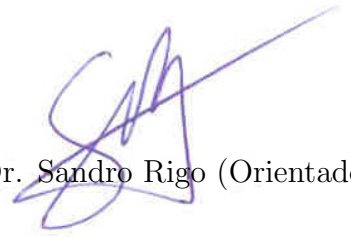
Viabilizando a Simulação *Multi-Threaded* para Modelos Escritos em SystemC

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Rodrigo Richard Cantos Faveri e aprovada pela Banca Examinadora.

Campinas, 25 de Outubro de 2010.



Prof. Dr. Sandro Rigo (Orientador)

Prof. Dr. Rodolfo Jardim de Azevedo
(Co-orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Fabiana Bezerra Müller – CRB8 / 6162

Faveri, Rodrigo Richard Cantos
F278v Viabilizando a simulação multi-threaded para modelos escritos em SystemC/Rodrigo Richard Cantos Faveri-- Campinas, [S.P. : s.n.], 2010.

Orientadores : Sandro Rigo; Rodolfo Jardim de Azevedo.
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1.SystemC. 2.Multiprocessadores. 3.Simulação (Computadores). 4.Hardware - Arquitetura. I. Rigo, Sandro. II. Azevedo, Rodolfo Jardim de. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Título em inglês: Enabling the multi-threaded simulation for models written in SystemC

Palavras-chave em inglês (Keywords): 1. SystemC. 2. Multiprocessors. 3. Computer simulation. 4. Hardware (Architecture).

Titulação: Mestre em Ciência da Computação

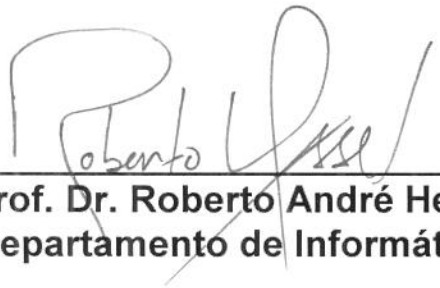
Banca examinadora: Prof. Dr. Sandro Rigo (IC – UNICAMP)
Prof. Dr. Roberto André Hexsel (Depto. Informática - UFPR)
Prof. Dr. Paulo Cesar Centoducatte (IC - UNICAMP)

Data da defesa: 25/10/2010

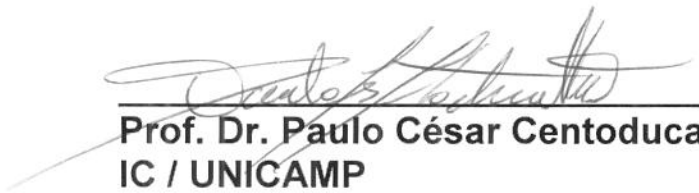
Programa de Pós-Graduação: Doutorado em Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 25 de outubro de 2010, pela Banca examinadora composta pelos Professores Doutores:



Prof. Dr. Roberto André Hexsel
Departamento de Informática / UFPR



Prof. Dr. Paulo César Centoducatte
IC / UNICAMP

Prof. Dr. Sandro Rigo
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Viabilizando a Simulação *Multi-Threaded* para Modelos Escritos em SystemC

Rodrigo Richard Cantos Faveri[1]

Outubro de 2010

Banca Examinadora:

- Prof. Dr. Sandro Rigo (Orientador)
- Prof. Dr. Paulo Cesar Centoducatte
  Instituto de Computação – UNICAMP
- Prof. Dr. Roberto A. Hexsel
  Universidade Federal do Paraná – UFPR
- Prof. Dr. Mário Lúcio Côrtes (Suplente)
  Instituto de Computação – UNICAMP
- Prof.ª Dr.ª Nahri Balesdent Moreano (Suplente)
  Universidade Federal de Mato Grosso do Sul – UFMS

[1]Suporte financeiro de: Bolsa da Fundação de Amparo à Pesquisa do Estado de São Paulo - FAPESP (processo 2008/07810-0)

Resumo

SystemC é uma linguagem de desenvolvimento de sistemas de hardware como, por exemplo, os modelos arquiteturais SoC (*Systems-on-Chip*) e, em conjunto com a biblioteca e metodologia TLM (*Transaction Level Modeling*), oferece a infraestrutura de simulação necessária capaz de realizar a simulação de software e hardware rapidamente em um alto nível de abstração. O seu núcleo de simulação foi construído como uma cadeia de *threads*, que são executadas uma por vez. Sendo assim, essa modelagem do núcleo de simulação do SystemC não é capaz de se beneficiar dos recursos oferecidos pelos novos processadores com mais de um núcleo de processamento, para obter ganhos de desempenho de simulação. Com o aumento da complexidade dos projetos de circuitos eletrônicos e a diminuição dos prazos para que um produto de SoC se torne comercial, o desempenho das simulações se tornou essencial. No presente trabalho, apresenta uma nova versão do SystemC capaz de executar em processadores multinúcleos com ganhos de desempenho de 2,0× à 22,029× à versão original em máquinas de 4 e 12 núcleos de processamento simulando plataformas contendo de 4 a 64 *threads*. Além disso, também foram realizadas mudanças nas interfaces TLM, para que a sincronização dos processos paralelos seja independente dos eventos hoje presentes no SystemC e, devido às alterações no núcleo de simulação do SystemC, a linguagem de descrição de arquitetura ArchC também foi adaptada para conseguir executar em um ambiente paralelo de simulação.

Abstract

SystemC is a modeling language for development of hardware systems, such SoCs (Systems-on-Chip) architectural models, and integrated with the methodology and library TLM (Transaction Level Modeling), it offers the required simulation platform infrastructure capable to simulate software and hardware in a fast way at different abstration levels. However, its single thread simulation kernel prevents it from utilizing the potential computing power of multi-core machines to speed up the simulation. With the complexity and the functionality of new circuits and applications size increasing and the time-to-market becoming shorter, the simulation speed-up is essential. In the present work, we introduce a new SystemC version, able to perform in multi-core machines and, consequently, with performance gains of 2.0× to 22.029× to the original version on machines with 4 and 12 cores simulating platforms with 4 to 64 threads. Furthermore, changes were made on the TLM interfaces for parallel process can synchronize independently of SystemC events, and because the changes in the SystemC simulation kernel, Archc also had to be adapted for execute in a parallel simulation environment.

Agradecimentos

Desenvolver um trabalho de pesquisa e escrever uma dissertação de mestrado, apesar do tempo que se passa sozinho em busca de soluções para os melhores resultados, é um processo que envolve colaboração. Muitas pessoas deixaram suas impressões digitais espalhadas por todo esse trabalho. Primeiramente, gostaria de agradecer ao meu orientador Prof. Dr. Sandro Rigo, pela paciência dispensada, pelo auxílio em todos os momentos do mestrado, ao meu co-orientador Prof. Dr. Rodolfo Azevedo, pelas sugestões de grande valor.

Em seguida, tenho a honra de agradecer ao Instituto de Computação e a Unicamp, por fazer parte dessa incrível comunidade acadêmica. Ao CNPq e à Fapesp, por todo o auxílio financeiro oferecido durante a execução deste trabalho.

Aos colegas de trabalho do LSC, Leonardo Piga, João, Felipe Klein, Rafael Auler, Alexandro Baldassin, Leonardo Ecco, Raoni, George, Daniel Nicácio, pela amizade, churrascos, kart e principalmente pelas ideias compartilhas, que muito ajudaram na execução deste trabalho. Fica aqui o meu agradecimento ao Bruno Albertini, pela ajuda ao decorrer do trabalho.

Aos meus amigos espalhados por todo o IC, em especial Douglas, Thiago, Maria, Daniel, Willian, Tripodi, Cristianno, Isaura, Robson e Marcos que, mais do que amigos, foram, praticamente, uma família para mim nesses anos de mestrado em Campinas.

Um agradecimento muito especial à minha família, que sempre apoiou minha ideia de vir tão longe realizar os meus sonhos. Um carinho especial aos meus avós: José, Euda, Alzira e Antonio. A minha irmã Juliana e seu esposo Ossival, por sempre estarem por perto nos bons e maus momentos e, principalmente, por fazerem parte da minha vida.

Dedico este trabalho aos meus pais Laercio e Clelia.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Com o aumento da frequência de relógio dos processadores se tornando tecnologicamente inviável e o número de transistores por circuito integrado crescendo, os desenvolvedores de arquiteturas de microprocessadores estão rapidamente adotando uma nova organização baseada na tecnologia *multi-cores*, em que dentro de um único circuito integrado é possível encontrar mais de um núcleo de processamento. Essa tecnologia abre novos caminhos para a realização computação realmente paralela, explorando o paralelismo no nível de *threads*, em que múltiplas partes de um programa são realmente executadas na mesma porção de tempo, além de colaborar com a redução do consumo de energia do circuito integrado e, consequentemente, reduzir a temperatura. Aplicações originalmente escritas de forma sequencial não podem se beneficiar dos núcleos extras, pois apresentam uma única *thread* de execução. Isto faz com que a melhor utilização dos recursos oferecidos pelos processadores multinúcleos dependa diretamente do avanço das tecnologias dos sistemas de software, como os compiladores e o sistema de execução, para extrair as *threads*, além de novas técnicas de programação [11, 29].

O constante aumento da capacidade de integração de componentes eletrônicos, representado pelos *Systems-on-Chip* (SoCs), circuitos capazes de integrar uma coleção de sinais, conversor de sinais, armazenamento de dados, processamento de sinais e funcionalidades de entrada/saída (I/O) dentro de um único chip, tem sido um dos responsáveis pelo aumento da complexidade dos projetos de circuitos integrados. Porém, a medida que essa complexidade dos projetos de circuitos integrados aumenta, o tempo de lançamento do produto no mercado também se torna elevado, devido ao tempo de simulação dispensado pelas linguagens e plataformas de simulação, que são modeladas como uma única *thread*, não permitindo utilizar todo o potencial da computação paralela, presente nos novos processadores multinúcleos[1].

SystemC [13, 22] é uma linguagem de desenvolvimento de sistemas de hardware em

[1]Do inglês *multi-core*

alto nível de abstração que, assim como as linguagens SpecC e System Verilog, surgiu em resposta ao aumento da complexidade citado anteriormente e tem ganho larga aceitação como um ambiente de simulação de modelos arquiteturais SoC e processadores embarcados [3], oferecendo a infraestrutura de plataforma de simulação capaz de realizar a simulação de software e hardware de maneira eficiente em um alto nível de abstração. O núcleo de simulação do SystemC foi construído como uma cadeia de *threads*, que são executadas de forma sequencial. O resultado é um núcleo sólido e estritamente determinístico, incapaz de tirar proveito da existência dos múltiplos núcleos de processamento existentes nos novos processadores. Esse efeito é agravado quando as aplicações a serem simuladas se tornam mais complexas.

Transaction Level Modeling (TLM) [12, 23] é uma metodologia de modelagem de sistemas de hardware baseada em transações em que, dentro de um sistema, a comunicação é separada da computação. Este tipo de modelagem é bem suportada em linguagens de alto-nível, como o SystemC. TLM permite que a precisão e o escopo do modelo sejam conhecidos e bem definidos em diferentes níveis de abstração, abrangendo todos os níveis intermediários entre o software puro e a descrição detalhada *Register Transfer Level* (RTL), favorecendo o reuso de componentes.

Baseada na metodologia TLM, a biblioteca SystemC TLM, tem como objetivos principais a definição de um conjunto de interfaces de comunicação que possa ser mapeado com clareza e eficiência tanto para hardware como para software, além de proporcionar o reuso de *Intelectual Property* (IPs), favorecendo ainda mais as metodologias de projetos de SoCs e sistemas semelhantes.

Recentemente foi lançada a biblioteca TLM-2 [23], a qual novos recursos foram incorporados à modelagem baseada em transações, tornando-a uma interface padrão para SystemC, já que esta oferece o ambiente necessário para a análise da arquitetura, análise do desenvolvimento, análise de desempenho do software e verificação de hardware. Além de oferecer maior interoperabilidade, velocidade e flexibilidade na modelagem, a TLM-2 apresenta três estilos diferentes de modelagem (ou codificação) em relação ao tempo, oferecendo diferentes relações de custo-benefício entre precisão de simulação e velocidade. Além disso, a biblioteca TLM-2 consiste de um conjunto de interfaces, *sockets*, *generic payload* e protocolo base, utilitários, interfaces de análise e portas, e as interfaces TLM-1.

Apesar das facilidades que o SystemC e o TLM proporcionam à modelagem de sistemas de hardware, certas etapas de projeto de um SoC continuam sendo bastante penosas, destacando-se a modelagem de microprocessadores e arquiteturas programáveis, partes fundamentais desse tipo de sistema [27].

A fim de resolver o problema de modelagem de arquiteturas programáveis, foram desenvolvidas as linguagens de descrição de arquiteturas (*Architecture Description Language* - ADLs), como a linguagem ArchC [28], desenvolvida no Laboratório de Sistemas de Com-

putação (LSC) do Instituto de Computação (IC) da Universidade Estadual de Campinas (UNICAMP). ArchC permite a criação de um modelo de processador em alto nível de abstração, gerando um simulador para ele baseado na linguagem e abstração do SystemC. Possibilitando assim a integração entre o modelo ArchC e modelos SoC e outros sistemas.

Visando o tempo de simulação necessário para realizar a simulação desses modelos de SoCs, a proposta do trabalho apresentado nesta dissertação é uma extensão do SystemC capaz de tirar proveito dos novos processadores multinúcleos, a favor da melhoria de desempenho, permitindo que seus processos sejam executados em paralelo. Por ser uma biblioteca muito utilizada na modelagem de sistemas, as alterações em seu núcleo de simulação não podem ser codificadas de maneira que exija grandes alterações em seu código. Por esse motivo, utilizar os recursos já oferecidos pela biblioteca do SystemC, na busca pelo paralelismo e, contornar os paradigmas existentes em seu núcleo de simulação se tornam grandes desafios.

Outro foco do trabalho é utilizar a biblioteca TLM como um novo meio de manter a sincronização da simulação do modelo, como uma primeira etapa na tentativa de contornar a utilização da sincronização por eventos, presente atualmente no SystemC.

Alterar o SystemC e o TLM, consequentemente, exige que adaptações na linguagem ArchC sejam feitas. Essas mudanças são direcionadas à geração dos simuladores, para que possam trabalhar com processadores multinúcleos e à comunicação em um ambiente paralelo de simulação que precisa ser especializada, uma vez que os eventos não se fazem mais presentes na sincronização das *threads* que simulam seus comportamentos.

1.1 Organização do Documento

Esta dissertação está organizada da seguinte maneira:

Capítulo 2: Este capítulo faz uma compilação dos conceitos, ferramentas e linguagens utilizadas para a realização deste trabalho e uma relação dos trabalhos acadêmicos relacionados.

Capítulo 3: Discorre detalhadamente sobre este trabalho, apresentado o novo modelo de execução do núcleo de simulação e as principais estruturas de dados envolvidas.

Capítulo 4: Descreve a plataforma de testes utilizada e os programas executados, além de apresentar os principais resultados obtidos nas simulações.

Capítulo 5: Contribuições deste trabalho, conclusões que podem ser feitas até o momento e sugestões de continuidade.

Capítulo 2

Conceitos Básicos e Trabalhos Relacionados

Este capítulo aborda conceitos, ferramentas e linguagens utilizados para a elaboração do presente trabalho. A seção 2.1 contextualiza o TLM; a seção 2.2 apresenta o SystemC em suas principais estruturas para o trabalho; a modelagem de sistemas em SystemC com TLM é apresentada na seção 2.3; na seção 2.4, são apresentados alguns dos recursos mais interessantes da nova biblioteca TLM-2; na seção 2.5 é apresentada a linguagem ArchC e suas ferramentas; finalmente, na seção 2.6 os trabalhos relacionados são apresentados.

2.1 *Transaction Level Modeling* (TLM)

Com o crescente aumento da complexidade dos projetos de sistemas, como os SoCs (*System-on-Chip*), e a pressão para que projetos se tornem funcionais o mais rápido possível, faz-se necessário o aumento do nível de abstração da modelagem de modo a ter uma visão geral do sistema desde o início.

As metodologias de projeto em nível de sistema são baseadas em criar, inicialmente, um modelo em alto nível de abstração que represente o sistema como um todo e, então, refinar esse modelo, descendo seu nível de abstração até que ele se torne a implementação de um sistema real [27].

A terminologia dos tipos de modelos utilizada nesse trabalho é baseada no padrão do OSCI TLM [23]. Os níveis *Algorithmic* e *Register Transfer Level* foram incluídos somente para fornecer contexto para os níveis entre eles e, portanto, não fazem parte do escopo do TLM. A seguir, há uma breve descrição de alguns tipos de modelos (alguns citados no presente trabalho), e seus nomes [10]:

***Algorithmic* (ALG):** Esse nível consiste na implementação, codificada totalmente em

software, do modelo comportamental da funcionalidade do sistema. Modelos implementados nesse nível, o desenvolvedor não tem preocupação com detalhes arquiteturais ou de implementação final.

***Communicating Processes* (CP).** Nesse nível, o comportamento do sistema é particionado em processos concorrentes que se comunicam ponto-a-ponto, trocando informações encapsuladas em estruturas de dados de alto-nível. Sistemas modelados nesse nível têm a arquitetura e implementação independentes entre si. Devido aos processos paralelos, há a preocupação do desenvolvedor com a arquitetura do sistema.

***Programmer's View* (PV):** Esse modelo oferece muito mais detalhes arquiteturais do que o modelo CP, especialmente quanto a comunicação. Dessa maneira, modelos de barramentos genéricos são instanciados como mecanismos de transporte e alguma arbitragem na infraestrutura de comunicação é aplicada. Segundo [27], os elementos computacionais são claramente definidos como mestres ou escravos. Portanto, a principal função do modelo PV é oferecer precisão de mapeamento de memória e de registradores, de tal forma que o modelo possa executar *drivers* de software em baixo-nível, os quais acessem dispositivos diretamente, como num sistema real.

***Cycle Accurate* (CA):** Modelos de precisão de ciclos, além da temporização precisa no nível do ciclo de *clock*, capturam detalhes microarquiteturais e, tipicamente, possuem interfaces em nível de bits, em que a comunicação é baseada em palavras do tamanho da largura dos barramentos e, portanto, não utiliza as estruturas de dados de alto-nível citadas anteriormente.

***Register Transfer Level* (RTL):** Modelos RTL refletem com precisão a arquitetura e a implementação do sistema até o nível de sinais e pinos. O comportamento existente entre as bordas do *clock* também é modelado e simulado. Modelos RTL são tipicamente usados para a síntese de hardware.

Segundo [10], modelos em nível de abstração *Register Transfer Level* (RTL) por não terem seu nível de abstração bem definido, podem não ter boa escalabilidade para apoiar a modelagem de sistemas cada vez mais complexos. Porém, em um modelo RTL é possível precisar a quantidade de detalhes que ele representa, ao contrário de um modelo em nível de abstração superior.

Na busca pela padronização de um nível de abstração, trabalhos sobre modelagem em nível de transações (TLM) [5, 10], buscam por níveis de abstração mais bem definidos, inseridos entre o puramente algorítmico e o RTL. Essa padronização faz com que a precisão e o escopo do modelo sejam conhecidos, permitindo assim que esse seja utilizado para

várias finalidades, como: o aumento de desempenho da simulação; a síntese de hardware; ou ainda a verificação e prototipagem de um sistema.

Apesar do nome, "nível de transação" não denota um único nível de detalhamento. Mais que isso, o termo refere-se a um conjunto de níveis de detalhamento, cada um variando no grau de detalhamento funcional ou temporal que representam [10].

Na modelagem em nível de transações, os detalhes de comunicação entre os componentes de computação são separados dos detalhes dos componentes de computação. Dessa maneira, é possível representar os componentes computacionais em um nível de detalhe e as estruturas de comunicação em um nível de detalhe completamente diferente.

Portanto, o TLM, como metodologia de projeto, tem por finalidades o gerenciamento da complexidade do projeto e a melhoria do fluxo do projeto, permitindo o desenvolvimento dos componentes computacionais em paralelo com as estruturas de comunicação.

A comunicação em TLM é toda modelada com o uso de canais SystemC. As requisições de transações acontecem por meio de chamadas de funções de interface (os conceitos de canais e interfaces são fornecidos na seção 2.2). As interfaces de TLM são implementadas dentro de canais, para encapsular os protocolos de comunicação e para estabelecer comunicação. Um processo simplesmente precisa acessar essas interfaces através das portas locais dos módulos [12].

A modelagem com TLM é incremental e, como uma modelagem em nível de sistema, tem inicialmente um modelo em alto nível de abstração escondendo detalhes desnecessários de um determinado módulo, adicionando-os conforme a necessidade para o desenvolvimento do modelo como um todo.

Apresentadas as características básicas do TLM, é preciso definir os tipos de modelagem de TLM possíveis e os níveis de abstração de cada um desses. Como explicado anteriormente, a abordagem mais comum de modelagem atualmente é considerar níveis de abstração separados para computação e comunicação.

No gráfico apresentado na figura 2.1, adaptada de [5], o eixo X representa a computação e o eixo Y representa a comunicação. Cada eixo tem três níveis de detalhamento de temporização em que os domínios podem estar: não-temporizado (UT - *untimed*), aproximadamente temporizado (AT - *approximately timed*) e temporizado com precisão de ciclos (CA - *cycle-accurated* ou CT - *cycle-timed*).

Não-temporizado diz respeito aos modelos puramente funcionais sem detalhes de implementação. Aproximadamente temporizado contém implementações de detalhes em nível de sistema, como o mapeamento das relações entre os processos da especificação do sistema e os elementos processantes da arquitetura do sistema. O tempo de execução é estimado em nível de sistema sem precisão de ciclos RTL. Temporizado com precisão de ciclos contém detalhes de implementação tanto na comunicação quanto na computação e em nível RTL, de tal forma que a estimativa da precisão de ciclos pode ser obtida.

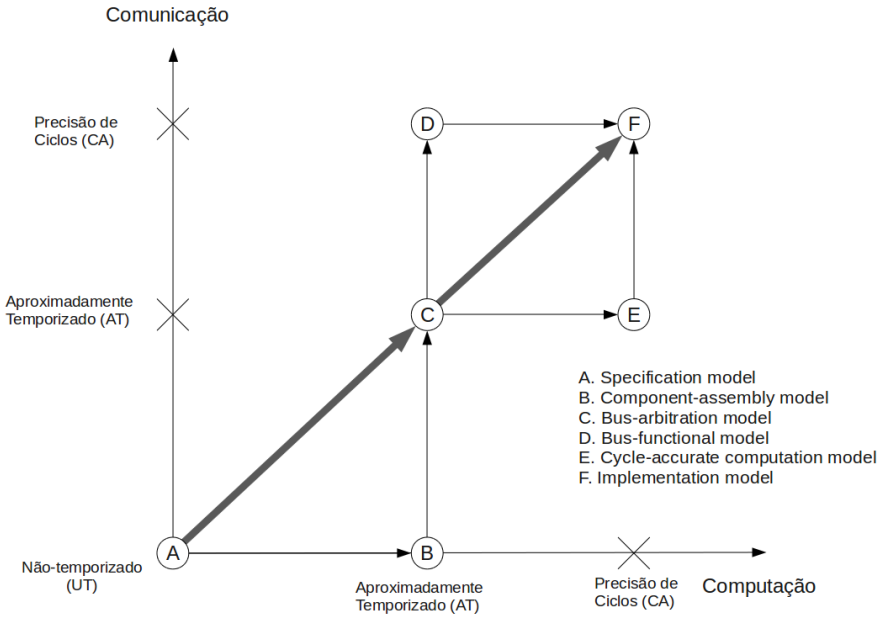


Figura 2.1: Grafo de Modelagem de Sistemas.

