*. A análise de escalonabildade foi realizada em linguagem de alto nível e verificou a satisfação das restrições temporais *hard* em redes em chip que implementam a estratégia de escalonamento por prioridades fixas, considerando os seus inter-relacionamentos, sejam eles diretos, indiretos e de autobloqueio. Os resultados mostraram que a escalonabilidade pode ser alcançada, inclusive em relação às interferências de autobloqueio. O processo de análise de escalonabilidade é fundamental para a política de atribuição de prioridades e consequentemente para a otimização da performance do sistema.

.

Palavras-Chave: Tempo Real, Escalonabilidade, Rede em chip, Arquitetura de comunicação, Hermes, Chaveamento *wormhole*, Prioridade fixa.

ABSTRACT

The reduction of transistor's size has favored the emergence of different computer architectures like multiprocessors on a single chip (Multiprocessor System on Chip - MPSoC) , we can use buses or networks as on-chip interconnect structure and communication. The bus is the most used element of communication, but the on-chip networks are seen as the solution to support the functional diversity and the complex structure of the MPSoC. The use of these architectures in critical systems resulted in new challenges to hardware designers and software developers. Dealing with time constraints is one of the challenges, especially when the goal is to implement real-time systems, real time is hard or soft. One of the known approaches focus of this study is to analyze the schedulability of tasks in a system. The schedulability analysis verifies the temporal parameters of tasks in order to ensure that deadlines are met even in adverse situations. By adverse situations can understand preemption and competition for network resources, such as virtual channels on physical links and input buffers on routers, in worst case scenarios latency of network communication chip. Besides the issue of verification, the schedulability analysis serves as a tool along with an algorithm for assigning priorities to validate a specific ordering of priorities of tasks. Thus, this research investigated the scheduling policy by priority guided by a schedulability analysis method based on interference of flows in a network on chip, aiming to support hard real-time services. Schedulability analysis was performed in a high level language and verified to the satisfaction of timing constraints in hard on-chip networks that implement scheduling strategy for fixed priorities, considering their inter-relationships , whether direct, indirect and self-blocking. The results showed that the scalability can be achieved, also with regard to interference self-blocking. The process of analyzing scalability is crucial for policy prioritization and hence the optimization of system performance.

Keywords: Real Time, Scalability, Network on Chip, Communication Architecture, Hermes, wormhole switching, Fixed priority.

LISTA DE TABELAS

Tabela 1: Fluxos de Tempo Real e suas Características (Shi, 2009). 38
Tabela 2: Descrição dos Benchmark. (Baseada em Ramaprasad & Mueller, 2006). 41
Tabela 3: Caracterísitcas e Latências de resposta do primeiro grupo de tarefas obtidas usando a análise de escalonabilidade (Baseada em Ramaprasad & Mueller, 2006)..... 43
Tabela 4: Descrição do Benchmark acrescido no segundo conjunto de aplicações. (Baseada em Ramaprasad & Mueller, 2006). 44
Tabela 5: Caracterísitcas e tempos de resposta do segundo grupo de tarefas. (Baseada em Ramaprasad & Mueller, 2006). 45
Tabela 6: Descrição do Terceiro Grupo de Tarefas da Aplicação do Veículo autônomo. (Baseado em Shi, 2009) 47
Tabela 7: Descrição do Segundo Benchmark. (Baseada em SHI, 2009). 48

LISTA DE FIGURAS

Figura-1. Layout de Topologias..... 18
Figura-2. Organização de SoC baseada em rede em chip. 18
Figura-3. Bloqueio encadeado de flits. 20
Figura-4. Rede em Chip e Fluxos de Tarefas. 37
Figura-5. Mapeamento do Grupo de Tarefas do experimento 1 em processadores interconectados por uma rede em chip em grelha 2x3. 42
Figura-6. Mapeamento do Segundo Grupo de Tarefas de Aplicações em processadores interconectados em uma rede em chip Mesh 3x3..... 44
Figura-7. Mapeamento do Terceiro Grupo de Tarefas de Aplicações em processadores interconectados em uma rede em chip Mesh 4x4..... 47

LISTA DE SIGLAS

B&B	*Branch and Bound*
BW	*Band Width*
CV	*Canal Vitual*
CMP	*Chip Multiprocessor*
EDF	*Earliest Deadline First*
FFD	*First Fit Decreasing*
Flit	*FLow control unit*
GT	*Guaranteed Throughtput*
IP	*Intellectual Property*
LL	*Least Laxity*
MPSoC	*MultiProcessor System-on-Chip*
NP	*Non-Deterministic Polynomial Time*
RM	*Rate Monotonic*
SDM	*Spacial Division Multiplexing*
SoC	*System-on-Chip*
STR	*Sistemas de Tempo Real*
TDM	*Time Division Multiplexing*
VCT	*Virtual Cut-Through*
WCET	*Worst Case Execution Time*

SUMÁRIO

1. INTRODUÇÃO 12
1.1 OBJETIVOS 15
1.2 ESTRUTURA DO DOCUMENTO 15
2 REDES EM CHIP E SISTEMAS DE TEMPO REAL 17
2.1 REDES EM CHIP 17
2.1.1 Chaveamento 19
2.1.2 Controle de Fluxo 21
2.1.3 Roteamento 21
2.1.4 Arbitragem 22
2.1.4.1 Arbitragem por prioridades 23
2.2 SISTEMAS DE TEMPO REAL (STR) 24
2.2.1 Escalonamento 25
2.2.2 Análise de Escalonabilidade 26
2.3 CONFIGURAÇÃO EXPERIMENTAL 27
3 ESCALONAMENTO 29
3.1 ESCALONAMENTO BASEADO EM FLUXOS 29
3.2 ABORDAGENS DE ESCALONAMENTO 30
3.2.1 Escalonamento de Prioridades Fixas 31
3.3 MODELO DE SISTEMA, RESTRIÇÕES E MÉTRICAS DE ESCALONABILIDADE 32
3.3.1 Interrelacionamentos entre Fluxos de Tarefas 34
3.4 IMPLEMENTAÇÃO COMPUTACIONAL 35
3.4.1 Modelagem em Forma de Grafo 35
3.4.2 Aplicação para Análise de Escalonabilidade 36
3.5 EXEMPLOS NUMÉRICOS E RESULTADOS 37
4 ESTUDO DE CASO 40
4.1 ATRIBUIÇÃO DE PRIORIDADES 40
4.2 EXPERIMENTO 1 41
4.3 EXPERIMENTO 2 44
4.4 EXPERIMENTO 3 46
5 CONCLUSÃO 50
6 TRABALHOS FUTUROS 52
7 REFERÊNCIAS 53
ANEXO 57

1. INTRODUÇÃO

Escalonamento é uma área essencialmente voltada para o agendamento de processos computacionais no âmbito dos sistemas. As atividades de escalonamento possibilitam executar processos de forma viável e concorrente, priorizando determinados tipos de processos (Li, 2003). O escalonamento é um dos principais mecanismos relacionados à comunicação nos sistemas tanto no nível de software dentro do kernel como no nível de hardware na comunicação dos componentes intrachip.

Considerando a evolução das arquiteturas de computadores que surgiram com o encolhimento dos transistores às escalas nanométricas, essa evolução intrachip conduziu o avanço tecnológico de simples barramentos à implantação de redes de comunicação chaveada. Essa nova arquitetura denomina-se de redes em chip e se apresentam como alternativa de interconexão para as arquiteturas CMP (*Chip Multiprocessor*) e MPSoC (*Multiprocessor System-on-Chip*). Enquanto a arquitetura CMP integra vários núcleos de processamento em uma única pastilha de silício, os MPSoC integram sistemas computacionais completos.

As redes em chip surgiram como alternativa para superar as limitações da estrutura de barramento, pois maximizam a largura de banda de comunicação, minimizam o gargalo do desempenho por meio dos acessos concorrentes e da escalabilidade (Benini and DeMicheli, 2002). O desempenho dos sistemas baseados em redes em chip não é medido somente pela performance dos núcleos de processamento ou núcleos IP (*Intectual Property* – IP), mas também pela eficiência na colaboração entre esses componentes.

A eficiência de comunicação é determinada pelos mecanismos da rede que tratam inclusive do escalonamento das tarefas. Além dos outros mecanismos que influenciam no atendimento dos requisitos das aplicações o escalonamento têm de ser específicado para atender os serviços implementados. Sendo que as redes em chip têm a capacidade de fornecer diferentes níveis de serviços a várias aplicações.

Um dos grandes desafios dos sistemas multiprocessados baseados em redes em chip é o escalonamento de tarefas, especialmente quando se trata da implementação de serviços de tempo real. Para essas arquiteturas o problema de alocação e escalonamento de tarefas de tempo real em sistemas multiprocessadores se torna um problema NP-Completo devido a inevitável existência de comunicação paralela (Srikanth, G., Shanthi, A. P., Maheswari, V. U., Siromoney, A., 2012).

Segundo Li (2003) as aplicações, especialmente de tempo real, tem de ser decompostas em pequenas tarefas e têm de ser escalonadas para executar em ordem e desempenhar a funcionalidade requida pelo sistema. Uma forma de determinar que o escalonamento seja o mais eficiente e que os

prazos sejam encontrados é por meio da análise de escalonamento. Esta técnica emprega algoritmos de escalonamento enquanto tenta alcançar a utilização ótima do processador. É preciso observar que essa análise visa somente à forma de o sistema cumprir os requisitos temporais e não os funcionais.

Muitos pesquisadores propuseram diferentes abordagens para o desenvolvimento de pesquisas em análise de escalonabilidade. Todos eles visam minimizar o tempo de conclusão dos programas paralelos e maximizar a utilização dos processadores buscando sempre o cumprimento das exigências dos prazos das aplicações e dos sistemas.

Um dos pesquisadores precursores foi Mutka (1994), o qual primeiramente resolveu esse problema em sistemas cuja comunicação se dava por meio de redes. A técnica consistiu da utilização do algoritmo *Rate Monotonic* (RM) como parâmetro para atribuição das prioridades fixas nos fluxos que compartilhavam links comuns. No algoritmo *Rate Monotonic* os fluxos de menores períodos têm maiores prioridades. Liu e Layland (Liu and Layland, 1973) anteriormente já haviam solucionado esse problema de escalonabilidade de tarefas por meio de um teste de limite de utilização pelo RM. Contudo, para um conjunto de tarefas em um único processador.

Mutka aplicou o teste de utilização em cada link da rede supondo que os fluxos que compartilhassem os links físicos pudessem encontrar seus prazos. Mas é necessário destacar que o chaveamento no qual estava trabalhando era *wormhole* e o teste não seria adequado para garantia de entrega de pacotes. Isso porque cada link é analizado independentemente em termos de utilização da largura de banda, condição somente verdadeira se para cada canal virtual (CV) houvesse espaço suficiente no buffer para armazenar o pacote completo em saltos intermediários.

Sendo assim a técnica de Mutka não éra capaz de determinar a escalonabilidade necessária. Balakrishnan (Balakrishnan & Ozguner, 1998) apresentou uma técnica de análise *off-line* denominado *lumped link model*. Nesse modelo todos os *links* envolvidos na travessia de um fluxo em um nível de prioridade específico são considerados como um recurso compartilhado. A partir desse recurso é calculada a interferência dos fluxos de alta prioridade que utilizam um ou mais links ao longo do caminho do fluxo em destaque.

Nessa técnica primeiramente é encontrado o conjunto máximo de interferência para em seguida com o conjunto encontrado calcular o pior caso de interferência. Esssa técnica apesar de ser eficiente é bastante pessimista desde que as interferências diretas e indiretas não se distinguem sendo tratadas da mesma maneira. O que induz a interferência excessiva desde que todos os fluxos, não importando se são diretos ou indiretos, devem ser calculados em sequência.

O modelo de Kim (Kim et al., 1998) permite preempção no nível de flit e avalia as latências no pior caso da rede por diagramas de linha de tempo. O algoritmo de Kim funciona pela geração do diagrama de tempo inicial direto modificando-o e apagando os slots de tempo indireto que não

seriam contidos. Apesar de o algoritmo de Kim diferenciar fluxos de maior prioridade diretos e indiretos ainda havia interferência excessiva.

Lu (Lu, Jantsch, Sander, 2005) utiliza uma abordagem parecida com a de Kim na busca pela latência da rede de pior caso propondo um modelo de árvore de contenção para distinguir interferências diretas e indiretas e capturar a utilização atual dos *links*. Com o modelo de árvore de contenção por ele proposto, foi considerado o escalonamento paralelo em links de comunicação disjunta, o que reduziu a possível interferência dos fluxos de maior prioridade.

É também provado que encontrar o cenário de pior caso é um problema NP-*Hard* quando ocorre comunicação paralela. Porém este modelo considera o cenário de pior caso onde a liberação de todos os fluxos é ao mesmo tempo, situação que não é verdadeira quando se trabalha com intereferências indiretas.

Então a abordagem de Shi (Shi, 2009) aplicada neste trabalho é baseada nos métodos anteriores, resolvendo os problemas encontrados e implementando um novo método de análise de latência de pior caso da rede. Pela avaliação de diversos interrelacionamentos e atributos do serviço entre os fluxos de tráfego, essa abordagem prediz a latência de transmissão com três interferências diferentes quantificáveis: interferências diretas, indiretas e de autobloqueios. Devido a inevitável existência de comunicação paralela essa abordagem é capaz de determinar a exata escalonabilidade dos fluxos de tempo real provando que é um problema geral NP-*hard*.

Tal técnica visa à implementação de serviços de tempo real para plataformas baseadas em redes em chip. O serviço de tempo real está ligado diretamente ao mecanismo de arbitragem influenciado pelo escalonamento. Para o escalonamento baseado em classe, os tráfegos de dados são caracterizados em um número reduzido de prioridades fazendo com que a rede lide com serviços com maior granularidade. Quando o escalonamento é baseado em fluxos cada prioridade é considerada um serviço, limitado pelo número de fluxos, caracterizando os serviços por uma granularidade fina.

Quando serviços de tempo real são tratados por uma rede surgem dois conceitos de tempo real que são o tempo real *hard* e *soft*. No tempo real *hard* o serviço de comunicação da rede deve cumprir com o prazo das aplicações. Enquanto que o tempo real *soft* que mantém uma média de transmissão mesmo que não possa garantir totalmente o tempo de resposta da rede para as aplicações.

O escalonamento de tarefas em tempo real minimiza o tempo total de execução de aplicações ao mesmo tempo em que maximiza a utilização dos componentes de processamento satisfazendo as restrições de tempo.

O escalonamento é executado por meio de parâmetros de tempo como período de execução e prazo para execução. Utilizando esses parâmetros o escalonador da rede certifica-se que as

restrições de tempo da rede sejam garantidas pelo tempo de resposta da rede. Esse tempo de resposta é o tempo que a rede leva para carregar as mensagens de um núcleo origem até um núcleo destino.

O âmbito da pesquisa se dá no escalonamento da rede em chip para serviços de tempo real. Aplicando técnicas de análise de tempo de resposta das tarefas distribuídas pela rede. A análise dos atributos de tempo das tarefas é realizada baseada nas interferências entre as tarefas. A análise é reealizada considerando o escalonamento baseado em prioridades fixas.

1.1 OBJETIVOS

O objetivo principal desse trabalho é investigar a política de escalonamento por prioridade guiada por um método de análise de escalonabilidade baseada nas interferências dos fluxos em uma rede em chip. O intuito de aplicar essa técnica às redes em chip é dar suporte a serviços de tempo real *hard*.

A organização e a definição das características da rede é uma parte crucial na implementação desse serviço de tempo real. São exploradas essas características de modo a propiciar previsibildiade à rede. Para isso são definidos os mecanismos de chaveamento, roteamento, arbitragem e topologia.

Por meio de aplicações testes com diferentes configurações de rede o método de análise de escalonabilidade é submetido para avaliar o tempo de resposta da rede e cumprimento dos prazos. Conforme as interferências causadas pela preempção dos fluxos de maior prioridade sob os outros de menores ocorre que o tempo de resposta da rede deve ser baixo o suficiente para os fluxos não perderem os prazos.

1.2 ESTRUTURA DO DOCUMENTO

Este documento está dividido em 6 capítulos. O capítulo 2 aborda os sistemas baseados em rede e faz relação com os sistemas de tempo real apresentando os mecanismos da rede envolvidos com o serviço de tempo real e a problemática por trás da implementação desse serviço nas redes em chip. No capítulo 3 o método de análise de escalonamento é apresentado, mostrando como resolver o problema de escalonabilidade. No capítulo 4 são mostrados os resultados de aplicações utilizando o

método apresentado no capítulo 3. O quinto capítulo traz as considerações finais da pesquisa e o sexto capítulo apresenta os trabalhos futuros.

2 REDES EM CHIP E SISTEMAS DE TEMPO REAL

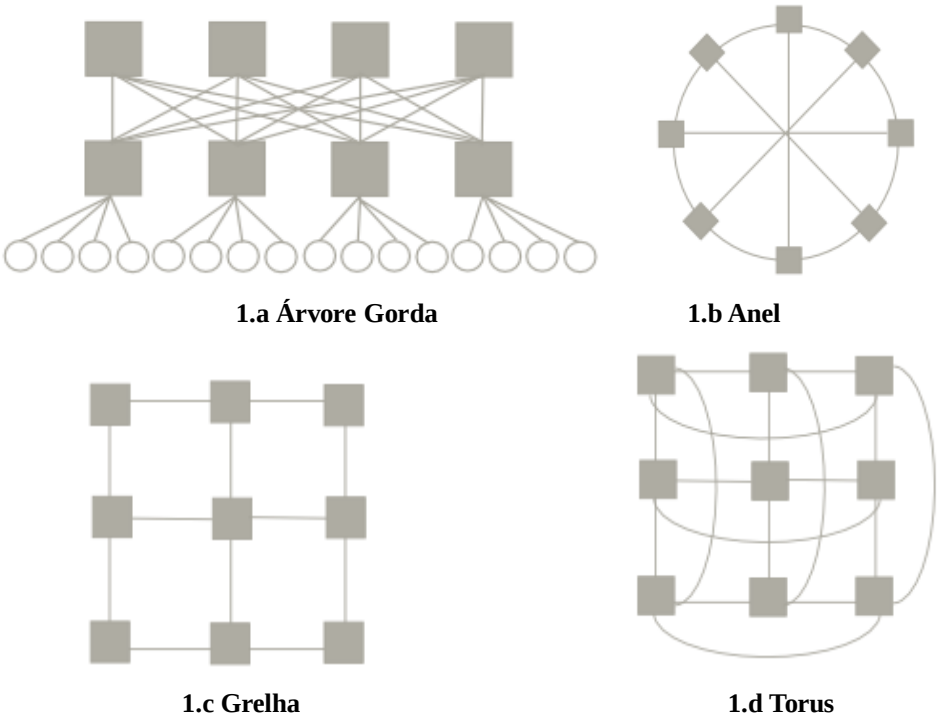
Este capítulo trata dos conceitos fundamentais das redes em chip, dos mecanismos de comunicação, bem como, dos conceitos relacionados ao serviço de comunicação de tempo real suportado pelas redes em chip.

2.1 REDES EM CHIP

As redes em chip adotam as características das redes de computadores convencionais em um ambiente intrachip considerando a topologia e os mecanismos de comunicação da rede (Benini & Micheli, 2002). A topologia diz respeito à lógica estrutural relacionada à organização dos componentes da rede. Enquanto que os mecanismos especificam os recursos da rede e as regras que incidem sobre a comunicação dos roteadores.

A organização dos roteadores e dos links, considerando a conectividade da rede, é de responsabilidade da topologia. Então as topologias podem ser classificadas como regulares e irregulares devido à forma como ocupam a área do chip, e diferenciam-se na padronização no tamanho de seus componentes. As topologias regulares podem ser organizada em dimensões 2D ou 3D. Sendo que, as configurações de uma topologia regular 2D têm menor complexidade de implementação se comparado com a 3D. As topologias 2D regulares, por sua vez, podem ser classificadas em forma de grelha, árvore gorda, anel e em forma de tórus.

A Figura -1 ilustra o *layout* de cada uma dessas topologias. A topologia árvore gorda está ilustrada na Figura 1.a, onde todos os roteadores se conectam e os nós folhas representam os núcleos ou componentes da rede. A Figura 1.b ilustra a topologia em formato de anel totalmente ligada, cujos saltos de pacotes na rede jamais serão superiores a dois. A Figura 1.c representa a topologia em forma de grelha, e a Figura 1.d representa a topologia tórus, que direfencia-se da grelha devido os roteadores das extremidades se conectarem.



1.a Árvore Gorda **1.b Anel**

1.c Grelha **1.d Torus**

Figura-1. Layout de Topologias

As topologias podem ser representadas por meio de grafos. A representação em forma de grafo exibe a disposição dos roteadores, de forma que os vértices representam os roteadores e as arestas representam os *links* de comunicação.

A Figura-2 apresenta um exemplo de MP-SoC com rede em chip de topologia em grelha 4x4, composta por roteadores e links que interconectam 16 núcleos IP (*Intelectual Property*). Cada roteador possui portas de comunicação que interligam os roteadores vizinhos e sendo uma delas a porta denominada local que conecta o núcleo IP à rede. Entre o roteador e o núcleo existe uma interface de comunicação com a rede.

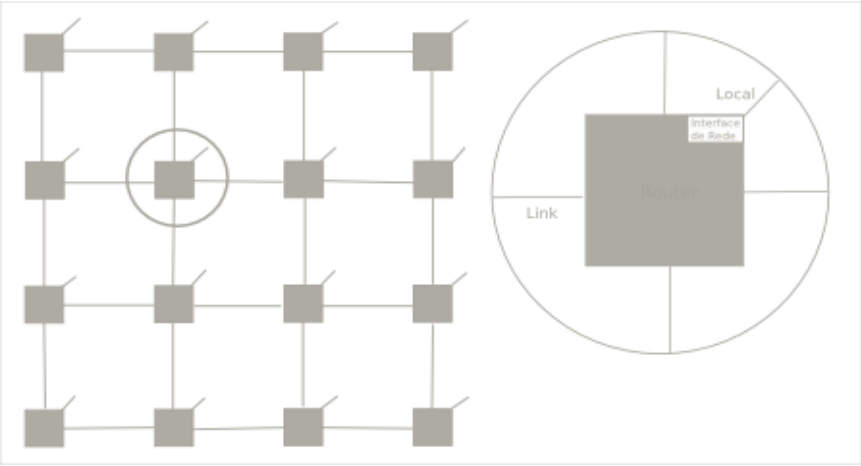


Figura-2. Organização de SoC baseada em rede em chip.

Nas subseções a seguir os mecanismos de rede serão apresentados. Na seção 2.1.1. são mostrados os tipos de chaveamento e como influenciam os pacotes. Na seção 2.1.2. é mostrado como é feito o controle de fluxo na rede, sua relação com o chaveamento e o seu impacto na comunicação. Na seção 2.1.3. é mostrado o mecanismo de roteamento e como é realizado o direcionamento dos fluxos na rede. Na seção 2.1.4. é mostrado a arbitragem da rede e as políticas de escalonamento .

2.1.1 Chaveamento

A comunicação da rede se dá através da infraestrutura de links e roteadores, contudo devido à complexidade envolvida na transmissão de dados os mecanismos da rede atuam para que as mensagens atinjam o destino.

As duas principais formas de chavear mensagens na rede são o chaveamento por circuitos e o chaveamento por pacotes, assim como nas redes convencionais. Dependendo do tipo de chaveamento os dados assumem diferentes tamanhos, como grandes pacotes ou pequenos *flits*.

Quando é necessário o uso de recursos dedicados a um serviço específico da rede ou aplicação o chaveamento por circuito reserva caminhos fim-a-fim de uma fonte a um destino. Essa reserva de caminhos é feita por um processo orientado a conexão. Contudo, neste tipo de processo a largura de banda não é totalmente aproveitada, no sentido que em certos períodos as tarefas não ocuparão toda a largura de banda da rede.

Para minimizar o desperdício de banda exigido pelo chaveamento por circuito o chaveamento baseado em pacotes surge como uma solução. Este tipo de chaveamento não faz reservas de recursos e os pacotes percorrerem os caminhos independentes de conexões entre fonte e destino. Há três principais técnicas de chaveamento por pacotes que são: *store and foward*, *virtual cut-through* (VCT), e *wormhole*.

Essas técnicas cada qual têm suas maneira de tratar os pacotes, o que envolve vantagens e desvantagens na adoção de cada uma delas. O primeiro tipo de chaveamento a surgir foi o *store and foward*, nele os *buffers* são maiores para armazenar todo um pacote. Essa situação implica em um problema, a reserva de *buffer*, que impede que outro pacote possa ser armazenado no roteador até que o primeiro pacote tenha sido repassado. Além disso, o *buffer* permanece ocupado até que o pacote nele seja completamente enviado para o roteador.

Isto fez surgir uma lacuna de desempenho para todo serviço que implementasse prioridades e interrupções no nível de pacote. Em resposta a esse problema o chaveamento *virtual cut-through* resolve o problema de repasse usando a técnica de *pipeline* nos pacotes. O *pipeline* corrige o

problema de repasse permitindo que os pacotes que chegam ao roteador continuem pelo caminho caso haja espaço no roteador seguinte sem armazenar por completo o pacote. Contudo, ainda há a necessidade de armazenar todo um pacote no caso deste não poder ser repassado. Essa característica trás altos custos de área no chip por necessitar de buffers que comportem todo o pacote.

Para contornar esse tipo de situação o chaveamento *wormhole* divide os pacotes de dados em unidades menores denominados *flits* para enviá-los pela rede. Na divisão de um pacote em *flits* este são empacotados em dois tipos de *flits*, os *flits* de cabeçalho e os de carga. Nos *flits* cabeçalhos estão às informações provenientes do roteamento, do controle de fluxo e de tempo se for o caso de o árbitro necessitar. Já nos *flits* de carga estão os dados referentes à aplicação executada pela rede.

Assim os *buffers* para esse tipo de chaveamento em específico são projetados para suportarem *flits* ao invés de pacotes inteiros. E assim como o chaveamento *cut-through* não há a necessidade de armazenamento de todo um pacote para repassar os *flits* para o próximo roteador.

O chaveamneto *wormhole* apresenta problemas quando considerado um alto nível de congestionamento gerado pelo pipeline dos *flits*, fenômeno conhecido como bloqueio encadeado. Esse fenômeno ocorre quando o canal por onde os *flits* irão passar está ocupado. Enquanto ocupado à medida que os *flits* vão chegando eles vão preenchendo o *buffer*. Quando sua capacidade é completada os *flits* restantes vão permanecer bloqueados nos roteadores anteriores.

Na figura-3 está representado o comportamento de bloqueio encadeado no nível de roteador. O roteador 1 é a fonte de envio de pacotes, seguindo pelos roteadores 2 e 3 que fazem parte da rota, contudo o roteador 4 está ocupado, então os *flits* vão ocupando os buffers nos roteadores 2 e 3 até que não possam mais armazenar qualquer *flit*.

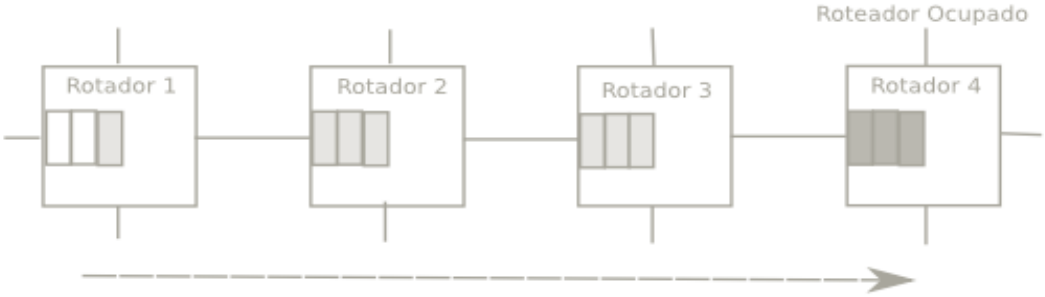


Figura-3. Bloqueio encadeado de flits.

O problema de encadeamento é comum a sistemas que implementam o chaveamento *wormhole*. Porém para diminuir a incidência desse fenômeno a técnica de canais virtuais é empregada no mecanismo de controle de fluxo. Os canais virtuais dividem o *link* físico logicamente

de forma que mais *flits* possam passar por ele e aumentando a utilização da largura de banda da rede. Na seção a seguir os mecanismos de controle de fluxo serão abordados.

2.1.2 Controle de Fluxo

Os mecanismos de controle de fluxo são responsáveis por evitar problemas tais como disponibilidade para envio, estouro da pilha de *buffer* e altas cargas tráfegos de dados. Para fazer o controle de fluxo em redes em chip são implementadas algumas técnicas provenientes das redes convencionais, como *handshake*, créditos e canais virtuais (CV).

Os canais virtuais juntamente com o mecanismo de chaveamento *wormhole* procuram aliviar o congestionamento de pacotes dentro do meio físico. A técnica de canais virtuais funciona pela divisão lógica do meio físico, o *link* e o *buffer* em filas independentes. Essa alocação virtual caracteriza o aumento da vazão da rede para cada canal virtual implementado. Desta forma vários pacotes podem dividir um canal físico com eficiência de largura de banda, trazendo, contudo, maior complexidade para o árbitro. Outra vantagem na utilização dos CV é o suporte a diferentes níveis de serviços, o que também torna a rede livre de *deadlocks*. Por conseqüência os canais virtuais trazem flexibilidade e aumento no desempenho.

O mecanismo de controle fluxo influencia na propagação rápida dos *flits*, não se restringindo simplesmente aos canais virtuais, mas também está relacionada com a alta utilização dos *buffers*. As técnicas comumente encontradas para controle de fluxo nas filas dentro do *buffer* são: *handshake* e o esquema baseado em créditos.

Ambas as técnicas tradicionais nas redes convencionais se encaixam perfeitamente no ambiente em chip. As redes que implementam a estratégia *handshake* de controle de fluxo, lidam com a lacuna de tempo que envolve a troca de mensagens sinalizadoras entre os roteadores. Neste caso a técnica de créditos é empregada com maior eficiência por o roteador ter noção da quantidade de *flits* que há no próximo. No caso do esquema baseado em créditos os roteadores da rede mantêm contadores em suas portas de saída e esses contadores monitoram a capacidade dos *buffers* de entrada nos roteadores vizinhos a quem estão interligados. Dessa forma quando o contador chega à zero significa que ele não poderá mais transmitir até que o buffer do próximo sinalize que haja espaço

2.1.3 Roteamento

O roteamento é o mecanismo que dá sentido e direção aos pacotes dentro da rede. Essa característica da rede guia as mensagens para seus destinos de modo que a comunicação seja

efetivada. Então, o mecanismo de roteamento é o responsável por transferir dados de um roteador para outro determinando a rota que os *flits* irão seguir.

A complexidade de implementação desse mecanismo pode ser considerada alta ou baixa dependendo do seu tipo, se é determinístico ou adaptativo. No roteamento adaptativo a complexidade se dá pela necessidade de o pacote encontrar uma rota alternativa, em tempo de execução, caso haja bloqueios e congestionamento no percurso.

Na comunicação por meio de um roteamento determinístico o caminho é delimitado pelo endereço de destino, produzindo a mesma rota dado o par fonte/destino. Considerando que os serviços de tempo real devem ter alta previsibilidade o roteamento determinístico não é a única escolha, porém a relação com a previsibilidade é bastante explorado para serviços de tempo real.

O roteamento determinístico também têm uma estreita relação com a topologia na qual a rede está organizada. No caso de uma topologia regular 2D em grelha uma técnica de roteamento comumente utilzada é a XY ou mesmo a YX, por conta principalmente de sua simplicidade. Em ambas as estratégias o caminho que o pacote percorre em um sentido, ou horizontal ou vertical, e em seguida percorrendo o sentido oposto.

2.1.4 Arbitragem

A arbitragem é o mecanismo da rede responsável por implementar a política de escalonamento. Inspirado nas redes convencionais, a comunicação nas redes em chip adota regras para definir qual tráfego irá usufruir dos recursos da rede.

Seguindo essas regras o mecanismo de arbitragem resolve os problemas de contenção e de tomada de decisão em relação à concorrência pelo acesso ao meio físico. As regras de arbitragem são denominadas de políticas de escalonamento e são classificadas em relação ao seu domínio, se por reserva de recursos ou por tempo de execução.

A reserva de recursos se caracteriza por alocação prévia dos recursos da rede para evitar contenções antes de iniciar a comunicação. Neste tipo, necessariamente, não é aplicado uma forma de escalonamento, porém a alocação se dá de duas formas como multiplexação por divisão de tempo (*Time Division Multiplexing* - TDM), e multiplexação por divisão de espaço (*Spacial Division Multiplexing* - SDM).

A arbitragem em tempo de execução por sua vez resolve problemas de contenção via escalonamento. As principais políticas de arbitragem para essa disciplina são o escalonamento *Round-Robin* e o escalonamento baseado em prioridades.

O escalonamento *Round-Robin* é um tipo de escalonamento dito justo, por permitir tempo de execução igual para todos os fluxos que estão esperando nos *buffers*. Contudo, não é um

escalonamento apropriado para sistemas tempo real. O escalonamento por prioridades se destaca por permitir que algumas tarefas se sobressaiam em relação a outro por conta de suas prioridades.

Estruturalmente, na implementação do escalonamento por prioridades, o árbitro é composto por duas entidades, o árbitro e o controle de admissão. O árbitro é o responsável por alocar a transmissão dinamicamente permitindo assim que alguma das mensagens no *buffer* possa ser enviada.

Dessa forma, a entidade de controle de admissão decide se algum novo fluxo de tarefa deve ser aceito pelo árbitro, considerando o equilíbrio da escalonabilidade dos outros fluxos. O controle de admissão realiza análises dos fluxos de modo *online* ou *offline*. Quando o controle de admissão trabalha em tempo de execução em um sistema, este é executado pelo sistema operacional. Já na questão *offline*, o controle de admissão é feito no momento do projeto.

A vantagem de ter o controle de admissão de forma *offline* é manter a previsibilidade do sistema desde a sua fase de projeto. Porém para execução correta das tarefas é necessário definir os mecanismos da rede de forma que seja o mais determinística possível. Por meio do determinismo da rede é calculado a latência que a rede irá incidir nas comunicações e verificar a escalonabilidade que o controle de admissão é responsável por manter. O método envolvido no calculo da latência é visto nas seções que se seguem.

2.1.4.1 Arbitragem por prioridades

A arbitragem por prioridades é um método amplamente difundido e aplicado na área de sistemas de tempo real. Essa ampla utilização se dá porque a partir da atribuição das prioridades a rede é capaz de diferenciar serviços, como, por exemplo, os serviços de tempo real.

O escalonamento dirigido por prioridades pode ser classificado pela granularidade das tarefas que pode ser grossa ou fina. Na granularidade fina para cada fluxo de tarefa é atribuído uma prioridade distinta e um canal virtual exclusivo, também denominada como prioridades baseadas em fluxo.

Considerando a granularidade das tarefas, a complexidade no projeto de uma rede em chip pode ser complexa ou não. No caso da granularidade fina para cada prioridade um canal virtual é alocado dentro dos *buffers* da rede.

Já na granularidade grossa a complexidade de implementação é menor em comparação por conta dos fluxos que são classificados em forma de serviços. Para cada serviço é atribuído uma prioridade e um canal virtual ao invés de um canal virtual para cada fluxo, e é denominado de prioridade baseadas em classes.

Com base na alocação de canais virtuais no método de prioridades baseado em fluxo as prioridades podem ser fixas ou dinâmicas. Na política de prioridades dinâmicas cada pacote recebe uma prioridade individual dinamicamente. Porém essa característica não é apropriada nos sistemas de tempo real que necessitam de determinismo. Esse determinismo é encontrado na política de prioridades fixas onde cada fluxo requer uma prioridade distinta antes da comunicação que não muda durante seu período de serviço.

2.2 SISTEMAS DE TEMPO REAL (STR)

Os sistemas de tempo real são caracterizados pela importância dada à exatidão temporal. Essa exatidão temporal não significa execução com tempos de resposta imediatos, mas sim o cumprimento de prazos dentro de um período de tempo estipulado. Embora os dados estejam processados de modo correto, quando o prazo não é cumprido o dado permanece obsoleto para o sistema.

Os ambientes para os quais os sistemas de tempo real são projetados são dinâmicos, onde eventos ocorrem de modo síncrono e assíncrono. Por conta desses ambientes as informações têm um prazo específico de validade em resposta aos eventos. Sendo assim, é de responsabilidade do sistema garantir o tempo de resposta para a aplicação e tornar viável sua execução.

Os STR são classificados de duas maneiras, STR *hard* ou STR *soft* por algumas características próprias dos sistemas. Como, por exemplo, a tolerância às falhas temporais, a utilidade dos resultados após o prazo e a severidade como é lidado com os resultados pela perda do prazo.

Nessa seção os sistemas de tempo real são definidos e classificados quanto à seriedade imposta pelas aplicações na validade temporal de suas informações. Outro aspecto é a problemática de implementação de tempo real como um serviço nas redes em chip.

- Tempo Real *Hard*

Os sistemas de tempo real *Hard* possuem restrições temporais rígidas implicando em situações onde é imperativo o cumprimento dos prazos. Quando essas restrições não são obedecidas à ocorrência de uma falha é inevitável o que causa a todo o sistema sérias penalidades, como riscos à vida humana ou ao meio ambiente (*Li & Yao, 2003*).

A necessidade principal neste tipo de aplicação é a previsibilidade para garantir que os requisitos funcionais e temporais sejam respeitados, mas principalente os temporais. Quando os

resultados de computação não respeitam as restrições de tempo à depreciação na validade incorre em sérias penalidades para a aplicação.

Sistemas de tempo real *hard* são intolerantes quanto às falhas temporais e tem alto grau de severidade quanto a utilidade dos resultados por conta dessas falhas.

- Tempo Real *Soft*

Sistemas de tempo real com restrições de tempo *soft* têm algum grau de flexibilidade na modelagem de aplicações dirigidas por tempo. Por flexibilidade entende-se que há níveis de tolerância para com os prazos e resultados na aceitabilidade dos tempos de resposta.

O serviço de tempo real *soft* para sistemas baseados em redes em chip é conhecido por manter uma média na vazão de dados na rede. Esse serviço garante para as aplicações uma reserva de recursos para os piores cenários (Rijpkem et al., 2003).

Um exemplo de aplicação seria um sistema de *streaming* de vídeo que implica em reservar largura de banda para certa taxa de transferência mínima para execução, mesmo quando sua taxa média de comunicação é inferior (Rijpkema et al., 2003). O problema com tal abordagem trata-se da ociosidade dos recursos por conta de subutilização da largura de banda da rede.

No projeto de STR baseados em redes em chip é necessário que as aplicações apresentem parâmetros para representar as restrições temporais. Considerando que às redes em chip podem classificar os fluxos de tráfego por prioridade no nível de pacote é possível anexar outras informações no flit de cabeçalho.

Como o pacote da rede é dividido em *flits* de carga e de cabeçalho as informações de roteamento, de controle de fluxo e das restrições temporais podem ser alocadas nos *flits*. Os parâmetros de tempo de um pacote são representados pelo prazo ou *deadline* e pelo período. É interessante que as redes em chip sejam projetadas para tratar essas informações, de modo que as aplicações de tempo real sejam executadas no tempo correto. A questão principal é conseguir com que a rede mantenha o tempo de resposta satisfatório para cumprir com as marcações de tempo de prazo e período no pacote.

2.2.1 Escalonamento

O escalonamento de tarefas é uma atividade de responsabilidade do árbitro que juntamente com outros mecanismos controlam os pacotes da rede. Contudo, sabendo da classificação dos STR em *hard* ou *soft*, as aplicações de tempo real devem ser separadas daquelas que não necessitam cumprir

com prazos. Conforme o árbitro seja capaz de separar os fluxos ou serviços de forma individual, essa divisão se traduz no escalonamento baseado em prioridades.

A forma como as prioridades devem ser enxergadas dentro deste tipo de escalonamento relaciona outro conceito denominado de preemptividade. A preemptividade é um fenômeno de sobreposição de tarefas, pois uma tarefa impedirá outra de executar por ter uma prioridade maior.

A ocorrência de preemptividade também surge durante o lançamento de pacotes na rede de uma tarefa de prioridade maior para que essa seja atendida mesmo quando ela for disparada por algum evento assíncrono. Essa característica de preemptividade garante liberdade de execução de um serviço com maior prioridade para que tenha melhores chances de encontrar seu prazo e garantir o tempo de resposta.

Sobressaindo essas características, o escalonamento é uma peça importante no funcionamento da rede, porém para serviços de tempo real é necessário um estudo sobre a escalonabilidade do sistema e do tráfego de aplicações. A efetividade do escalonamento para serviços de tempo real deve-se ao elemento da previsibilidade. Então para escalonar tarefas em tempo real uma forma de garantir a previsibilidade é prever o tempo que irá tomar cada uma e os recursos que serão empregados pela rede na comunicação.

2.2.2 Análise de Escalonabilidade

Considerando a importancia da análise de escalonabilidade para o escalonamento de tarefas de tempo real, essa seção introduzirá um método de análise de escalonabilidade que busca identificar as interferências causadas pela preemptividade na rede (Shi, 2009). O método de análise de escalonabilidade realiza uma previsão por meio de cálculos de latência baseado nas interferências que as tarefas sofrem ao longo de sua travessia na rede. Nesse método as interferências são classificadas como: direta, indireta e de autobloqueio.

As interferências são influenciadas diretamente pelos mecanismos que a rede emprega na comunicação, tais como chaveamento, roteamento e controle de fluxo. A arbitragem é o principal mecanismo, a qual aplica o escalonamento para tarefas baseado em prioridades para esta análise.