A composição de um desses níveis de detalhe no domínio da computação, com outro nível de detalhe no domínio da comunicação dá origem aos vários tipos de TLM. Portanto, existem nove tipos possíveis de modelagem TLM sob essa abordagem. Segundo [5], alguns tipos de modelagem são um tanto incomuns ou pouco úteis e, devido a isso, não foram incluídos no grafo.

A seta preta representa o fluxo de desenvolvimento. Essa seta passa através dos modelos A, C e F, e representa a funcionalidade do sistema, abstração da arquitetura do sistema e a implementação da precisão de ciclo do sistema, respectivamente. Geralmente, diferentes fluxos de modelagem incluem diferentes modelos. Por exemplo, um fluxo pode passar pelos modelos A, B, D e F, um outro modelo pode passar pelos modelos A, C, E e F ou então por A, B, C, D e F.

Além da nomenclatura para os principais modelos utilizados, o trabalho apresentado em [5] também define uma nomenclatura para os cinco domínios de projeto nos quais o uso de TLM é relevante, sendo eles [27]:

Domínio de Modelagem: Diz respeito à criação de modelos em alto nível de abstração, modelos em nível de sistema, cujo detalhamento seja apenas o suficiente para considerar um sistema como um conjunto de módulos e ter uma ideia básica de qual a semântica desses módulos, além de quais deles se comunicam entre si. O tipo de modelo que normalmente lida com esse domínio é o *Specification Model*, no qual

há somente uma estrutura geral do sistema sem detalhes de implementação, e a computação e a comunicação não são temporizados.

Domínio de Validação: Afirma que o modelo representa fielmente as propriedades do sistema. Nesse domínio, usa-se TLM para verificar a corretude e fidelidade de uma implementação ou modelo detalhado em relação às suas especificações ou mesmo etapas anteriores de projeto que tenham usado TLMs. A corretude do modelo pode ser validada por diferentes métodos como a cosimulação ou a verificação formal. Além disso, se o componente que se deseja validar for modelado em nível de precisão de ciclos e o resto dos componentes do sistema forem modelados em nível aproximadamente temporizado, o tempo de simulação pode ser menor do que o tempo necessário para simular um modelo puramente em precisão de ciclos, gerando ganho de desempenho.

Domínio de Refinamento: Toda vez que um novo detalhe é adicionado à modelagem, o modelo original deve ser reescrito ou refinado, para que o novo detalhe seja incluído. O domínio de refinamento refere-se ao uso de TLM no fluxo de projeto de um sistema. Parte-se de um modelo funcional, que implemente o sistema, puramente usando software, refinando-o primeiramente para um TLM básico como o *Specification Model* com a estrutura desse sistema, conduzindo-o posteriormente por diversas etapas de refinamento desse TLM, adicionando temporização a ele, até obter um *Implementation Model* que, possuindo precisão de ciclos, poderá ser refinado para RTL e daí por diante. Esse fluxo pode ser melhor visualizado no grafo da figura 2.1.

Domínio de Exploração: Corresponde a etapa de desenvolvimento cuja finalidade é estudar as características de um sistema, como desempenho, consumo de energia, etc, permitindo com que os desenvolvedores possam tomar as melhores decisões de projeto. Diferentes TLMs requerem diferentes fontes de estimativas. Por exemplo, para um elemento processante em *Component-assembly model* (B) é necessário estimar o tempo de computação, para canais abstratos de barramentos em *Bus-arbitration model* (C) estima-se o tempo aproximado de comunicação e o tempo de comunicação em precisão de ciclos para canais detalhados de barramento em *Bus-functional model* (D). Explorar a arquitetura no *bus-arbitration model*, em que a computação e a comunicação são aproximadamente temporizadas, pode resultar em melhoria de desempenho da simulação e aumento do espaço de exploração do projeto.

Domínio de Síntese. Nessa modalidade, realiza-se o refinamento dos modelos automaticamente, através de ferramentas específicas. Essas ferramentas devem ser capazes de produzir soluções ideais de refinamento do projeto, seja nos seus componentes

computacionais ou nas suas estruturas de comunicação, para as dadas restrições e métricas de otimização.

Na próxima seção é apresentado o SystemC, uma linguagem de modelagem de sistemas de hardware em software, e seus principais recursos para a realização do presente trabalho.

2.2 SystemC

SystemC [4, 13, 22] é uma linguagem de modelagem que tem evoluído em resposta a uma crescente necessidade por um sistema que melhore a produtividade para os desenvolvedores de projetos eletrônicos.

SystemC é uma biblioteca de classes em C++ que permite ao projetista criar modelos de precisão de ciclo para software, arquitetura de hardware, interfaces para SoC (*System On Chip*) etc. Essa biblioteca, juntamente com o compilador C++, pode ser usada para desenvolver modelos do sistema desejado em nível de transação, realizar a simulação e otimização do projeto, e permitir o desenvolvimento de hardware e software com uma especificação executável do sistema [8]. Uma especificação executável é essencialmente um programa C++ que exibe o mesmo comportamento do sistema.

A biblioteca de classes do SystemC oferece os construtores necessários para a modelagem da arquitetura de um sistema, incluindo temporização de hardware, concorrência e comportamento reativo. O SystemC possui diversas estruturas que facilitam a descrição de hardware, software e interfaces em C++, tais como: módulos, processos, portas, sinais, frequência de relógio etc.

Em sistemas complexos, muitas funcionalidades são proporcionadas por componentes independentes. Os módulos são utilizados para representar estes componentes. Podemos entender um módulo como um contêiner de funcionalidades, com estado, comportamento e uma estrutura hierárquica. Um processo é a unidade básica de execução. Desde o início da simulação até o seu término, todo código em execução é inicializado por um ou mais processos. Por definição, processo é uma função ou um método de classe de um módulo que é invocado pelo escalonador no núcleo do SystemC [4].

O diagrama da figura 2.2 ilustra os principais componentes do SystemC. As subseções a seguir descrevem os componentes e estruturas mais relevantes para a realização do trabalho proposto.

2.2.1 *Threads* e Métodos

Os processos de simulação são funções de módulos (SC_MODULE) que simulam o comportamento de um dispositivo ou sistema. Esses processos podem ser descritos como

Figura 2.2: Arquitetura do SystemC

métodos (SC_METHOD), *threads* (SC_THREAD) e *clocked threads* ($SC\_CTHREAD$). Cada um desses três tipos tem um comportamento diferente.

Um método (SC_METHOD) é o tipo mais básico de processo, pois é simplesmente uma função membro de uma classe do tipo módulo, que é constantemente chamada pelo núcleo do simulador, não havendo a presença de argumentos de chamada da função e muito menos um valor de retorno. Processo do tipo SC_METHOD somente é sincronizado através de eventos externos (*sc_event*), definidos em sua lista de sensibilidade [4].

Outro tipo básico de processo é a *thread* (SC_THREAD) que, diferentemente do método, é invocada pelo núcleo de simulação somente uma vez e pode ter sua execução suspensa ou reativada. Uma vez inicializada a execução de uma *thread*, ela retém todo o controle de simulação até que haja um comando de parada de execução. O SystemC oferece dois meios de parar a execução e retornar o controle ao simulador. O primeiro é um simples comando de fim de execução. Uma vez que uma *thread* é finalizada dessa forma, ela não pode mais ser reativada. Sendo assim, as *threads* tipicamente contém um laço infinito com pelo menos uma chamada a um método $wait()$. Esse método permite que o processo seja interrompido temporariamente e o controle da simulação seja passado para o núcleo do simulador, até que um evento ao qual o processo é sensível ocorra. O processo do tipo *thread* é um processo mais genérico e pode ser usado para realizar vários tipos de modelagem [4, 8].

Existe também um caso especial de processo *thread* que é conhecido por processo *clocked thread* ($SC\_CTHREAD$), pois esse é sensível somente à variação do sinal do relógio do sistema. *Clocked threads* são muito utilizadas em ferramentas de síntese comportamental. Essa popularidade, em parte, devido à lógica de tais ferramentas produzirem códigos sincronizados e, também, pelas simplificações que esses processos proporcionam à codificação [8].

2.2.2 Canais e Interfaces

A hierarquia de módulos, presente no SystemC, sem uma forma de comunicação entre eles, se torna algo inútil. Portas em SystemC (*sc_port*) oferecem um meio dos módulos acessarem os canais de comunicação além de suas fronteiras. Uma porta encaminha as chamadas ao método da interface do canal ao qual a porta está vinculada. Esta interface define um conjunto de serviços que são requisitados pelo módulo que a contém [4, 22].

Quando há uma chamada a uma função membro pertencente ao canal instanciado dentro de um módulo-filho, esta chamada pode ser feita diretamente, sem a utilização de canais, utilizando somente um tipo especializado de porta, chamada de *sc_export* [4].

Uma porta do tipo *sc_export* tem sua sintaxe de declaração semelhante às portas do tipo *sc_port*, mas diferem na conectividade. A utilização de uma porta do tipo *sc_export* faz com que um módulo forneça uma interface para seu módulo pai, movendo o canal para dentro do módulo e utilizando a porta externamente, como se fosse um canal. Fornecer uma interface através deste tipo de porta é uma alternativa para simplificar a implementação de interfaces. O uso explícito de portas *sc_export* permite que uma simples instância de um módulo forneça múltiplas interfaces de maneira estruturada [4, 22].

```
1 SC_MODULE (my_Module1){
2         sc_port<sc_fifo_in_if<float>>    a_in;
3         sc_export<sc_fifo_in_if<float>> b_in
4 };
```

Figura 2.3: Exemplo de interconexão de portas (1 de 3)

```
1 SC_MODULE (my_Module2){
2         sc_port<sc_fifo_out_if<float>>   a_out;
3         sc_port<sc_fifo_out_if<float>>   b_out;
4 };
```

Figura 2.4: Exemplo de interconexão de portas (2 de 3)

As figuras 2.3, 2.4 e 2.5 exemplificam a diferença de utilização entre *sc_port* e *sc_export*. Na declaração dos módulos (figuras 2.3, 2.4), todas as portas implementam um tipo de interface ***sc_fifo_if***, seja ela de entrada ou saída. Na figura 2.3 é declarada uma porta do tipo *sc_export*, ***b_in***. A diferença de utilização fica clara na figura 2.5. Para conectar as *sc_ports* ***a_in*** e ***a_out*** foi criado um canal do tipo *sc_fifo* (linha 9). Nas linhas 12 e 13 esse canal é associado a ambas as portas, para enfim, estabelecer a comunicação entre os módulos. Utilizar *sc_export* elimina a necessidade da utilização explícita de um canal, devido a própria natureza da porta, na linha 16, pode-se observar como acontece a conexão entre as portas ***b_in*** e ***b_out*** sem a necessidade do canal.

```
1  CONNECT::CONNECT( sc_module_name nm)
2  : sc_module(nm)
3  {
4          //Instantiate
5          my_Module1 my_Module1_i("my_Module1_i");
6          my_Module2 my_Module2_i("my_Module2_i");
7
8          //Channel
9          sc_fifo<float> my_channel;
10
11         //Connecting via channel
12         my_Module1_i.a_in(my_channel);
13         my_Module2_i.a_out(my_channel);
14
15         //Connecting via sc_export
16         my_Module2_i.b_out(my_Module1_i.b_in);
17 };
```

Figura 2.5: Exemplo de interconexão de portas (3 de 3)

Em C++ é definido o conceito de classe abstrata. Uma classe abstrata é uma classe que nunca é usada diretamente, mas é utilizada somente através de classes derivadas. Em geral, classes abstratas contém somente funções puramente virtuais, ou seja, essas funções não contém as respectivas implementações. Portanto, a classe derivada deve fornecer uma implementação para todas as funções puramente virtuais. Se uma classe não contém dados e, somente, possui métodos puramente virtuais, ela é denominada como sendo uma interface. Uma interface SystemC é uma classe abstrata derivada de uma classe *sc_interface*, mas não derivada de uma classe *sc_object*. Portanto, uma interface contém um conjunto de métodos puramente virtuais que devem ser definidos em um ou mais canais, derivados dessa interface [4, 22].

O SystemC oferece uma variedade de interfaces padrão que podem ser utilizadas para a construção de canais de comunicação entre módulos. Essas interfaces são compostas basicamente de métodos de leitura e escrita. Podem ser bloqueantes ou não-bloqueantes, bem como unidirecionais ou bidirecionais. As interfaces padrão do SystemC podem ser do tipo *sc_fifo*, *sc_signal*, *sc_mutex* e *sc_semaphore*.

O código da figura 2.6 é um exemplo de interface extraída da biblioteca do SystemC. Esse código se refere à uma interface FIFO não-bloqueante, que oferece dois métodos: o primeiro, é chamado de ***nb_read()*** e permite que um dado seja lido da FIFO; o segundo é o método ***data_written_event()***, que pode ser utilizado para dinamicamente esperar até que um novo valor esteja disponível na FIFO. Ainda neste texto, é possível encontrar mais discussões sobre interfaces. Porém, mais detalhes sobre a utilização de interfaces podem ser encontrados em [4].

```
1  template <class T>
2  class sc_fifo_nonblocking_in_if
3  : virtual public sc_interface
4  {
5  public:
6
7      // non-blocking read
8      virtual bool nb_read( T& ) = 0;
9
10     // get the data written event
11     virtual const sc_event& data_written_event() const = 0;
12 };
```

Figura 2.6: Exemplo de interface SystemC fifo

2.2.3 Núcleo de Simulação

Muitas atividades em um sistema real ocorrem ao mesmo tempo, ou concorrentemente. Devido a isso, o SystemC utiliza processos para modelar concorrência. Como em muitos simuladores dirigidos por eventos, a concorrência não é uma execução concorrente real, mas sim uma concorrência emulada como um sistema multitarefas (*multi-tasking*). Em outras palavras, cada processo no simulador executa um pequeno pedaço de código e, então, voluntariamente cede o controle à execução de outro processo no mesmo espaço de tempo de simulação [4].

O núcleo de simulação é responsável pela inicialização dos processos e gerenciamento de quais serão os próximos processos a serem executados. Devido à natureza cooperativa do modelo do simulador, os processos são responsáveis por suspender a si mesmos e permitir a execução de outros processos concorrentes. O simulador do SystemC possui três fases principais de operação: elaboração, inicialização e simulação. A fase de simulação ainda pode ser subdividida em três outras fases: avaliação (ou execução), atualização e avanço do tempo. Uma quarta fase principal ocorre no fim da execução, e pode ser caracterizada como um pós-processamento ou limpeza.

Durante a elaboração, os módulos do SystemC são construídos e diversos parâmetros de simulação são estabelecidos. A fase de elaboração é seguida por uma chamada à função *sc_start*(), que invoca o núcleo de simulação. Essa chamada realiza a fase de inicialização. Os processos definidos durante a fase de elaboração precisam ser inicializados e, durante a fase de inicialização, todos os processos são armazenados em uma fila de processos prontos para a execução [4, 8].

Uma vez realizada a inicialização dos processos, o núcleo avança para a fase de simulação. A figura 2.7 [16] ilustra o diagrama de fluxo do escalonador do SystemC.

Segundo [6, 16, 24], inicialmente, todos os processos são executáveis. Mesmo que os processos sejam executados como POSIX *pthreads*, todos eles são executados sequencialmente. Se algum evento imediato ocorre, então todos os processos sensíveis a esse evento

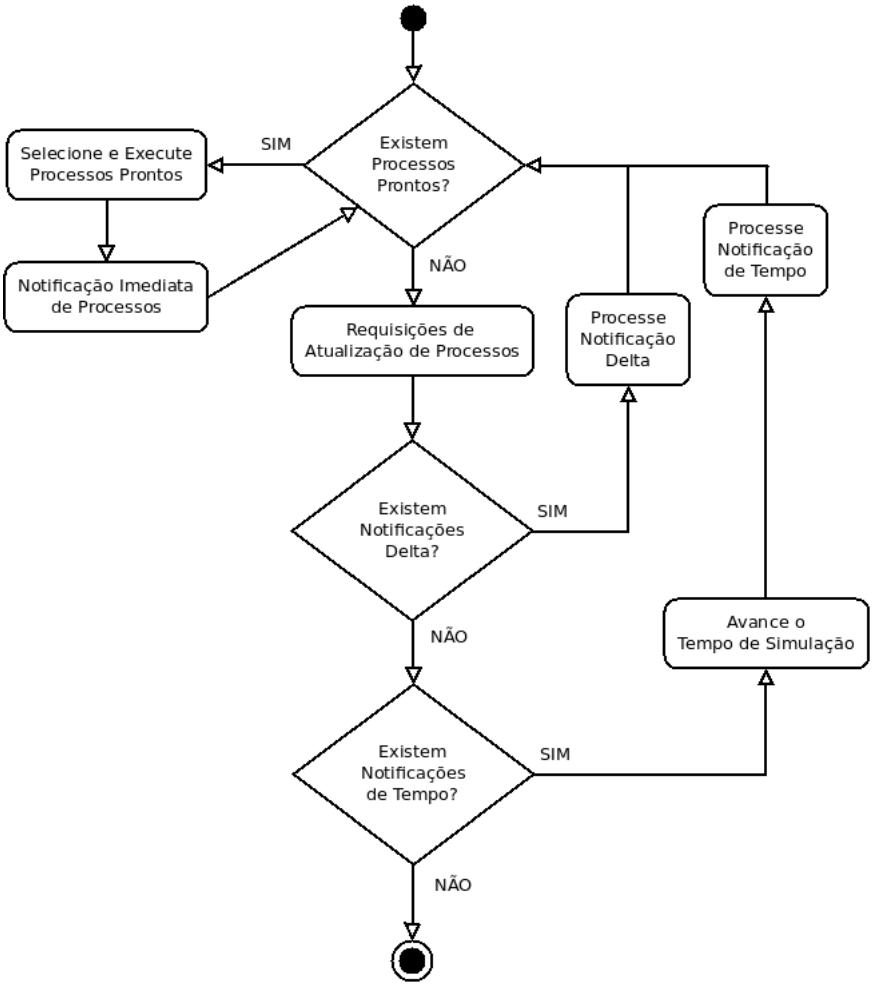


Figura 2.7: Escalonador do SystemC

se tornam imediatamente executáveis. A execução de processos e o engatilhamento de eventos imediatos constituem a fase de avaliação (*evaluation phase*). Essa fase é executada até que não existam mais processos executáveis.

Após a fase de avaliação, o escalonador executa todos os pedidos de atualização. Esse estágio é chamado de fase de atualização (*update phase*). A simulação do SystemC obedece ao paradigma de avaliação e atualização em que os processos prontos para executar são executados e, somente então, seus sinais de saída são atualizados [8]. As requisições de atualização são geradas na fase de avaliação quando um processo escreve um novo valor em um canal. A atualização do canal pode gerar eventos delta ou eventos de tempo. Uma vez que, todo o processo de avanço do escalonador da fase de avaliação para a fase de atualização é chamado de ciclo delta. Então, o paradigma de avaliação e atualização faz com que os processos não considerem novos valores imediatamente, antes do próximo

ciclo delta.

Depois que todas as requisições de atualização são processadas, o escalonador avança para a fase de notificação delta. Durante essa fase, todos os processos sensíveis ao evento delta se tornam executáveis e a simulação retorna, novamente, à fase de avaliação. Nesse caso, diz-se que a simulação avançou em um ciclo delta.

Caso não existam mais notificações delta, o simulador avança para a fase de notificação de eventos de tempo. Nessa fase, o tempo de simulação é avançado para o primeiro evento de tempo. Todos os processos sensíveis ao atual evento de tempo se tornam executáveis. Após realizada a fase de notificação de eventos de tempo, a simulação volta à fase de avaliação. A simulação é encerrada somente quando não há mais eventos de tempo.

Na próxima seção é apresentado o SystemC TLM, em que o SystemC é integrado com uma biblioteca especialmente desenvolvida para facilitar a modelagem de sistemas em TLM.

2.3 SystemC TLM

Nas seções 2.1 e 2.2, foram apresentados separadamente os conceitos de modelagem em nível de transações (TLM) e a biblioteca SystemC para modelagem de sistemas, respectivamente. Nessa seção, é apresentado o SystemC integrado com a biblioteca para modelagem TLM. Com o objetivo de facilitar a modelagem de sistemas e facilitar a interoperabilidade entre módulos e IPs.

2.3.1 O Padrão de Modelagem SystemC TLM

O padrão de modelagem TLM em SystemC [26] tem como objetivos principais a definição de um conjunto de interfaces de comunicação que possa ser mapeado com clareza e eficiência tanto para hardware como para software, além de proporcionar o reuso de *Intelectual Property* IPs em um mesmo projeto (através de seus diversos níveis de abstração), ou entre projetos diferentes [27].

As interfaces SystemC TLM são interfaces de comunicação padronizadas, que oferecem basicamente métodos de leitura e escrita, e são capazes de facilitar a interoperabilidade entre módulos e IPs. Assim como as interfaces anteriormente citadas, as interfaces SystemC TLM podem ser bloqueantes ou não-bloqueantes, bem como unidirecionais ou bidirecionais.

A definição de interface bloqueante e interface não-bloqueante está diretamente relacionada ao comportamento de sincronização dos métodos de uma interface. Como visto na seção 2.2.1, um processo do tipo SC_THREAD pode ter sua execução suspensa por uma chamada de $wait()$, enquanto que os processos do tipo SC_METHOD somente

são sincronizados na presença de eventos externos, que devem estar relacionados em sua lista de sensibilidade. Portanto, interfaces bloqueantes são aquelas cujos métodos podem fazer chamadas de *wait*(), enquanto que as interfaces não-bloqueantes são aquelas que garantidamente não o fazem. No primeiro caso, as chamadas aos métodos da interface somente são realizadas por processos SC_THREAD, enquanto que no segundo caso, tanto SC_THREAD quanto SC_METHOD podem fazer chamadas.

As interfaces unidirecionais implementadas em SystemC TLM são baseadas no modelo de canal *sc_fifo*. Interfaces unidirecionais modelam a comunicação entre dois pontos, caracterizando o modelo produtor-consumidor. Para operar sobre esse modelo de filas, três métodos são oferecidos pela interface: *put*() para escrever; *get*() para leitura; e *peek*() que faz a leitura sem consumir o dado, mantendo-o no FIFO. Além disso, esses métodos são oferecidos para interfaces bloqueantes e não-bloqueantes, no último caso, os métodos são acompanhados do prefixo "nb_".

As interfaces bidirecionais do SystemC TLM são utilizadas para modelar transações existentes na comunicação entre dois pontos de um sistema. No modelo de comunicação bidirecional, ambos os pontos atuam tanto como produtores quanto consumidores, ou seja, o primeiro atua como o produtor, enviando uma requisição para o segundo ponto que a consome. O segundo ponto também pode atuar como produtor, enviando uma resposta a essa transação, com o primeiro ponto consumindo-a. A diferença entre eles é que o produtor (ou mestre) inicia a transação produzindo uma requisição e consome uma resposta enquanto que o consumidor (ou escravo) consome uma requisição e produz uma resposta.

A biblioteca SystemC TLM implementa duas interfaces bidirecionais mais básicas, *tlm_master_if* e *tlm_slave_if*. Essas interfaces combinam, por herança, os métodos *put* e *get* das interfaces unidirecionais. Neste caso, o dado enviado pela interface mestre, através do método *put*, é uma requisição e o dado recebido por *get* uma resposta. No escravo, o que acontece na interface é o contrário, o método *put* envia as respostas e o método *get* recebe as requisições.

Além dessas duas interfaces bidirecionais, existe uma mais interessante, chamada de *tlm_transport_if*. Essa interface combina os métodos *put*() e *get*() bloqueantes em um único método chamado *transport*(), que é capaz de enviar uma requisição como argumento e receber a resposta como valor de retorno. Esse acoplamento entre requisições e respostas, faz com que para toda requisição haja uma resposta, o que não necessariamente acontece quando se trabalha com interfaces bidirecionais mestre e escrava.

Uma ou mais das interfaces descritas anteriormente podem ser implementadas em canais ou diretamente no módulo alvo utilizando uma porta *sc_export*. A biblioteca SystemC TLM oferece três implementações de canais que podem ser utilizadas com essas interfaces.

O canal *tlm_fifo* implementa todas as interfaces unidirecionais citadas anteriormente,

com o FIFO baseado em *sc_fifo*. Um outro canal chamado de *tlm_req_rsp_channel* consiste de dois fifos, um para enviar a requisição do mestre para o escravo e outro para enviar a resposta do escravo para o mestre. Por fim, há o canal *tlm_transport_channel*, utilizado para modelar situações em que cada requisição está fortemente vinculada a uma resposta, esse canal basicamente implementa a interface *tlm_transport_if* utilizando dois FIFOs de tamanho 1.

Modelagem em SystemC TLM

As interfaces e métodos apresentados nessa seção formam a base da proposta OSCI TLM. Os usuários podem e devem desenvolver seus próprios canais implementando uma ou todas as interfaces, ou então, implementá-las diretamente no módulo escravo utilizando *sc_export*.

Em [26] é apresentada uma interessante recomendação de modelagem de comunicação intermódulos através de camadas, escondendo detalhes de comunicação entre os módulos processantes. Esse tipo de arquitetura de comunicação em camadas é utilizada quando se deseja modelar em nível de PV (do inglês *Programmers View*). Consequentemente, o modelo PV permite tanto o reuso de código quanto o refinamento do código de um módulo processante PV, de implementação abstrata, até a comunicação com componentes RTL.

Essa arquitetura de modelagem de comunicação pode ser vista em três camadas [26, 27]:

Camada do Usuário: Corresponde aos métodos de comunicação de alto nível oferecidos aos elementos processantes modelados em software. Esses métodos são do tipo *read*() e *write*() e devem ser definidos de acordo com a interface que irão compor, do modelo de comunicação proposto.

Camada de Protocolo: Diz respeito à estrutura dos pacotes de informações que irão trafegar de um módulo para outro. Esses pacotes devem ser modelados como classes ou estruturas que encapsulam as informações transmitidas, definindo o que são requisições e respostas. Junto a camada de protocolo também são encontradas classes que fazem a comunicação do módulo com a camada de transporte, do lado do módulo mestre encontra-se a classe *initiator_port*, responsável por implementar a interface definida na camada do usuário e utilizada pelo mestre, montando o pacote de requisição e o enviando através do método *transport* de uma porta de saída, assim como também é responsável pelo tratamento do pacote de resposta recebido. Do lado do módulo escravo, o método *transport*() é implementado de forma a desmontar o pacote de requisição tratando-o de acordo com seu tipo, o que geralmente implica em chamadas de método na camada do usuário.

Camada de Transporte: Refere-se à estrutura de portas, tanto *sc_port* como *sc_export*, e canais de SystemC responsáveis por passar adiante as chamadas de *transport*() pelo mestre até chegar ao escravo.

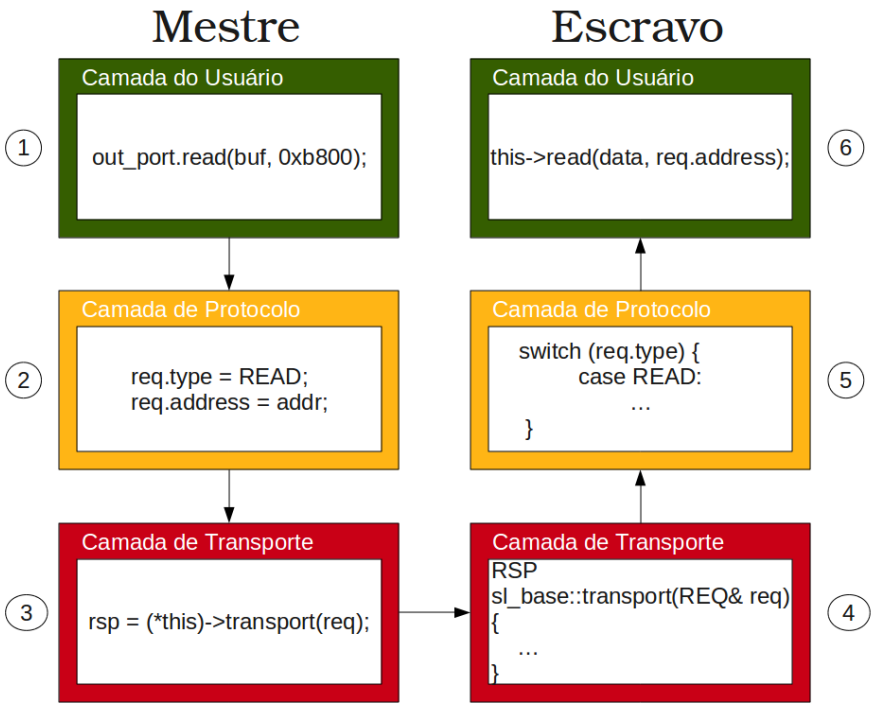


Figura 2.8: Fluxo de requisição de leitura em memória em um modelo PV.

A figura 2.8, adaptada de [27], ilustra o fluxo de uma requisição na arquitetura de comunicação em camadas TLM para uma transação de leitura de memória. A transação inicia-se quando um processo no módulo mestre invoca o método de leitura na porta de saída deste módulo (1). Esse método, que já está implementado na camada de protocolo, irá montar o pacote de requisição com o tipo de operação que será realizada na memória e o endereço de leitura dos dados (2). Feito isso, a requisição será encaminhada através da chamada do método *transport*() que, ligada à camada de transporte, irá encaminhar a requisição para o módulo escravo, neste caso a memória (3).

No módulo de memória, a implementação do método *transport*() faz parte da classe *sl_base*, que é herdada pelo módulo (4). Essa implementação faz parte da camada de protocolo e irá decodificar o pacote recebido, identificando-o como uma requisição de leitura (5). Isso faz com que o método *read*(), implementado na camada do usuário, seja disparado e o módulo de memória faça a leitura dos dados no endereço especificado na requisição (6).

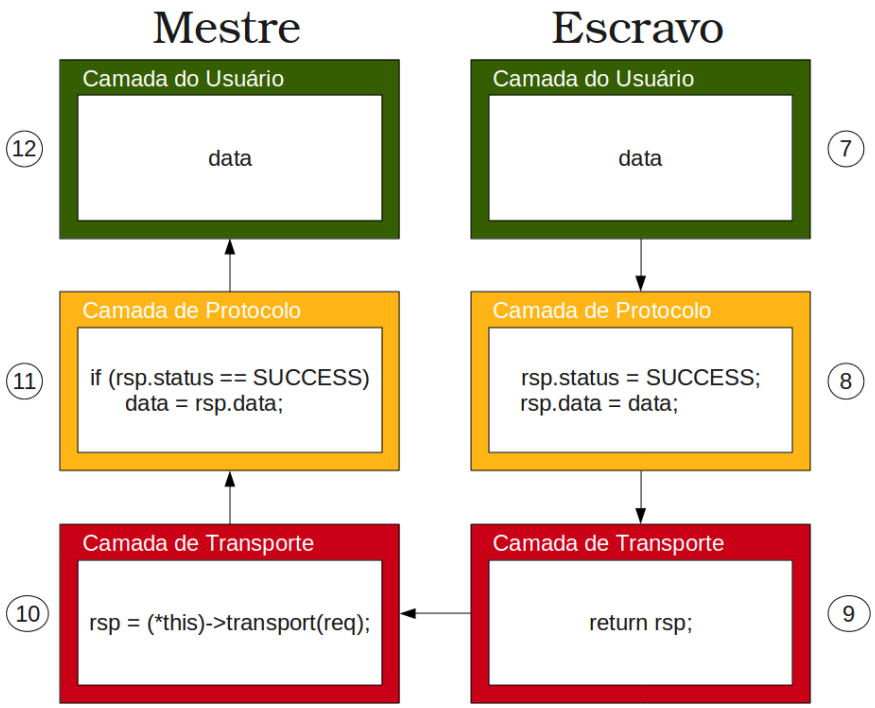


Figura 2.9: Fluxo de resposta da leitura em memória em um modelo PV.

O fluxo do retorno da resposta está ilustrado na figura 2.9. O dado lido pelo método *read*() está no *buffer data* (7). Na camada de protocolo, esse dado é então adicionado ao pacote resposta, juntamente com a sinalização de que a leitura foi realizada com sucesso (8). Em seguida, o retorno do método *transport*() devolve esse pacote de resposta como valor (9), que retorna através da camada de transporte e é armazenado em *rsp* (10), na *initiator_port* do mestre. Novamente, o pacote segue para a camada de protocolo, onde esse será decodificado pelo próprio método *read*(), que encaminhou a requisição (11). Uma vez que a leitura do dado foi realizada com sucesso, o dado pode ser armazenado no *buffer data* e encaminhado para o mestre em sua camada de usuário.

Encaminhamento de Tráfego com Roteador

Um problema muito comum, encontrado durante a modelagem de SoCs é o encaminhamento do tráfego gerado por um módulo mestre para um dos módulos escravos conectados a ele. Uma maneira fácil de codificar e de bom desempenho é a utilização de um roteador[1], presente nos padrões de projeto do SystemC TLM [26].

Para uma melhor compreensão desse padrão de projeto, o exemplo de uma transação

[1] do inglês *router*

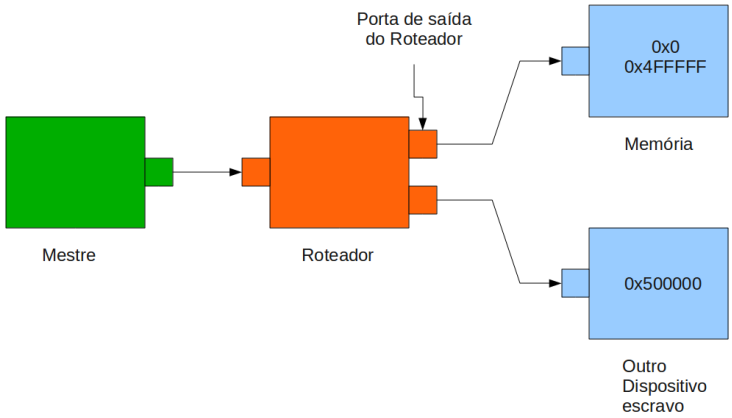


Figura 2.10: Padrão de projeto de um roteador.

de leitura em memória será mantido. O modelo implementado, ilustrado na figura 2.10, é composto pelo módulo mestre como elemento processante (programado para trabalhar com memórias de cinco megabytes), conectado à porta de entrada do roteador. Conectados à porta de saída do roteador, temos dois módulos escravos, sendo o primeiro um módulo de memória, mapeado na faixa de endereços de 0x0 à 0x4FFFFF, e um segundo módulo escravo qualquer, mapeado no endereço 0x500000.

O roteador é um módulo composto pelas camadas de protocolo e transporte. O mapa de endereçamento do roteador faz o mapeamento de uma faixa de endereços em uma ou mais portas de saída. Se o roteador não encontrar um endereço que não esteja mapeado, este retorna uma mensagem de erro do protocolo. Caso contrário, o roteador despachará a requisição através da chamada de *transport*() para o módulo escravo correto.

O processo de envio de uma requisição está ilustrado nas figuras 2.11 e 2.12. O fluxo interno ao mestre (1 a 3) permanece o mesmo. Agora, quem primeiro recebe essa requisição é o roteador. A requisição é recebida pela camada de transporte do roteador (4). A implementação do método *transport*(), como nos escravos, também faz parte da camada de protocolo e irá decodificar o pacote de requisição (5), identificando a faixa de endereços que contenha o endereço contido no pacote. Neste caso, o endereço é 0xb800, o qual está contido na faixa de endereços da memória. Uma vez identificada a porta ao qual o endereço está associado (6), o roteador irá despachar a requisição através da chamada de *transport*() para o módulo escravo de memória (7). Para, finalmente, enviar a resposta de volta ao mestre. O fluxo interno ao escravo de memória (8 a 10) também permanece o mesmo.

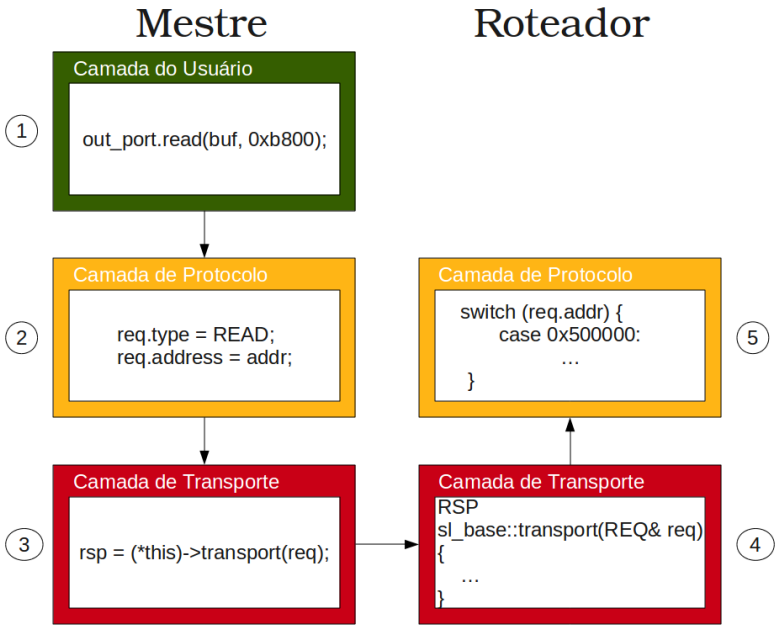


Figura 2.11: Fluxo da requisição Mestre - Roteador.

Recentemente, novos recursos de interfaces e estilos de modelagem foram incorporados à biblioteca TLM, dessas interfaces e estilos de modelagem, os mais interessantes para o trabalho são descritos na seção 2.4, a seguir.

2.4 TLM 2.0

Recentemente, a *Open SystemC Initiative* (OSCI) [22] lançou a segunda versão do TLM, chamada de TLM-2. A biblioteca TLM-2 estende a biblioteca padrão anterior OSCI TLM para um número maior de detalhes para a modelagem de SoCs. Além disso, TLM-2 apresenta três estilos diferentes de modelagem (ou codificação) em relação ao tempo, oferecendo diferentes relações de custo-benefício entre precisão de simulação e velocidade.

TLM-2 consiste de um conjunto de interfaces, *sockets*, *generic payload* e protocolo base, utilitários, interfaces de análise e portas, e as interfaces TLM-1.

A biblioteca TLM-2 reconhece diversos estilos de modelagem que podem ser utilizados. Sendo os principais estilos de modelagem da TLM-2: não-temporizado (UT - *untimed*), vagamente-temporizado (LT - *Loosely timed*) e aproximadamente temporizado (AT - *approximately timed*) [23].

O estilo de modelagem não-temporizado (UT) desconsidera o tempo. Esse estilo de modelagem não é nativo do TLM 2.0, mas é suportado pelas interfaces TLM-1 e, como

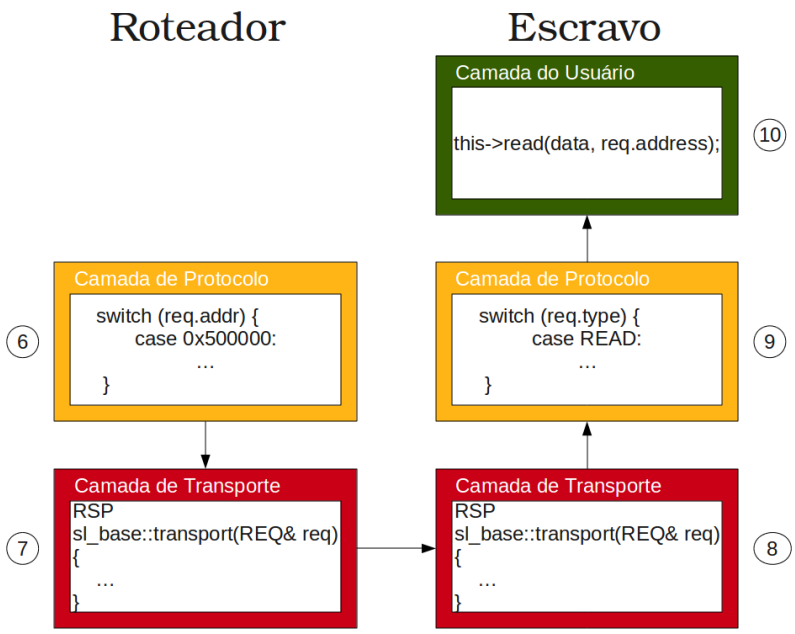


Figura 2.12: Fluxo da requisição Roteador - Escravo.

dito anteriormente, diz respeito aos modelos puramente funcionais sem detalhes de implementação.

O estilo de modelagem vagamente-temporizado (LT) surgiu recentemente no lançamento da OSCI TLM-2.0. Com essa modelagem é possível sinalizar uma transação com um tempo de atraso (*delay*) d, para marcá-la (no caso $d > 0$) como uma transação futura. Ou seja, esse estilo de modelagem permite que partes do modelo executem sob um espaço de tempo local (*time warp*) até que eles alcancem o ponto onde devem sincronizar com o resto do sistema, esse método é chamado de dissociação temporal[2]. A ideia de utilizar esse método é que as trocas de contexto no simulador sejam reduzidas e a simulação ganhe um maior desempenho [9].

Quando a dissociação temporal é implementada, um processo pode executar, de acordo com o seu tempo local, até encontrar uma dependência em uma variável atualizada por outro processo, ou então precise interagir com um outro processo. Nesse ponto, o processo é responsável por determinar se continua com a sua simulação ou retorna o controle para o núcleo de simulação. Se um processo pode executar sua simulação sem um tempo limite, então o núcleo de simulação pode se tornar inoperante e outros processos podem nunca ter a chance de executar. Para organizar isso, o TLM-2 fornece um mecanismo que impõe um limite máximo de tempo que um processo pode executar, chamado de *quantum keeper*.

[2]do inglês *temporal decoupling*

Tipicamente, um pequen *quantum* de tempo aumenta a precisão da simulação, enquanto que, um *quantums* maiores melhoram o desempenho da simulação.

O terceiro estilo de modelagem é o aproximadamente temporizado (AT). Esse estilo é utilizado para casos de exploração arquitetural e análise de desempenho do modelo. Cada transação é dividida em múltiplas fases, com uma marcação de tempo entre as fases. A modelagem aproximadamente temporizada não pode explorar o conceito de dissociação temporal, principalmente pela necessidade da precisão do tempo. Em vez disso, cada processo executa em modo *lock-step* com o escalonador do SystemC.

Com relação as novas interfaces oferecidas pela TLM-2, destacam-se a *direct memory interface* (DMI) e a *debug transport interface* (DTI). Essas interfaces especializadas são distintas das interfaces de transporte e oferecem acesso direto e acesso depurado à uma região de memória do módulo escravo, respectivamente. A interface DMI é capaz de contornar o caminho normal de múltiplas chamadas de *transport* que partem do módulo mestre, passando pelos componentes de interconexão, até o módulo escravo, fazendo com que a simulação ganhe em desempenho. Enquanto que, DTI é uma interface que fornece uma maneira de ler e escrever na memória do módulo escravo, porém, fazendo o mesmo caminho, entre mestre e escravo, utilizado pela interface de transporte, mas sem a existência de atrasos (*delays*), *waits*, notificações de eventos ou qualquer outro evento associado às interfaces de transporte. DTI foi originalmente desenvolvida para oferecer um modelo com uma ferramenta de depuração em alto nível de abstração. Por exemplo, a interface DTI permite que o mestre tenha conhecimento do conteúdo do sistema de memória durante a simulação, para fins de diagnósticos, ou para inicializar uma certa área de memória no fim da elaboração.

Tanto em DMI quanto em DTI, as transações são do tipo **tlm_generic_payload**. *Generic payload* é um novo recurso oferecido pela TLM-2 que possibilita uma maior interoperabilidade para modelos de memória mapeada em barramento.

Generic payload tanto pode garantir imediata interoperabilidade para modelos abstratos de memória mapeada em barramento - em que detalhes de protocolo não são importantes - quanto pode ser utilizado como base para modelar protocolos mais específicos, com a vantagem de reduzir o custo de implementação e melhorar o desempenho da simulação quando há a necessidade de trabalhar com diferentes protocolos em um mesmo sistema.

A tabela 2.1 [14] apresenta os atributos da classe *generic payload* com seus tipos de dados e valores padrão.

Command é uma variável do tipo *tlm_command* e pode assumir um dos três valores: TLM_READ_COMMAND, TLM_WRITE_COMMAND e TLM_IGNORE_COMMAND. *Address* armazena o endereço de memória que será lido ou escrito durante a transação. *Data pointer* é um ponteiro para o array de dados. O tamanho desse array de dados fica

Atributo	Tipo	Valor Padrão
Command	tlm_command	TLM_IGNORE_COMMAND
Address	sc_dt::uint64	0
Data pointer	unsigned char*	0
Data length	unsigned int	0
Byte enable pointer	unsigned char*	0
Byte enable length	unsigned int	0
Streaming width	unsigned int	0
DMIallowed	bool	false
Response status	t1m_response_status	TLM_INCOMPLETE_RESPONSE
Extension	pointers	0

Tabela 2.1: Atributos do *generic payload*

armazenado como um valor em *data length*. *Byte enable pointer* é um ponteiro para o array de *byte enable*. Esse array é aplicado repetitivamente no array de dados. Os valores do array são interpretados da seguinte maneira: o valor 0 indica que o byte correspondente está desabilitado, mas se o valor for 0xff, então o byte correspondente está habilitado. O tamanho desse array fica armazenado em *byte enable value*. O atributo *streaming width* indica o número de bytes que estão sendo transferidos. O atributo *dmi_allowed* determina se o recurso de DMI pode ser usado ou não. *Response status* sempre retorna um valor dizendo se a operação foi realizada com sucesso ou não. Finalmente, *extension pointer(s)* serve para facilitar a inclusão de extensões de qualquer tipo para o objeto *generic payload*.

A seção à seguir descreve o ArchC, uma linguagem para descrição e geração de simuladores executáveis de processadores, que utiliza o TLM como interface de comunicação com os demais módulos SystemC de um sistema.