Descobrir qual o tempo de resposta da rede, devido a essas interferências, é à base dos cálculos de análise de escalonabilidade. Essas interferências são encontradas por meio das rotas dos pacotes, desde que os *flits* de cada pacote herdam a mesma prioridade e rota.

Nessa situação os pacotes de maior prioridade simplesmente causam preempção naqueles de menor e estes precisam também cumprir com seus prazos. Então ao descobrir as interferências os parâmetros de tempo daqueles fluxos de menores prioridades são usados para medir o tempo de resposta da rede e verificar se serão capazes de cumpri-los.

2.3 CONFIGURAÇÃO EXPERIMENTAL

As premissas adotadas para este trabalho são baseadas no modelo da rede em chip Hermes (Moraes, Calazans, & Mello, 2004). A topologia assumida é a direta 2D em forma de grelha, onde os roteadores estão distribuídos de forma regular. Na figura 3 é ilustrado como se dá essa distribuição e é observado facilmente que na topologia em grelha os nós interligados formam uma espécie de matriz linear (Duato, Yalamanchili, & Ni, 2002).

Este trabalho assume que a arbitragem é realizada por prioridade fixa preemptiva, onde as prioridades são fixadas no início de toda execução. Esta técnica incorre em menores sobrecargas no sistema em comparação com a política de prioridade dinâmica, desde que não é necessário determinar a prioridade de uma tarefa em tempo de execução. Além do que a fixação das prioridades podem se dá em nível de hardware (*Li & Yao, 2003*).

A técnica utilizada para o chaveamento é baseada em *wormhole*, cujo escalonamento é realizado no nível de *flits*. Desde que os *flits* sejam as unidades escalonáveis dentro da rede, o *flit* de cabeçalho ganha importância na análise de escalonabilidade. O *flit* de cabeçalho governa a rota do fluxo avançando pela rede com as informações importantes de roteamento construindo o caminho para os *flits* restantes que o seguem em forma de *pipeline* (Agarwal, 2009; Lee, 2003; Shi & Burns, 2008).

De acordo com (Kim et al., 1998; Lee, 2003; Mutka, 1994; Shi, Burns, Indrusiak, 2010), o chaveamento *wormhole* desencadeia bloqueios no decorrer da transmissão dos pacotes na espera de canais ocupados ao longo do caminho. É assumido que sistemas de tempo real com escalonamento baseado em prioridades preemptivas dão preferência ao *flit* cabeçalho de maior prioridade acontecendo assim o bloqueio das mensagens de menor prioridade.

As técnicas de chaveamento e de controle de fluxo complementam-se para diminuir o problema de congestionamento, aumentando as taxas de garantia de entrega (Lee, 2003; Mello et al., 2005). Admite-se que o número de canais virtuais é igual à quantidade de níveis de prioridade.

Ter conhecimento da rota que os pacotes percorrerão é fundamental nos cálculo de latência para previsibilidade da rede, sob esse contexto este estudo é realizado com a implementação do roteamento XY. Essa estratégia de roteamento é classificada como determinista e a locomoção dos *flits* atravessando a rede desenvolve-se deslocando primeiro na posição horizontal e em seguida na posição vertical (Agarwal, 2009; Singh et al., 2010; Zeferino & Susin, 2009).

Para flexibilizar a comunicação entre os roteadores e determinar se os *buffers* têm capacidade de armazenar os *flits* que vêm chegando o mecanismo de controle de fluxo baseado em créditos facilita a troca de informações. O controle de fluxo baseado em créditos mantém a

contagem de *flits* disponíveis no *buffer* do roteador vizinho tornando-o ciente das transferências de *flits* para evitar sobrecargas e congestinamentos na rede (Duato et al., 2002; Gratz et al., 2006; Mello et al., 2005).

A motivação dessa configuração experimental surge pela necessidade de manter os sistemas com redes em chip o mais deterministas quanto possível para que possam fornecer serviços em tempo real. O comportamento determinístico dos componentes da rede também é o que mantém a previsibilidade. Conforme a previsibilidade seja mantida, o método análise de escalonabilidade é associado a essas características da rede. Método este que é baseado no cenário de pior caso de latência de pacotes da rede.

3 ESCALONAMENTO

Este capítulo apresenta a análise de escalonabilidade a ser aplicada às redes em chip visando atender os requisitos das aplicações de tempo real *hard*.

3.1 ESCALONAMENTO BASEADO EM FLUXOS

Nas redes em chip a política de escalonamento utilizada é dependente dos serviços implementados pela rede. Para os serviços de tempo real a política de escalonamento frequentemente utilizada é baseada em prioridades.

O escalonamento por prioridades diferencia-se quanto à granuliaridade das prioridades em relação ao fluxo de tarefas. Quando se tem granularidade grossa de prioridades por fluxos de tarefas o escalonamento é baseado em classes. Contudo, quando se têm granularidade fina o escalonamento é baseado em fluxos.

O escalonamento por prioridade baseado em fluxos define por qual canal o fluxo de comunicação deverá passar, constituindo uma relação de um para um entre prioridades e fluxos. Em consequência dessa relação é estabelecido um circuito lógico virtual da fonte ao destino pela reserva de canais virtuais ao longo do caminho do fluxo.

Considerando a prioridade fixa, a cada fluxo é atribuído uma prioridade distinta que não é alterada ao longo do período de execução. Adicionalmente à utilização de prioridades fixas surge o efeito de preempção. A preempção nas redes em chip ocorre quando um fluxo é impedido de ser enviado, devido a um fluxo de maior prioridade estar ocupando os recursos da rede.

As preempções dos fluxos no decorrer do percursso pela rede tornam-se contenções aumentando o atraso no envio dos pacotes. Esse aumento de atraso para serviços de tempo real deve ser considerado para que o tempo de resposta da rede atenda ao prazo do fluxo.

Consequentemente, nos serviços de tempo real, é importante determinar o atraso máximo que o fluxo de tarefa pode sofrer por conta da preemptividade, sendo possível fazê-lo pelo cálculo da previsibildade do limite superior de atraso na rede. A abordagem tratada no decorrer deste trabalho envolve o cálculo do limite máximo de latência da rede baseado nos relacionamentos de interferência causados pela preemptividade e pelas prioridades de cada um dos fluxos envolvidos na comunicação.

Nas seções seguintes serão apresentados tais relacionamentos e o impacto que elas causam a escalonabilidade de um ambiente em chip. A seguir algumas abordagens serão comparadas com o escalonamento de prioridade fixa e suas principais diferenças serão apresentadas.

3.2 ABORDAGENS DE ESCALONAMENTO

O esquema de escalonamento em redes em chip é determinado por dois principais elementos, o módulo para escalonamento e uso dos recursos da rede, e o módulo de análise e previsibilidade do comportamento do sistema para aplicar o escalonamento aos fluxos.

O escalonamento pode ser aplicado em nível de sistema operacional ou tempo de execução, e também pode surgir no nível de hardware ou arquitetura de comunicação, como as redes em chip. Em conjunção, a análise de previsibilidade verifica os requisitos temporais de um conjunto de fluxos para serviços de tempo real.

As estratégias de escalonamento podem ser divididas como fixas e dinâmicas. A diferença entre esses dois tipos é a variação de prioridades que podem ou não ocorrer em tempo de execução. As abordagens de escalonamento fixas e dinâmicas frequentemente implementadas são o escalonamento de prioridades fixas, escalonamento EDF (*Earliest Deadline First*) e o escalonamento *Least Laxity* (LL). Os algoritmos EDF e LL são algoritmos de escalonamento dinâmico. Os algoritmos de escalonamento dinâmico são aqueles que a prioridade das tarefas muda conforme o tempo de execução.

Os algoritmos baseados na LL alocam a prioridade máxima para aquelas tarefas de menor laxidez (Davis & Kato, 2012; Mesidis, 2011; Sha et al., 2004). A laxidez de uma tarefa é definida como a diferença do espaço de tempo do prazo e o tempo de execução restante indicando a flexibilidade para o escalonamento da tarefa correspondente.

Contudo a troca de contexto em demasia quando duas ou mais tarefas possuem a mesma laxidez torna inadequada a implementação desse escalonamento para serviços de tempo real. Essa situação é conhecida com laço de laxidez e reduz a previsibilidade da rede. O escalonamento LL é conhecido por ser totalmente dinâmico.

No algoritmo EDF as prioridades são definidas por aquela tarefa, cujo deadline está mais próximo de expirar. O conjunto de tarefas é escalonado de acordo com a finalização de seus deadlines.

Ambas as possibilidades de escalonamento tornam-se de difícil implementação quando o assunto é serviço de tempo real, justamente pela imprecisão e dinamicidade que as prioridades alcançam. Na seção a seguir é apresentado o escalonamento de priodades fixas.

3.2.1 Escalonamento de Prioridades Fixas

Para alcançar a previsibilidade na comunicação em redes em chip é necessário um escalonamento de comportamento preciso e imutável, essas características são encontradas na abordagem baseada em prioridades fixas. Nesse escalonamento as prioridades das tarefas não mudam do começo ao fim da execução.

Para as redes em chip que implementam o escalonamento por prioridades fixas com chaveamento *wormhole* os pacotes ao serem divididos em uma quantidade *i* de *flits* cada um destes herda a prioridade atribuída ao pacote como um todo. Uma forma de interpretar as prioridades é considerar o valor 1 como o maior valor de prioridade e qualquer valor inteiro acima prioridades menores.

Fatores como esses são determinantes para os cálculos de análise de escalonabilidade dos fluxos na rede. Considerando a escalonabilidade, a análise de cenários de pior caso no cálculo da latência é o foco da abordagem para encontrar os atrasos na comunicação baseado nos relacionamentos entre os fluxos.

Essa análise de escalonabilidade para escalonamento de prioridades fixas avalia os relacionamentos entre os fluxos buscando prever todos os possíveis atrasos de comunicação conhecendo a ordenação das prioridades e determinando se são escalonáveis ou não.

Guan (Guan et al., 2010) considera o limite de utilização do WCET (*Worst Case Execution Time*) para cada tarefa caracterizada por seu período e prazo. A ordem é atribuída de forma não-crescente e o índice representa as prioridades da tarefa. Para o escalonamento das tarefas é aplicado o Algoritmo RM (*Rate Monotonic*) em cada processador.

No entanto, o teste de RM não é suficiente para garantir a entrega dos pacotes de dados no chaveamento *wormhole*, uma vez que um problema surge no espalhamento dos *flits* dos pacotes pela rede. O chaveamento *wormhole* baseado em prioridades estudado por Mutka (Mutka, 1998; Leung & Zhao, 2005) foi o primeiro a abordar a análise de escalonabilidade utilizando o algoritmo RM.

Em (Dorin et al., 2010) é proposto um escalonamento semi-particionado em tempo real *hard* (SPHRTS) e um paradigma baseado nesse conceito de escalonamento semi-particionado com migrações restritas. A estratégia proposta é periódica e demora um período de dois jobs de tarefas subsequentes para que possam ser atribuídas tarefas a diferentes processadores. Essas tarefas são escalonadas entre os processadores usando o algortimo *First Fit Decreasing* (FFD).

Nesse modelo as tarefas caracterizam cada uma por três parâmetros: WCET, o prazo relativo e o tempo mínimo de chegada. As tarefas são escalonadas por um escalonador EDF com migrações

restritas e presume-se que todas as tarefas sejam independentes. O escalonamento EDF possui um arbitro que define as prioridades dos pacotes individuais de forma dinâmica.

Pelo escalonamento EDF ser dinâmico e não ser adequado para sistemas de tempo real *hard*, Shi (Shi & Burns, 2008, 2009; Shi, Burns, & Indrusiak, 2010) usa da política de prioridades fixas para tornar o comportamento da rede previsível e a passagem dos pacotes o mais determinístico possível. Além disso, a abordagem proposta por Shi (Shi & Burns, 2008, 2009; Shi, Burns, & Indrusiak, 2010) torna possível à análise escalonabildade quando o prazo é maior do que o período.

Os trabalhos de Leung e Zhao, e Zhou (Leung & Zhao, 2005; Zhou, Qiao, Lin, 2011), da mesma maneira empregam uma estratégia de atribuição de prioridades dinâmicas que dificilmente é aplicável para tempo real *hard*. Nesses trabalhos um algoritmo de escalonamento multi-núcleo em tempo real que extende o escalonamento Pfair de tarefas globais (E-Pfair) utiliza de parâmetros híbridos. Os parâmetros são encontrados nas tarefas correspondem ao tempo de liberação, o tempo de execução, a prioridade, o período e o deadline.

O período e o tempo de execução permitem calcular o peso da tarefa por conta do algoritmo de escalonamento que utiliza o deadline e o peso destas tarefas para definir as prioridades. Nesta abordagem, a comunicação do núcleo é reduzida, pois as tarefas adjacentes e as tarefas altamente relacionadas são mantidas no mesmo núcleo, atingindo equilíbrio de carga de trabalho do sistema com uma política de duplicação de tarefas.

3.3 MODELO DE SISTEMA, RESTRIÇÕES E MÉTRICAS DE ESCALONABILIDADE

O modelo teórico de sistema diz respeito às características e as suposições feitas acerca da plataforma a se analisar. O modelo de sistema trabalhado é baseado nas suposições em (Shi, Burns, & Indrusiak, 2010) representado por uma rede de tempo real com chaveamento *wormhole* (Γ). Essa rede por sua vez compreende *n* fluxos de tarefas de tempo real, $\Gamma = \{\tau_1, \tau_2, \tau_3, ..., \tau_N\}$. Cada um dos fluxos de tráfego é caracterizado pelos atributos $\tau_i = (P_i, C_i, T_i, D_i, J^R_i, J^I_i)$.

Considerando que todos os fluxos que requerem entrega temporal são periódicos ou esporádicos, *T* é considerado o menor intervalo de limite de tempo entre os lançamentos de pacotes sucessivos. Esse intervalo de lançamento dos pacotes também é conhecido como período.

Sobre a perspectiva dos serviços de tempo real e a implementação de prioridades fixas, cada fluxo de tarefas tem um valor de prioridade *P*. Esses valores são distintos e todos os respectivos pacotes pertencentes ao fluxo herdam a mesma prioridade. É considerado como maior valor de prioridade o número 1 e os números inteiros sucessivos maiores denotam as menores prioridades.

Os fluxos de tarefas que são de tempo real têm agregados aos seus pacotes o prazo *D* para haver o controle da validade temporal, o que significa que todos os pacotes que pertencem ao fluxo

têm a restrição de ser entregue dentro de certo limite de tempo. Sendo esse prazo decisivo durante o escalonamento, no cenário de pior caso, o prazo de cada um dos fluxos têm de ser menor ou igual ao seu período, $D_i \leq T_i$, $\tau_i \in \Gamma$. Contudo, nas seções a seguir é visto que essas restrições podem ser ultrapassadas sem repercussões negativas no desempenho ou na validade dos dados.

Neste modelo de sistema o desvio sucessivo máximo dos pacotes liberados durante o período é denominado *jitter* (J^R_I). O jitter diz respeito ao atraso adicional na geração dos pacotes, que no caso de o pacote τ_i ser gerado em um dado momento então ele está apto à transmissão no tempo $J^R_I + a$. Essa medida também influencia no prazo, denominando-o de prazo absoluto e é tido como $a + D_i$. Pela existência do *jitter* a restrição entre o prazo e o período é atualizada para $D_i \leq T_i - J^R_i$.

Em uma situação de interferência onde haja lançamento de dois pacotes sucessivos ocorre o fenômeno de jitter de interferência. O Jitter de interferência (J^I_i) é melhor definido como o desvio máximo de tempo no início de transmissão entre dois pacotes sucessivos.

Esses parâmetros são determinados para descobrir a latência do tempo de resposta da rede. A latência básica da rede é uma medida base calculada na situação onde não há qualquer interferência de outros fluxos de maior prioridade sobre um de menor na rede. Esta medida é determinada pela distância de roteamento origem/destino, pelo tamanho do pacote e pela largura de banda para fins de avaliação. A fórmula de latência básica da rede então é dada pela equação (1):

$$C_i = \frac{L_i^{max}}{f_{size}} \cdot \frac{f_{size}}{B_{link}} + H \cdot S \qquad (1)$$

De acordo com (Liu et al., 2011), a latência básica máxima da rede (C_i) são dados pelo tamanho dos pacotes, pelo atraso no roteamento e pela quantidade de saltos na rede. O primeiro termo da equação determina o tempo de transmissão do pipeline de pacotes com base no tamanho do pacote e a largura de banda da rede. No primeiro termo temos as medidas do tamanho do pacote e do tamanho do *flit*, onde L^{max}_i e o f_{size} pertencem a τ_i e B_{link} é a largura de banda do *link*. No segundo termo o atraso de roteamento do *flit* cabeçalho é contabilizado. As medidas são representadas por H que indica a quantidade de saltos relacionados com o caminho da origem/destino. A constante de tempo de atraso de processamento de cada roteador é determinada pelo S.

Na seção a seguir é apresentado o interrelacionamento dos fluxos, caracterizando-os pela forma como a preempção direta ou indireta influencia no tempo de resposta da rede. Esses relacionamentos são utilizados no cálculo da latência de pior caso, que também é apresentada nas seções seguintes.

3.3.1 Interrelacionamentos entre Fluxos de Tarefas

Para a análise dos relacionamentos dos fluxos é necessário formalizar algumas características da rede. Um dos elementos a se formalizar é a topologia da rede, a qual para os fins deste trabalho está sob a forma de grelha. A topologia está representada por um grafo *G:VxE*, com *V* sendo um conjunto de elementos chamados nós e *E* sendo um conjunto de elementos de ligação chamados arestas. Dado um conjunto de *n* fluxos de tarefas que podem ser mapeados para a rede alvo, $\Re_n$ representa o roteamento de cada uma delas. Se o fluxo τ_i compartilha ao menos uma ligação com o fluxo τ_j o conjunto de intersecção é calculado pela interssecção, $\Re_j \cap \Re_i$.

- **Relacionamento Direto entre os Fluxos de Tarefas**

Durante a transmissão dos pacotes a interferência de contenção pode ser classificada como uma relação de competição direta pelo recurso da rede. Definindo este conjunto de interferência como $S_i^D = \{\tau_i \mid \Re_i \cap \Re_j \neq \varphi$ e $P_j > P_i \ \forall \tau_i \in \Gamma\}$ a latência da rede máxima do fluxo observado τ_i pode ser encontrado pela equação (2):

$$R_i = \quad {}_{\forall \tau_j \in S_i^D} \ \frac{R_i + J_j^R}{T_j} * C_j + C_i \qquad (2)$$

- **Relacionamento Indireto entre os Fluxos de Tarefas**

Quanto há interferência indireta durante a transmissão de pacotes, o conjunto é definido como $S^I_i = \{\tau_k \mid \tau_k$ tem uma relação indireta competindo com τ_i e $P_k > P_j > P_i \ \forall \tau_i \in \Gamma$, onde τ_j é qualquer fluxo intermediário$\}$. Neste caso, a latência máxima da rede do fluxo observado τ_i pode ser encontrado por (3):

$$R_i = \quad {}_{\forall \tau_j \in S_i^D} \ \frac{R_i + J_j^R + J_i^I}{T_j} * C_j + C_i \qquad (3)$$

- **Relacionamento de Auto-Bloqueio entre os Fluxos de Tarefas**

Quando a latência da rede é maior do que o período, o auto-bloqueio pode ocorrer, isto é, quando os *flits* finais do pacote anterior estão sendo enviados e os *flits* iníciais do próximo pacote já

foram introduzidos na rede. O conceito de período de ocupação é aplicado. O período de ocupação corresponde à duração de tempo contíguo que o link de transmissão correspondente mantém servindo todos os pacotes de fila com a prioridade *i* ou superior.

Com a implementação de tal técnica a restrição de $D_i \leq T_i$, $\tau_i \in \Gamma$ é flexibilizada para $D_i \geq T_i$, $\tau_i \in \Gamma$, onde o prazo da tarefa em questão pode ser maior que o período de lançamento de pacotes. A equação (4) apresenta a fórmula para o cálculo do período de ocupação:

$$B_i = \quad {}_{\forall \tau_j \in S_i^D} \frac{R_i + R_j - C_j + J_j^R}{T_j} * C_j + \frac{B_i + J_i^R}{T_j} * C_i \qquad (4)$$

Além disso, de acordo com Shi (Shi, 2009), a latência máxima da rede para t_i é dada pela equação (5):

$$R_i = max(w_i(q) - (q-1)T_i + J_i^R) \qquad (5)$$

Em que q é o índice de ocorrência do pacote, e w_i (q) é a janela de tempo para transmissão do pacote dada pela equação:

$$w_i \ q \ = qC_i + \quad {}_{\forall \tau_j \in S_i^D} \frac{w_i \ q \ + J_j^R + J_j^I}{T_i} * C_i \quad (6)$$

3.4 IMPLEMENTAÇÃO COMPUTACIONAL

Para avaliar a eficácia da técnica de análise de escalonabilidade proposta por Shi (2009) foi desenvolvida em Java uma aplicação que calcula e analisa estaticamente dados experimentais. Para a modelagem das tarefas em forma de grafo e melhor representação das mesmas foi utilizado uma biblioteca específica denominado Jung. As subseções a seguir apresentam as questões de implementação.