2.5 ArchC

Esta seção apresenta o ArchC [28], uma linguagem de descrição de arquitetura (ADL), desenvolvida no Laboratório de Sistemas de Computação (LSC) do Instituto de Computação (IC) da Universidade Estadual de Campinas (UNICAMP).

ArchC tem sua sintaxe totalmente baseada em C++ e SystemC, assim facilitando o uso e aprendizado por parte daqueles projetistas já familiarizados com SystemC. Além disso, ArchC é classificado como uma ADL mista, pois seus modelos são compostos tanto por descrições estruturais (elementos arquiteturais, componentes internos) como comportamentais (informações gerais sobre o conjunto de instruções e comportamentos de instruções).

2.5.1 Sintaxe e Semântica

Um modelo ArchC é dividido em duas partes: a descrição do conjunto de instruções (AC_ISA) e a descrição de elementos arquiteturais (AC_ARCH). Na descrição AC_ISA, são declarados os detalhes de cada instrução, como: (a) formato, tamanho e sintaxe assembly; (b) opcodes necessários para decodificá-las; (c) o comportamento de cada instrução. A descrição de AC_ARCH contém a declaração de elementos arquiteturais de memórias internas, *caches*, bancos de registradores, registradores em separado, além de estágios de *pipeline* e seus respectivos registradores de estado. É também nessa descrição que são definidas determinadas características da arquitetura, como o tamanho de uma palavra ou a ordem dos bytes (*little-endian* ou *big-endian*). Baseado nessas duas descrições, o ArchC pode gerar simuladores interpretados (utilizando SystemC) e simuladores compilados (utilizando C++), entre outras ferramentas de software.

2.5.2 Ferramentas ArchC

O ArchC 2.0 é composto basicamente pelo conjunto de ferramentas: o pré-processador **acpp**, o gerador de simuladores interpretados **acsim** e o emulador de chamadas de sistema.

O pré-processador **acpp** é uma ferramenta compilada como uma biblioteca que extrai os dados descritos em AC_ARCH e AC_ISA para a geração do simulador. Essa ferramenta é composta por um analisador léxico e um analisador sintático.

A ferramenta **acsim** é capaz de gerar simuladores funcionais interpretados. Isso significa que cada instrução do programa deve ser buscada, decodificada e executada. Normalmente, esses simuladores utilizam uma *cache* de instruções decodificadas para evitar que a busca de uma instrução já decodificada seja feita novamente. A ferramenta **acsim** utiliza as informações extraídas por **acpp** para criar todas as classes C++ e/ou módulos SystemC necessários na construção da arquitetura do simulador [2, 27].

A simulação interpretada em ArchC se baseia em um esqueleto de simulador. Isso significa que a máquina de simulação é fixa, executando sempre as mesmas etapas, independente do processador simulado. Essas etapas de simulação seguem a seguinte ordem: decodificação; passagem de parâmetros para definir o comportamento da instrução; seleção do comportamento adequado e; finalmente, a execução.

Essas principais etapas de simulação ocorrem dentro de um processo SystemC do tipo *SC_THREAD*, chamado de $modelo :: behavior()$, que podem ser bloqueados naturalmente, com uma chamada $wait()$ SystemC. Durante a simulação, por padrão as instruções são processadas em lotes de 500 instruções (esse valor pode ser configurado), até que o laço principal seja interrompido por uma chamada de $wait()$.

Já os simuladores compilados executam estaticamente as operações que, normalmente, são executadas dinamicamente pelos simuladores interpretados. Para que isso seja possível,

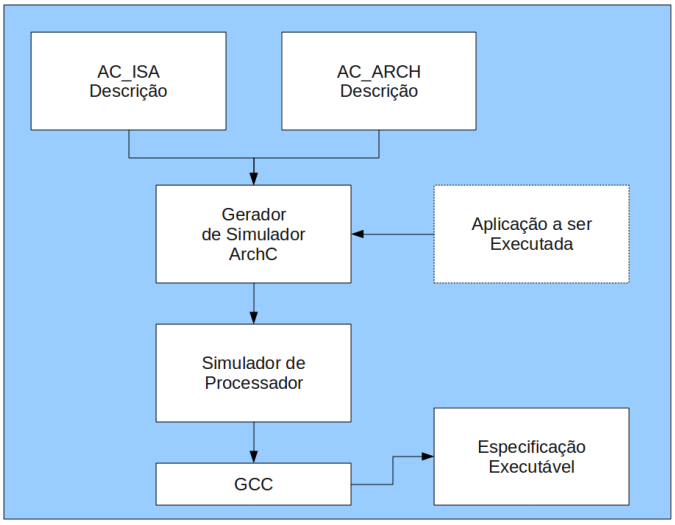


Figura 2.13: Fluxo de Geração de Simuladores.

a decodificação das instruções é movida para o tempo de compilação. Portanto, para a geração do simulador, tanto a descrição arquitetural como a aplicação que será executada são fornecidas ao gerador de simuladores. Sendo assim, o desempenho dos simuladores compilados é alto quando comparados aos simuladores interpretados. A ferramenta **accsim** é responsável pela geração desses simuladores. Diferentemente da ferramenta **acsim**, as informações extraídas por **acpp** são utilizadas para criar somente classes C++, devido à própria natureza do simulador [2].

A figura 2.13, adaptada de [2], ilustra o fluxo utilizado pelo ArchC para gerar simuladores. O fluxo seguido pelas ferramentas **acsim** e **accsim** são muito parecidos. Embora ambos geradores façam uso das informações contidas dos descritores AC_ARCH e AC_ISA, extraídas por **acpp**, para a geração do simulador, a diferença entre os simuladores interpretados e os simuladores compilados, descrita anteriormente, está representada pela caixa pontilhada ligada ao gerador de simulador, que é a aplicação a ser executada sendo embutida na geração do simulador.

A geração de simuladores através do ArchC possibilita que eles já sejam integrados com um mecanismo de emulação de chamadas do sistema operacional. Isso faz com que os modelos ArchC sejam capazes de simular aplicações que realizem operações de I/O sem que a aplicação simulada seja modificada. Agora, na modelagem de novos sistemas com ArchC, é preciso informar a partir de quais elementos de armazenamento (memória, banco de registradores, etc) os argumentos para a chamada do sistema serão buscados. Uma forma de fazer isso é por meio de interfaces de funções que fornecem a informação necessária para o simulador ArchC. Toda a funcionalidade do sistema operacional é implementada

na classe *ac_syscall*, que possui métodos virtuais que precisam ser implementados para cada novo processador segundo seu manual *Application Binary Interface* (ABI) [28].

2.5.3 Interfaces e Protocolo ArchC

A partir da versão 2.0 do ArchC [27], a comunicação externa do simulador de processadores com módulos SystemC é feita através de interfaces TLM 1.0. Isso não significa que um simulador codificado em ArchC tenha que se comunicar somente com módulos que compartilhem a mesma interface e protocolo de comunicação.

Uma maneira simples de integrar módulos com diferentes interfaces TLM e protocolos de comunicação é utilizar módulos intermediários que realizam a tradução entre esses protocolos. Esses módulos são chamados de *wrappers*. A modelagem de *wrappers* permite até que módulos TLM se conectem a módulos RTL. Mesmo assim, esse módulo dever ser capaz de converter sinais RTL em pacotes TLM (e vice-versa) e, provavelmente, permitir comunicação bloqueante e não-bloqueante.

O ArchC TLM foi desenvolvido para permitir que os módulos dos simuladores gerados em ArchC possam ser integrados com os modelos PV SystemC, citados na seção 2.3. Em particular, a interface escolhida para compor o ArchC TLM foi a interface *tlm_transport_if*, devido às suas características, em especial, a comunicação bidirecional com acoplamento entre requisições e respostas.

O protocolo ArchC TLM corresponde às definições dos pacotes de requisições e respostas. Esses pacotes são estruturas, cujos campos são descritos a seguir:

Pacote de Requisição:

- type: Descreve o tipo da transação. Pode assumir um dos cinco valores: *READ, WRITE, LOCK*(para bloquear um dispositivo ou barramento)*, UNLOCK* e *REQUEST_COUNT*(para propósitos de depuração, requisita o número de pacotes da transação).

- dev_id: *Device ID*. Toda porta do tipo *initiator* em ArchC é automaticamente atribuída a um identificador de dispositivo que fornece informação para dispositivos com múltiplas portas, como os barramentos.

- addr: Endereço de operação da requisição.

- data: Dados enviados.

Pacote de Resposta:

- status: Corresponde ao estado da transação, podendo assumir um dos dois valores: *ERROR* ou *SUCCESS*.

- req_type: Tipo da transação.
- data: Dados recebidos.

A utilização dos recursos citados nesta seção permite ao modelo ArchC implementar métodos de entrada e saída em uma camada mais genérica de comunicação com dispositivos externos (escravos), ***ac_memport***. Sendo o módulo simulador responsável pela inicialização das operações de leitura ou escrita através dessa camada. Portanto, com o protocolo de comunicação ArchC TLM, a comunicação entre o módulo simulador e um dispositivo externo é feita de maneira idêntica ao apresentada na seção 2.3. A diferença é que as chamadas de métodos da camada de usuário, responsáveis pela inicialização de uma transação, são feitas por *ac_memport* ao invés de aparecerem diretamente no código do comportamento da instrução.

Na seção a seguir, são apresentados os trabalhos realizados na tentativa de paralelização do SystemC. Também são apresentados comparativos entre esses trabalhos e o este projeto de pesquisa em relação a eles.

2.6 Trabalhos Relacionados

Esta seção trata de outros trabalhos realizados na tentativa de paralelização do simulador SystemC. Esses trabalhos podem ser classificados segundo a forma de comunicação: compartilhada ou distribuída.

Primeiramente, são apresentados os trabalhos em que a paralelização é realizada sobre sistemas de memória compartilhada e não exige a criação de novos mecanismos de comunicação e sincronização. Em seguida, são apresentados os trabalhos com distribuição do paralelismo em sistemas de memória geograficamente distribuída.

Pérez *et al.* [25] reescreveram o núcleo de simulação do SystemC a partir do zero. Esse novo núcleo foi desenvolvido para simuladores em nível de ciclo e suporta somente um pequeno subconjunto da sintaxe do SystemC, além de não trabalhar com TLM. O foco do projeto foi o desenvolvimento de um novo algoritmo de escalonamento dos processos.

Segundo os autores, o número de vezes que os processos são acordados (*wake-ups*) pode ter um forte impacto no desempenho. Se um processo retoma sua execução naturalmente, então o valor do sinal ao qual o processo é sensível foi alterado. Na técnica de escalonamento do SystemC, em um ciclo simulado do relógio, um processo pode ser acordado entre, aproximadamente, 1 a 12 vezes. Simulações com SystemC são realizadas de maneira sequencial sendo que, a cada ciclo simulado de relógio, os processos que são sensíveis à borda de subida são acordados e, durante suas execuções, eles modificam os seus sinais de saída, acordando outros processos. Sendo assim, a ordem em que os

processos são acordados, no começo de cada ciclo simulado de relógio, não é conhecida a priori.

O trabalho desenvolvido em [25] propõem uma técnica de escalonamento intermediária, entre um escalonamento estático e o escalonamento dinâmico do SystemC (descrito anteriormente nesse texto), chamada de escalonamento acíclico. No escalonamento estático, a arquitetura do sistema simulado é vista como um grafo, em que cada processo é um vértice e cada sinal é uma aresta orientada e o grafo é analisado para determinar o menor caminho possível através de todos os processos. Esse caminho, corresponde ao escalonamento do processo que é então compilado. O desafio presente na análise do grafo está na procura e solução do escalonamento de processos representados num grafo cíclico.

No escalonamento acíclico, o objetivo não é procurar e escalonar os componente fortemente conectados mas sim interpretar o grafo como um ou mais grafos acíclicos. Se um grafo tiver ciclos então ele é arbitrariamente quebrado e os vértices são ranqueados através dos ciclos para, então, escalonar os processos de acordo com seus ranqueamentos, fazendo com que o escalonamento agora passe a ser dinâmico.

Por fim, no trabalho de [25], os autores obtiveram ganhos de desempenho de 1,93× a 3,56× nas simulações, para processadores superescalares e RISC, respectivamente. Demonstrando que esta proposta pode ser uma boa solução. Porém, para obter um ótimo desempenho é necessário que os programadores informem ao núcleo do simulador as dependências existentes no código.

Baseados no problema apresentado em [25], em que os processos são acordados por diversas vezes durante a simulação e também na execução desnecessária de código, Naguib e Guindi [20], desenvolveram o SplitPro, uma ferramenta capaz de analisar automaticamente e extrair as dependências de sinais do modelo SystemC para, então, utilizar essas informações para dividir grandes processos em processos menores.

Segundo os autores, a execução desnecessária de código diz respeito à avaliação de sinais de saídas cujas entradas não sofreram alterações. Isso inclui a execução de porções de código, em um processo, que não alteram o estado final do sistema em um ciclo simulado de relógio.

A solução proposta em [20] é baseada no particionamento dos processos. Para que os processos possam ser particionados, os sinais de entrada e saída de um processo devem permitir que ele seja dividido em dois ou mais processos que, juntos, se comportem exatamente como o original. Para realizar o particionamento o SplitPro supõe que, sendo P um processo, I seu conjunto de entradas e O seu conjunto de saídas. O processo P pode ser particionado, se e somente se, existir um elemento O_j em O que é função de um subconjunto I_k de I. Para o particionamento quebrar um ciclo no grafo ou apresentar alguma melhoria, I_k não pode ser igual a I, caso contrário, o particionamento vai resultar em atraso no escalonamento de um novo processo. Para cada um dos sinais de saída do

sistema, o SplitPro constrói uma árvore de dependências, segundo um conjunto de regras apresentada em [20].

Naturalmente, a execução dos processos produz pedidos de atualização nos sinais de saída. Para remover avaliações desnecessárias das saídas, o SplitPro examina a conectividade entre os módulos e calcula a melhor ordem de execução dos processos baseado no grafo de dependências, para finalmente criar um ranqueamento dos processos. Mas, pode existir um processo que dependa do escalonamento de um outro processo no mesmo ciclo delta. Sendo assim, o processo com o maior ranqueamento não é chamado imediatamente, devido às suas dependências de sinais.

Como resultado do trabalho, foi observado um ganho de desempenho em torno de 23% nos resultados obtidos por [25]. Entretanto, o projeto está limitado por não rastreamento de dependências de sinais através de mais de uma função ou mais de um processo.

Ezudheen *et al.* [24], propõem que a paralelização do núcleo de simulação do SystemC seja feita utilizando a semântica já existente no processo de simulação em que dentro de um ciclo delta a ordem de execução dos processos não é pré-definida. Com base nesta semântica, os autores modificaram o escalonador do SystemC, fazendo com que este execute todos os processos, que estejam prontos para execução, em paralelo no mesmo instante de tempo para então atualizar seus canais e demais mudanças.

No começo da fase de evolução do núcleo de simulação, todos os processos são colocados em um *array*. O escalonador cria um ambiente de múltiplas execuções, que é responsável por manter as informações de estado relacionadas aos processos em execução e às múltiplas *threads*. Este ambiente é único e está associado a todas as CPUs.

Segundo os autores, nenhuma sincronização é necessária no estágio de inicialização do escalonador paralelo. A semântica do SystemC permite que múltiplos processos leiam um canal, mas somente um deles tem permissão para escrever neste canal. Quando múltiplos processos estão em execução paralela e criam pedidos de atualização no mesmo instante de tempo, esses podem tentar colocar pedidos de atualização na fila de atualização ao mesmo instante. Isso representa um potencial gerador de inconsistência de dados. Para prevenir esse tipo de inconsistência os autores modificaram as filas de atualização para listas ligadas. Assim, quando há um novo pedido de atualização, um novo nó é inserido na lista de atualizações. Essa inserção é realizada dentro de uma região crítica definida pela diretiva openmp (#pragma omp critical). Um mecanismo similar foi implementado para tratar a inconsistência de dados para processos SystemC, que engatilham notificações imediatas e criam novos processos executáveis no mesmo instante.

O particionamento do problema pode ser feito automaticamente baseado na taxa de utilização dos núcleos de processamento, no número de núcleos de simulação ou pode ser feito manualmente pelo programador. Durante a realização dos testes, quando o número de núcleos da máquina foi maior que oito, o particionamento manual foi o que apresentou

maior *speedup*. Quanto maior o número de módulos em execução e maior a computação requerida por cada módulo, melhor é o desempenho. Porém, quanto maior for número de núcleos de máquina simulando e o número de módulos em execução não for grande o suficiente, menor é o ganho de desempenho, por exemplo, a simulação de um sistema contendo 128 módulos, com 7 núcleos processando o desempenho foi aproximadamente 5,3×, já com 15 núcleos simulando o desempenho foi de aproximadamente 8,0×.

Nakamura *et al.* [21] apresentam um novo meio de realizar o escalonamento das *threads* no SystemC, reduzindo a frequência de trocas de contexto entre as *threads* em execução. Ao invés de utilizar a função *wait*() do SystemC, o programador pode utilizar a função *consume*() para sinalizar ao núcleo que uma determinada quantidade de tempo vai ser consumida pela *thread*. O ganho de desempenho não é apresentado, mas certamente existe, uma vez que foi eliminado em torno de 96% das trocas de contexto. Essa solução não resolve o problema de execução de múltiplas tarefas em múltiplos processadores.

Mello *et al.* [19] apresentam uma nova versão paralela do núcleo de simulação do SystemC, chamada de SystemC-SMP. Segundo os autores nenhuma modificação nos modelos de simulação que executam com a distribuição padrão do SystemC é necessária. Em SystemC-SMP as *threads* são sincronizadas por *sc_event*, porém, nem o tempo e, muito menos as notificações imediatas são mais utilizadas, o que significa que o paradigma de evolução e atualização não está mais presente no núcleo de simulação. Para cada núcleo de processamento da máquina o SystemC-SMP associa um escalonador local, sendo este responsável pela execução de todas as *SC_THREADs* no mesmo processador físico. Cada escalonador local é executado como uma Posix-*thread* (*pthread*) pelo escalonador do sistema operacional. Através de macros o usuário pode especificar no modelo a afinidade da *SC_THREADs* com os núcleos de processamento. Vale a pena dizer que as *SC_THREADs* são mapeadas como *quick threads* e somente os escalonadores locais são *pthreads*.

A sincronização do modelo, como descrito anteriormente, é feito por meio de eventos. Neste caso, *sc_event* assume somente três valores, que podem ser alterados por diferentes *threads*, *IDLE*, *WAITING* e *NOTIFIED*. Os primeiros resultados mostram um ganho de 1,8 vezes quando comparado à simulação sequencial.

De maneira geral, paralelizar o SystemC em um sistema de memória distribuída exige a criação de um novo ambiente de simulação, responsável pela comunicação e sincronização entre os processos. Os trabalhos a seguir demonstram como a paralelização do SystemC para ambientes de memória distribuída é realizada e como a sincronização por meio de mensagens pode ser eficientemente empregada nesse tipo de ambiente.

O trabalho de Chopard *et al.* [6] apresenta uma solução em que cada nó de processamento do *cluster* executa uma cópia do escalonador do SystemC, simulando um subconjunto de módulos do sistema modelado. Isso é possível devido à própria semântica do SystemC, que diz: dentro de um ciclo delta a ordem de execução dos processos não

é pré-definida; quais processos serão executados depende do ciclo delta anterior. Mesmo que os processos possam ser executados em paralelo, os escalonadores locais devem ser sincronizados no final de cada ciclo delta.

A sincronização de um canal envolve somente os nós que esse conecta e, assim como no SystemC *serial*, se o valor num dado canal muda, então os nós conectados a ele engatilham o evento associado, e os processos sensíveis são realocados como executáveis. Para eventos de tempo, os autores definem um nó mestre que coleta os próximos eventos de tempo de todos os outros nós, calcula o próximo tempo de simulação e então envia esse valor para todos os nós para atualização local.

A única alteração na linguagem do SystemC foi a adição de um novo tipo de módulo, chamado *sc_node_module*, que internamente reúne todos os módulos que devem ser gerenciados por um nó de processamento. Do ponto de vista do desenvolvedor não há mudanças mas esse novo tipo de módulo impede a existência de hierarquia entre os módulos. Sendo assim, o particionamento do sistema modelado é manual e de responsabilidade do desenvolvedor.

Finalmente, em testes com um sistema modelado como um *pipeline*, os autores obtiveram ganhos de desempenho de aproximadamente 3,07 vezes em um cluster com até 52 nós com processadores mono-núcleo. Dependendo da modelagem empregada, a comunicação entre os módulos pode se tornar um gargalo no desempenho, uma vez que esse pode envolver troca de sinais entre as múltiplas instâncias do núcleo do SystemC.

Dando continuidade ao trabalho apresentado em [6], no trabalho de Combes *et al.* [7] os autores apresentam algumas maneiras de reduzir o *overhead* ocasionado pela sincronização. Baseados nisso, são apresentados dois esquemas de sincronização: sincronização dos canais e sincronização descentralizada do tempo de simulação.

No primeiro esquema, a sincronização de todos os canais ocorre durante a fase de atualização. Seja P um módulo processante do tipo *sc_node_module*, seja I_P o conjunto de canais de entrada desse módulo e O_P o conjunto de canais de saída desse módulo. Mesmo que a atualização aconteça durante a fase de atualização, os valores de O_P podem ser enviados antes, desde que esses não sofram mais alterações durante o mesmo ciclo delta. Do mesmo modo, os valores de um canal em I_P não precisam ser recebidos na fase de atualização se nenhum processo é sensível às mudanças nesse canal. Portanto, é possível atrasar a finalização da operação de recebimento até uma próxima leitura ou então até a próxima fase de atualização, se nenhum processo precisar ler o valor.

O esquema de sincronização descentralizada do tempo de simulação faz com que todo nó calcule localmente o próximo tempo de simulação θ. Este esquema faz uso da troca de mensagens entre os nós processantes para garantir o sincronismo e um nó executa seu ciclo delta até que não hajam mais processos para serem executados. No fim do laço de ciclo delta, esse nó calcula o próximo tempo de simulação e o envia como uma requisição

do tipo END_REQ para todos os outros nós, que respondem com uma mensagem com o valor END_ANS. Para realizar essas trocas de mensagens, os nós executam um processo concorrente que manipula essas mensagens enquanto a simulação está em andamento. Quando um nó não tem mais processos para executar, esse deve aguardar até que todos as mensagens, esperadas dos outros nós, sejam enviadas.

Utilizando a mesma plataforma de testes de [6], Combes *et al.* obtiveram um ganho de desempenho de 2,98 vezes em relação à versão original do SystemC em um *cluster* de máquinas de núcleo simples, conectadas por uma rede do tipo *Giga-Ethernet*. Os autores ainda comentam que bons desempenhos somente são possíveis quando a carga do sistema está bem balanceada entre os nós do *cluster*.

Huang *et al.* [16] desenvolveram uma biblioteca distribuída do SystemC (SCD) com suporte para simulações em ambientes de computação paralela geograficamente distribuídos. A técnica apresentada é portável para distribuição funcional e *approximated-timed* TLM. A biblioteca SCD é integrada com um *framework* completo MPSoc DSE, facilitando a geração e distribuição automática da simulação em ambientes Linux.

O método de distribuir a simulação do SystemC em [16] é executar o ciclo delta de simulação combinado com chamadas ao método $sc\_start(SC\_ZERO\_TIME)$ repetidamente, já que este método executa o ciclo delta somente uma única vez. Após todo ciclo delta, a simulação retorna à função $main()$, por exemplo, $sc\_main()$, onde a comunicação e a sincronização são declaradas. Comunicação, neste caso, significa a transmissão de dados entre os simuladores. Sincronização refere-se ao controle global da simulação. Sendo assim, um ponto fixo de sincronização para o propósito de comunicação não é necessário, fazendo com que a comunicação e a sincronização estejam desacoplados no modelo. O tempo de simulação somente é avançado se $sc\_start(timeval)$ for chamado, independentemente da existência de eventos ou do tempo de simulação corrente ou passado.

Os pontos chave em [16] são: (a) a sincronização e comunicação; (b) o controle sobre diferentes simuladores. Para isso, um simples simulador pode estar em três estados de simulação: *Busy, Idle* ou *Done*.

Para gerenciar o estado global da simulação, um mecanismo de mestre-escravo é aplicado. O mestre coleta as informações dos simuladores escravos e avança o tempo de simulação ou termina a simulação.

O particionamento da simulação é feito automaticamente pelo *framework* MPSoc DSE com a biblioteca SCD tendo como referência os dados arquiteturais do modelo SystemC, especificados de maneira abstrata e contendo somente dados estruturais, de desempenho e paramétricos necessários para decisões de modelagem, e no mapeamento dos canais de comunicação entre módulos do modelo.

Durante os testes os autores verificaram que, no pior caso, a execução do modelo proposto em uma única máquina, do ambiente distribuído, com um único simulador, não

obteve ganho algum. No melhor caso, utilizando um total de cinco máquinas e cada uma executando um total de quatro simuladores, o ganho obtido foi de 4,5 vezes.

Ziyu *et al.* [31], apresentam um novo ambiente para a simulação paralela do SystemC, chamado ArchSC. Segundo os autores, para a modelagem do ArchSC foram consideradas as plataformas de simulação paralela de grande escala no nível de sistema, tendo a ferramenta ArchSim [17] como base para o paralelismo.

O paralelismo do ArchSC dá-se pela divisão do sistema simulado em vários subsistemas relativamente independentes que são dispersos em um ambiente paralelo com vários simuladores SystemC e esses subsistemas cooperam para executar a simulação completa.

A modificação do mecanismo de escalonamento faz-se necessária. A ferramenta ArchSim é utilizada para controlar a execução da simulação no SystemC, incluindo a inicialização, execução, pausa e parada. O ArchSC também é capaz de enviar e receber pacotes de dados com os módulos remotos. Portanto, o ArchSC paraleliza o SystemC sob dois aspectos: o núcleo de simulação e os canais de comunicação.

Segundo [17], a ferramenta ArchSim é composta por um servidor global (GS), um controle central de toda a simulação, agentes de serviços locais (LSA), que são os elementos básicos mapeados nos processadores físicos ou núcleos de processamento e, por fim, as entidades que são os objetos de simulação encapsulados como entidades que são escalonadas pelos LSAs.

O escalonador SystemC é dividido em duas partes, chamadas de *Forward* e *Backward*. A fase *Forward* encapsula a fase de avaliação do SystemC, já a fase *Backward* compreende a fase de atualizações. Entre a execução das novas fases, os subsistemas podem trocar dados entre si.

Quanto à comunicação entre módulos, o ArchSC gerencia todos os canais remotos dos subsistemas. Uma vez que podem existir vários canais entre dois subsistemas, todos os canais de mesmo tipo, por exemplo, um canal de *input*, são unidos em uma mesma porta do mesmo tipo no ArchSim.

O particionamento do problema é sempre feito manualmente pelo desenvolvedor da plataforma a ser simulada. Os autores consideram que o usuário é capaz de particionar o problema em subsistemas relativamente independentes, sem que isso interfira no processamento e nos resultados.

Testes em uma plataforma modelada como um *pipeline* em um *cluster* de 8 máquinas mostraram que o desempenho melhorou com o aumento de número de módulos (ou estágios), a quantidade de computação realizada por cada estágio, e o número de processos. Em média, os testes apresentaram desempenho próximo do número de processos em dois casos: quando o número de estágios foram de 4096 e 8192 com a quantidade fixa de computação[3] igual a 1000; e a quantidade de computação realizada por cada estágio

[3]Ou seja, o número de iterações que um número é operado

de $1E6$ e $1E7$, com o número fixo de 128 estágios.

Conclusões

Para melhor visualização dos resultados, a tabela 2.2 relaciona todos os trabalhos citados anteriormente. Essa tabela é composta por três colunas: Nome/Autor, com o nome do trabalho ou os autores; Comunicação, vem a ser o meio de comunicação e sincronização entre processos; e *Speedup*, o ganho obtido pela implementação, em relação ao SystemC de execução sequencial. Ou no caso da ferramenta SplitPro [20], o ganho é sobre os resultados obtidos por Pérez *et al.* [25].

Nome/Autor	**Comunicação**	**Speedup**
Pérez *et al.* [25]	SMP	1,93X a 3,56X
SplitPro [20]	SMP	23%
Ezudheen *et al.* [24]	SMP	5,3X a 8,0X
Mello *et al.* [19]	SMP	1,9X
Chopard *et al.* [6]	MPI	3,07X
Combes *et al.* [7]	MPI	2,98X
Huang *et al.* [16]	MPI	4,5X
ArchSC [31]	Unix Socket	-

Tabela 2.2: Relação de trabalhos relacionados.

A paralelização proposta nesta dissertação foi direcionada para arquiteturas multinúcleos de hardware e, portanto, não foi preciso criar um ambiente próprio de paralelização, comunicação e sincronização de processos, como acontece em [6, 7, 16, 31]. Nesses trabalhos, é possível verificar que, no momento da simulação há uma grande quantidade de mensagens sendo trocadas entre processos (possível gargalo no desempenho) e todo um ambiente deve ser gerenciado, além da simulação propriamente dita.

Uma vez que o SystemC já implementa Posix Pthreads, ao compilar o SystemC com a biblioteca *pthreads.h* espera-se que toda SC_THREAD seja mapeada como uma *pthread*. No entanto, como vimos na seção 2.2, o escalonador do SystemC tende a sequencializar a execução das *threads*. Por esse motivo propõem-se um novo tipo de *thread*, uma especialização da classe SC_THREAD, chamada de $SC\_DTHREAD$ ou *detached thread*. No trabalho de [24], os autores fazem com que essas *pthreads* sejam executadas em paralelo, somente se elas forem processos prontos para a execução em um determinado ciclo delta, e são colocadas para dormir em seguida. A motivação do emprego de $SC\_DTHREAD$ é que uma *thread* nunca durma e, que numa chamada de $wait()$, essa *thread* somente conceda o processador à outra *thread* e seja reescalonada posteriormente.

A partir do momento em que a simulação é executada com $SC\_DTHREADs$, toda a

sincronização é realizada diretamente no sistema modelado, os eventos permanecem, mas não interferem na execução das *detached threads*. Sendo assim, parte da paralelização do modelo e a sincronização dos processos é de responsabilidade do desenvolvedor. O capítulo 3 contextualiza esse trabalho de paralelização do SystemC em seus detalhes.

Capítulo 3

SystemC *Multi-Threaded*

Os conceitos apresentados no capítulo anterior (seção 2.2) sobre a linguagem SystemC, permitem identificar quais melhorias podem ser aplicadas ao núcleo de simulação da linguagem e como esse deve se comportar em um ambiente simulação paralela.

A proposta deste projeto de pesquisa é obter uma extensão do SystemC capaz de executar em processadores com mais de um núcleo de processamento, sem a necessidade de realizar grandes alterações no código original do SystemC.

O foco principal desse trabalho é utilizar os recursos já oferecidos pelo SystemC para obtenção de execução paralela e, consequentemente, ganhos de desempenho de simulação. O SystemC, originalmente, possibilita que toda *SC_THREAD* seja mapeada como uma *pthread* para o sistema operacional e, ao mesmo tempo, tenha todos os atributos de um processo SystemC. Mesmo com esse mapeamento, quando o SystemC é compilado com a biblioteca *pthreads.h*, a execução dessas *threads* fica restrita à política de escalonamento de núcleo de simulação, apresentada na seção 2.2.3, que continuam a ser executadas sequencialmente.

Por esse motivo, a proposta apresentada aqui é que as *pthreads* sejam executadas de maneira independente do escalonador do SystemC. Assim, uma vez que a simulação é iniciada, essas *threads* tem suas execuções disparadas e, na ocorrência de uma chamada de *wait*(), a *thread* em que a chamada foi iniciada concede o processador para que outra *thread* seja executada e ela é posteriormente reescalonada. Como resultado, foi desenvolvida uma nova classe, chamada de *SC_DTHREAD* (*detached thread*). Toda a paralelização do modelo de sistema é feita explicitamente, assim, todas as *SC_THREADs* que o programador opte que sejam executadas em paralelo, agora devem ser declaradas como *SC_DTHREADs*.

Entretanto, o trabalho não se limita somente à paralelização do núcleo de simulação do SystemC. Outra contribuição importante deste trabalho diz respeito à comunicação e à sincronização entre processos no ambiente paralelo de simulação. Para que as *dthreads*

consigam executar de maneira independente do escalonador, elas abstraem a existência dos eventos que, por sua vez, foram substituídos por um outro tipo de mecanismo de sincronização. Sendo assim, a comunicação e sincronização são feitas diretamente sobre os canais TLM do modelo. Portanto, o modelo do escalonador direcionado a eventos, apresentado na figura 2.7 da subseção 2.2.3, não se aplica inteiramente as *dthreads* que executam independentemente do escalonador.

O texto deste capítulo é dividido em três partes: na seção 3.1, é apresentado o novo modelo de execução do núcleo de simulação do SystemC; na seção 3.2, são apresentadas as novas estruturas de dados utilizadas na implementação do novo modelo de execução; e por fim, as alterações realizadas no ArchC e no TLM são apresentadas na seção 3.3.

3.1 Modelo de Execução

A análise feita na seção 2.2.3 mostra claramente que o SystemC, embora permita que cada *sc_thread* seja mapeada como uma Posix *pthread*, o algoritmo de escalonamento do núcleo de simulação limita essas *threads* à uma execução sequencial, obedecendo ao paradigma de avaliação e atualização. Portanto, ao utilizar os novos processos *dthreads* é possível fazer com que essas *pthreads* sejam, realmente, executadas em paralelo.

A introdução do conceito de *dthreads* ao escalonador do SystemC impõe mudanças em seu fluxo de execução. Uma vez iniciada a fase de simulação do SystemC, as *dthreads* têm suas execuções disparadas e, à partir deste momento, não possuem mais vínculos com o núcleo de simulação. Essa desvinculação ocorre porque as *dthreads* não são incluídas na fila de processos prontos para execução e, portanto, não obedecem ao paradigma de avaliação e atualização. Sendo assim, surge novo problema, uma vez que o núcleo de simulação do SystemC é sempre encerrado juntamente com os processos *dthread*, por não existirem notificações de tempo e, muito menos, processos para serem executados em sua fila de processos prontos.

A figura 3.1 representa o diagrama de fluxo do código do novo modelo de execução do núcleo de simulação do SystemC. Uma vez iniciado, o diagrama apresenta uma nova tarefa, em que o escalonador libera todas as *dthreads* para execução. A sequência do fluxo se aplica somente aos outros tipos de processos, se existirem declarações dos mesmos. Caso contrário, no final do diagrama de fluxo foi inserido um laço, responsável por aguardar que todas as *dthreads* tenham finalizado suas respectivas execuções.

Esse laço foi implementado introduzindo um ponto de junção (Posix *pthread_join*) no núcleo de simulação, resolvendo o problema de encerramento prematuro, mantendo-o em execução enquanto todas as *dthreads* não terminarem suas execuções. Essa implementação resulta em uma primeira versão do núcleo de simulação, em que todas as *dthreads* declaradas são executadas em paralelo, independentemente do núcleo de simulação. Vale

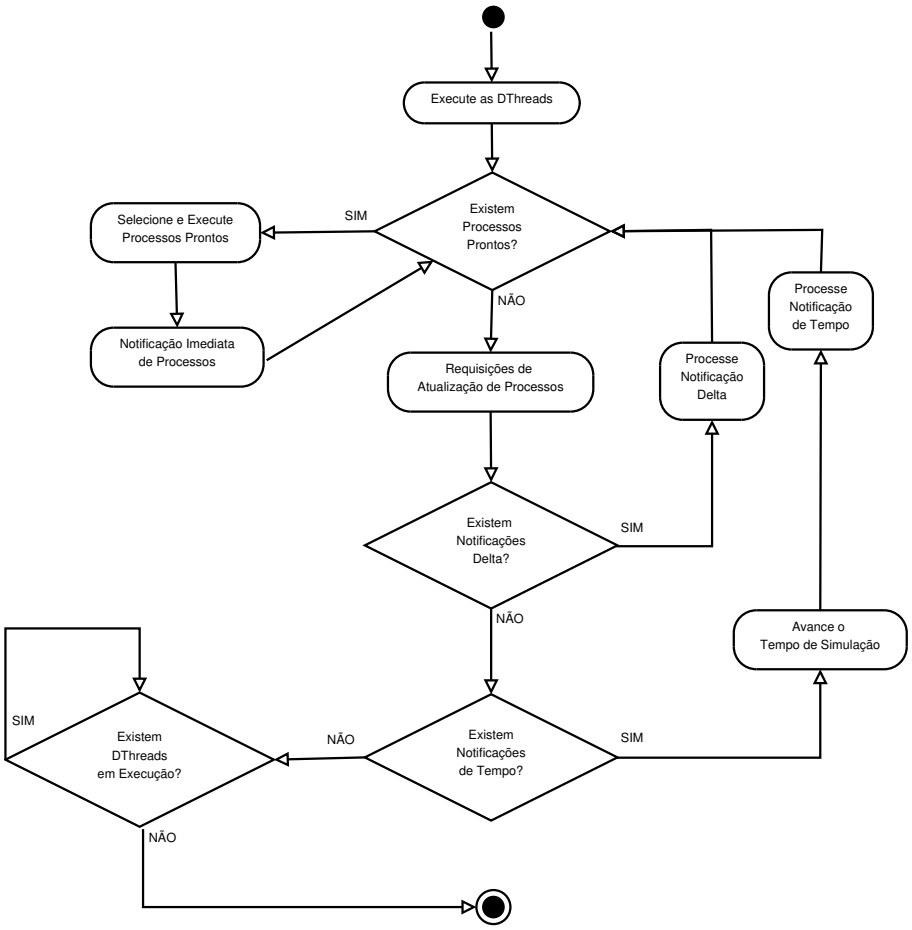


Figura 3.1: Diagrama de fluxo do novo modelo de execução

a pena lembrar que, se não existirem outros tipos de processos declarados, o núcleo do SystemC não está fazendo nada de útil e somente esperando a finalização das *dthreads*.

Baseado nesse problema, é proposto um segundo modelo de execução, no qual a primeira declaração *SC_DTHREAD* seja sempre uma *sc_thread* que execute internamente ao SystemC, caso já não exista uma *sc_thread* previamente declarada e as demais declarações de *dthreads* se comportem realmente como *detached threads*. Se, em todo o modelo, existir somente uma declaração de *dthread*, então ela será registrada e executada como uma *sc_thread*.

Em termos de código, quando uma *SC_DTHREAD* é declarada no modelo do usuário, a macro da figura 3.2 (linha 1), declarada em **sc_module.h**, invoca a macro *declare_dthread_process* (linha 2). O código da figura 3.3 mostra como a *thread* interna é criada. Na classe ***sc_simcontext*** foi criada uma nova variável do tipo booleana, chamada *m_system_thread*, que indica se a *thread* interna já foi criada, essa variável é iniciada com o valor *false*. Na linha 3, essa variável é consultada, através do método *sc_simcontext::system_thread*. Caso

```
1  #define SC_DTHREAD(func)                                    \
2      declare_dthread_process( func ## _handle,               \
3                               #func,                         \
4                               SC_CURRENT_USER_MODULE,        \
5                               func )
```

Figura 3.2: Definição da macro *SC_DTHREAD*

nenhuma *sc_thread* ainda tenha sido criada, a macro *declare_thread_process* é executada (linha 4) e a *sc_thread* é criada e *m_system_thread* atualizada. Caso contrário, o método *sc_simcontext::create_dthread_process* é executado e um objeto do tipo *dthread* é criado.

```
1  #define declare_dthread_process(handle, name, host_tag, func)                    \
2      {                                                                            \
3              if (::sc_core::sc_get_curr_simcontext()->system_thread() == false){ \
4                  declare_thread_process(handle, name, host_tag, func);            \
5              }else{                                                               \
6          ::sc_core::sc_process_handle handle =                                    \
7               sc_core::sc_get_curr_simcontext()->create_dthread_process(          \
8                   name, false,                                                    \
9                   SC_MAKE_FUNC_PTR( host_tag, func ), this, 0 );                  \
10             }                                                                    \
11     }
```

Figura 3.3: Definição do método de declaração de *detached threads*

O desempenho das *dthreads* é proveniente do seu comportamento de execução independente do núcleo de execução do SystemC. Porém, o desempenho global da simulação é diretamente afetado pelo desempenho da *thread* que está executando internamente ao núcleo de simulação do SystemC.

Simplesmente compilar o SystemC com a biblioteca *pthreads*, faz com que, além das *dthreads*, todos processos do tipo *sc_thread* também sejam mapeados como *pthreads*. No capítulo 4 é mostrado que, compilar o SystemC com *pthreads* pode não ser a melhor opção, pois seu desempenho é em média 36 vezes pior do que utilizar *Quick Threads* (um tipo interno de *thread* do SystemC). Essa diferença de desempenho ocorre devido ao seu comportamento de espera ocupada em chamadas de *wait*, em que a *pthread* deve esperar pelo engatilhamento de um evento dentro de um *pthread_cond_wait*. Esse desempenho certamente interfere no desempenho do novo modelo de simulação, quando a *thread* interna for mapeada como *pthread*.

Por esse motivo, na implementação de um terceiro modelo de simulação, utilizou-se um mapeamento híbrido das *threads*, em que as *dthreads* são mapeadas como *pthreads* e a *thread* interna passa a ser mapeada como uma *quick thread*. Desta maneira, qualquer outra *thread* que seja declarada no modelo do usuário também será mapeada como uma *quick thread*.

A presença de *dthreads* e *threads* internas do SystemC executando em um mesmo modelo fazem com que as chamadas de *wait*() tenham que ser tratadas de uma nova maneira. Quando uma chamada de *wait*() ocorre no modelo do usuário, ela não é realizada diretamente na *thread* e sim em uma sequência de chamadas, partindo de ***sc_module***, ***sc_wait*** e, finalmente, na ***sc_thread***. A chamada de *wait*() no módulo somente organiza os argumentos passados na assinatura do método e faz a chamada do método correspondente em *sc_wait*. Nesse ponto, a chamada de *wait* é realizada na *thread* que se encontra atualmente em execução, registrada em ***sc_simcontext::m_curr_proc_info***. No novo modelo de execução uma *dthread* nunca é registrada como um processo atualmente em execução, pois não é escalonada pelo núcleo de simulação. Desta forma, esse método de chamadas de *wait* não se aplica e pode resultar em erros se não for tratado. Note que uma chamada de *wait* em uma *dthread* pode fazer uma interrupção em uma *sc_thread*, que não esteja esperando, ou em um ponteiro nulo, caso não existam mais *sc_threads* em execução.

Portanto, um novo modelo para realizar as chamada de *wait*() é incorporado ao SystemC para este suportar a existência das *dthreads*. Quando uma chamada de *wait* chega em *sc_module*, não é possível identificar qual *thread* a fez (pode haver mais de uma *thread* registrada em um módulo) e se essa *thread* é uma *dthread* ou uma *sc_thread*. Assim, propõem-se que todas as *threads* sejam registradas no momento de criação em tabelas *hash*, sendo uma tabela local ao módulo e uma tabela de acesso global, quando a chamada de *wait* acontece em um módulo externo acessado pela *thread*. O registro e a estrutura das tabelas são explicados em detalhes na seção 3.2.

Uma vez registradas, quando as *threads* acessarem o método *wait*() do módulo elas poderão ser devidamente identificadas. Essa identificação acontece por meio da extração do identificador da *thread* através da função Posix *pthread_self* e consulta à tabela *hash* local. Caso a entrada não exista para o identificador na tabela *hash* local então a tabela *hash* global é acessada.

Quando uma *sc_thread* for identificada, essa segue a sequência de chamadas descrita anteriormente para o núcleo de simulação sequencial. Já uma *dthread* tem um comportamento diferente, uma *dthread* nunca dorme e para que esta tenha um comportamento mais justo e permita que outras *threads* também possam utilizar o processador é fundamental que ela implemente um método que a faça liberar o processador para que outra *thread* execute. Por esse motivo, quando o método ***sc_wait::dthread_wait(dthread_process)*** é invocado por uma *dthread* previamente identificada, o método ***pthread_yield()*** é aplicado nessa *dthread*, fazendo com que ela seja colocada no fim da lista de execução de sua prioridade estática do processador e outra *thread* seja escalonada para execução.

Porém, vale a pena lembrar que o escalonador Linux também pode retirar essa *thread* de execução e reescaloná-la posteriormente no processador, segundo sua política de esca-

lonamento. Desta forma, provavelmente uma *thread* vai ser submetida ao escalonamento mais de uma vez, entre duas chamadas de *wait*.

Por conta do problema acima descrito optou-se por deixar que nada seja feito em uma chamada de *wait* e o *kernel* Linux seja responsável pelo escalonamento. Além disso, o novo *kernel* Linux 2.6 [18] beneficia o desempenho da simulação por ter melhor escalabilidade, ser capaz de escalonar usando um algoritmo $O(1)$ (o número de processos não afeta o seu desempenho), ter melhor eficiência em ambiente multiprocessado (nenhum núcleo de processamento fica ocioso se existe trabalho a fazer), garante a afinidade do processo ao núcleo que está executando (evita troca de núcleos) e, por fim, garante a justiça, uma vez que nenhum processo deixa de receber ao menos um pequeno *quantum* de tempo de processamento e também, nenhum processo recebe um grande *quantum* de tempo de processamento (respeitando a prioridade de cada processo).

Concluindo, a implementação do novo modelo de execução resultou em três versões que são testadas e avaliadas no capítulo 4. Na primeira versão, todas *dthreads* são executadas em paralelo e o SystemC aguarda a finalização delas; na segunda versão, a primeira declaração de *dthread* é registrada e executada como *sc_thread* interna ao núcleo de simulação e esta é mapeada como uma *pthread*; a terceira versão é semelhante à segunda, exceto pelo fato de que a *sc_thread* é mapeada como uma *quick thread*.

A próxima seção apresenta em detalhes as novas estruturas de dados e mecanismos de controle apresentados durante a descrição do novo modelo de execução. Entre essas novas estruturas destacam-se a classe *sc_dthread_process*, as tabelas *hash sc_dthread_hash*, utilizadas na identificação do processos nas chamadas de *wait* e a classe *sc_dthread_join*, responsável por manter o núcleo de simulação em execução até que todas as *dthreads* sejam finalizadas.