3.4.1 Modelagem em Forma de Grafo

Segundo as proposições de Shi (2009) as tarefas de uma aplicação são modeladas em forma de grafo. Para essa modelagem cada uma dessas tarefas inclui parâmetros como a prioridade, a latência básica, o período, o prazo, o jitter de liberação, básicas para o modelo de tarefas e também os parâmetros do número de vértices e as arestas que formam o grafo. Cada uma das tarefas modeladas

em forma de grafo é definida como um conjunto de roteadores por onde passa o fluxo de dados, representado pelos vértices, e um conjunto de *links* que molda o caminho da tarefa, representado pelas arestas. O conjunto de vértices e arestas representam os recursos da rede em chip que são compartilhadas pelas tarefas e esse compartilhamento, dependendo de como ocorra, define os relacionamentos das tarefas como direto, indireto e de autobloqueio. O conjunto de vértices é representado por (v_1, v_2, v_3, ..., v_n) e o conjunto de arestas são pares ordenados de vértices (e_{12}, e_{23}, e_{34}, ..., $e_{n\text{-}1n}$).

3.4.2 Aplicação para Análise de Escalonabilidade

A aplicação encontra-se dividida em seis classes estas são, respectivamente, BBSA, CompareTask, PriorityAssigment, SchedulabilityTest, Task e TaskGraph. Primeiramente a classe que têm o método principal é a TaskGraph. Essa classe é a responsável por inicializar o processo de análise construindo cada uma das tarefas e indicando o caminho que cada uma percorre pela definição dos vértices e das arestas. Essa construção das tarefas é definida pela classe Task. Quando a lista das tarefas está montada e adequadamente configurada dá-se início ao processo de análise de escalonabilidade.

A classe que executa atribuição de prioridades das tarefas é a BBSA, onde está implementado o algoritmo de *Branch and Bound* (B&B). Na execução deste algoritmo são testados os limites superiores de latência das tarefas a fim de atribuir inicialmente prioridades àqueles que satisfazem a condição de o limite superior de latência ser menor ou igual ao prazo.

Quando esta condição não é satisfeita é realizado o cálculo do limite inferior das tarefas que não foram atribuídas a qualquer prioridade. A partir deste ponto o algoritmo B&B usufrui de funções heurísticas para a escolha do melhor candidato dentre as tarefas de uma lista de candidatos para atribuir a tarefa ao seu devido nível de prioridade.

A final desse processo, onde é alcançada uma ordenação ótima para uma lista de tarefas baseado no algoritmo B&B guiado por uma função heurística, é ativado o método de análise de escalonabilidade para aquele conjunto de tarefas encontrado na classe *SchedulabilityTest*. Durante essa análise são testadas situações onde o cenário de pior caso de latência da rede é menor ou igual aos prazos das tarefas, para as situações de interferência direta e indireta. Além de analisar os casos de interferência de autobloqueio quando o prazo é menor que o cenário de pior caso de latência da rede para aquele conjunto de tarefas em uma determinada ordenação.

As outras classes correspondem a classes de suporte para as operações principais, a classe *CompareTask auxilia* o encontro das intersecções das tareafas, enquanto que *PriorityAssignment* nos cálculos das latências supeior e inferior.

3.5 EXEMPLOS NUMÉRICOS E RESULTADOS

Os resultados obtidos foram alcançados pelo teste de dados experimentais com a aplicação da análise de escalonabilidade a fim de testar a exatidão da implementação (Vidal, 2013). As características do modelo de rede são baseados na rede em chip HERMES (Moraes, F, Calazans, N., & Mello, A. 2004). A rede Hermes possui as características e as implementações dos mecanismos comparáveis ao modelo hipotético do sistema. Tais como o chaveamento wormhole, o escalonamento por prioridades fixas, roteamento XY, baixa latência e pode multiplexar o canal físico em mais de um canal virtual.

A Figura 1 mostra a forma como os roteadores estão dispostos e interligados para este exemplo. Como na rede em chip HERMES (Moraes, F, Calazans, N., & Mello, A. 2004) os roteadores são numerados a partir da origem e o conjunto de fluxos na rede $\Gamma = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$ é distribuído entre os roteadores.

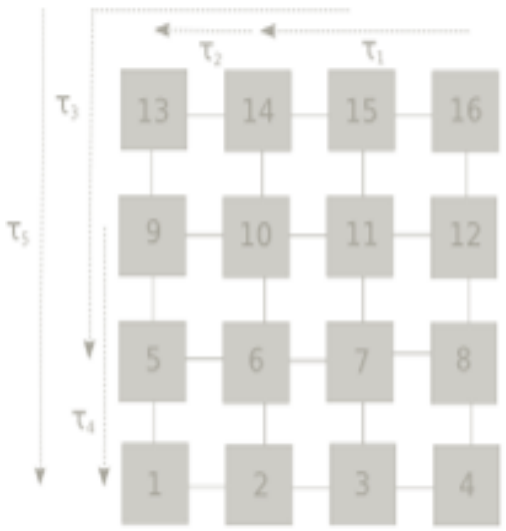


Figura-4. Rede em Chip e Fluxos de Tarefas.

Para obtenção das interferências da tarefa a estratégia utilizada é a modelagem dos fluxos como grafo e encontrar as interseções nos *links* e roteadores compartilhados por eles. No caso de a interseção nula, $\Re_i \cap \Re_j = 0$, os fluxos são considerados disjuntos e não há interferência entre eles. Este método é fundamental para descobrir as interrelações entre as tarefas, supondo que na comparação de dois fluxos exista interferências. O roteamento das tarefas com base na Figura-4 pode ser representado da seguinte maneira:

$$R_1 = \{e_{1615}, e_{1514}\}, R2 = \{e_{1413}\},$$

$$R3 = \{e_{1514}, e_{1413}, e_{139}, e_{95}\}, R4 = \{e_{95}, e_{51}\}, R5 = \{e_{139}, e_{95}, e_{51}\}$$

Seguindo as premissas sobre interrelações e sobre o conhecinhemento das interferências é fácil descobrir as interferências diretas do conjunto de fluxos de maior prioridade (S^D_i) e as interferências indiretas do conjunto de fluxos de maior prioridade (S^I_i), as quais são:

- τ_1 e τ_2 têm as mais altas prioridades e eles são disjuntos:

$$S^D_1 = S^I_1 = S^D_2 = S^I_2 = 0$$

- O fluxo τ_3 compartilha dos recursos de *link* com τ_1 e τ_2 a qual a interssecção permanece igual à:

$$S^D_3 = \{\tau_1, \tau_2\}, \text{ e } S^I_3 = 0$$

- O fluxo τ_4 diretamente contende com τ_3 e indiretamente concorre com τ_1 e τ_2:

$$S^D_4 = \{\tau_3\}, \text{ and } S^I_4 = \{\tau_1, \tau_2\}$$

- O τ_5 fluem diretamente contende com o τ_3 e τ_4 indiretamente concorre com τ_1 e τ_2:

$$S^D_5 = \{\tau_3, \tau_4\}, \text{ e } S^I_5 = \{\tau_1, \tau_2\}$$

A Tabela 1 abaixo descreve todos os fluxos de tarefas e as características consideradas para avaliação da análise de escalonabilidade (Kim, Kim, Hong, & Lee, 1998), com base na latência do pior caso da rede.

Tabela 1: Fluxos de Tempo Real e suas Características (Shi, 2009).

Real-Time Traffic-flows	***P***	***C***	***T***	***D***	J^R	J^I
τ_1	1	1	5	5	0	0
τ_2	2	2	7	7	0	0
τ_3	3	2	9	9	0	3
τ_4	4	4	12	12	0	2
τ_5	5	3	8	12	0	0

O pior caso de análise de latência da rede para τ_1 e τ_2 é igual a sua latência da rede básica pois eles têm as prioridades mais elevadas não existindo interferência entre eles.

$$R_1 = 1 \text{ e } R_2 = 2$$

No τ_3 há interferência direta dos fluxos de tarefa de maiores prioridades τ_1 e τ_2 de acordo com a Figura 4. Por conseguinte, a latência de pior caso da rede atribuído a este fluxo é:

$$R_3 = 5$$

O fluxo τ_5 sofre interferência direta do τ_4 e interferência indireta do τ_3. Sendo que o fluxo τ_4 sofre interferência direta do τ_3 e interferências indiretas de τ_1 e τ_2. Dessa forma suas latências de pior caso da rede são:

$$R_4 = 6 \text{ and } R_5 = 12$$

É observado que o cenário de pior caso para todos os fluxos é encontrado o prazo satisfatóriamente sendo este menor ou igual ao prazo, R ≤ D. Inclusive para a tarefa de τ_5 cujo prazo *D* é maior que o período *T* de lançamento de pacotes. Por meio da análise de escalonabilidade é satisfeito a restrição temporal do τ_5 caracterizando-o como interferência de autobloqueio.

Por meio da implementação do método analítico para previsão do cenário de pior caso de latência de rede é possível alcançar bons resultados, encontrando as interferências diretas, indiretas e de autobloqueio dos fluxos de tarefas. A automação do processo é possível por meio da modelagem em forma de grafo e da análise de interferência.

Esta análise é fundamental para a política de atribuição de prioridade, e pode ser vista como uma verificação analítica para a ordenação de prioridades, otimização do desempenho e compartilhamento de recursos a fim de atingir os prazos de comunicação na rede em chip com serviços de tempo real *hard*. As demais abordagens consideram o equilíbrio de carga de trabalho entre os núcleos ou o tempo de execução no pior caso das tarefas.

4 ESTUDO DE CASO

Neste capítulo são apresentadas diferentes aplicações na forma de estudo de caso para avaliação do desempenho da análise de escalonabilidade mostrada na seção anterior comparando os resultados obtidos com os valores reais das aplicações dentro de uma plataforma de rede em chip. Em cada teste são utilizadas variações de organização da rede e diferentes tarefas.

Seguindo o foco da pesquisa em serviços de tempo real *hard* as aplicações são descritas destacando a necessidade de garantias de tempo de resposta às comunicações na rede. Dessa forma é aplicada a análise de escalonabilidade para o cálculo da latência na rede para os cenários de pior caso das aplicações.

4.1 ATRIBUIÇÃO DE PRIORIDADES

Além de avaliar as interferências dos fluxos, a análise de escalonabilidade também pode ser aplicada para otimizar a atribuição de prioridades. Por isso além da avaliação de latência da rede, o teste de escalonabilidade pode ser utilizado em relação a sua atribuição e ordenação das prioridades para otimização do escalonamento dos fluxos de tarefas. É utilizado um algoritmo de busca B&B (*Branch and Bound)* guiado por uma função heurística.

A função do algoritmo de busca é alocar um nível de prioridade a partir da quantidade de interferências enfrentadas por um determinado fluxo. A função heurística auxilia na seleção do candidato mais apto para o nível de prioridade testado baseado na suposição de tolerância a interferência adicional. Esta tolerâcia representa quanta interferência um fluxo em certo nível de prioridade pode tolerar sem a perda do prazo. Dessa forma é dada a chance para aqueles fluxos que são menos tolerantes a conseguir maiores prioridades. A função heurística utilizada é a diferença entre o prazo D_i e a o limite inferior de latência de pior caso R'_i para o fluxo τ_i.

$$H_i = D_i - R'_i \qquad (6)$$

Nas seções seguintes são apresentados os experimentos para o estudo de caso. Este está dividido em três experimentos para as seguintes organizações: rede em chip em grelha 2x3 para o experimento 1; rede em chip em grelha 3x3 para o experimento 2; e rede em chip em grelha 4x4 para o experimento 3.

4.2 EXPERIMENTO 1

O primeiro modelo de aplicações utilizadas no experimento é baseado em (Ramaprasad & Mueller, 2006) seguindo fundamentalmente algumas suposições impostas por ele. Como por exemplo, supor que as aplicações naturalmente fazem acessos às regiões de memória.

Adicionalmente, algumas características são delimitadas para manter a previsibilidade. Primeiramente, os limites de laço da aplicação precisam ser conhecidos em tempo de compilação. Segundo, expressões subscritas de arrays devem ser funções das variáveis de indução de loop. E terceiro não deve haver acessos de memória dinâmicos ou baseados em ponteiros. Seguindo essas recomendações, na tabela 2 é possível ver as aplicações referenciais as quais serão submetidas à avaliação de escalonabilidade.

Na tabela 2 as aplicações respectivamente são, um benchmark para implementação de um filtro de convolução, outro para implementação de um filtro de resposta a um impulso finito, uma aplicação para implementar um filtro da média de um quadrado, outro para executar *n* atualizações em tempo real de uma fórmula e um aplicativo para encontrar o produto de duas matrizes.

Tabela 2: Descrição dos Benchmark. (Baseada em Ramaprasad & Mueller, 2006).

Aplicações	**Descrição**
Convolution	Programa para implementar filtro de convolução.
Fir	Programa para implementar um filtro de resposta ao impulso finito.
lms	Programa para implementar um filtro de média-quadrado.
n-real-update	Programa para executar *n* atualizações reais da formula D (i) = C (i) + A (i) * B (i), onde A (i), B (i), C (i) e D (i) são números reais, e i = 1, ..., n
Matrix 1	Programa para encontrar o produto de duas matrizes

Para o primeiro experimento é utilizado uma plataforma *SoC* (*System-on-Chip*) com 5 elementos processantes e uma memória secundária de dados com o seguinte conjunto das 5 tarefas, convolution, Fir, Lms, n-real-update e Matrix1. Para cada tarefa é gerado um fluxo de tráfego. Dessa forma as tarefas são distribuídas entre os núcleos e tentam acessar a memória desse sistema.

Na Figura 5 está representado o sistema em questão com as devidas tarefas executando nos núcleos. É possível ver o sentido que os dados seguem baseados na busca por dados na memória fazendo seu caminho pela rede.

A configuração da rede é do tipo determinístico para fortalecer a previsibilidade para a análise de escalonabilidade. Como a configuração da rede segue o modelo da rede HERMES esta possui roteamento deterministico XY, o chaveamento em pacotes usando a técnica *wormhole* e tem canais virtuais pra controle de fluxo. O controle da passagem dos pacotes é feita por crédito implementado juntamente no mecanismo memorização nas portas de entrada do roteador.

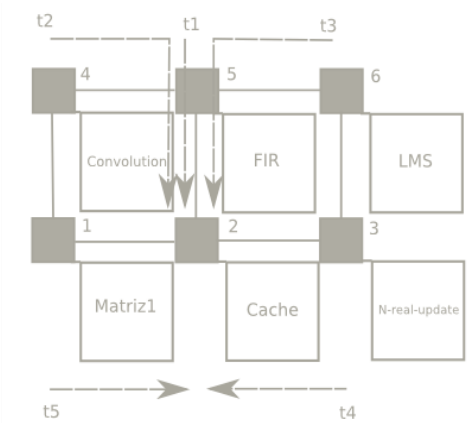


Figura-5. Mapeamento do Grupo de Tarefas do experimento 1 em processadores interconectados por uma rede em chip em grelha 2x3.

Considerando essas características da rede, segundo (Ramaprasad & Mueller, 2006) os acessos à memória causam bloqueio em tarefas de baixa prioridade quando uma tarefa de maior prioridade busca por dados na memória. Essa ocorrência causa o efeito de atraso denominado de pontos de preempção. Os pontos de preempção são as interrupções que uma tarefa sofre por conta da busca de dados na memória.

Tendo como representação do sistema a Figura 5, no momento da liberação os fluxos t_4, t_5 que estão em núcleos adjacentes à memória, estes não sofrem de interferência quanto à comunicação na rede o que faz com que suas latências no cenário de pior caso de comunicação sejam iguais às latências básicas da rede, respectivamente. Apesar disso ambas as tarefas sofrem por bloqueio de preempção em relação ao acesso na cache. O fluxo t_1 por ser o fluxo de maior prioridade encaixa-se em uma situação, onde os flits que herdam a mesma prioridade do pacote atravessam a rede sem encontrar preempções. Sendo assim a tarefa t_1 comunica-se com a cache sofrendo apenas de sua latência básica da rede no pior caso da rede.

Analisando o fluxo t_2, no momento de sua liberação este irá sofrer preempções sempre que o fluxo t_1 estiver ativo, por conta da interferência direta que existe entre os dois por haver links em comum entre os dois. Porém, mesmo com a existência de preempção, o tempo de resposta da rede

em relação ao fluxo t_2 permanece escalonável como é mostrado na tabela 3, onde o valor de latência de pior caso da rede é inferior ao prazo da tarefa.

Finalmente o fluxo t_3 em comparação com os outros fluxos é o que mais sofre preempção, proveniente das interferências dos fluxos t_1 e t_2. No momento de sua liberação é possível que os outros fluxos estejam utilizando o link físico recebendo interferência direta de t_1 e interferência indireta de t_2 no cenário de pior cado. Interpretando como um resultado positivo, sua latência de pior caso da rede, mesmo havendo interferências, direta e indireta, este continua escalonável como está representado na tabela 3, onde a latência é inferior ao prazo da tarefa.

A tabela 3 apresenta a organização das tarefas segundo (Ramaprasad & Mueller, 2006) e suas características. Estas tarefas seguem o modelo estabelecido por Shi (Shi & Burns, 2008; Shi, Burns, & Indrusiak, 2010) onde mostra a prioridade (P), o deadline (*D*), a latência da rede sem contenções (*C*), ou latência básica, e o cenário de pior caso de latência da rede (*R*).

A tabela mostra o tipo de aplicação, sua respectiva prioridade, período e prazo ou *deadline*. As prioridades para estas tarefas são dadas seguindo o esquema mais comum de atribuição denominado *Rate Monotonic*.

Tabela 3: Caracterísitcas e Latências de resposta do primeiro grupo de tarefas obtidas usando a análise de escalonabilidade (Baseada em Ramaprasad & Mueller, 2006).

Aplicação	Prioridade (P)	Periodo (*T*)	Prazo (Deadline) (D)	Latência Básica da Rede (C_i)	Latência da Rede de pior caso (R)
τ_1(convolution)	1	62500	62500	7491	7491
τ_2(fir)	2	125000	125000	9537	17028
τ_3(lms)	3	125000	125000	14536	31564
τ_4(lms)	4	250000	250000	16738	16738
τ_5(Matrix 1)	5	250000	250000	54168	54168

Seguindo o modelo de rede e tarefas de Shi (Shi & Burns, 2008; Shi, Burns, & Indrusiak, 2010), as tarefas possuem os seguintes parâmetros $t_n = \{P, C, T, D, J^R\}$. Considerando o caso para este conjunto de tarefas com interferência direta e interferência indireta, os valores no campo de latência de rede de pior caso é o resultado dos cálculos de latência e interferência do comportamento das tarefas na rede, sem considerar o atraso de bloqueio da memória.

O resultado da análise de escalonabilidade para o primeiro conjunto de aplicações mostrou-se escalonável nos níveis de prioridades definidos. Essa satisfatoriedade é notada pelos cenários de

pior caso das tarefas estarem menor que o valor dos prazos estipulados para cada tarefa como é visto na tabela 3.

4.3 EXPERIMENTO 2

No segundo conjunto de Benchmark a ser testado houve a preocupação de envolver mais casos com interferência indireta para tornar claro a funcionalidade da ténica de escalonabilidade e ter a certeza de incluir situações possíveis ainda não cobertas no grupo de aplicações anteriores na análise de escalonabilidade. Neste segundo conjunto de tarefas houve o acréscimo de uma tarefa, que é apresentada na tabela 4. A tarefa extra representa a operação para encontrar o produto dos pontos de dois vetores.

Tabela 4: Descrição do Benchmark acrescido no segundo conjunto de aplicações. (Baseada em Ramaprasad & Mueller, 2006).

Benchmark	Descrição
dot-product	Programa para encontrar o produto de pontos de dois vetores.

Com o acréscimo desta nova tarefa a estruturação é modificada de maneira significativa, como está representado na Figura 6. Por meio da figura é possível notar a mudança de topologia e o acréscimo de outras funcionalidades. As principais mudanças encontram-se nos roteadores 1 e 4. Existem blocos de memórias onde as tarefas fazem acessos em busca de dados para suas operações.

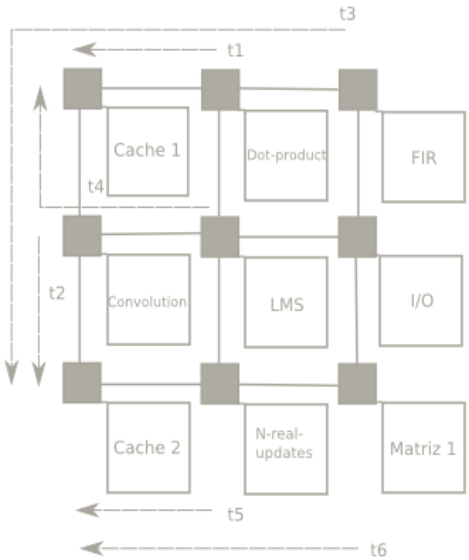


Figura-6. Mapeamento do Segundo Grupo de Tarefas de Aplicações em processadores interconectados em uma rede em chip Mesh 3x3.