3.2 Novas Estruturas de Dados

Para que o núcleo de simulação do SystemC consiga executar, mesmo sem a presença de processos registrados em sua lista de prontos para execução e desvinculados de seu escalonador, novas estruturas de dados e mecanismos de controle para o ambiente *multi-threaded* foram incluídos ao núcleo de simulação. Primeiramente, é apresentada a classe *sc_dthread_process*, principal classe do novo modelo de simulação; em seguida são apresentadas as estruturas de tabela *hash* e ponto de junção; por fim, são apresentadas as modificações no ArchC e adaptações para sincronização pelos canais TLM.

3.2.1 Implementação de *SC_DTHREADS*

Para que as *threads* possam ser corretamente manuseadas em um ambiente paralelo, novos métodos devem ser adicionados à classe *sc_thread_process* e os comportamentos dos seus métodos devem ser alterados. Por esse motivo, foi criada uma nova classe, chamada de *sc_dthread_process*, ou simplesmente, *dthread* (*detached thread*).

Essa nova classe não é uma especialização da classe *sc_thread_process* e sim uma extensão da classe *sc_process_b*, a classe base dos processos do tipo *thread*, garantindo a interoperabilidade do novo processo com o resto do sistema, além de distinguir *sc_dthread_process* e *sc_thread_process*, pois muitos dos métodos presentes em *sc_thread_process* foram eliminados ou têm seus comportamentos muito alterados em *sc_dthread_process*. A declaração da classe *sc_dthread_process* pode ser vista na figura 3.4. Além disso, a figura também mostra a quantidade de métodos que não são mais necessários para as *dthreads*.

O texto a seguir fornece mais informações sobre a classe *sc_dthread_process*, desde a instanciação de um objeto do tipo *sc_dthread* às novas estruturas de dados envolvidas nas operações com *dthreads*, e as principais diferenças entre as classes *sc_thread_process* e *sc_dthread_process*.

Quando uma *dthread* é declarada no modelo do usuário, uma série de macros são utilizadas para definir se essa declaração será uma instância de *sc_dthread* ou *sc_thread*, como descrito na seção 3.1. Se a declaração for realmente uma *sc_dthread*, então um objeto correspondente à classe deve ser criado. A instanciação dos objetos de processos ocorre durante a fase de elaboração no núcleo de simulação, mais especificamente no objeto da classe *sc_simcontext*, que mantém o contexto da simulação.

Para dar suporte à instanciação de *dthreads*, foi criado um novo método em *sc_simcontext*, chamado ***create_dthread_process***. Esse método é responsável por chamar o construtor da classe *sc_dthread_process* e manter a lista de processos criados.

Os métodos construtores das classes *sc_thread_process* e *sc_dthread_process* são muito parecidos, desde a assinatura dos métodos a instanciação do processo base (*sc_process_b*), exceto por valores que caracterizam as *threads*, como o atributo de tipo de processo *m_process_kind* e a necessidade das *sc_threads* estarem vinculadas à listas de sensibilidade a eventos, o que não acontece com as *sc_dthreads* e, portanto, foi eliminada.

A lista de objetos de processos *sc_methods*, *sc_thread*, *sc_cthread* e *sc_dthread* é mantida pela classe nativa do SystemC *sc_process_table*. Essas listas são utilizadas para a criação das corotinas dos processos. Essa classe não precisa necessariamente manter listas inteiras, mas somente o primeiro elemento ou cabeça de cada lista. Usando como exemplo *dthreads*, quando um novo objeto *dthread* é instanciado ele deve ser inserido na lista correspondente. Todas as classes de processos possuem um ponteiro *m_exist_p* para um outro processo existente. Portanto, se a lista for vazia, então *process_table* fará com que o novo processo seja a cabeça da lista e *m_exist_p* um ponteiro nulo. À medida que novos processos forem

Figura 3.4: Diagrama de classes ilustrando a herança entre as classes sc_process_b, sc_thread_process e sc_dthread_process.

criados, eles serão adicionados na cabeça da lista mas com *m_exist_p* apontando para o antigo primeiro elemento. Esse encadeamento é feito pelo método *set_next_exist*(∗*next_p*) em *sc_dthread_process*, simplesmente atribuindo *next_p* a *m_exist_p*.

Passada a fase de elaboração, obrigatoriamente toda *dthread* deve ser mapeada como uma Posix *pthread* e esse mapeamento é feito através do método ***prepare_for_simulation***, percorrendo[1] toda a lista de processos *dthreads* previamente criada. Esse mapeamento é feito durante a fase de inicialização do núcleo de simulação. Ao invocar esse método, uma corotina para o comportamento da *dthread* é criada.

Em SystemC uma corotina é uma instância de uma classe *sc_cor_qt* (*Quick Threads*), *sc_cor_pthread* (*PThreads*) ou *sc_cor_fiber* (WIN32), derivadas da classe abstrata *sc_cor*. Para que as corotinas possam ser criadas, abortadas ou então escalonadas para simulação, o SystemC define uma outra classe chamada de *sc_cor_pkg* (*coroutine pac-*

[1]Para percorrer a lista de processos a partir do início, basta utilizar o método *next_exist()* em *sc_dthread_process*.

kage), que é implementada conforme a especialização da corotina. Portanto, para que a classe derivada de *sc_cor* e a classe *sc_cor_pkg* deem suporte à criação das *dthreads*, novos métodos foram incorporados a essas classes, como: ***invoke_module_method_detached*** e ***create_detached***, respectivamente. Visto que as *dthreads* não estão mais vinculadas ao escalonador do SystemC, não foi necessária a inclusão de métodos que as manuseiam, como *yield* e *abort*.

O método ***invoke_module_method_detached*** é responsável pela inicialização da execução da corotina da *thread* de *sc_dthread_cor_fn(void* arg)*. Quando uma corotina é criada e é mapeada como *pthread*, a rotina que é passada como argumento, no momento de criação, e que será imediatamente executada pela *pthread* é a função *invoke_module_method_detached* (figura 3.5).

```
1  void* sc_cor_qt::invoke_module_method_detached(void* context_p)
2  {
3          sc_simcontext* sim_c = sc_core::sc_get_curr_simcontext();
4
5          sc_cor_qt* p = (sc_cor_qt*)context_p;
6
7          pthread_mutex_lock( &create_mutex );
8          pthread_cond_signal( &create_condition );
9          pthread_mutex_unlock( &create_mutex );
10         pthread_barrier_wait(&sim_c->barrier);
11
12         (p->m_cor_fn)(p->m_cor_fn_arg);
13
14         return 0;
15 }
```

Figura 3.5: Código do método invoke_module_method_detached

Originalmente, ao criar uma corotina para a *thread*, o SystemC coloca essa *thread* em espera sob um *pthread_cond_wait*, e somente será executada se a variável condicional for verdadeira. No novo modelo de execução essa espera foi removida e para evitar que a corotina para a *dthread* criada inicie sua execução antes que o núcleo entre na fase de simulação ela é imediatamente travada em uma barreira (linha 10). Essa barreira é declarada em *sc_simcontext* e está disponível globalmente (linha 3). Ela é iniciada entre o final da fase de elaboração, quando todas as *dthreads* estão declaradas, e antes da fase de inicialização quando as corotinas são criadas, com o valor $num = num\_dthreads + 1$, que é a quantidade de *dthreads* declaradas mais a *thread* SystemC. Dessa maneira, as *dthreads* só têm suas execuções liberadas quando o núcleo de simulação também atingir a barreira. Para manter o padrão do núcleo de simulação, o ponto em que a *thread* do núcleo atinge a barreira e libera a execução das *dthreads* foi alocado no início do escalonador e é executado uma única vez, por ficar fora do laço de escalonamento. Na figura 3.1, o local de execução das *dthreads* pode ser melhor visualizado pela atividade *execute as dthreads* no topo do diagrama.

```
1  sc_cor*
2  sc_cor_pkg_pthread::create_detached( std::size_t stack_size, sc_cor_fn* fn, void* arg )
3  {
4      sc_cor_pthread* cor_p = new sc_cor_pthread;
5
6      // INITIALIZE OBJECT'S FIELDS FROM ARGUMENT LIST:
7      cor_p->m_pkg_p = this;
8      cor_p->m_cor_fn = fn;
9      cor_p->m_cor_fn_arg = arg;
10
11     // SET UP THREAD CREATION ATTRIBUTES:
12     pthread_attr_t attr;
13         pthread_attr_init( &attr );
14         if ( stack_size != 0 )
15         {
16                 pthread_attr_setstacksize( &attr, stack_size );
17         }
18
19     // CREATING DTHREAD
20     pthread_mutex_lock( &create_mutex );
21     if ( pthread_create( &cor_p->m_thread, &attr,
22              &sc_cor_pthread::invoke_module_method_detached, (void*)cor_p ) )
23     {
24         std::fprintf(stderr, "ERROR - could not create thread\n");
25     }
26
27     //Inserting DThread into the hash table
28     sc_process_b* dthread_p = (sc_process_b*) cor_p->m_cor_fn_arg;
29
30     //Local Hash
31     sc_module* parent_module = (sc_module*) dthread_p->get_parent_object();
32     parent_module->get_dthread_hash()->set_new_entry(cor_p->m_thread, dthread_p);
33
34     //Global Hash
35     ::sc_core::sc_get_curr_simcontext()->get_dthread_hash()->set_new_entry(cor_p->
           m_thread, dthread_p);
36
37     //Inserting DThread for Joining
38     ::sc_core::sc_get_curr_simcontext()->get_join_dthread()->set_join(cor_p->m_thread);
39
40     pthread_cond_wait( &create_condition, &create_mutex );
41     pthread_attr_destroy( &attr );
42     pthread_mutex_unlock( &create_mutex );
43
44     return cor_p;
45 }
```

Figura 3.6: Código do Método sc_cor_pkg::create_detached

A implementação do método ***sc_cor_pkg_pthreads::create_detached*** é apresentada na figura 3.6. Nesse método, o tamanho da pilha, a semântica da *thread* e os argumentos do comportamento da *thread* são passados nos argumentos (linha 2). A primeira tarefa deste método é criar a corotina propriamente dita para a *dthread* (linha 4). Uma vez criada a corotina, ela precisa ser iniciada com a semântica do objeto *dthread* e seus argumentos (linhas 8 e 9), que serão utilizados mais tarde nesse mesmo código. Antes que a *pthread* seja criada é preciso definir o tamanho mínimo de alocação de sua pilha (linhas

12 a 17), isso é feito a partir do valor de *stack_size*, passado como valor nos argumentos do método. Uma vez concluída a fase de inicializações de variáveis do método, toda a criação das *pthreads* acontece dentro de uma região de código protegida por *mutex*, devido a necessidade de escrita em variáveis e estruturas de dados compartilhadas. Na linha 21, a *pthread* para o método *invoke_module_method_detached* é criada, informando o tamanho de sua pilha e argumentos do método.

Concluída a criação da *pthread*, é necessário incluir referências sobre essa *thread* nas estruturas de dados especialmente desenvolvidas para o novo modelo de execução. Entre essas novas estruturas estão duas tabelas *hash*, que são utilizadas em chamadas de *wait*. Essas tabelas são compostas por dois campos: o identificador da *pthread* e o objeto do processo *dthreads*. Sendo assim, na linha 28, *sc_process_b* é extraído da corotina por meio de *casting*. Para uma nova inserção na tabela *hash* local, primeiramente deve-se obtê-la acessando o módulo pai da *dthread* (linha 31). A inserção na tabela *hash* local ocorre na linha 32. Para acessar a tabela *hash* global e inserir a nova entrada, basta obter a referência para o objeto instanciado da classe *sc_simcontext* (linha 35).

Por fim, a referência para a *pthread* é inserida no objeto global instanciado da classe *sc_join_dthread* (linha 38), responsável por fazer com que o simulador não encerre até que todas as *pthreads* tenham terminado suas execuções.

Uma vez pronta e liberada para execução a *dthread* processa até que não exista mais nada para fazer. Nesse ponto sua corotina (*sc_dthread_cor_fn*) faz uma chamada ao método *kill_process*(), responsável pela remoção da instância do objeto em uso. Em *sc_thread*, o método local *kill_process* primeiramente faz uma chamada ao método *kill_process*() em sua classe base *sc_process_b* para remover o processo de listas de eventos dinâmicos e estáticos ao qual possa estar inserido, em seguida, verifica se esse processo é o processo atualmente em execução e aborta sua execução ou então o remove da lista de processos prontos para execução. Em *sc_dthread* o método *kill_process* é simplificado, pois somente o método *kill_process* da classe base é executado, porém, sem a remoção de listas de eventos, sendo assim, esse método somente remove as referências existentes à instância de objeto da classe base no núcleo de simulação. Uma vez finalizada a remoção, a *thread* do processo é naturalmente encerrada pelo fim do seu código.

3.2.2 Tabelas *Hash*

Durante a apresentação do novo modelo de simulação (seção 3.1) foi descrita a metodologia adotada para a identificação do processo *thread* quando uma chamada ao método *wait*() é realizada. Conforme descrito anteriormente, o identificador da *pthread* extraído por *pthread_self()* é utilizado como índice de busca pelo processo *sc_process_b* na tabela *hash*. Na seção 3.2.1, foi apresentado o local do código onde as novas entradas são inseridas nas

tabelas *hash*, durante a criação da corotina da *dthread* na fase de inicialização.

Essa seção apresentada as motivações para a utilização de tabelas *hash* e o que são as tabelas *hash* em SystemC.

Como mostrado na seção 3.1, no novo modelo de execução, quando uma chamada ao método *wait*() é realizada no modelo do usuário não se sabe quem de fato realizou essa chamada, se foi uma *dthread* ou qualquer outro processo SystemC. O meio encontrado para que o SystemC seja capaz de identificar qual processo está realizando a chamada de *wait*() foi obter o identificador da *pthread* mas, somente esse identificador não representa quem é esse processo internamente ao SystemC. Portanto, é necessário que exista uma relação entre o identificador da *pthread* e o processo base correspondente.

Uma tabela *hash* (ou tabela de dispersão) é a melhor escolha para armazenar esses dados e, consequentemente, recuperá-los com maior eficiência. Assim, a partir de uma simples chave é possível recuperar rapidamente o valor associado a ela. O objetivo das tabelas *hash* é que a função de espalhamento seja capaz de realizar a busca em $O(1)$, não importando o número de entradas da tabela.

Uma vez que o núcleo de simulação do SystemC já implementa tabelas *hash*, não há necessidade de adicionar uma nova estrutura de dados ao núcleo. Tabela *hash* em SystemC é definida pela classe ***sc_phash***, com atributos e métodos apropriados para o manuseio da tabela e, inclusive, funções de dispersão prontas para uso, segundo o tipo de dado da chave.

Para que as tabelas *hash* locais ao módulo e a tabela *hash* global não sejam diretamente instanciadas e definidas nas classes *sc_module* e *sc_simcontext*, respectivamente, foi criada uma nova classe chamada de ***sc_dthread_hash***. Essa nova classe é capaz de criar uma nova tabela, inserir e recuperar dados dela. A figura 3.7 ilustra a associação entre as classes *sc_module*, *sc_dthread_hash* e *sc_simcontext*. É possível verificar que somente a associação de uma tabela *hash* local é sempre um para um com um módulo e a tabela *hash* global também é um para um com *simcontext*.

A inserção de novas entradas nas tabelas *hash* foi descrita anteriormente neste capítulo através da figura 3.6 (linhas 37 e 40). Quando uma nova entrada é inserida, o seu identificador único *pid* (extraído por *pthread_self*) é utilizado como chave para a função de dispersão. A função de dispersão utilizada (já implementada por *sc_phash*), calcula o índice da seguinte forma: $indice = pid * 3141592661U$ e todo o tratamento de colisões já está previsto na implementação de *sc_phash*.

A utilização de dois tipos de tabelas *hash* (local e global) se deve a questões de desempenho. A consulta a uma tabela *hash* implementada em SystemC envolve diversas chamadas a funções e essas funções retornam valores, e neste caso, o valor esperado é o ponteiro para *sc_process_b*. Uma vez que a leitura não é feita diretamente sobre a tabela, em um ambiente paralelo, o valor de retorno pode ser alterado antes que a *thread* possa

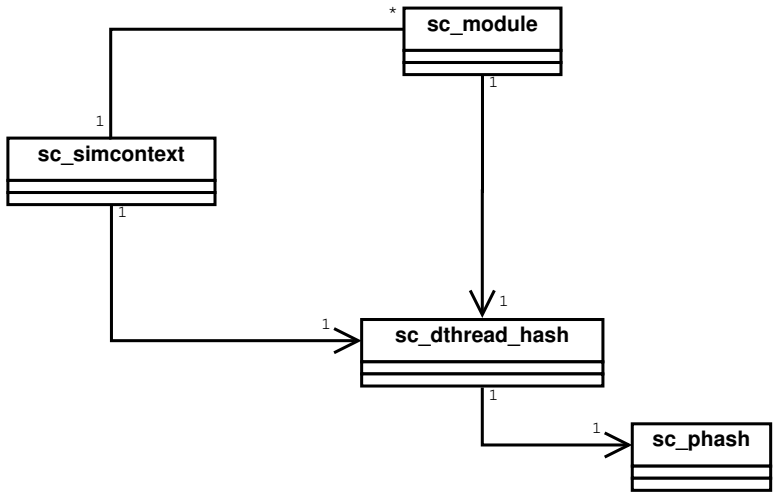


Figura 3.7: Diagrama de classes ilustrando a associação entre as classes sc_module, sc_dthread_hash e sc_simcontext.

lê-lo. Sendo assim, a leitura da tabela *hash* global é realizada dentro de um *mutex* global, o que representa um gargalo no sistema, se todas as *threads* tiverem que ler a todo momento essa tabela. Por esse motivo, são implementadas tabelas *hash* locais aos módulos. Essas tabelas também são lidas dentro de mutexes locais, porém, a concorrência dentro de um módulo pode ser facilmente controlada pela quantidade de processos declarados dentro dele. Por fim, caso a entrada não seja encontrada na tabela local porque a *thread* que chamou o *wait* não foi declara no módulo, então a tabela global é consultada.

3.2.3 Ponto de Junção

A seção 3.1 descreve que, ao utilizar processos tipo *dthreads*, o núcleo do SystemC não possui alguma referência à existência delas durante a simulação e, portanto, encerrará a simulação sem que as *dthreads* tenham finalizado suas rotinas. Para manter o núcleo em execução até que todas as *dthreads* tenham concluído suas tarefas, foi inserido um ponto de junção no final do código do escalonador do SystemC.

Esse ponto de junção deve ter conhecimento de todas as *dthreads* criadas e aguardar que todas elas terminem para, então permitir que o núcleo de simulação encerre. O ponto de junção aqui proposto foi modelado como uma classe (figura 3.8) com um único atributo, que é um vetor de *pthread_t* e dois métodos: o primeiro para a inserção de elementos no vetor; e o segundo que realiza a operação de *pthread_join* da biblioteca Posix.

Assim que a corotina para a *dthread* é criada, ela também deve ser inserida no vetor da junção. Na figura 3.6 linha 38, é apresentado esse local onde todas as corotinas são inseridas. Vale lembrar que essa operação é feita durante a fase de elaboração e, portando,

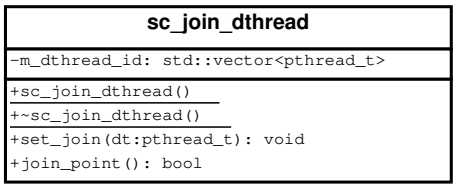


Figura 3.8: Declaração da classe sc_join_dthread.

não adiciona custos ao tempo de simulação.

O ponto de junção é um método que percorre todo o vetor de *pthreads*, bloqueando a *thread* do núcleo de simulação e aguardando a finalização de todas as *dthreads* desse vetor. Uma vez finalizada com sucesso a junção de todos os métodos, o núcleo pode enfim encerrar a simulação.

3.3 Alterações em ArchC e TLM

Uma vez que as plataformas de testes utilizadas para a validação deste trabalho foram modeladas utilizando a linguagem ArchC descrita na seção 2.5, esta também deve ser adaptada para dar suporte à simulação paralela real.

Durante a elaboração do trabalho de Sigrist 2007 [27], que resultou no ArchC 2.0, além da integração com TLM, muitas das variáveis estáticas presentes no código foram eliminadas a favor das simulações de plataformas com mais de um modelo de processador, gerando benefícios para a simulação *multi-threaded*, objeto deste trabalho. Porém, falhas ainda foram encontrados e corrigidos na biblioteca.

Uma situação que sempre requer a atenção do projetista em um ambiente paralelo é o compartilhamento de recursos por dois ou mais processos. Em ArchC isso não é diferente, uma vez que dois ou mais modelos tenham que compartilhar a mesma biblioteca da aplicação. Especialmente se o comportamento de cada modelo de processador for declarado e executado como uma *dthread*. Portanto, qualquer recurso dessa biblioteca que seja utilizado para manter o estado de um modelo de processador dentro de uma plataforma, e tenha definição global, implicará em erro durante a simulação.

Nos modelos de processadores, quando uma instrução é enviada para execução, ela primeiro tem que ser decodificada, para uma instrução do hardware descrito, para que enfim o seu comportamento possa ser aplicado. Durante essa decodificação o método ***DecodeAsInstruction*** (figura 3.9) é chamado para montagem de um vetor com os campos da instrução decodificada. Na declaração desse método a variável *fields* foi declarada como estática (linha 3). Essa declaração afeta diretamente o comportamento das várias instâncias de processadores da plataforma, já que é possível ter mais de um processador decodificando uma instrução em um dado instante de tempo.

```
1  unsigned* ac_decoder_full::DecodeAsInstruction(ac_decoder_full * decoder, ac_dec_instr *
       instruction, unsigned char *buffer, int quant)
2  {
3    static unsigned* fields = 0;
4    unsigned counter;
5    ac_dec_field *field;
6    //ac_dec_list *list = instruction -> dec_list;
7    ac_dec_field *list ;
8
9    ac_dec_format *format;
10
11   ...
12 }
```

Figura 3.9: Trecho do código do método ac_decoder_full::DecodeAsInstruction

Retirar a palavra chave *static* da declaração da variável, resolve o problema, mas ainda é necessário avaliar o impacto dessa alteração para modelos de processadores que trabalham com palavras de mais de 32 bits. Tanto essa como as demais modificações serão repassadas ao ArchC *Team*, para serem revistas e oficialmente anexadas à versão final da ferramenta.

Feita essa modificação, é necessário que o ArchC possibilite a criação de modelos com suporte ao modelo de execução com *dthreads*.

Como descrito na seção 2.5, a única *thread* em execução de um modelo ArchC é definida pela rotina do método *modelo::behavior()*, responsável pela parte principal da execução (decodificação, passagem de parâmetros para os comportamentos de instrução, seleção do comportamento adequado e, finalmente, execução). Devido à presença da execução de instruções em lotes, a cada 500 instruções executadas o laço principal de execução é interrompido por uma chamada de *wait*.

Sendo assim, a primeira alteração foi realizada na ferramenta *acsim*, possibilitando a criação de modelos que implementem suas execuções como *SC_DTHREAD*. A princípio a seleção entre *sc_thread* ou *sc_dthread* é feita por meio de uma macro (*USE_SC_DTHREADS*) que pode ser definida pelo usuário antes da compilação do ArchC. Portanto, quando o usuário definir esta macro, a ferramenta acsim, na criação da declaração do módulo do modelo, automaticamente declara *modelo::behavior()* como sendo uma *SC_DTHREAD*, e executará como tal, quando a simulação da plataforma for iniciada.

A segunda e última alteração no ArchC é direcionada as interfaces TLM das portas de comunicação do modelo gerado. Nas plataformas modeladas até o momento, a concorrência no roteador ou barramento podia ser facilmente controlada por meio de chamadas de *wait*, por um árbitro do barramento ou então pela própria falta de concorrência real do sistema. Dessa maneira, é necessário garantir que somente um processador esteja ocupando o dispositivo de comunicação em um certo instante de tempo.

Seja o padrão de projeto utilizando roteador, descrito na seção 2.3 (figura 2.10). O

roteador é um módulo que implementa uma interface TLM e realiza o encaminhamento das requisições vindas dos processadores por meio da função *transport()* e retorna as respostas a essas requisições aos seus respectivos emissores. Um vez que o roteador não possui qualquer mecanismo de arbitragem, em um ambiente de paralelismo real, esse dispositivo deve garantir que a requisição seja corretamente encaminhada ao dispositivo de destino, sem que seja alterada no trajeto. Além disso, é preciso garantir que a resposta a essa requisição retorne corretamente, sem interferência em seus dados.

Para resolver esse problema, é preciso fazer com que os canais de comunicação entre os modelos de processadores e o roteador garantam o acesso exclusivo ao roteador. Portanto, para garantir o acesso exclusivo, é necessário que toda a comunicação seja executada dentro de um *mutex*. A melhor forma de implementar um *mutex* no canal é inseri-lo diretamente na interface que este canal implementa, no caso do ArchC, interfaces TLM.

```
1  template < typename REQ , typename RSP >
2  class tlm_transport_if : public virtual sc_core::sc_interface
3  {
4  private:
5
6    pthread_mutex_t tlmMutex;
7
8  public:
9    tlm_transport_if(){
10         mutexInit();
11   }
12
13   virtual RSP transport( const REQ & ) = 0;
14
15   virtual void transport( const REQ &req , RSP &rsp ) {
16     rsp = transport( req );
17   }
18
19   virtual void transport_mutexed( const REQ &req , RSP &rsp ) {
20      pthread_mutex_lock(&tlmMutex);
21      rsp = transport( req );
22      pthread_mutex_unlock(&tlmMutex);
23   }
24
25   virtual int mutexInit() {
26      return pthread_mutex_init(&tlmMutex, NULL);
27   }
28
29 };
```

Figura 3.10: Classe da interface tlm_transport_if remodelada

A figura 3.10 apresenta o código da classe *tlm_transport_if*, presente na biblioteca TLM. Originalmente implementado no ArchC, quando uma chamada de *read* ou *write* acontece numa porta, uma chamada ao método de interface $RSP\,transport(const\,REQ\&)$ da linha 13 é efetuada. Para esse mecanismo de sincronização, o interessante é a assinatura do método da linha 15. Com essa assinatura é possível fazer com que a chamada de *transport*

aconteça dentro de um *mutex*, garantindo exclusividade do roteador. Para não alterar as assinaturas originais, foi implementado um novo método chamado ***transport_mutexed*** (linha 19), para acessar o canal com exclusividade. Uma variável de *mutex* e sua inicialização também foram adicionadas à classe. O método construtor da classe também é alterado, para permitir a inicialização do *mutex* quando a classe herdeira de *tlm_transport_if* é instânciada, caso contrário o usuário deverá fazer a inicialização diretamente no modelo.

Mesmo com essas alterações na interface TLM o ArchC não é capaz de reconhecer o novo método de imediato exigindo que mais alterações sejam feitas, neste caso, na classe *ac_tlm_port* de sua biblioteca. Além da assinatura $RSP\,transport(const\,REQ\&)$ que, obrigatoriamente deve ser reconhecida para os casos em que não seja necessária a comunicação protegida por *mutex*, o método $transport\_mutexed(const\,REQ\&\,req, RSP\&\,rsp)$ também deve se reconhecido pela classe. Essa diferenciação é feita por meio de definição de macros de pré-processamento.

```
1  void ac_tlm_port::do_transport( ac_tlm_req &req, ac_tlm_rsp &rsp )
2  {
3  #ifdef USE_MUTEXED_TRANSPORT
4          (*this)->transport_mutexed(req, rsp);
5  #else
6          rsp = (*this)->transport(req);
7  #endif
8  }
9
10 void ac_tlm_port::do_transport( ac_tlm_req &req)
11 {
12         ac_tlm_rsp rsp;
13         do_transport(req, rsp);
14 }
```

Figura 3.11: Assinaturas do método do_transport

A solução encontrada para este caso foi criar um novo método para fazer *transport* dentro de *ac_tlm_port*, chamado ***do_transport***. A figura 3.11, apresenta as duas assinaturas criadas para *do_transport*. Se a macro *USE_MUTEXED_TRANSPORT* for previamente definida, então a comunicação pelo canal será protegida por *mutex* (linha 3), devido ao método *transport mutexed*, fazendo com que a requisição e a resposta sejam passadas por referência. Caso contrário, o método *transport* original é executado. A assinatura presente na linha 1 é utilizada pelos métodos de *read* e *write* implementados pela classe *ac_tlm_port*. Já a assinatura da linha 10 é utilizada por métodos de bloqueio e desbloqueio de dispositivos.

É importante dizer que, se o comportamento do modelo de processador for declarado como uma *dthread*, não necessariamente a comunicação deverá ser feita por um canal de acesso exclusivo. Esse canal pode ser implementado para qualquer outro dispositivo da plataforma que implemente o protocolo TLM do ArchC. Utilizar um canal de acesso

exclusivo para modelos em que não há *dthreads* pode não ser uma boa opção, devido à falta de paralelismo real e um sincronismo desnecessário de concorrência no dispositivo alvo.

O próximo capítulo, descreve, passo-a-passo, a implementação do modelo de plataforma utilizado para a realização dos testes, a preparação dos *benchmarks* para a avaliação do presente trabalho, além dos resultados obtidos pelos testes.

Capítulo 4

Resultados

Primeiramente, este capítulo apresenta um rápido guia de como modificar uma plataforma existente de maneira que ela execute com *dthreads*. Em seguida, é apresentada a plataforma utilizada para a realização dos testes, os programas do pacote Splash2 [30] escolhidos para serem executados nessa plataforma, como eles foram paralelizados, e por fim são apresentados os resultados das execuções, com cada programa, sob diferentes configurações, bem como a quantidade de modelos de processadores e a carga de trabalho em cada um deles.

A arquitetura de plataforma de testes foi proposta segundo modelos comumente utilizados por usuários para validação de sistemas. Por suas próprias características, essa arquitetura de plataforma não beneficia a simulação paralela, devido à grande contenção existente ao acessar o roteador e o módulo único de memória. Os testes realizados na seção 4.3 mostram que mesmo sob essa arquitetura desfavorável o modelo de simulação do SystemC *Multi-Threaded* é capaz de registrar bons desempenhos. Na seção 4.4 são apresentadas algumas formas de tornar essa plataforma de simulação mais favorável à simulação *multi-threaded* e, consequentemente, melhorar o desempenho da simulação.

4.1 Modificação de Modelos

Esta seção descreve como fazer modificações de modelos existentes para utilizarem processos *dthreads* e como é possível obter canais de comunicação exclusiva entre dois ou mais módulos. Para explicar tais modificações é utilizada uma plataforma composta por dois modelos de processadores ArchC, memória e um módulo IP, todos conectados por um roteador.

Segundo a seção 3.3, o comportamento (*behavior*()) do modelo é automaticamente declarado como uma *SC_THREAD*. Para fazer com que os modelos de processadores executem como *dthreads*, o ArchC deve ser compilado com a macro *USE_SC_DTHREADS*

definida. Dessa maneira, todo modelo de processador da plataforma é automaticamente gerado e tem o seu comportamento declarado como sendo uma *dthread*. A figura 4.1, apresenta uma parte do código da declaração da classe módulo *powerpc*, gerado automaticamente pelo ArchC. A linha 8 é a assinatura do método C++ *behavior*, na linha 14, esse método é declarado como uma *dthread* para o núcleo de simulação do SystemC e será executado segundo o novo modelo de simulação.

```
1  class powerpc: public ac_module, public powerpc_arch {
2  private:
3    typedef cache_item<powerpc_parms::AC_DEC_FIELD_NUMBER> cache_item_t;
4    typedef ac_instr<powerpc_parms::AC_DEC_FIELD_NUMBER> ac_instr_t;
5  public:
6  ...
7    //!Behavior execution method.
8    void behavior();
9  ...
10   SC_HAS_PROCESS( powerpc );
11
12   //!Constructor.
13   powerpc( sc_module_name name_ ): ac_module(name_), powerpc_arch(), ISA(*this), syscall
         (*this) {
14     SC_DTHREAD( behavior );
15     bhv_pc = 0;
16     has_delayed_load = false;
17     start_up=1;
18     id = 1;
19   }
20 ...
21 };
```

Figura 4.1: Método *behavior* do PowerPC declarado como SC_DTHREAD

Seja o módulo IP (figura 4.2) um módulo que trabalhe tanto como escravo, recebendo um valor no registrador *RxDataR*, quanto um módulo mestre que grave um valor na memória, por meio de sua *thread* interna *send_thread*(). A figura 4.2, apresenta a implementação do método *transport* do módulo IP. Quando um novo dado é enviado, imediatamente esse dado é gravado no registrador *RxDataR* e a *thread send_thread* tem sua execução liberada. Por fim, o pacote de resposta é enviado ao módulo mestre (processador) indicado o sucesso da operação.

Na figura 4.3, os métodos da classe módulo *my_IP* são implementados. No construtor da classe (linha 2), as portas são instanciadas e os registradores do módulo iniciados. Ainda no construtor, o método *send_thread* é declarado para executar como sendo uma *SC_THREAD* (linha 9). Na linha 21, está implementado o comportamento do método *send_thread*. Neste exemplo, o valor de *RxDataR* é utilizado para gerar um novo valor que é armazenado no registrador de saída *TxDataR*. Já que esse método é uma *thread*, então ele é executado dentro de um laço infinito (linha 26), porém, sua execução depende da existência de um novo valor no registrador *RxDataR*, daí o motivo de se utilizar a

variável booleana *send_wakeup*. Quando essa variável se torna verdadeira (linha 27) a *thread* calcula o valor de *TxDataR* (linha 33) e prepara o pacote de envio desse valor para um determinado endereço de memória (linhas 35 a 37), em seguida, esse pacote é enviado pela porta de saída do módulo por meio do método *transport* (linha 39) e seguirá diretamente ao roteador e, enfim, será direcionado à memória.

Fazer com que a *thread send_thread* seja executada como uma *dthread*, nesse caso, é uma tarefa simples, exigindo somente que, ao invés de declarar a *thread* como *SC_THREAD* ela seja declarada como *SC_DTHREAD*. Dessa maneira, o SystemC é capaz de reconhecê-la e executá-la como sendo uma *dthread*, essa alteração é apresentada na figura 4.4, linha 9.

```
1  class my_IP: public sc_module,
2               public ac_tlm_transport_if // Using ArchC TLM protocol
3  {
4  private:
5      int RxDataR;
6      int TxDataR;
7      bool send_wakeup;
8
9  public:
10     //Sets thread behavior
11     SC_HAS_PROCESS(my_IP);
12     //Destructor
13     ~my_IP();
14     //Constructor
15     my_IP(sc_module_name name);
16     //Threads
17     void send_thread();
18     //Ports
19     sc_export< ac_tlm_transport_if > target_export;
20     ac_tlm_port sendPort;
21     // TLM transport method
22     ac_tlm_rsp transport( const ac_tlm_req &request ){
23         ac_tlm_rsp response;
24
25         switch (request.type){
26                 case (WRITE): RxDataR = request.data;
27                               send_wakeup = true;
28                               response.status = SUCCESS;
29                 break;
30                 default:      response.status = ERROR;
31                 break;
32         }
33         return response;
34     }
35 };
```

Figura 4.2: Declaração da classe *my_IP*

Uma vez definidos os processos *dthread*, o controle de concorrência anteriormente imposto pelo núcleo de simulação do SystemC não se aplica mais. Dessa maneira, é preciso reavaliar a plataforma, por exemplo, para determinar quais módulos escravos podem suportar acessos concorrentes e quais precisam ser acessados exclusivamente pelos módulos

```
1  //Construtor
2  my_IP::my_IP(sc_module_name name)
3      : sc_module(name),
4      target_export("p1port"),
5      sendPort("sendPort", 8388608U)
6  {
7      target_export( *this );
8
9      SC_THREAD(send_thread);
10
11     send_wakeup = false;
12
13     RxDataR       = 0x0;
14     TxDataR       = 0x0;
15 }
16
17 //Destructor
18 inline my_IP::~my_IP(){}
19
20 //Send Thread
21 void my_IP::send_thread(){
22     int tempData;
23     ac_tlm_req request;
24     ac_tlm_rsp response;
25
26     while(1){
27         while(!send_wakeup);
28
29         send_wakeup = false;
30
31         tempData = RxDataR;
32
33         TxDataR = TxDataR * (tempData << 6);
34
35         request.addr = 0x2000000;
36         request.type = WRITE;
37         request.data = tmpdata;
38
39         response = sendPort->transport(request);
40      }
41 }
```

Figura 4.3: Implementação dos métodos da classe *my_IP*

processantes. O acesso exclusivo ao módulo escravo não é definido diretamente nele e sim na interface do canal de acesso a esse módulo, como descrito na seção 3.3, nas alterações realizadas na interface TLM. Portanto, são os módulos processantes da plataforma que devem acessar os módulos escravos pelo canal exclusivo e esse acesso é feito por meio do método *transport_mutexed*(). O módulo acessado não tem conhecimento sobre o acesso exclusivo.

Seja o roteador o módulo escravo que é acessado de maneira exclusiva pelos demais módulos processantes. Para os modelos ArchC, o processador é capaz de realizar esse acesso exclusivo aos módulos escravos se a macro *USE_MUTEXED_TRANSPORT* for previamente definida durante o tempo de compilação do ArchC (seção 3.3). Para o módulo

```
1  //Construtor
2  my_IP::my_IP(sc_module_name name)
3      : sc_module(name),
4      target_export("p1port"),
5      sendPort("sendPort", 8388608U)
6  {
7      target_export( *this );
8
9      SC_DTHREAD(send_thread);
10     ...
11 }
12
13 //Send Thread
14 void my_IP::send_thread(){
15     int tempData;
16     ac_tlm_req request;
17     ac_tlm_rsp response;
18
19     while(1){
20         while(!send_wakeup);
21         ...
22         sendPort->transport_mutexed(request, response);
23      }
24 }
```

Figura 4.4: Implementação dos métodos da classe *my_IP*, segundo o novo modelo de execução

IP, esse acesso exclusivo deve ser implementado manualmente, alterando a maneira como o método *transport* é chamado. Para que isso seja possível basta utilizar a nova assinatura $transport\_mutexed(req, rsp)$ do método, figura 4.4, linha 22. Dessa maneira, é possível garantir que somente uma *dthread* por vez terá acesso ao roteador da plataforma, evitando problemas de corridas sobre este módulo.

4.2 Plataforma de Testes

A modelagem da plataforma de testes foi realizada sobre a estrutura da ARP *ArchC Reference Platform*, um espaço de trabalho, descrito em [1], dedicado à modelagem de plataformas utilizando modelos do ArchC. A ARP pode ser vista como uma estrutura de diretórios, para que o usuário possa separar os diversos módulos da plataforma e trabalhar separadamente em cada um. Esses diretório são listados a seguir:

- **platforms**: Descrição da plataforma do usuário;
- **processors**: Simuladores gerados pelo ArchC e utilizados como núcleos de processamento nas plataformas;
- **is**: Barramentos e estruturas de interconexão;

- **ip**: IPs, dispositivos adicionais;
- **sw**: *Programa* que é executado nos simuladores de processadores;
- **wrappers**: Tradutores entre os simuladores de processadores, IPs e níveis de abstração.

A arquitetura da plataforma modelada para os testes pode ser vista na figura 4.5. Essa plataforma representada na figura 4.5 é composta por dois modelos de processadores, mas durante os testes esse número é elevado até sessenta e quatro processadores. Especificamente, para essa plataforma optou-se pelos simuladores de processadores PowerPC, por estarem amplamente testados em projetos internos ao LSC e por realizarem a definição de suas pilhas de instruções internamente ao modelo, evitando que essas definições sejam feitas dentro de cada programa que será executado pela plataforma, o que não acontece em outros modelos como o MIPS.

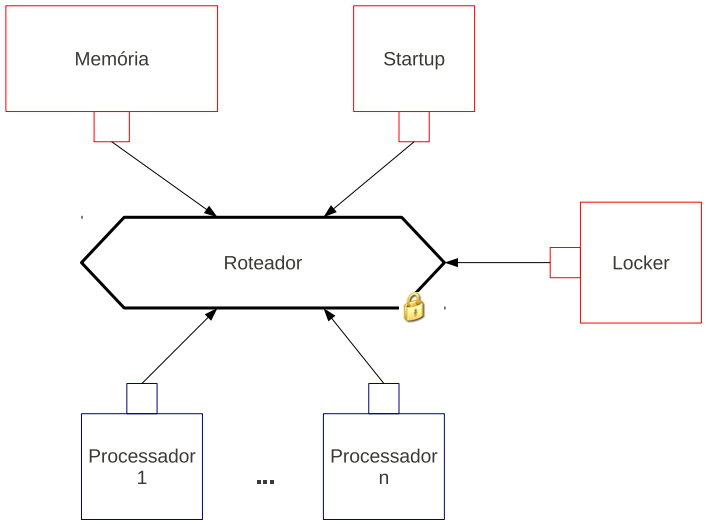


Figura 4.5: Arquitetura da plataforma de testes.

Quanto aos IPs, foram modelados: um módulo TLM de memória (***ac_tlm_mem***); um módulo de inicialização dos processadores (***ac_tlm_startup***); e por fim, um módulo de bloqueio global (***ac_locker_module***).

O módulo TLM de memória (***ac_tlm_mem***) é único para todos os processadores e possui 8MB de capacidade de armazenamento; um módulo de inicialização dos processadores (***ac_tlm_startup***) é responsável por atribuir um identificador único a cada processador,

esse identificador será posteriormente utilizado na divisão das tarefas a serem executadas pelo programa; e por fim, um módulo de bloqueio global (***ac_locker_module***) que trabalha como uma variável de *mutex* global à plataforma, fazendo com que certas operações de software ou certos dispositivos somente possam ser acessados pelos processadores, se esse *mutex* assim permitir.

Toda a comunicação entre os dispositivos da plataforma (IPs, processadores, etc) foi modelada com base no padrão de projeto de roteador, apresentado na seção 2.3. O módulo que representa o roteador está descrito em ***ac_tlm_router*** no diretório **is**. Com base na seção 3.3, o roteador foi implementado de modo que somente um processador consiga acessá-lo por vez, além disso, todo o mapeamento de memória e IPs é feito pelo próprio roteador, dentro do método *transport* da interface TLM.

O diretório ***multi_powerpc***, dentro do diretório **platforms**, representa a plataforma implementada para os testes e o arquivo de definições, chamado **defs.arp**, contém a estrutura da plataforma. O código mostrado na figura 4.6 apresenta as definições padrão da plataforma de teste. Na linha 1, são definidos os diretórios dos IPs (memória, *locker* e *startup*). A linha 2, informa o diretório em que o roteador da plataforma é declarado. O diretório do modelo de processador, que executará nesta plataforma, está referenciado na linha 3. Na linha 4, é informado o diretório do programa que será executado pela plataforma de simulação. A linha 5 não está completamente preenchida, porque esta plataforma não necessita de *wrapper*.

```
1 IP := ac_tlm_mem ac_locker_module ac_tlm_startup
2 IS := ac_tlm_router
3 PROCESSOR := powerpc
4 SW := ocean
5 WRAPPER :=
```

Figura 4.6: Arquivo de definições contendo a estrutura da plataforma

Ainda no diretório ***multi_powerpc***, o arquivo **main.cpp** é responsável pela instanciação dos componentes da plataforma, como: processadores, IPs e roteador. Todas as conexões de portas e canais são estabelecidas e, além disso, é nele em que o programa a ser simulado é carregado nos processadores e a simulação é inicializada.

4.2.1 Programas de Testes

Os testes realizados neste trabalho são executados utilizando cinco programas do pacote Splash2 [30]. Este pacote foi desenvolvido especialmente para sistemas paralelos multiprocessados, especificamente com espaço de endereçamento compartilhado. Estes programas são descritos a seguir:

FFT: (Transformada Rápida de Fourier[1]) é uma versão em seis passos distintos que utiliza o algoritmo $radix - \sqrt{n}$ onde n é o número de pontos do vetor. Os pontos são organizados em matrizes $\sqrt{n} \times \sqrt{n}$ que são distribuídas entre os processadores de modo que cada um receba um conjunto contíguo de pontos. Esse algoritmo exige que o número de *threads*, nesse caso de processadores, seja potência de dois e o tamanho da matriz seja maior ou igual ao número de *threads*. A verificação dos resultados obtidos foi feita com a própria função de teste do algoritmo, que calcula a transformada inversa e compara os resultados com o conjunto inicial de pontos.

LU: [2] fatora uma matriz densa no produto de uma matriz triangular inferior (*lower*) e uma matriz triangular superior (*upper*). A matriz densa A de $n \times n$ elementos é dividida em um *array* $N \times N$ de $B \times B$ blocos ($n = NB$). Um bloco de tamanho B deve ser grande o suficiente para manter a taxa de *cache miss* baixa e pequeno o suficiente para obter um bom balanceamento da carga nos processos. Os elementos de um bloco são alocados de forma contígua e os blocos são alocados localmente aos processadores que os estão computando.

Ocean: este programa estuda os movimentos de oceanos em grande escala baseado nos ventos e correntes. Nessa versão, o problema é dividido em matrizes quadradas e estas são representadas como matrizes de 4 dimensões, com todas as submatrizes alocadas de maneira contígua ao local de execução. Para o cálculo do problema o programa utiliza o algoritmo *red-black Gauss-Seidel*.

Water-NSquared: esta aplicação calcula as forças e potenciais que ocorrem ao longo do tempo em um sistema de moléculas de água. Toda a computação é feita utilizando um algoritmo $O(n^2)$. Cada processador precisa de todos os dados de entrada, mas somente faz um subconjunto de cálculos e armazena os resultados localmente. No final de cada etapa de cálculos, os processadores gravam seus resultados em uma área de memória compartilhada. Portanto, existem alternâncias entre leitura compartilhada e fases de atualização.

Water-Spatial: esta aplicação resolve o mesmo problema que Water-NSquared, mas utiliza uma abordagem diferente no algoritmo. Nesse programa, o problema é modelado como uma matriz de 3 dimensões de células e utilizam um algoritmo $O(n)$, que é mais eficiente que o algoritmo de Water-NSquared para um número maior de moléculas. Cada processador que possua uma célula somente precisa procurar pelas moléculas dentro desse subconjunto. O movimento de moléculas para dentro

[1] Do inglês *Fast Fourier Transform*
[2] Do inglês *Lower and Upper*

ou para fora das células faz com que atualizações sejam necessárias, resultando em comunicação entre os processadores.