Também no roteador 6 o núcleo é responsável por operações de entrada e de saída da rede. O roteador 2 está conectado ao núcleo que realiza a operação “n-real-updates”, o roteador 3 executa operações sobre matrizes, o núcleo ligado ao roteador 4 executa operações sobre um filtro de convolução e o núcleo no roteador 2 executa o filtro de média-quadrado. Os roteadores 8 e 9 se responsabilizam por comunicar as operações *dot-product* e FIR.

Segundo a Figura 6 é possível analisar a situação de escalonabilidade dos fluxos na rede no momento de liberação dos fluxos. Os fluxos t_1, t_2, t_4 e t_5 no momento da liberação não sofrem de preempções, baseado no caminho percorrido pelos mesmos, configurando assim suas latências de pior caso como iguais as suas respectivas latências básicas na rede. Contudo o fluxo t3 no momento de sua liberação pode ocorrer preemptividade causada pelos fluxos t1, t2 e t4 no seu cenário de pior caso. É visto na tabela 5 que a latência causada por essas preempções não é superior ao prazo estabelecido para realização dessa tarefa.

A tabela 5 determina a mudança nas características das tarefas quando a nova operação assume a prioridade mais alta. As latências de pior caso das tarefas t_1, t_2 e t_5 são iguais às suas respectivas latências básicas, pois o salto na rede em busca de dados nos módulos de memória é mínino não acontecendo interferências com outros fluxos de menores prioridades.

Tabela 5: Caracterísitcas e tempos de resposta do segundo grupo de tarefas. (Baseada em Ramaprasad & Mueller, 2006).

Aplicação	Prioridade (P)	Periodo (*T*)	Prazo (D)	Latência da Rede sem contenção (C_i)	Latência da Rede de pior caso (R)
τ_1(Dot-product)	1	50000	50000	750	750
τ_2(convolution)	2	62500	62500	7491	7491
τ_3(Fir)	3	125000	125000	9537	17778
τ_4(lms)	4	125000	125000	9537	14536
τ_5(n-real-update)	5	250000	250000	16738	16737
τ_6(Matrix 1)	6	250000	250000	54168	70905

É possível visualizar na figura 6 a comunicação da rede para esse segundo conjunto de aplicações testadas.

Comparando os resultados encontrados nos testes baseados em (Ramaprasad & Mueller, 2006) os resultados de tempo de resposta e de latência da rede não se diferem de forma radical, com exceção daquelas tarefas que não sofrem por interferências. Os resultados de latência de pior caso

pernamencem dentro da margem do prazo das tarefas como é apresentado na tabela 5. Conclui-se que conforme é incluído um novo fluxo de tarefa no grupo de testes a escalonabilidade é mantida pela previsão da análise.

4.4 EXPERIMENTO 3

O terceiro benchmark testado no experimento 3 está relacionado a um modelo de veículo autônomo sendo compreendido por um sistema de controle e de navegação do veículo. A aplicação também inclui tarefas para controle de estabilidade e direção do veículo hipotético (Indrusiak et al., 2008; Maatta et al., 2008; Shi, 2009).

A figura 7 apresenta a forma como as tarefas da aplicação estão distribuídas dentro da plataforma. A rede em chip está em uma topologia em grelha 4x4 executando com 16 roteadores para executar as 38 tarefas distintas. As funções das tarefas que compõem a aplicação são descritas em detalhe na tabela 6.

A plataforma que dá suporte à aplicação é composta por um par de câmeras, para reunir informações sobre os obstáculos desconhecidos. Essas duas câmeras captam informações de obstáculos no espaço desconhecido, durante funcionamento estas alimentam o sistema como 25 quadros por segundo em uma resolução de 640x480 com 8 bits de representação para a cor, particionadas em quatro quadrantes. E para cada quadrante é feito o processamento de estimação de fundo e extração de características para reuní-las novamente formando o objeto de interesse.

A distância dos obstáculos é analisada pela rotina de fotogrametria. Após a análise as informações sobre os obstáculos são adicionadas ao banco de dados por meio dos sensores ultrasônicos e pelos cálculos obtidos com a fotogrametria. A reunião destas informações é utilizada pelo veículo no processo de navegação autônoma.

A outra parte do sistema resopnsável pela estabilidade busca parâmetros de vibração e pressão do pneu para ajustar a velocidade de acordo com os dados dos respectivos sensores para o veículo melhor se adaptar a superfície na qual está se movendo.

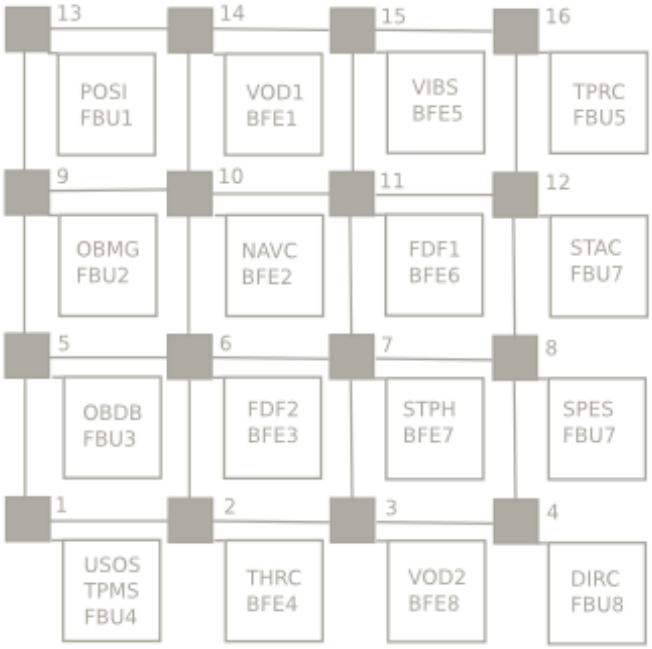


Figura-7. Mapeamento do Terceiro Grupo de Tarefas de Aplicações em processadores interconectados em uma rede em chip Mesh 4x4.

Tabela 6: Descrição do Terceiro Grupo de Tarefas da Aplicação do Veículo autônomo. (Baseado em Shi, 2009)

Tarefa	Descrição
TPMS	Sistema de monitoramento de pressão do pneu
VIBS	Sensor de vibração
SPES	Speed Sensor
POSI	Interface de sensor de posição
USOS	Sensor ultrasônico
FBU1	Buffer de Frame – Camera esquerda, quadrante superior esquerdo
FBU2	Buffer de Frame – Camera esquerda, quadrante superior direito
FBU3	Buffer de Frame – Camera esquerda, quadrante inferior esquerdo
FBU4	Buffer de Frame – Camera esquerda, quadrante inferior direito
FBU5	Buffer de Frame – Camera direita, quadrante superior esquerdo
FBU6	Buffer de Frame – Camera direita, quadrante superior direito
FBU7	Buffer de Frame – Camera direita, quadrante inferior esquerdo
FBU8	Buffer de Frame – Camera direita, quadrante inferior direito
STAC	Controle de estabilidade
TPRC	Controle de pressão do pneu
DIRC	Controle de direção
NAVC	Controle de navegação
OBDB	Banco de dados de obstáculos
BFE1	Estimação de fundo e extrato de características 1
BFE2	Estimação de fundo e extrato de características 2

BFE3	Estimação de fundo e extrato de características 3
BFE4	Estimação de fundo e extrato de características 4
BFE5	Estimação de fundo e extrato de características 5
BFE6	Estimação de fundo e extrato de características 6
BFE7	Estimação de fundo e extrato de características 7
BFE8	Estimação de fundo e extrato de características 8
FDF1	Fusão de dados de características 1
FDF2	Fusão de dados de características 2
STPH	Photogrametria stereo
THRC	Controle do acelerador
VOD1	Odometria visual 1
VOD2	Odometria visual 2
OBMG	Gerenciador de banco de dados de obstáculos

Segundo Shi (Shi, 2009) o sistema de reconhecimento de obstáculos opera tarefas complexas sendo necessário dividir a aplicação de forma independente, como o pré-processamento, a extração de características e o cálculo de distância. As tarefas se agrupam em duas ou três para cada núcleo dentro da *SoC*.

A seguir é mostrado na tabela 7 às especificações formais dos fluxos da aplicação do veículo de navegação automotíva, mostrando as respectivas fontes e destinos de cada um e os valores associados às características das tarefas descritas na tabela 6.

Tabela 7: Descrição do Segundo Benchmark. (Baseada em SHI, 2009).

Fluxos	**Fonte**	**Destino**	**Prioridade**	**Latência Básica da Rede (µs)**	**Período (s)**	**Deadline (s)**	**Jitter (%)**	**Latência da Rede de Pior Caso (µs)**
1	POSI	NAVC	31	10.36	0.5	0.5	0.0	420.28
2	NAVC	OBDB	32	20.60	0.5	0.5	0.0	20.6
3	OBDB	NAVC	33	163.96	0.5	0.5	0.0	942.74
4	OBDB	OBMG	34	163.92	0.5	0.5	0.0	215.39999999999998
5	NAVC	DIRC	24	5.32	0.1	0.1	0.0	25.88
6	SPES	NAVC	25	5.28	0.1	0.1	0.0	819.9
7	NAVC	THRC	26	10.36	0.1	0.1	0.0	51.6
8	FBU3	VOD1	1	384.16	0.04	0.04	0.0	384.16
9	FBU8	VOD2	2	384.08	0.04	0.04	0.0	384.08
10	VOD1	NAVC	3	5.20	0.04	0.04	0.0	5.2
11	VOD2	NAVC	4	5.26	0.04	0.04	0.0	389.42
12	FBU1	BFE1	5	384.08	0.04	0.04	0.0	384.08

13	FBU2	BFE2	6	384.08	0.04	0.04	0.0	384.08
14	FBU3	BFE3	7	384.08	0.04	0.04	0.0	768.24
15	FBU4	BFE4	8	384.08	0.04	0.04	0.0	384.08
16	FBU5	BFE5	9	384.08	0.04	0.04	0.0	384.08
17	FBU6	BFE6	10	384.08	0.04	0.04	0.0	384.08
18	FBU7	BFE7	11	384.08	0.04	0.04	0.0	384.08
19	FBU8	BFE8	12	384.08	0.04	0.04	0.0	768.16
20	BFE1	FDF1	13	20.60	0.04	0.04	0.0	20.6
21	BFE2	FDF1	14	20.56	0.04	0.04	0.0	20.56
22	BFE3	FDF1	15	20.60	0.04	0.04	0.0	20.6
23	BFE4	FDF1	16	20.64	0.04	0.04	0.0	41.24
24	BFE5	FDF2	17	20.64	0.04	0.04	0.0	25.84
25	BFE6	FDF2	18	20.60	0.04	0.04	0.0	41.24
26	BFE7	FDF2	19	20.56	0.04	0.04	0.0	20.56
27	BFE8	FDF2	20	20.60	0.04	0.04	0.0	25.86
28	FDF1	STPH	21	82.00	0.04	0.04	0.0	82.0
29	FDF2	STPH	22	82.00	0.04	0.04	0.0	102.6
30	STPH	OBMG	23	41.12	0.04	0.04	0.0	41.12
31	POSI	OBMG	35	10.32	0.5	0.5	0.0	10.32
32	USOS	OBMG	27	10.36	0.1	0.1	0.0	51.48
33	OBMG	OBDB	37	41.04	1	1	0.0	61.64
34	TPMS	STAC	36	20.72	0.5	0.5	0.0	435.76
35	VIBS	STAC	28	5.24	0.1	0.1	0.0	5.24
36	STAC	TPRC	38	20.56	1	1	0.0	20.56
37	SPES	STAC	29	10.32	0.1	0.1	0.0	10.32
38	STAC	THRC	30	10.44	0.1	0.1	0.0	446.12

Na tabela 7 as unidades de medida dos períodos e dos deadlines das tarefas estão em segundos (s), enquanto que as Latências básicas e de pior caso estão em micro-segundos (μs). O jitter é medido em forma de porcentagem de atraso relacionada ao início de transmissão por conta da geração do pacote. O resultado da análise de escalonabilidade manteve-se escalonável mesmo com tráfego pesado de troca de dados entre os núcleos.

5 CONCLUSÃO

Este trabalho investigou a política de escalonamento por prioridade com base na técnica proposta por Shi (Shi, 2009). Considerando um MPSoC ou CMP baseados em uma interconexão de redes em chip e sistema de tempo real, o objetivo principal foi observar a análise de escalonabilidade a partir das interferências dos fluxos na rede. Considerando a técnica de chaveamento *Wormhole*.

Tendo em vista as características do chaveamento Wormhole, que promove menores requisitos de buffer e limite inferior de latência, além de maior vazão de dados entre os roteadores da rede, foram realizados experimentos de análise de escalonabilidade cujos resultados obtidos apontam para a eficiência da técnica empregada, no que diz respeito ao compartilhamento de links na rede.

Os resultados obtidos a partir dos experimentos realizados apontam que a latência de pior caso da rede calculada sobre os diferentes tipos de interferências aumenta a previsibilidade dos fluxos. Podendo ser utilizada como um meio de determinar a execução dos prazos das tarefas de forma mais rígida ocasionando a implementação do serviço de tempo real *hard*.

A análise de escalonabildade foi realizada em linguagem de alto nível e verificou a satisfação das restrições temporais *hard* em redes em chip que implementam a estratégia de escalonamento por prioridades fixas.

A estratégia apresentada no capítulo 3 avaliou diversos relacionamentos entre os fluxos e seus parâmetros permitindo assim a previsibilidade baseada em três interferências diferentes: a interferência direta de fluxos de maior prioridade, a interferência indireta de fluxos de maior prioridade e interferência de autobloqueio.

Adotando a abordagem de escalonabilidade, foram avaliados três conjuntos de aplicações. Sendo o segundo experimento uma extensão do primeiro, os resultados obtidos não se distanciaram dos números apresentados por (Ramaprasad & Mueller, 2006). Considerando que as aplicações comunicam-se pela estrutura de rede em chip e que as trocas de mensagem podem viajar por uma quantidade considerável de roteadores, os prazos são mantidos mesmo recebendo interferências ao longo do caminho.

O último grupo de tarefas foi baseado em uma aplicação para um veículo de navegação autônoma divida em diversos sistemas menores e caracterizados pelo tráfego pesado de informações (Shi, 2009). Todos esses fluxos e suas características são baseadas em dados reais para serviços de tempo real *hard*, em uma rede em chip. Os resultados mostram que a escalonabilidade é encontrada até mesmo em relação às interferências de autobloqueio.

O interesse da obtenção dos valores de latência nos cenários de pior caso utilizando análises de escalonabilidade é a automatização da análise de interferência. Esse processo de análise é fundamental para a política de atribuição de prioridades e assim para a otimização da performance do sistema. Os resultados alcançados geraram esse trabalho de dissertação e uma publicação na Conferência Internacional Ibero Americana.

6 TRABALHOS FUTUROS

Seria interessante ter um mecanismo de simulação que permitisse a realização da técnica aqui empregada. Portanto, uma proposta de trabalho futuro seria desenvolver esse mecanismo.

Neste trabalho o escalonamento e análise de escalonabilidade se dão de forma off-line e em implementação em alto nível. Assim um trabalho futuro poderia ser a implementação da abordagem aqui estudada em uma linguagem de descrição de hardware para comparação e teste em forma de simulação.

Esta pesquisa considerou as tarefas como sendo independentes entre si. Então, um possível trabalho futuro seria acrescentar à análise de aplicações e tarefas dependentes. Outra sugestão de trabalho futuro seria a realização da análise de escalonabilidade baseada em prioridades dinâmicas em outros tipos de sistemas.

Considerando a atribuição de prioridades, seria interessante estudar/pesquisar outras heurísticas que pudessem ser adotadas além da *branch and bound*.

7 REFERÊNCIAS[1]

Adriahantenaina, A. et. al., "SPIN: a scalable, packet switched, on-chip micro-network," *Design, Automation and Test in Europe Conference and Exhibition, 2003*, vol., no., pp.70,73 suppl., 2003. doi: 10.1109/DATE.2003.1253808.

Agarwal, A., Iskander, C., and Ravi Shankar, "Survey of Network on Chip (NoC) Architectures & Contributions", *Journal of Engineering, Computing and Architecture*[online], vol.3, no.1, 2009 [cited Nov. 21, 2010], available : http://www.scientificjournals.org/journals2009/articles/1.

Zhou, B., Qiao, J., Lin, S. "Research on parallel Real time scheduling algorithm of hybrid parameters task on MC platform" , Applied Mathematics & Information Sciences, 5, pp.211S-217S, 2011.

Balakrishnan, S., & Ozguner, F. "A priority- driven flow control mechanism for real-time traffic in multiprocessor networks". IEEE Transactions on Parallel and Distributed Systems, 9(7), 664–678, 1998. doi:10.1109/71.707545

Benini, L. and De Micheli, G., "Networks on chips: a new SoC paradigm," Computer , vol.35, no.1, pp.70,78, Jan 2002. doi: 10.1109/2.976921

Bjerregaard, T. and Mahadevan, S. "A survey of research and practices of Network-on-chip". *ACM Comput. Surv.* 38, 1, Article 1 (June 2006), 2006. DOI=http://doi.acm.org/10.1145/1132952.1132953 http://doi.acm.org/http://doi.acm.org/10.1145/1132952.1132953

D. C. Black, J. Donovan, "*SystemC: From the Ground Up"*, 2nd ed., Springer 2009. ISBN 0-387-69957-0

Bolotin, E., Cidon, I., Ginosar, R., Kolodny, A. "QNoC: QoS architecture and design process for network on chip", JOURNAL OF SYSTEMS ARCHITECTURE, vol. 50, pp.105,128, 2004.

CORRÊA, Edgard F. "Redes-em-Chip para sistemas embarcados visando à otimização de medidas de Qualidade de Serviços para aplicações de tempo real". 2007. 131f. Tese (Doutorado) - Universidade Federal do Rio Grande do Sul. Programa de Pós Graduação em Computação, Porto Alegre, 2007.

Dally, W., "Virtual-channel flow control," *Parallel and Distributed Systems, IEEE Transactions on*, vol.3, no.2, pp.194,205, Mar 1992. doi: 10.1109/71.127260.

Dally, W., and Towles, B. "*Principles and Practices of Interconnection Networks"*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

Davis, R. and Kato, S. "FPSL, FPCL and FPZL schedulability analysis". *Real-Time Syst.* 48, 6 (November 2012), pp. 750-788, 2012. DOI=10.1007/s11241-012-9149-x http://dx.doi.org/10.1007/s11241-012-9149-x

[1] Baseadas na norma NBR 6023, de 2002, da Associação Brasileira de Normas Técnicas (ABNT).

Dorin, F., Yomsi, P., Goossens, J., Richard, P. “Semi-Partitioned Hard Real-Time Scheduling with Restricted”. Migrations upon Identical Multiprocessor Platforms, CoRR, vol 1006.2637, 2010. http://arxiv.org/abs/1006.2637

Duato, J., S. Yalamanchili, and L. Ni. “*Interconnection Networks: An Engineering Approach”*, 2nd printing, Morgan Kaufmann Publishers, San Francisco, 2002.

Freitas, H. C. ; Alves, M. A. Z. ; Navaux, P. O. A. (2009) “NoC e NUCA: Conceitos e Tendências para Arquiteturas de Processadores Many-Core.” 9ª Escola Regional de Alto Desempenho (ERAD). Caxias do Sul, RS, Brasil, p. 5-37, 2009.

Gratz, P., Kim, C., McDonald, R., Keckler, S. W., & Burger, D.” Implementation and Evaluation of On-Chip Network Architectures”. 2006 International Conference on Computer Design*, 477–484, 2006. doi:10.1109/ICCD.2006.4380859*

Guan, N., Stigge, M., Wang, Y., Yu, G., "Fixed-Priority Multiprocessor Scheduling with Liu and Layland's Utilization Bound," *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE* , vol., no., pp.165,174, 12-15 April 2010 doi: 10.1109/RTAS.2010.39

Guerrier, P.; Greiner, A., "A generic architecture for on-chip packet-switched interconnections," *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings* , vol., no., pp.250,256, 2000. doi: 10.1109/DATE.2000.840047.

Goossens, K., "Formal methods for networks on chips," *Application of Concurrency to System Design, 2005. ACSD 2005. Fifth International Conference on* , vol., no., pp.188,189, 7-9 June 2005 doi: 10.1109/ACSD.2005.36

Harmanci, M.D.; Escudero, N.P.; Leblebici, Y.; Ienne, P., "Providing QoS to connection-less packet-switched NoC by implementing DiffServ functionalities," *System-on-Chip, 2004. Proceedings. 2004 International Symposium on* , vol., no., pp.37,40, 16-18 Nov. 2004. doi: 10.1109/ISSOC.2004.1411140.

Karim, F.; Nguyen, A.; Dey, S., "An interconnect architecture for networking systems on chips," *Micro, IEEE* , vol.22, no.5, pp.36,45, Sep/Oct 2002. doi: 10.1109/MM.2002.1044298.

Kim, B., Kim, J., Hong, S. J., Lee, S.. “A real-time communication method for wormhole switching networks”. In ICPP ’98: Proceedings of the Interna- tional Conference on Parallel Processing, pages 527–534, 1998.

Lee, S. “Real-time wormhole channels”. *J. Parallel Distrib. Comput.* 63, 3 (March 2003), 299-311, 2003. DOI=10.1016/S0743-7315(02)00055-2. http://dx.doi.org/10.1016/S0743-7315(02)00055-2.

Leung, J., & Zhao, H. “*Real-Time Scheduling Analysis”*, Technical report, p. 134, 2005.

Li, Q. & Yao, C. *“Real-Time Concepts for Embedded Systems”*. Published by CMP Books an imprint of CMP Media LLC, 2003.

Liu, L. and Layland, J. “Scheduling algorithms for multiprogramming in a hard-real-time environment”. J. ACM, 20(1):46–61, 1973.

Liu, W., Xu, J., Wu, X., Ye, Y., Wang, X., Zhang, W.; Nikdast, M.; Wang, Z. "A NoC Traffic Suite Based on Real Applications," *VLSI (ISVLSI), 2011 IEEE Computer Society Annual Symposium on* , vol., no., pp.66,71, 4-6 July 2011 doi: 10.1109/ISVLSI.2011.49

Lu, Z., Jantsch, A., Sander, I. "Feasibility analysis of messages for on- chip networks using wormhole routing". In ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation, pages 960–964, 2005.

Mello, A., Tedesco, L., Calazans, N., and Moraes, F. "Virtual channels in networks on chip: implementation and evaluation on hermes NoC". In *Proceedings of the 18th annual symposium on Integrated circuits and system design* (SBCCI '05). ACM, New York, NY, USA, 178-183, 2005. DOI=10.1145/1081081.1081128 http://doi.acm.org/10.1145/1081081.1081128

Mesidis, P. "*Mapping of Real-time Applications on Network-on-Chip based MPSOCS"*. University of York. Department of Computer Science, York, 2012, Retrieved from http://etheses.whiterose.ac.uk/2764

Moraes F., Calazans N., Mello A., Moller L., Ost L. "HERMES: An infrastructure for low area overhead packet-switching networks on chip" *Integration, the VLSI Journal*, 38 (1) , pp. 69-93, 2004.

Moraes, A. Mello, L. Möller, L. Ost, N. Calazans, "A low area overhead packet-switched network on chip: architecture and prototyping", in: IFIP Very Large Scale Integration (VLSI-SOC), pp. 318-323, 2003.

Mutka, Matt W., "Using rate monotonic scheduling technology for real-time communications in a wormhole network," *Parallel and Distributed Real-Time Systems, 1994. Proceedings of the Second Workshop on* , vol., no., pp.194,199, 28-29 Apr 1994 doi: 10.1109/WPDRTS.1994.365629

Nicopoulos, C., Narayanan, V., Das, C. "A Holistic Design Exploration". Springer. Series: Lecture Notes in Electrical Engineering, Vol. 45, 2010, XXII, 223p.

Rijpkema, E.; Goossens, K.; Radulescu, A.; Dielissen, J.; van Meerbergen, J.; Wielage, P.; Waterlander, E., "Trade-offs in the design of a router with both guaranteed and best-effort services for networks on chip," *Computers and Digital Techniques, IEE Proceedings* - , vol.150, no.5, pp.294-302, doi: 10.1049/ip-cdt:20030830

Rego, R. "Projeto e Implementação de uma Plataforma MP-SoC usando SystemC". 2007. Dissertação (Mestrado) - Universidade Federal do Rio Grande do Norte. Programa de Pós Graduação em em Sistemas e Computação, Natal, 2007.

Samman, Faizal A.; Hollstein, Thomas; Glesner, Manfred, "Flexible Wormhole-Switched Network-on-chip with Two-Level Priority Data Delivery Service", International Journal of Electrical & Electronics Engineering;Aug2009, Vol. 3 Issue 5, p242, 2009.

Sha, L., Abdelzaher, T., Karl-Erik, Cervin, A., Baker, T., Burns, A., Buttazzo, G., Caccamo, M., Lehoczky, J., and K. Mok, A. "Real Time Scheduling Theory: A Historical Perspective". *Real-Time Syst.* 28, 2-3 (November 2004), 101-155, 2004. DOI=10.1023/B:TIME.0000045315.61234.1e http://dx.doi.org/10.1023/B:TIME.0000045315.61234.1e

Shi, Z. and Burns, A. “Real-Time Communication Analysis for On-Chip Networks with Wormhole Switching”. In *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip* (NOCS '08). IEEE Computer Society, Washington, DC, USA, 161-170, 2008.

Shi, Z. “Real-Time Communication Services. 2009”. Tese (Doutorado) – University of York. Department of Computer Science, York, 2009.

Zheng Shi; Burns, A., "Real-Time Communication Analysis with a Priority Share Policy in On-Chip Networks," *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on* , vol., no., pp.3,12, 1-3 July 2009 doi: 10.1109/ECRTS.2009.17

Shi, Z., Burns, A., & Indrusiak, L. S. (2010). “Schedulability Analysis for Real Time On-Chip Communication with Wormhole Switching”. *International Journal of Embedded and Real-Time Communication Systems* (Vol. 1, pp. 1–22). doi:10.4018/jertcs.2010040101

Singh, A., Srikanthan, T., Kumar, A., and Jigang, W. “Communication-aware heuristics for run-time task mapping on NoC-based MPSoC platforms”. *J. Syst. Archit.* 56, 7 (July 2010), 242-255, 2010. DOI=10.1016/j.sysarc.2010.04.007 http://dx.doi.org/10.1016/j.sysarc.2010.04.007

Srikanth, G. U., Shanthi, A. P., Maheswari, V. U., Siromoney, A. “A Survey on Real Time Task Scheduling”, *69*(1), 33–41, 2012.

Vidal, R. R., “Contention Aware Scheduling for Real-Time Wormhole Network-on-Chip”, Proceedings XIX Workshop IBERCHIP, Cusco, Peru, fevereiro 2013.

Zeferino, C.A.; Susin, A.A., "SoCIN: a parametric and scalable network-on-chip," *Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings. 16th Symposium on* , vol., no., pp.169,174, 8-11 Sept. 2003. doi: 10.1109/SBCCI.2003.1232824.

ANEXO

1. Código respectivo à implementação realizada para processar o cálculo da análise de escalonabilidade.

```
package taskgraph;

import edu.uci.ics.jung.graph.SparseMultigraph;
import java.util.ArrayList;
import java.util.List;

/**
 *
 * @author ronnison
 */
public class Task extends SparseMultigraph<Integer, String>
{

    /**
     * constantes da rede-em-chip
     */
    private static int MaxLength = 32;
    private static int FlitSize = 4;
    private static int BW = 32;
    private static int Rhop = 25;

    /**
     * características dos fluxos
     */
    public int priority;
    public double MaxLatency;
    public double period;
    public double deadline;
    public double jitter;
    public double intJitter;
    public double ReleaseTime;

    // atribuição de prioridade
    public boolean mark;

    //roteamento
    List<List<Task>> direct = new ArrayList<List<Task>>();
    List<List<Task>> indirect = new
ArrayList<List<Task>>();

    /**
     * Análise de escalonabilidade
     */
    public double WorstCaseNetworkLatency;

    /**
     * construtor da tarefa
     */
    public Task (int priority, double period, double jitter) {
        super();
        this.priority = priority;
        this.period = period;
        this.deadline = period;
        this.jitter = jitter;
        this.intJitter = 0.0;
        this.MaxLatency = 0.0;
        this.ReleaseTime = 0.0;
        this.mark = false;
    }

    /**
     * getters das propriedades do fluxo
     */

    public void setPriority(int i) {
        this.priority = i;
    }

    public int getPriority() {
        return priority;
    }

    public double getDeadline() {
        return deadline;
    }

    public double getJitter() {
        return jitter;
    }

    public double getIntJitter() {
        return intJitter;
    }

    public double getPeriod() {
        return period;
    }

    public double getReleaseTime() {
        return ReleaseTime;
    }

    public void setMaxLatency(int hops) {
        this.MaxLatency =
(MaxLength/FlitSize)*(FlitSize/BW)+hops*Rhop;
    }

    public void setMaxLatency(double value) {
        this.MaxLatency = value;
    }

    public void setDeadLine (double value) {
        this.deadline = value;
    }

    /**
     * método para definir a latência básica máxima da rede
     */
    public double getMaxLatency() {
        return MaxLatency;
    }

    /**
    * métodos para encontrar as interferências diretas e
indiretas
    */
    public List<String> getRoute () {
        List<String> l = new ArrayList<String>();
        for (String e : this.getEdges()) {
            l.add(e);
        }
        return l;
    }

    public boolean priorityTest (Task t) {
        if (this.priority > t.priority)
            return true;
        else
            return false;
    }

    public void unassigned() {
```

```
        this.setPriority(0);
        this.mark = false;
    }

    public void assigned(int P) {
        this.setPriority(P);
        this.mark = true;
    }

    public boolean isAssigned () {
        if (this.mark == true) {
            return true;
        } else {
            return false;
        }
    }

    void getTrafficFlowWithPriority(int i) {
        this.unassigned();
    }
}

package taskgraph;

import edu.uci.ics.jung.graph.util.EdgeType;
import java.util.ArrayList;
import java.util.List;
import
org.apache.commons.collections15.list.AbstractLinkedList;

/**
 *
 * @author ronnison
 */
public class TaskGraph {

    /**
     * @param args the command line arguments
     */
    public static double JITTER = 0.0;

    public static void main(String[] args) {
        // Tarefa 15,6,9
        Task task1 = new Task(31, 500000.0, JITTER);
        task1.addVertex(1);
        task1.addVertex(2);
        task1.addVertex(6);
        task1.addEdge("e12", 1, 2, EdgeType.DIRECTED);
        task1.addEdge("e26", 2, 6, EdgeType.DIRECTED);
//        task1.setMaxLatency(task1.getEdgeCount());
        task1.setMaxLatency(10.36);

        // Tarefa 2
        Task task2 = new Task(32, 500000.0, JITTER);
        task2.addVertex(6);
        task2.addVertex(5);
        task2.addVertex(9);
        task2.addEdge("e65", 6, 5, EdgeType.DIRECTED);
        task2.addEdge("e59", 5, 9, EdgeType.DIRECTED);
//        task2.setMaxLatency(task2.getEdgeCount());
        task2.setMaxLatency(20.60);

        // Tarefa 3
        Task task3 = new Task(33, 500000.0, JITTER);
        task3.addVertex(9);
        task3.addVertex(10);
        task3.addVertex(6);
        task3.addEdge("e910", 9, 10, EdgeType.DIRECTED);
        task3.addEdge("e106", 10, 6, EdgeType.DIRECTED);
//        task3.setMaxLatency(task3.getEdgeCount());
        task3.setMaxLatency(163.96);

        // Tarefa 4
        Task task4 = new Task(34, 500000.0, JITTER);
        task4.addVertex(9);
        task4.addVertex(5);
        task4.addEdge("e95", 9, 5, EdgeType.DIRECTED);
//        task4.setMaxLatency(task4.getEdgeCount());
        task4.setMaxLatency(163.92);

        // Tarefa 5
        Task task5 = new Task(24, 100000.0, JITTER);
        task5.addVertex(6);
        task5.addVertex(7);
        task5.addVertex(8);
        task5.addVertex(12);
        task5.addVertex(16);
        task5.addEdge("e67", 6, 7, EdgeType.DIRECTED);
        task5.addEdge("e78", 7, 8, EdgeType.DIRECTED);
        task5.addEdge("e812", 8, 12, EdgeType.DIRECTED);
        task5.addEdge("e1216", 12, 16,
EdgeType.DIRECTED);
//        task5.setMaxLatency(task5.getEdgeCount());
        task5.setMaxLatency(5.32);

        // Tarefa 6
        Task task6 = new Task(25, 100000.0, JITTER);
        task6.addVertex(12);
        task6.addVertex(11);
        task6.addVertex(10);
        task6.addVertex(6);
        task6.addEdge("e1211", 12, 11,
EdgeType.DIRECTED);
        task6.addEdge("e1110", 11, 10,
EdgeType.DIRECTED);
        task6.addEdge("e106", 10, 6, EdgeType.DIRECTED);
//        task6.setMaxLatency(task6.getEdgeCount());
        task6.setMaxLatency(5.28);

        // Tarefa 7
        Task task7 = new Task(26, 100000.0, JITTER);
        task7.addVertex(6);
        task7.addVertex(10);
        task7.addVertex(14);
        task7.addEdge("e610", 6, 10, EdgeType.DIRECTED);
        task7.addEdge("e1014", 10, 14,
EdgeType.DIRECTED);
//        task7.setMaxLatency(task7.getEdgeCount());
        task7.setMaxLatency(10.36);

        // Tarefa 8
        Task task8 = new Task(1, 40000.0, JITTER);
        task8.addVertex(9);
        task8.addVertex(10);
        task8.addVertex(6);
        task8.addVertex(2);
        task8.addEdge("e910", 9, 10, EdgeType.DIRECTED);
        task8.addEdge("e106", 10, 6, EdgeType.DIRECTED);
        task8.addEdge("e62", 6, 2, EdgeType.DIRECTED);
//        task8.setMaxLatency(task8.getEdgeCount());
        task8.setMaxLatency(384.16);

        // Tarefa 9
        Task task9 = new Task(2, 40000.0, JITTER);
        task9.addVertex(16);
        task9.addVertex(15);
        task9.addEdge("e1615", 16, 15,
EdgeType.DIRECTED);
```

```
//      task9.setMaxLatency(task9.getEdgeCount());
        task9.setMaxLatency(384.08);

        // Tarefa 10
        Task task10 = new Task(3, 40000.0, JITTER);
        task10.addVertex(2);
        task10.addVertex(6);
        task10.addEdge("e26", 2, 6, EdgeType.DIRECTED);
//      task10.setMaxLatency(task10.getEdgeCount());
        task10.setMaxLatency(5.20);

        // Tarefa 11
        Task task11 = new Task(4, 40000.0, JITTER);
        task11.addVertex(15);
        task11.addVertex(14);
        task11.addVertex(10);
        task11.addVertex(6);
        task11.addEdge("e1514", 15, 14,
EdgeType.DIRECTED);
        task11.addEdge("e1410", 14, 10,
EdgeType.DIRECTED);
        task11.addEdge("e106", 10, 6, EdgeType.DIRECTED);
//      task11.setMaxLatency(task11.getEdgeCount());
        task11.setMaxLatency(5.26);

        // Tarefa 12
        Task task12 = new Task(5, 40000.0, JITTER);
        task12.addVertex(1);
        task12.addVertex(2);
        task12.addEdge("e12", 1, 2, EdgeType.DIRECTED);
//      task12.setMaxLatency(task12.getEdgeCount());
        task12.setMaxLatency(384.08);

        // Tarefa 13
        Task task13 = new Task(6, 40000.0, JITTER);
        task13.addVertex(5);
        task13.addVertex(6);
        task13.addEdge("e56", 5, 6, EdgeType.DIRECTED);
//      task13.setMaxLatency(task13.getEdgeCount());
        task13.setMaxLatency(384.08);

        // Tarefa 14
        Task task14 = new Task(7, 40000.0, JITTER);
        task14.addVertex(9);
        task14.addVertex(10);
        task14.addEdge("e910", 9, 10, EdgeType.DIRECTED);
//      task14.setMaxLatency(task14.getEdgeCount());
        task14.setMaxLatency(384.08);

        // Tarefa 15
        Task task15 = new Task(8, 40000.0, JITTER);
        task15.addVertex(13);
        task15.addVertex(14);
        task15.addEdge("e1314", 13, 14,
EdgeType.DIRECTED);
//      task15.setMaxLatency(task15.getEdgeCount());
        task15.setMaxLatency(384.08);

        // Tarefa 16
        Task task16 = new Task(9, 40000.0, JITTER);
        task16.addVertex(4);
        task16.addVertex(3);
        task16.addEdge("e43", 4, 3, EdgeType.DIRECTED);
//      task16.setMaxLatency(task16.getEdgeCount());
        task16.setMaxLatency(384.08);

        // Tarefa 17
        Task task17 = new Task(10, 40000.0, JITTER);
        task17.addVertex(8);
        task17.addVertex(7);
        task17.addEdge("e87", 8, 7, EdgeType.DIRECTED);
//      task17.setMaxLatency(task17.getEdgeCount());
        task17.setMaxLatency(384.08);

        // Tarefa 18
        Task task18 = new Task(11, 40000.0, JITTER);
        task18.addVertex(12);
        task18.addVertex(11);
        task18.addEdge("e1211", 12, 11,
EdgeType.DIRECTED);
//      task18.setMaxLatency(task18.getEdgeCount());
        task18.setMaxLatency(384.08);

        // Tarefa 19
        Task task19 = new Task(12, 40000.0, JITTER);
        task19.addVertex(16);
        task19.addVertex(15);
        task19.addEdge("e1615", 16, 15,
EdgeType.DIRECTED);
//      task19.setMaxLatency(task19.getEdgeCount());
        task19.setMaxLatency(384.08);

        // Tarefa 20
        Task task20 = new Task(13, 40000.0, JITTER);
        task20.addVertex(2);
        task20.addVertex(3);
        task20.addVertex(7);
        task20.addEdge("e23", 2, 3, EdgeType.DIRECTED);
        task20.addEdge("e37", 2, 7, EdgeType.DIRECTED);
//      task20.setMaxLatency(task20.getEdgeCount());
        task20.setMaxLatency(20.60);

        // Tarefa 21
        Task task21 = new Task(14, 40000.0, JITTER);
        task21.addVertex(6);
        task21.addVertex(7);
        task21.addEdge("e67", 6, 7, EdgeType.DIRECTED);
//      task21.setMaxLatency(task21.getEdgeCount());
        task21.setMaxLatency(20.56);

        // Tarefa 22
        Task task22 = new Task(15, 40000.0, JITTER);
        task22.addVertex(10);
        task22.addVertex(11);
        task22.addVertex(7);
        task22.addEdge("e1011", 10, 11,
EdgeType.DIRECTED);
        task22.addEdge("e117", 11, 7, EdgeType.DIRECTED);
//      task22.setMaxLatency(task22.getEdgeCount());
        task22.setMaxLatency(20.60);

        // Tarefa 23
        Task task23 = new Task(16, 40000.0, JITTER);
        task23.addVertex(14);
        task23.addVertex(15);
        task23.addVertex(11);
        task23.addVertex(7);
        task23.addEdge("e1415", 14, 15,
EdgeType.DIRECTED);
        task23.addEdge("e1511", 15, 11,
EdgeType.DIRECTED);
        task23.addEdge("e117", 11, 7, EdgeType.DIRECTED);
//      task23.setMaxLatency(task23.getEdgeCount());
        task23.setMaxLatency(20.64);

        // Tarefa 24
        Task task24 = new Task(17, 40000.0, JITTER);
        task24.addVertex(3);
```

```
        task24.addVertex(2);
        task24.addVertex(6);
        task24.addVertex(10);
        task24.addEdge("e32", 3, 2, EdgeType.DIRECTED);
        task24.addEdge("e26", 2, 6, EdgeType.DIRECTED);
        task24.addEdge("e610", 6, 10, EdgeType.DIRECTED);
//        task24.setMaxLatency(task24.getEdgeCount());
        task24.setMaxLatency(20.64);

        // Tarefa 25
        Task task25 = new Task(18, 40000.0, JITTER);
        task25.addVertex(7);
        task25.addVertex(6);
        task25.addVertex(10);
        task25.addEdge("e76", 7, 6, EdgeType.DIRECTED);
        task25.addEdge("e610", 6, 10, EdgeType.DIRECTED);
//        task25.setMaxLatency(task25.getEdgeCount());
        task25.setMaxLatency(20.60);

        // Tarefa 26
        Task task26 = new Task(19, 40000.0, JITTER);
        task26.addVertex(13);
        task26.addVertex(12);
        task26.addEdge("e1312", 13, 12,
EdgeType.DIRECTED);
//        task26.setMaxLatency(task26.getEdgeCount());
        task26.setMaxLatency(20.56);

        // Tarefa 27
        Task task27 = new Task(20, 40000.0, JITTER);
        task27.addVertex(15);
        task27.addVertex(14);
        task27.addVertex(10);
        task27.addEdge("e1514", 15, 14,
EdgeType.DIRECTED);
        task27.addEdge("e1410", 14, 10,
EdgeType.DIRECTED);
//        task27.setMaxLatency(task27.getEdgeCount());
        task27.setMaxLatency(20.60);

        // Tarefa 28
        Task task28 = new Task(21, 40000.0, JITTER);
        task28.addVertex(7);
        task28.addVertex(11);
        task28.addEdge("e711", 7, 11, EdgeType.DIRECTED);
//        task28.setMaxLatency(task28.getEdgeCount());
        task28.setMaxLatency(82.00);

        // Tarefa 29
        Task task29 = new Task(22, 40000.0, JITTER);
        task29.addVertex(10);
        task29.addVertex(11);
        task29.addEdge("e1011", 10, 11,
EdgeType.DIRECTED);
//        task29.setMaxLatency(task29.getEdgeCount());
        task29.setMaxLatency(82.00);

        // Tarefa 30
        Task task30 = new Task(23, 40000.0, JITTER);
        task30.addVertex(11);
        task30.addVertex(10);
        task30.addVertex(9);
        task30.addVertex(5);
        task30.addEdge("e1110", 11, 10,
EdgeType.DIRECTED);
        task30.addEdge("e109", 10, 9, EdgeType.DIRECTED);
        task30.addEdge("e95", 9, 5, EdgeType.DIRECTED);
//        task30.setMaxLatency(task30.getEdgeCount());
        task30.setMaxLatency(41.12);

        // Tarefa 31
        Task task31 = new Task(35, 500000.0, JITTER);
        task31.addVertex(1);
        task31.addVertex(5);
        task31.addEdge("e15", 1, 5, EdgeType.DIRECTED);
//        task31.setMaxLatency(task31.getEdgeCount());
        task31.setMaxLatency(10.32);

        // Tarefa 32
        Task task32 = new Task(27, 100000.0, JITTER);
        task32.addVertex(13);
        task32.addVertex(9);
        task32.addVertex(5);
        task32.addEdge("e139", 13, 9, EdgeType.DIRECTED);
        task32.addEdge("e95", 9, 5, EdgeType.DIRECTED);
//        task32.setMaxLatency(task32.getEdgeCount());
        task32.setMaxLatency(10.36);

        // Tarefa 33
        Task task33 = new Task(37, 1000000.0, JITTER);
        task33.addVertex(5);
        task33.addVertex(9);
        task33.addEdge("e59", 5, 9, EdgeType.DIRECTED);
//        task33.setMaxLatency(task33.getEdgeCount());
        task33.setMaxLatency(41.04);

        // Tarefa 34
        Task task34 = new Task(36, 500000.0, JITTER);
        task34.addVertex(13);
        task34.addVertex(14);
        task34.addVertex(15);
        task34.addVertex(16);
        task34.addVertex(12);
        task34.addVertex(8);
        task34.addEdge("e1314", 13, 14,
EdgeType.DIRECTED);
        task34.addEdge("e1415", 14, 15,
EdgeType.DIRECTED);
        task34.addEdge("e1516", 15, 16,
EdgeType.DIRECTED);
        task34.addEdge("e1612", 16, 12,
EdgeType.DIRECTED);
        task34.addEdge("e128", 12, 8, EdgeType.DIRECTED);
//        task34.setMaxLatency(task34.getEdgeCount());
        task34.setMaxLatency(20.72);

        // Tarefa 35
        Task task35 = new Task(28, 100000.0, JITTER);
        task35.addVertex(3);
        task35.addVertex(4);
        task35.addVertex(8);
        task35.addEdge("e34", 3, 4, EdgeType.DIRECTED);
        task35.addEdge("e48", 4, 8, EdgeType.DIRECTED);
//        task35.setMaxLatency(task35.getEdgeCount());
        task35.setMaxLatency(5.24);

        // Tarefa 36
        Task task36 = new Task(38, 1000000.0, JITTER);
        task36.addVertex(8);
        task36.addVertex(4);
        task36.addEdge("e84", 8, 4, EdgeType.DIRECTED);
//        task36.setMaxLatency(task36.getEdgeCount());
        task36.setMaxLatency(20.56);

        // Tarefa 37
        Task task37 = new Task(29, 100000.0, JITTER);
        task37.addVertex(12);
        task37.addVertex(8);
```

```
        task37.addEdge("e128", 12, 8, EdgeType.DIRECTED);
//        task37.setMaxLatency(task37.getEdgeCount());
        task37.setMaxLatency(10.32);

        // Tarefa 38
        Task task38 = new Task(30, 100000.0, JITTER);
        task38.addVertex(8);
        task38.addVertex(7);
        task38.addVertex(6);
        task38.addVertex(10);
        task38.addVertex(14);
        task38.addEdge("e87", 8, 7, EdgeType.DIRECTED);
        task38.addEdge("e76", 7, 6, EdgeType.DIRECTED);
        task38.addEdge("e610", 6, 10, EdgeType.DIRECTED);
        task38.addEdge("e1014", 10, 14,
EdgeType.DIRECTED);
//        task38.setMaxLatency(task38.getEdgeCount());
        task38.setMaxLatency(10.44);

        // Lista de Tarefas de 1 à 38
        List<Task> l = new ArrayList<Task>();

l.add(task1);l.add(task2);l.add(task3);l.add(task4);l.add(task5
);

l.add(task6);l.add(task7);l.add(task8);l.add(task9);l.add(task1
0);

l.add(task11);l.add(task12);l.add(task13);l.add(task14);l.add(t
ask15);

l.add(task16);l.add(task17);l.add(task18);l.add(task19);l.add(t
ask20);

l.add(task21);l.add(task22);l.add(task23);l.add(task24);l.add(t
ask25);

l.add(task26);l.add(task27);l.add(task28);l.add(task29);l.add(t
ask30);

l.add(task31);l.add(task32);l.add(task33);l.add(task34);l.add(t
ask35);
        l.add(task36);l.add(task37);l.add(task38);

        Task t1 = new Task(0, 5.0, JITTER);
        t1.addVertex(16);
        t1.addVertex(15);
        t1.addVertex(14);
        t1.addEdge("e1615", 16, 15);
        t1.addEdge("e1514", 15, 14);
        t1.setMaxLatency(1.0);

        Task t2 = new Task(0, 7.0, JITTER);
        t2.addVertex(14);
        t2.addVertex(13);
        t2.addEdge("e1413", 14, 13);
        t2.setMaxLatency(2.0);

        Task t3 = new Task(0, 9.0, JITTER);
        t3.addVertex(15);
        t3.addVertex(14);
        t3.addVertex(13);
        t3.addVertex(9);
        t3.addVertex(5);
        t3.addEdge("e1514", 15, 14);
        t3.addEdge("e1413", 14, 13);
        t3.addEdge("e139", 13, 9);
        t3.addEdge("e95", 9, 5);
        t3.setMaxLatency(2.0);

        Task t4 = new Task(0, 12.0, JITTER);
        t4.addVertex(9);
        t4.addVertex(5);
        t4.addVertex(1);
        t4.addEdge("e95", 9, 5);
        t4.addEdge("e51", 5, 1);
        t4.setMaxLatency(4.0);

        Task t5 = new Task(0, 8, JITTER);
        t5.addVertex(13);
        t5.addVertex(9);
        t5.addVertex(5);
        t5.addVertex(1);
        t5.addEdge("e139", 13, 9);
        t5.addEdge("e95", 9, 5);
        t5.addEdge("e51", 5, 1);
        t5.setMaxLatency(3.0);
        t5.setDeadLine(12);

        List<Task> l2 = new ArrayList<Task>();
        l2.add(t1); l2.add(t2); l2.add(t3); l2.add(t4); l2.add(t5);

//        CompareTask ct = new CompareTask();
//        SchedulableTest st = new SchedulableTest(l2);
//
System.out.println(l2.get(0).WorstCaseNetworkLatency);
//
System.out.println(l2.get(1).WorstCaseNetworkLatency);
//
System.out.println(l2.get(2).WorstCaseNetworkLatency);
//
System.out.println(l2.get(3).WorstCaseNetworkLatency);
//
System.out.println(l2.get(4).WorstCaseNetworkLatency);

        List<Task> l3 = new ArrayList<Task>();
//        l3.add(t1); l3.add(t2); l3.add(t3); l3.add(t4); l3.add(t5);
        l3.add(t5); l3.add(t4); l3.add(t3); l3.add(t2); l3.add(t1);


        ////////////////////////////////////////////////////////
//        Task tal1 = new Task(0, 50000.0, JITTER);
//        tal1.addVertex(5);
//        tal1.addVertex(2);
//        tal1.addEdge("e52", 5, 2);
//        tal1.setMaxLatency(750.0);

        Task tal1 = new Task(1, 62500.0, JITTER);
        tal1.addVertex(5);
        tal1.addVertex(2);
        tal1.addEdge("e52", 5, 2);
        tal1.setMaxLatency(7491.0);

        Task tal2 = new Task(2, 125000.0, JITTER);
        tal2.addVertex(4);
        tal2.addVertex(5);
        tal2.addVertex(2);
        tal2.addEdge("e45", 4, 5);
        tal2.addEdge("e52", 5, 2);
        tal2.setMaxLatency(9537.0);

        Task tal3 = new Task(3, 125000.0, JITTER);
        tal3.addVertex(6);
        tal3.addVertex(5);
        tal3.addVertex(2);
        tal3.addEdge("e65", 6, 5);
        tal3.addEdge("e52", 5, 2);
        tal3.setMaxLatency(14536.0);
```

```
        Task tal4 = new Task(4, 250000.0, JITTER);
        tal4.addVertex(3);
        tal4.addVertex(2);
        tal4.addEdge("e32", 3, 2);
        tal4.setMaxLatency(16738.0);

        Task tal5 = new Task(5, 250000.0, JITTER);
        tal5.addVertex(1);
        tal5.addVertex(2);
        tal5.addEdge("e12", 1, 2);
        tal5.setMaxLatency(54168.0);

        List<Task> l4 = new ArrayList<Task>();

l4.add(tal1);l4.add(tal2);l4.add(tal3);l4.add(tal4);l4.add(tal5);
//
l4.add(tal5);l4.add(tal4);l4.add(tal3);l4.add(tal2);l4.add(tal1);

        Task dotproduc = new Task(1, 50000.0, 0.0);
        dotproduc.addVertex(8);
        dotproduc.addVertex(7);
        dotproduc.addEdge("e87", 8, 7);
        dotproduc.setMaxLatency(750.0);

        Task convolution = new Task(2, 62500.0, 0.0);
        convolution.addVertex(4);
        convolution.addVertex(1);
        convolution.addEdge("e41", 4, 1);
        convolution.setMaxLatency(7491.0);

        Task fir = new Task(3, 125000.0, 0.0);
        fir.addVertex(9);
        fir.addVertex(8);
        fir.addVertex(7);
        fir.addVertex(4);
        fir.addVertex(1);
        fir.addEdge("e98", 9, 8);
        fir.addEdge("e87", 8, 7);
        fir.addEdge("e74", 7, 4);
        fir.addEdge("e41", 4, 1);
        fir.setMaxLatency(9537.0);

        Task lms = new Task(4, 125000.0, 0.0);
        lms.addVertex(5);
        lms.addVertex(4);
        lms.addVertex(7);
        lms.addEdge("e54", 5, 4);
        lms.addEdge("e47", 4, 7);
        lms.setMaxLatency(14536.0);

        Task nrealupdates = new Task(5, 250000.0, 0.0);
        nrealupdates.addVertex(2);
        nrealupdates.addVertex(1);
        nrealupdates.addEdge("e21", 2, 1);
        nrealupdates.setMaxLatency(16737.0);

        Task matrix1 = new Task(6, 250000.0, 0.0);
        matrix1.addVertex(3);
        matrix1.addVertex(2);
        matrix1.addVertex(1);
        matrix1.addEdge("e32", 3, 2);
        matrix1.addEdge("e21", 2, 1);
        matrix1.setMaxLatency(54168.0);

        List<Task> lista = new ArrayList<Task>();
        lista.add(dotproduc); lista.add(convolution);
lista.add(fir);
        lista.add(lms); lista.add(nrealupdates);
lista.add(matrix1);

        System.out.println(l);
    }
}

package taskgraph;

import java.util.ArrayList;
import java.util.List;

/**
 *
 * @author ronnison
 */
public class SchedulableTest {
    List<Task> e = new ArrayList<Task>();
    public SchedulableTest (List<Task> e) {
        this.e = e;
    }

    boolean SchedulableTest (List<Task> e) {
        boolean result = false;
        CompareTask ct = new CompareTask();
        ct.WorstCaseNetworkLatencyAnalysis(e);
        List<Double> TimeWindow = new
ArrayList<Double>();
        double BusyPeriod = 0.0;
        for (int i = 0; i < e.size(); i++) {
            if (e.get(i).getDeadline() <= e.get(i).getPeriod() -
e.get(i).getJitter()) {
                if (e.get(i).WorstCaseNetworkLatency <=
e.get(i).getDeadline()) {
                    System.out.println("Latencia da rede
"+e.get(i).WorstCaseNetworkLatency);
                    e.get(i).mark = true;
                } else {
                    e.get(i).mark = false;
                }
                result = true;
            } else if (e.get(i).getDeadline() > e.get(i).getPeriod() -
e.get(i).getJitter()) {
                ct.setIntJitter(e.get(3), 0.0);
                BusyPeriod = ct.SelfBlockingInterference(e.get(i));
                if (BusyPeriod <= e.get(i).getPeriod() -
e.get(i).getJitter()) {
                    System.out.println("somente uma instância do
pacote é gerada até o fim do período de ocupação");
                    e.get(i).WorstCaseNetworkLatency =
BusyPeriod;
                    result = true;
                } else {
                    System.out.println("mais de uma instância do
pacote é gerada até o fim do período de ocupação.");
                    double x =
Math.ceil(BusyPeriod/e.get(i).getPeriod());
                    int q = (int) x;
                    TimeWindow =
ct.TimeWindowFunction(e.get(i), q, BusyPeriod);
                    e.get(i).WorstCaseNetworkLatency =
ct.MaxTimeWindow(e.get(i), TimeWindow);
                    result = true;
                }
                BusyPeriod = 0.0;
                result = false;
            }
        }
        for (int i = 0; i < e.size(); i++) {
```

```
            if (e.get(i).direct != null) {
                if (ct.ParallelTest (e.get(i))) {
                    if (!ct.ParallelAnalysis(e.get(i))) {
                        System.out.println("Fluxo não escalonável");
                    }
                }
            }
        }

        for (int i = 0; i < e.size(); i++) {
            if (e.get(i).mark != true) {
                System.out.println("Tráfego de fluxo não
escalonável");
            } else {
                System.out.println("Teste de Escalonabilidade bem
sucedido!");
            }
        }
        return result;
    }
}

package taskgraph;

import java.util.ArrayList;
import java.util.List;

/**
 *
 * @author ronnison
 */
public class PriorityAssigment{

    CompareTask ct = new CompareTask();

    void PriorityOrderAssigmentLowerBound (List<Task> e,
Task t, int P) {
        int numPriority = e.size();
        System.out.println(numPriority);

        for (int i = 0; i < numPriority; i++) {
            List<Task> Sd = new ArrayList<Task>();
            System.out.println(e.get(i));
            System.out.println(e.get(i).mark);
            if (e.get(i).mark == false || e.get(i).equals(t)) {
                for (int j = i; j < numPriority; j++) {
                    if (e.get(j).mark == false || e.get(i).equals(t)) {
                        if (!e.get(j).equals(e.get(i))) {
                            if (ct.getInterference(e.get(i), e.get(j))) {
                                Sd.add(e.get(j));
                            }
                        }
                    }
                }
                if (!Sd.isEmpty()) {
                    System.out.println(e.get(i));
                    System.out.println(Sd);
                    System.out.println(e.get(i).direct);
                    if (e.get(i).direct == null) {
                        e.get(i).direct = new
ArrayList<List<Task>>();
                    }
                    e.get(i).direct.add(Sd);
                    System.out.println("Task "+e.get(i));
                    System.out.println("Sd "+Sd);
                    Sd = null;
                } else {
                    e.get(i).direct = null;
                }
                if (e.get(i).direct == null) {
                    e.get(i).WorstCaseNetworkLatency =
e.get(i).MaxLatency;

System.out.println(e.get(i).WorstCaseNetworkLatency);
                } else {
                    e.get(i).WorstCaseNetworkLatency =
ct.LowerBoundNetworkLatency(e.get(i));

System.out.println(e.get(i).WorstCaseNetworkLatency);
                }
            }
        }

        for (int i = 0; i < e.size(); i++) {
            e.get(i).direct = null;
            e.get(i).indirect = null;
        }
    }

    boolean PriorityOrderAssigmentUpperBound (List<Task>
e, Task t, int P) {
        int prio = 0;
        int numPriority = e.size();
        System.out.println(numPriority);

        for (int i = 0; i < e.size(); i++) {
            if (e.get(i).equals(t)) {
                System.out.println(e.get(i));
                e.get(i).assigned(P);
            }
        }

        for (int i = 0; i < numPriority; i++) {
            List<Task> Sd = new ArrayList<Task>();
            List<Task> Si = new ArrayList<Task>();
            prio = i;
            if (e.get(i).mark == false || e.get(i).equals(t)) {
                for (int j = 0; j < numPriority; j++) {
                    if (e.get(j).mark == false || e.get(i).equals(t)) {
                        if (!e.get(j).equals(e.get(i))) {
                            if (ct.getInterference(e.get(i), e.get(j))) {
                                Sd.add(e.get(j));
                                for (int k = j; k < numPriority; k++) {
                                    if (e.get(k).mark == false ||
e.get(i).equals(t)) {
                                        if (ct.getInterference(e.get(j),
e.get(k))) {
                                            if (!e.get(i).equals(e.get(j))) {
                                                if (!e.get(j).equals(e.get(k))) {
                                                    if (!e.get(i).equals(e.get(k)))
{
                                                        Si.add(e.get(k));
                                                    }
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
                for (int j = 0; j < Sd.size(); j++) {
                    Si.remove(Sd.get(j));

                }
```

```
            if (!Sd.isEmpty()) {
                if (e.get(i).direct == null) {
                    e.get(i).direct = new
ArrayList<List<Task>>();
                }
                e.get(i).direct.add(Sd);
                System.out.println("Task "+e.get(i));
                System.out.println("Sd "+Sd);
                System.out.println("Si "+Si+"\n");
                Sd = null;
                if (!Si.isEmpty()) {
                    if (e.get(i).indirect == null) {
                        e.get(i).indirect = new
ArrayList<List<Task>>();
                    }
                    e.get(i).indirect.add(Si);
                    Si = null;
                }
            } else {
                e.get(i).direct = null;
                e.get(i).direct = new ArrayList<List<Task>>();
                e.get(i).indirect = null;
                e.get(i).indirect = new ArrayList<List<Task>>();
            }
        }
    }
    if (ct.setIntJitterUpperBound(e) == false) {
        for (int i = 0; i < e.size(); i++) {
            e.get(i).direct = null;
            e.get(i).indirect = null;
        }
        return false;
    } else {
        for (int k = 0; k < e.size(); k++) {
            if (e.get(k).indirect == null) {
                e.get(k).WorstCaseNetworkLatency =
e.get(k).MaxLatency;
            } else {
                e.get(k).WorstCaseNetworkLatency =
ct.UpperBoundNetworkLatency(e.get(k));
            }

System.out.println(e.get(k).WorstCaseNetworkLatency);
        }
        for (int i = 0; i < e.size(); i++) {
            e.get(i).direct = null;
            e.get(i).indirect = null;
        }
        return true;
    }
  }
}

package taskgraph;

/**
 *
 * @author ronnison
 */
public class MathUtils {

  private static double gcd(double a, double b) {
    while (b > 0) {
        double temp = b;
        b = a % b; // % is remainder
        a = temp;
    }
  return a;
  }

  private static double gcd(double[] input) {
    double result = input[0];
    for(int i = 1; i < input.length; i++) result = gcd(result,
input[i]);
    return result;
  }

  private static double lcm(double a, double b) {
    return a * (b / gcd(a, b));
  }

  public static double lcm(double[] input) {
    double result = input[0];
    for(int i = 1; i < input.length; i++) result = lcm(result,
input[i]);
    return result;
  }

}

package taskgraph;

import java.util.ArrayList;
import java.util.List;

/**
 *
 * @author ronnison
 */
public class CompareTask {

  public boolean getInterference (Task t1, Task t2) {
    List<String> interf = new ArrayList<String>();
    for (String e : t1.getEdges()) {
        for (String f : t2.getEdges()) {
            if (e.equals(f)){
                interf.add(e);
            }
        }
    }
    if (!interf.isEmpty()) {
        return true;
    } else {
        return false;
    }
  }

  public boolean directInterference (List<Task> e) {
    List<Task> Sd = null;
    for (int j = 0; j < e.size(); j++) {
        Sd = new ArrayList<Task>();
        for (int i = 0; i < e.size(); i++) {
            if (e.get(j).getPriority() > e.get(i).getPriority()) {
                if (!e.get(j).equals(e.get(i))) {
                    if (getInterference(e.get(j), e.get(i))) {
                        Sd.add(e.get(i));
                    }
                }
            }
        }
        if (!Sd.isEmpty()) {
            if (e.get(j).direct == null) {
                e.get(j).direct = new ArrayList<List<Task>>();
            }
            e.get(j).direct.add(Sd);
            Sd = null;
        } else {
            e.get(j).direct = null;
```

```
        }
    }
    return true;
  }

  public boolean indirectInterference (List<Task> e) {
    List<Task> Si = null;
    for (int j = 0; j < e.size(); j++) {
      Si = new ArrayList<Task>();
      for (int k = 0; k < e.size(); k++) {
        for (int l= 0 ; l < e.size(); l++) {
          if ((e.get(j).getPriority() >
e.get(k).getPriority())){
            if ((e.get(k).getPriority() >
e.get(l).getPriority())) {
              if (!e.get(j).equals(e.get(k))) {
                if (!e.get(k).equals(e.get(l))) {
                  if (!e.get(j).equals(e.get(l))) {
                    if (getInterference(e.get(j), e.get(k)))
{
                      if (getInterference(e.get(k),
e.get(l))) {
                        Si.add(e.get(l));
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
      if (!Si.isEmpty()) {
        for (int k = 0; k < e.get(j).direct.get(0).size(); k++)
{
          for (int l = 0; l < Si.size(); l++) {
            if
(e.get(j).direct.get(0).get(k).equals(Si.get(l))) {
              Si.remove(Si.get(l));
            }
          }
        }
        if (e.get(j).indirect == null) {
          e.get(j).indirect = new ArrayList<List<Task>>();
        }
        e.get(j).indirect.add(Si);
        Si = null;
      } else {
        e.get(j).indirect = null;
      }
    }
    return true;
  }

  public double WorstCaseNetworkLatencyAnalysis
(List<Task> e) {
    double R = 0.0;
    if (e.isEmpty()){
      return 0.0;
    } else if (directInterference(e)) {
      for (int i = 0; i < e.size(); i++) {
        if (e.get(i).direct == null) {
          e.get(i).WorstCaseNetworkLatency =
e.get(i).getMaxLatency();
        } else {
          e.get(i).WorstCaseNetworkLatency =
DirectInterferenceFunction(e.get(i));
        }
      }
      if (indirectInterference(e)) {
        setIntJitter(e);
        for (int i = 0; i < e.size(); i++) {
          if (e.get(i).indirect != null) {
            e.get(i).WorstCaseNetworkLatency =
IndirectInterferenceFunction(e.get(i));
          }
        }
      }
    }
    return R;
  }

  public double SelfBlockingInterference (Task e) {
    double result = 0.0;
    double result2 = 0.0;
    double Q = 0;

    int i = 0;
    while (i != -1) {
      if (i == 0) {
        result = e.getMaxLatency();
      } else {
        for (int j = 0; j < e.direct.get(0).size(); j++) {
          result2 += Math.ceil((result +
e.direct.get(0).get(j).getIntJitter() +
e.direct.get(0).get(j).getJitter())/e.direct.get(0).get(j).getPerio
d())*e.direct.get(0).get(j).getMaxLatency();
        }
        Q = (Math.ceil((result +
e.getJitter())/e.getPeriod())*e.getMaxLatency());
        result2 += Q;
      }
      if (result == result2) {
        i = -1;
      } else {
        if (i != 0) {
          result = result2;
          result2 = 0;
        }
        i++;
      }
    }
    return result2;
  }

  public double LowerBoundNetworkLatency (Task e) {
    return DirectInterferenceFunction(e);
  }

  public double DirectInterferenceFunction(Task e) {
    double result = 0.0;
    double result2 = 0.0;

    int i = 0;
    while (i != -1) {
      if (i == 0) {
        result = e.getMaxLatency();
      } else {
        for (int j = 0; j < e.direct.get(0).size(); j++) {
          result2 += Math.ceil((result +
e.direct.get(0).get(j).getJitter())/e.direct.get(0).get(j).getPerio
d())*e.direct.get(0).get(j).getMaxLatency();
        }
        result2 += e.getMaxLatency();
      }
      if (result == result2) {
        i = -1;
      } else {
```

```
            if (i != 0) {
                result = result2;
                result2 = 0;
            }
            i++;
        }
    }
    return result2;
}

public double UpperBoundNetworkLatency (Task e) {
    return IndirectInterferenceFunction(e);
}

private double IndirectInterferenceFunction(Task e) {
    double result = 0.0;
    double result2 = 0.0;

    int i = 0;
    while (i != -1) {
        if (i == 0) {
            result = e.getMaxLatency();
        } else {
            for (int j = 0; j < e.direct.get(0).size(); j++) {
                result2 += Math.ceil((result +
e.direct.get(0).get(j).getIntJitter() +
e.direct.get(0).get(j).getJitter())/e.direct.get(0).get(j).getPerio
d())*e.direct.get(0).get(j).getMaxLatency();
            }
            result2 += e.getMaxLatency();
        }
        if (result == result2) {
            i = -1;
        } else {
            if (i != 0) {
                result = result2;
                result2 = 0;
            }
            i++;
        }
    }
    return result2;
}

public void setIntJitter (List<Task> e) {
    for (int i = 0; i < e.size(); i++) {
        if (e.get(i).indirect != null) {
            for (int j = 0; j < e.get(i).direct.get(0).size(); j++) {
                e.get(i).direct.get(0).get(j).intJitter =
(e.get(i).direct.get(0).get(j).WorstCaseNetworkLatency -
e.get(i).direct.get(0).get(j).MaxLatency);
            }
        }
    }
}

public boolean setIntJitterUpperBound (List<Task> e) {
    boolean x = false;
    for (int i = 0; i < e.size(); i++) {
        if (e.get(i).indirect != null) {
            if (!e.get(i).indirect.isEmpty()) {
                if (!e.get(i).direct.isEmpty()) {
                    System.out.println("teste "+e.get(i).direct);

System.out.println(e.get(i).direct.get(0).size());
                    for (int j = 0; j < e.get(i).direct.get(0).size();
j++) {
                        e.get(i).direct.get(0).get(j).intJitter =
(e.get(i).direct.get(0).get(j).getDeadline() -
e.get(i).direct.get(0).get(j).MaxLatency);
                    }
                    x = true;
                }
            }
        } else {
            x = false;
        }
    }
    return x;
}

public void setIntJitter (Task e, double value) {
    e.intJitter = value;
}

public boolean ParallelTest (Task e) {
    boolean parallel = false;
    for (int i = 0; i < e.direct.get(0).size(); i++) {
        for (int j = 0; j < e.direct.get(0).size(); j++) {
            if
(!e.direct.get(0).get(i).equals(e.direct.get(0).get(j))) {
                if (getInterference(e.direct.get(0).get(i), e)) {
                    parallel = true;
                }
            }
        }
    }

    return parallel;
}

public boolean ParallelAnalysis (Task e) {
    boolean result = false;
    double t1 = MaxReleaseTime(e) +
LeastCommonMultiplePeriod (e);
    double deadline = e.getDeadline();
    for (int i = 0; i < e.direct.get(0).size(); i++) {
        if (deadline < t1) {
            result = true;
        }
        deadline = e.direct.get(0).get(i).getDeadline();
    }

    return result;
}

public double MaxReleaseTime (Task e) {
    double MaxReleaseTime = e.getReleaseTime();

    for (int i = 0; i < e.direct.get(0).size(); i++) {
        if (MaxReleaseTime <
e.direct.get(0).get(i).getReleaseTime()) {
            MaxReleaseTime =
e.direct.get(0).get(i).getReleaseTime();
        }
    }

    return MaxReleaseTime;
}

public double LeastCommonMultiplePeriod (Task e) {
    double[] result = new double[e.direct.get(0).size()+1];

    for (int i = 0; i < e.direct.get(0).size(); i++) {
        if (i == 0) {
            result[i] = e.getPeriod();
```

```
            }
            result[i+1] = e.direct.get(0).get(i).getPeriod();
        }

        return MathUtils.lcm(result);
    }

    public List<Double> TimeWindowFunction (Task e,
double q, double BusyPeriod) {
        double result = 0.0;
        double result2 = 0.0;
        List<Double> TimeWindows = new
ArrayList<Double>();

        for (int j = 1; j <= q; j++) {
            int i = 0;
            while (i != -1) {
                if (i == 0) {
                    result = e.getMaxLatency()+j*e.getPeriod();
                } else {
                    for (int l = 0; l < e.direct.get(0).size(); l++) {
                        result2 += Math.ceil((result +
e.direct.get(0).get(l).getIntJitter() +
e.direct.get(0).get(l).getJitter())/e.direct.get(0).get(l).getPerio
d())*e.direct.get(0).get(l).getMaxLatency();
                    }
                    result2 += j*e.getMaxLatency();
                }
                if (result == result2) {
                    i = -1;
                } else {
                    if (i != 0) {
                        result = result2;
                        result2 = 0.0;
                    }
                    i++;
                }
            }
            if (j == q) {
                if (result2 > BusyPeriod) {
                    result2 = BusyPeriod;
                }
            }
            TimeWindows.add(result2);
            result2 = 0.0;
        }
        return TimeWindows;
    }

    public double MaxTimeWindow (Task e, List<Double>
TimeWindow) {
        double result = 0.0;
        List<Double> MaxResult = new ArrayList<Double>();

        for (int i = 0; i < TimeWindow.size(); i++) {
            result = (TimeWindow.get(i) - ((i)*e.getPeriod()) +
e.getJitter());
            MaxResult.add(result);
        }

        for (int i = 0; i < MaxResult.size(); i++) {
            if (result < MaxResult.get(i)) {
                result = MaxResult.get(i);
            }
        }
        return result;
    }
}
```

```
package taskgraph;

import java.rmi.Remote;
import java.util.ArrayList;
import java.util.List;
import
org.apache.commons.collections15.list.AbstractLinkedList;
import sun.org.mozilla.javascript.ast.Assignment;

/**
 *
 * @author ronnison
 */
public class BBSA {
    PriorityAssigment pa = new PriorityAssigment();
//    Task aux = null;
    int P;
    int priorityCurrent;
    //contrói-se n listas de candidatos
    List<List<List<Task>>> ListaCandidatos = new
ArrayList<List<List<Task>>>();
    List<Task> lista = new ArrayList<Task>();
    int p = 0;
    boolean resultado = false;
    int index = 0;

    void BBSA (List<Task> e) {
        P = e.size();

        ListaCandidatos.add(new ArrayList<List<Task>>());
        boolean b = false;

        for (int i = e.size()-1; i >= 0; i--) {
            //Atribui prioridade ao fluxo que passe na avaliação
de limite superior
            //reduz a prioridade
            boolean full = false;
            for (int j = 0; j < e.size(); j++) {
                if (!e.get(j).isAssigned()) {
                    full = false;
                    break;
                } else {
                    full = true;
                }
            }
            if (full == true) {
                SchedulableTest st = new SchedulableTest(e);
                if (!st.SchedulableTest(e)) {
                    System.out.println(ListaCandidatos.get(index-
1));

                    System.out.println(ListaCandidatos);
                    i = BackTrack(e, i);
                    System.out.println(i);

                } else {
                    System.out.println("escalonável");
                    resultado = true;
                }
            }
            if (!e.get(i).isAssigned()) {
                int w = 0;
                while (b == false) {
                    b = UpperBoundAnalysis(e);
                    if (b == false) {
                        b = true;
                    }
                }
                System.out.println(e.get(0).getPriority());
                System.out.println(e.get(1).getPriority());
```

```
            System.out.println(e.get(2).getPriority());
            System.out.println(e.get(3).getPriority());
            System.out.println(e.get(4).getPriority());
              if (LowerBoundAnalysis(e)) {
                System.out.println(ListaCandidatos);
                index++;
              } else {
                ListaCandidatos.add(new
ArrayList<List<Task>>());
                index++;
              }
            System.out.println(index);
            System.out.println(ListaCandidatos.size());
            System.out.println(P);
            System.out.println("lista candidatos
"+ListaCandidatos);
            while (ListaCandidatos.get(index-1).isEmpty()) {
              Task a = null;
              P++;
              i--;
              index--;
              if (P > e.size()) {
                //falha
                System.out.println("Falhou");
                return;
              } else {
                a = GetTrafficFlowWithPriority(e, P);
                System.out.println("fluxo removido"+a);
              }
              //remove elemente da lista
              // desatribua o fluxo da prioridade P+1
              // recomeça a busca por outro fluxo apropriado
            }
            //seleciona melhor candidato
            //atribui a prioridade P
            System.out.println(index);
            System.out.println(P);
            if (i == e.size()-1) {
              for (int j = 0; j < e.size(); j++) {
                if (!e.get(j).isAssigned()) {
                  p = index-1;
                  if (ListaCandidatos.size() > 1){
                    p = index-1;
                  }
                  break;
                } else {
                  p = index;
                  if (ListaCandidatos.size() > 1){
                    p = index-1;
                  }
                  break;
                }
              }
            } else {
              p = p+1;
            }
            System.out.println(p);
            System.out.println(ListaCandidatos.size());
            System.out.println("lista candidatos
"+ListaCandidatos);
            Task a = GetBestCandidate(ListaCandidatos, p);
            ListaCandidatos.get(p).remove(a);
            System.out.println(a);
            if (a != null) {
              for (int j = 0; j < e.size(); j++) {
                if (e.get(j) == a) {
                  e.get(j).assigned(priorityCurrent);
                }
              }
            }

            System.out.println(a);
            if (P == 1) {
              for (int j = 0; j < e.size(); j++) {
                if (!e.get(j).isAssigned()) {
                  e.get(j).assigned(P);
                  SchedulableTest st = new
SchedulableTest(e);
                  if (!st.SchedulableTest(e)) {
                    e.get(j).unassigned();

System.out.println(ListaCandidatos.get(index));

System.out.println(ListaCandidatos.get(index-1));
                    i = BackTrack(e, i);
                    break;
                  } else {
                    System.out.println("escalonável");
                    System.out.println(e.get(0).getPriority());
                    System.out.println(e.get(1).getPriority());
                    System.out.println(e.get(2).getPriority());
                    System.out.println(e.get(3).getPriority());
                    System.out.println(e.get(4).getPriority());
                    resultado = true;
                  }
                }
              }
            }
            System.out.println(e.get(0).getPriority());
            System.out.println(e.get(1).getPriority());
            System.out.println(e.get(2).getPriority());
            System.out.println(e.get(3).getPriority());
            System.out.println(e.get(4).getPriority());
          }
          ////////////////// modificado
          System.out.println(i);
          if (i == 0) {
            for (int j = 0; j < e.size(); j++) {
              if (e.get(j).isAssigned()) {
                if (e.get(j).getPriority() == 0) {
                  e.get(j).unassigned();
                }
              } else {
                i = e.size();
              }
            }
          }
          if (resultado == true) {
            break;
          }
        }
      }

      int BackTrack (List<Task> e, int i) {
        Task a = null;
        System.out.println(ListaCandidatos.get(index-1));
        while (ListaCandidatos.get(index).isEmpty()) {
          i++;
          index--;
          if (P > e.size()) {
            //falha
            System.out.println("Falhou");
            break;
          } else {
            if (!ListaCandidatos.get(index).isEmpty()) {
              if (ListaCandidatos.get(index).size() != 1) {
                a = GetTrafficFlowWithPriority(e, P);
                System.out.println("fluxo removido"+a);
```

```
                    System.out.println();
                    for (int j = 0; j <
ListaCandidatos.get(index).size(); j++) {
                        for (int k = 0; k <
ListaCandidatos.get(index).get(j).size(); k++) {

System.out.println(ListaCandidatos.get(index).get(j));
                            if
(ListaCandidatos.get(index).get(j).get(k).equals(a)) {

ListaCandidatos.get(index).get(j).remove(a);
                            }
                        }
                    }
                } else {
                    i++;
                    index--;
                }
            }
            System.out.println(ListaCandidatos.get(index));
            if (ListaCandidatos.get(index).size() != 0) {
                a = GetBestCandidate(ListaCandidatos, P);
                if (a == null) {
                    int x = ListaCandidatos.get(index).size();
                    ListaCandidatos.get(index).get(x-1).add(null);

System.out.println(ListaCandidatos.get(index).get(x-1));
                    i = BackTrack(e, i);
                }
            } else {
                i = BackTrack(e, i);
            }
            P++;
            break;
        }
        //remove elemente da lista
        // desatribua o fluxo da prioridade P+1
        // recomeça a busca por outro fluxo apropriado
    }
    return i;
}

boolean UpperBoundAnalysis (List<Task> e) {
    boolean x = false;
    for (int i = 0; i < e.size(); i++) {
        if (!e.get(i).isAssigned()) {
            if (pa.PriorityOrderAssigmentUpperBound(e,
e.get(i), P) == true) {

System.out.println(e.get(i).WorstCaseNetworkLatency);
                System.out.println(e.get(i).getDeadline());
                if (e.get(i).WorstCaseNetworkLatency <=
e.get(i).getDeadline()) {
                    System.out.println("deu certo: prioridade
"+P);
                    x = true;
                }else {
                    System.out.println("não deu certo: prioridade
"+P);
                    System.out.println(i);
                    System.out.println(e.size());
                    if (i == e.size()){
                        x = false;
                    }
                }
            } else {
                for (int j = 0; j < e.size(); j++) {
                    if (e.get(j).getPriority() == P) {
                        e.get(j).unassigned();
                    }
                }
                x = false;
            }
        }
        if (x == true) {
            P--;
        } else {
            System.out.println(P);
            for (int j = 0; j < e.size(); j++) {
                if (e.get(j).getPriority() == P) {
                    e.get(j).unassigned();
                }
            }
        }
        if (!lista.isEmpty()) {
            lista = null;
            lista = new ArrayList<Task>();
        }
    }
    System.out.println(P);
    return x;
}

boolean LowerBoundAnalysis (List<Task> e) {
    boolean x = false;
    int aux = e.size();
    int index = ListaCandidatos.size()-1;

    System.out.println(e.get(0).getPriority());
    System.out.println(e.get(1).getPriority());
    System.out.println(e.get(2).getPriority());
    System.out.println(e.get(3).getPriority());
    System.out.println(e.get(4).getPriority());
    System.out.println(e.get(4).mark);
            System.out.println(e.get(4).getPriority());
    for (int i = 0; i < e.size(); i++) {
        System.out.println("Task "+e.get(i));
        if (!e.get(i).isAssigned()) {
            pa.PriorityOrderAssigmentLowerBound(e, e.get(i),
P);
            if (e.get(i).WorstCaseNetworkLatency <=
e.get(i).getDeadline()) {
                System.out.println("deu certo: prioridade "+P);
                System.out.println("worstcase
"+e.get(i).WorstCaseNetworkLatency);
                x = true;
            }else {
                System.out.println("não deu certo: prioridade
"+P);
                System.out.println("worstcase
"+e.get(i).WorstCaseNetworkLatency);
                x = false;
            }
        }
        for (int j = 0; j < e.size(); j++) {
            if (e.get(j).getPriority() == 0) {
                e.get(j).unassigned();
            }
        }
        if (x == true) {
            if (!e.get(i).isAssigned()) {
                System.out.println(e.get(4).mark);
                System.out.println(e.get(4).getPriority());
                lista.add(e.get(i));
            }
            System.out.println("lista "+lista);
            if (i == aux-1) {
                ListaCandidatos.get(index).add(lista);
```

```
                ListaCandidatos.add(new                                    }
ArrayList<List<Task>>());
                index++;
                break;
            }
        } else {
            for (int j = 0; j < e.size(); j++) {
                if (e.get(j).getPriority() == P) {
                    e.get(j).unassigned();
                    System.out.println(e.get(j));
                }
            }
        }
    }
    System.out.println("Lista cand 1"+ListaCandidatos);
    if (x == true) {
        lista = null;
        lista = new ArrayList<Task>();
    } else if (!lista.isEmpty()) {
        ListaCandidatos.get(index).add(lista);
        ListaCandidatos.add(new ArrayList<List<Task>>());
        index++;
    }
    System.out.println("Lista cand 2"+ListaCandidatos);
    lista = null;
    lista = new ArrayList<Task>();
    priorityCurrent = P;
    P--;
    System.out.println(P);
    return x;
}

Task GetTrafficFlowWithPriority (List<Task> e, int P) {
    Task x = null;
    for (int i = 0; i < e.size(); i++) {
        System.out.println("fluxo "+e.get(i)+" prioridade
"+e.get(i).getPriority());
        if (e.get(i).getPriority() == P) {
            System.out.println(e.get(i));
            e.get(i).unassigned();
            x = e.get(i);
        }
    }
    return x;
}

Task GetBestCandidate (List<List<List<Task>>>
Candidatos, int P) {
    Task a = null;
    double H1, tmp = 0.0;
    System.out.println(P);
    System.out.println(Candidatos.get(P).size());
    for (int i = 0; i < Candidatos.get(P).size(); i++) {
        for (int j = 0; j < Candidatos.get(P).get(i).size(); j++)
{
            if (!Candidatos.get(P).get(i).get(j).isAssigned()) {
                H1 =
Candidatos.get(P).get(i).get(j).getDeadline() -
Candidatos.get(P).get(i).get(j).WorstCaseNetworkLatency;
                System.out.println(H1);
                if (tmp < H1) {
                    tmp = H1;
                    a = Candidatos.get(P).get(i).get(j);
                }
            }
        }
    }
    return a;
}
```