Em vários pontos dos programas utilizados é necessário trocar dados entre os processadores ou então fazer a sincronização entre eles. Além disso, por diversas vezes os processadores também precisam acessar exclusivamente variáveis globais. Para isso foi desenvolvida uma pequena biblioteca que implementa algumas das funções básicas da biblioteca Posix, que não está disponível para o *cross-compiler* utilizado pelos modelos ArchC. Essa biblioteca oferece funções, como barreira, *mutex* e *join point*. Atividades como incrementar ou decrementar uma barreira, adquirir ou liberar um *lock*, incrementar o *join point*, são todas realizadas dentro do *mutex* implementado pelo módulo ***ac_locker_module*** da plataforma.

```
1  unsigned int *STARTUP = 0x800000;
2
3  int main(int argc, char *argv[])
4  {
5    register int procNumber;
6
7    AGlobalLock();
8        procNumber = *STARTUP;
9    RGlobalLock();
10
11   if (procNumber == 1){
12       main0(argc, argv);
13   } else{
14       WorkStart();
15   }
16
17   exit(0);  // To avoid cross-compiler exit routine
18   return 0; // Never executed, just for compatibility
19
20 }
```

Figura 4.7: Divisão das tarefas entre os processadores no programa Water-NSquared

Além disso, os programas do pacote Splash2 são codificados para dividir as tarefas utilizando *pthreads*. No ambiente de testes desse trabalho a rotina executada pela *pthread* passa a ser diretamente executada pelos processadores da plataforma. O trecho de código da figura 4.7 foi extraído do programa *Water-NSquared*. Na linha 1, o módulo *ac_tlm_startup* é endereçado e será responsável por gerar um identificador único a cada processador. Na linha 8, esse identificador é atribuído ao processador dentro do *mutex* global da plataforma, por operações definidas na biblioteca implementada. Uma nova função *main* executa a divisão das tarefas entre os processadores, nesse caso, o processador com identificador 0 será o responsável por iniciar os dados executando o antigo *main* (agora *main0*), e pela captura e teste das soluções calculadas pelos demais processado-

res. Os outros processadores executarão a rotina anteriormente executada pelas *pthreads* $WorkStart()$.

4.3 Desempenho

A caracterização das máquinas utilizadas para os testes com a plataforma da figura 4.5 executando os programas descritos na seção 4.2.1 pode ser observada abaixo:

Máquina A: Intel Core i7 860 (2,80GHz) com 4 cores e 8 threads, 8MB de memória cache, 5599,99 bogomips, Ubuntu 9.10, kernel 2.6.31-20.

Máquina B: AMD Opteron Processor 6168 (1,9GHz) com 12 cores, 12x128KB de cache L1, 12x512KB de cache L2 e 12288KB de cache L3, 3800,07 bogomips, Ubuntu 10.04 (64bits), kernel 2.6.32-22.

Todos os experimentos foram executados com todas as compilações possíveis do SystemC, tanto as originais quanto as geradas nesse trabalho, para fins de comparação de desempenho. Para cada configuração (versão do SystemC, número de *threads*) apresentada nos gráficos desse capítulo, as simulações foram repetidas 4 vezes, devido às pequenas variações dos tempos de execução entre elas. O tempo de execução e as instruções executadas por segundo de cada configuração representam a média aritmética dos dados obtidos pelas 4 repetições. Para facilitar a leitura dos gráficos e tabelas do apêndice A, foram criadas nomenclaturas para cada compilação do SystemC que são descritas a seguir:

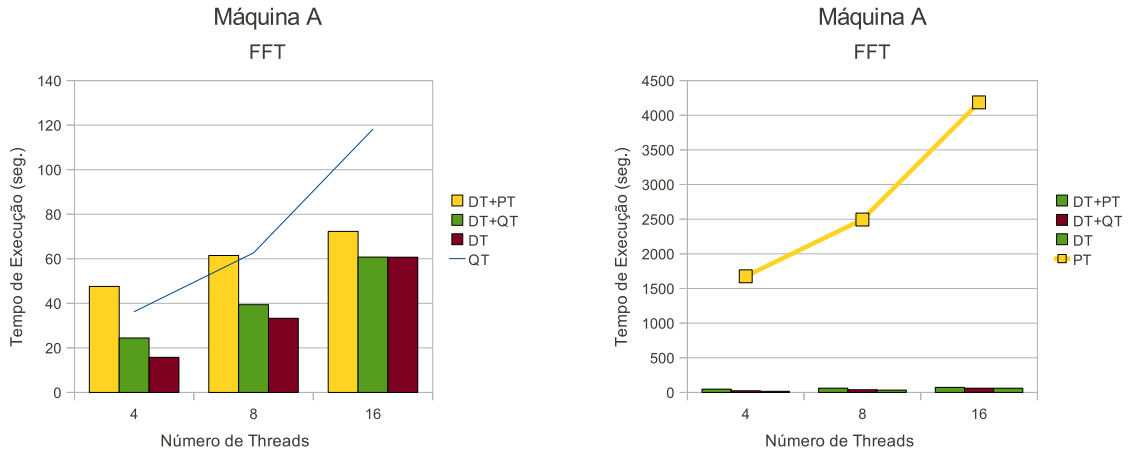
QT: Distribuição padrão do SystemC executando os processos tipo *threads* como *Quick Threads*.

PT: Mesma distribuição anteriormente citada porém executando as *threads* como *PThreads*.

DT+PT: Versão compilada do SystemC utilizando *dthreads*, porém, com a presença da *thread* interna ao núcleo do simulador, que é executada como uma *pthread*.

DT+QT: SystemC utilizando *dthreads* com a presença da *thread* interna, diferentemente de DT+PT, essa *thread* é executada como uma *quick thread*.

DT: Versão do SystemC executando os processos do tipo *thread* somente como *dthreads*, não existindo a *thread* interna.



(a) QT (b) PT

Figura 4.8: Tempos das simulações do programa FFT (Máquina A).

Vale a pena enfatizar que, nem todas as simulações foram executadas sob a compilação original do SystemC utilizando *pthreads* (PT). Isso é devido ao baixo desempenho dessas simulações e, consequentemente, à alta demanda de tempo para executá-los não servem como um bom referencial para comparações e cálculos de desempenho.

A figura 4.8 apresenta os gráficos com os tempos de execução das simulações do programa FFT na plataforma de testes na máquina A. Sendo que, no gráfico 4.8a, a linha representa o tempo de execução do SystemC somente com *quick threads* (QT), enquanto que no gráfico 4.8b a linha representa o tempo de execução do SystemC somente com *pthreads* (PT). O eixo X correspondente ao número de *threads* utilizadas para a execução do programa e o eixo Y o tempo de simulação necessário, para finalizar toda a execução. A tabela A.1 com os resultados detalhados das simulações de FFT na maquina A pode ser encontrada no Anexo A.

Devido as limitações de alocação de memória da plataforma de testes, as simulações com o programa FFT ficaram restritas em 16 processadores com o valor de entrada 10, o que corresponde à uma matriz de 1024 números complexos da transformada. Devido ao próprio tamanho do problema, esta é a única simulação realizada sob a configuração do SystemC com *pthreads* (PT).

Nos gráficos da figura 4.8, é possível verificar que, para todas as configurações do SystemC, a execução do programa em mais *threads* não apresenta melhoria de desempenho, porque o tamanho do problema não é grande o suficiente para justificar a presença dessas *threads* e toda a sincronização envolvida. Analisando a figura, individualmente

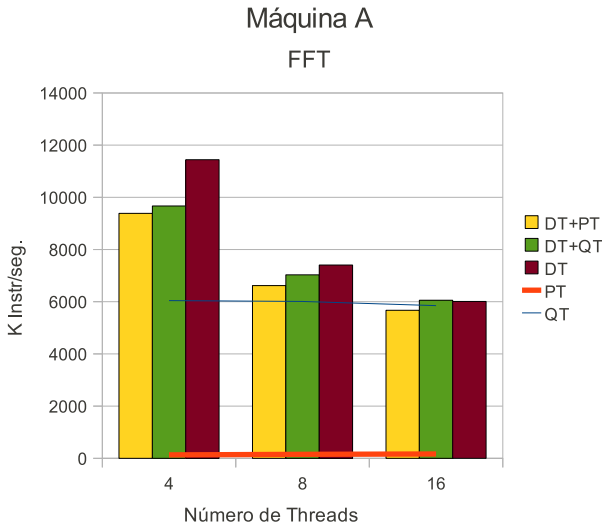
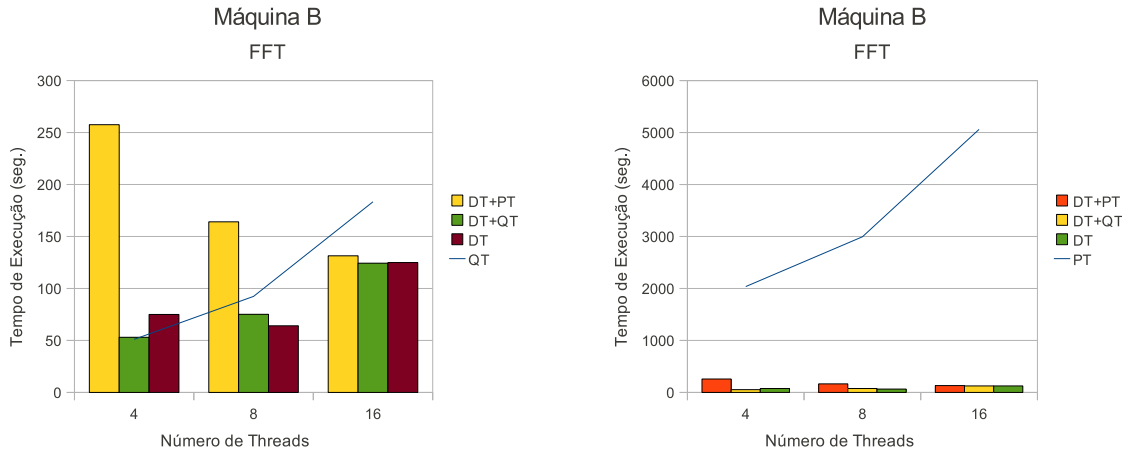


Figura 4.9: Instruções executadas por segundo das simulações do programa FFT (Máquina A).

para cada quantidade de *threads*, nota-se a evolução entre as versões do SystemC, com tempos de simulação cada vez menores. Com destaque para a versão executando somente *dthreads* (DT) seguida pela versão executando uma *quick thread* internamente ao núcleo de simulação (DT+QT) com tempos de simulação sempre abaixo das versões originais. Além disso, analisando o gráfico da figura 4.9 e as colunas da tabela A.1, que representam a quantidade de instruções executadas por segundo (K Instr/s) é possível avaliar melhor a evolução de desempenho entre as versões. Por exemplo, para as simulações com 8 *threads* a versão DT executou 7405,867K instr/s contra 6009,113K instr/s da versão original QT. Utilizando o tempo de simulação para calcular o total de instruções executadas por cada versão tem-se que a versão DT executou menos instruções que QT, 246.452.454 instruções contra 376.416.874 instruções, respectivamente. Sendo que o mesmo programa fora executado em ambas configurações, esperava-se que o número de instruções executadas fossem os mesmos ou próximos em ambas mas, devido à presença do *lock* na plataforma, associada ao desempenho do modelo de execução do SystemC utilizado e ao tamanho da região crítica do programa, que esse *lock* protege, esses números sofreram alterações. Por exemplo, no modelo QT, se uma *thread*, por algum motivo, adquirir o *lock* e for reescalonada, as demais *threads* continuam a ser escalonadas e a executar as instruções de leitura e aquisição do *lock* sem êxito, até que ele seja liberado e uma delas consiga adquiri-lo. Se a plataforma contiver um grande número de *threads*, então mais instruções serão executadas e mais tempo a *thread* detentora do *lock* levará para liberá-lo, por causa do escalonador



(a) QT (b) PT

Figura 4.10: Tempos das simulações do programa FFT (Máquina B).

do SystemC.

A figura 4.10 apresenta os gráficos os tempos de execução das simulações da plataforma com o programa FFT na máquina B. A disposição dos dados no gráfico segue o mesmo padrão dos gráficos da figura 4.8. A tabela A.2 com os resultados detalhados das simulações de FFT na maquina B pode ser encontrada no Anexo A.

Visivelmente os tempos das execuções na máquina B são maiores que os registrados na máquina A, devido ao tamanho de cache menor e à frequência de relógio ser inferior, entre outros aspectos do hardware. Na máquina B não há melhoras nos tempos de execução de DT+PT, DT+QT e DT, na configuração de 4 *threads*, em relação à QT. Sob a configuração de 8 *threads*, somente, DT+PT não melhora o seu tempo de execução. Mas, utilizando 16 *threads*, todas novas versões do modelo de execução do SystemC registram bons resultados. As quantidades de instruções executadas por segundo, pelas simulações com FFT na máquina B, podem ser visualizadas no gráfico da figura 4.11.

Com o auxílio dos gráficos da figuras 4.12 e 4.13, é possível uma melhor avaliação do desempenho (*speedup*) das novas versões do modelo execução do SystemC em relação às versões originais do mesmo. O eixo X dos gráficos representam a quantidade de *threads* em que a simulação foi executada (4, 8 ou 16). O eixo Y representa o ganho obtido pela simulação com a versão paralela do SystemC em relação à versão original, QT ou PT. A tabela A.3, com os resultados mais detalhados dos desempenhos das versões pode ser encontrada no Anexo A.

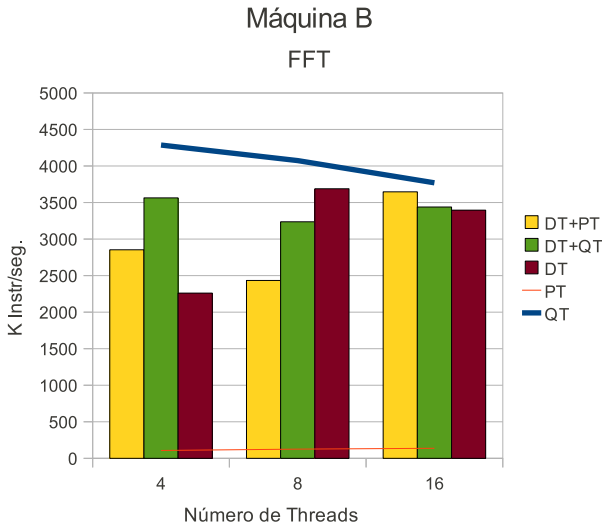


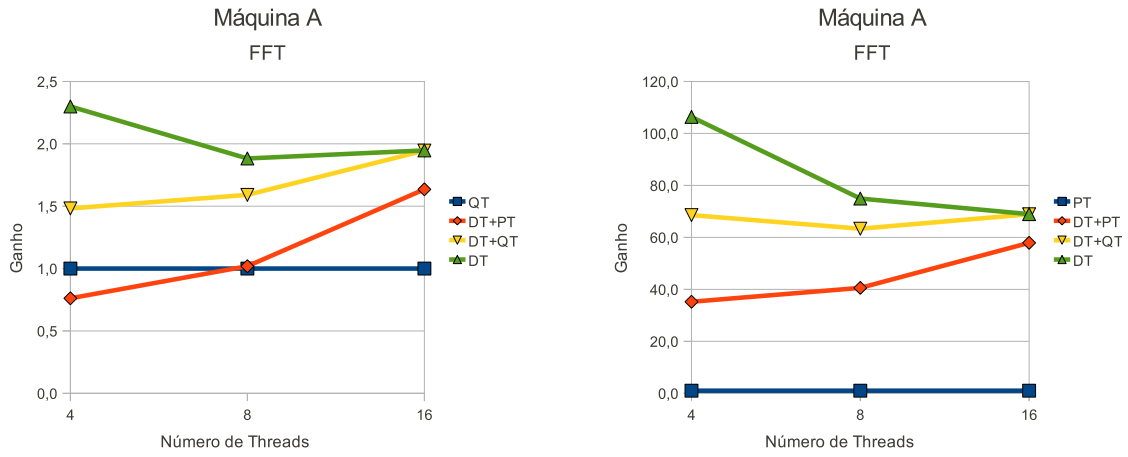
Figura 4.11: Instruções executadas por segundo das simulações do programa FFT (Máquina B).

O desempenho ao longo deste capítulo é medido pela equação:

$$desempenho = \frac{tempo\,versão\,original}{tempo\,nova\,versão}$$

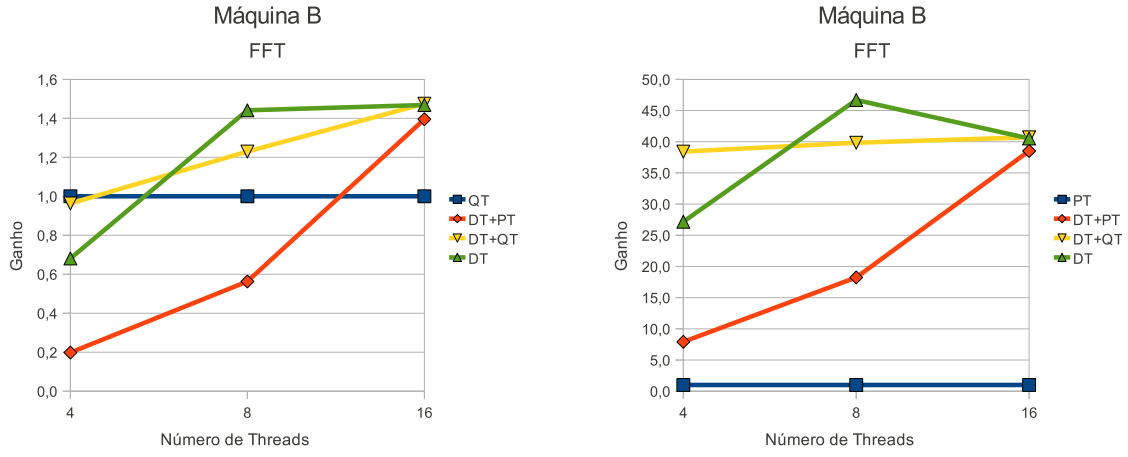
Os resultados apresentados nas figuras 4.12 e 4.13 que ficaram abaixo de 1 indicam que, sob essa configuração do programa FFT o modelo em questão não apresenta ganho de desempenho em relação à versão original do SystemC. Isto ocorre, por exemplo, com a versão DT+PT executando em 4 *threads* quando comparada à versão original QT na máquina A, em que o desempenho foi de 0,761×, ou seja, não houve ganho de desempenho. O melhor desempenho obtido na máquina A foi de 2,299× da versão DT, executando com 4 *threads*, quando comparada à versão QT. Na máquina B, o melhor desempenho, de 1,475×, foi registrado por DT+QT sob a configuração de 16 *threads*, quando comparada à versão QT.

Ainda pelas figuras 4.12 e 4.13, comparando os desempenhos das versões DT+QT e DT para todas as quantidades de *threads*, em ambas máquinas, é possível verificar a aproximação do desempenho entre as versões à medida que o número de *threads* aumenta. Por exemplo, o desempenho de DT+QT e DT, nessa ordem, com 16 *threads* na máquina A foram de 1,946× e 1,947× e na máquina B os desempenhos de 1,475× e 1,468×. Isso se deve, em parte, ao próprio modelo de plataforma utilizado, que faz com que o roteador



(a) QT (b) PT

Figura 4.12: Desempenho do programa FFT (Máquina A).



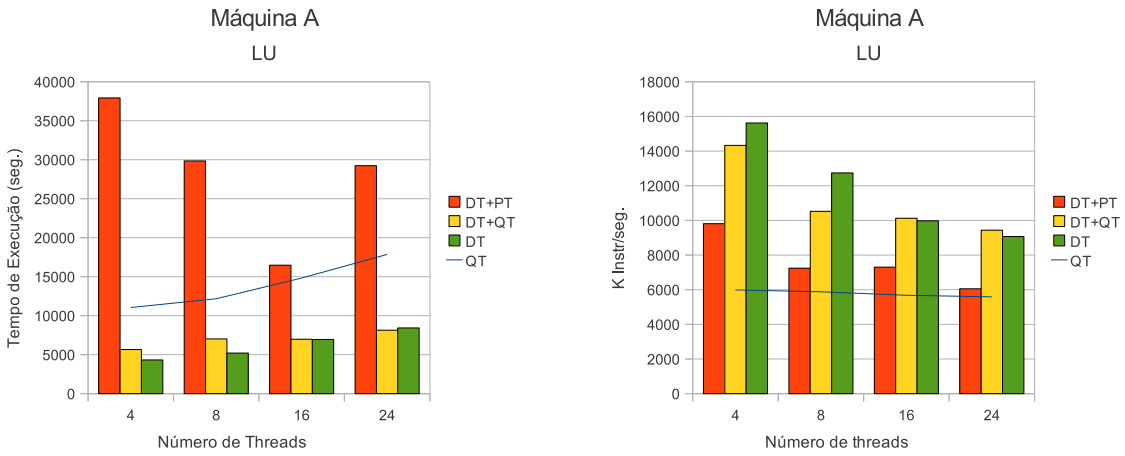
(a) QT (b) PT

Figura 4.13: Desempenho do programa FFT (Máquina B).

seja um gargalo de execução, devido ao acesso exclusivo ao dispositivo.

Os desempenhos obtidos em relação à versão original do SystemC PT são utilizados aqui somente para ilustrar a ineficiência do modelo de execução, uma vez que o seu desempenho, dependendo da máquina utilizada, pode variar de 7,908 à 106,353 vezes mais lento que as demais versões. Como mostram as figuras 4.12b e 4.13b. Este fato torna inviável a execução com os demais programas de testes com problemas maiores que o FFT. Além disso, esse modelo não serve como referência para cálculos de desempenho.

O segundo programa executado do pacote Splash2 é o LU. Diferentemente do programa anterior, ele não possui restrições de execução, o que permitiu utilizar até 24 *threads* (processadores) na execução paralela na máquina A. Os gráficos figura 4.14 ilustram o comportamento da simulação sob os modelos de execução do núcleo de simulação do SystemC e a quantidade de *threads* executadas. Os dados obtidos pelas simulações estão particionados entre as tabelas A.4 e A.5. Além das colunas com a versão compilada do SystemC (SystemC), tamanho do problema de entrada do programa (Carga), tempo real de execução em segundos (Tempo) e instruções executadas por segundo (K Instr/s), agora, o desempenho (*Speedup*) de cada configuração de simulação, é inserido na tabela, uma vez que somente a versão original do SystemC QT é a referência de cálculo de desempenho.



(a) Tempos de Execução

(b) Instruções/Seg.

Figura 4.14: Dados das simulações do programa LU (Máquina A).

Pelos gráficos da figura 4.14, utilizar a versão DT+PT não apresentou ganho de desempenho em relação à QT, se mostrando ineficiente na paralelização de LU. A quantidade de instruções executadas por segundo de DT+PT (figura 4.14b), por exemplo, com 4 *threads* foi alta, 9810,936 K Instr/s, com uma melhor avaliação, têm-se que o número total de

instruções executadas pelo modelo foi de 372.118.261.953 instruções, ou seja, 5,629 vezes maior ao que foi executado por QT. O mesmo vale para todas as configurações executadas com DT+PT. Isso porque no modelo de simulação DT+PT, a *thread* interna executando como *pthread* usa um tempo muito grande para executar o código protegido pelo *lock* para, enfim, liberá-lo, fazendo com que as demais *dthreads* executem muitas instruções, tentando adquirir o *lock*.

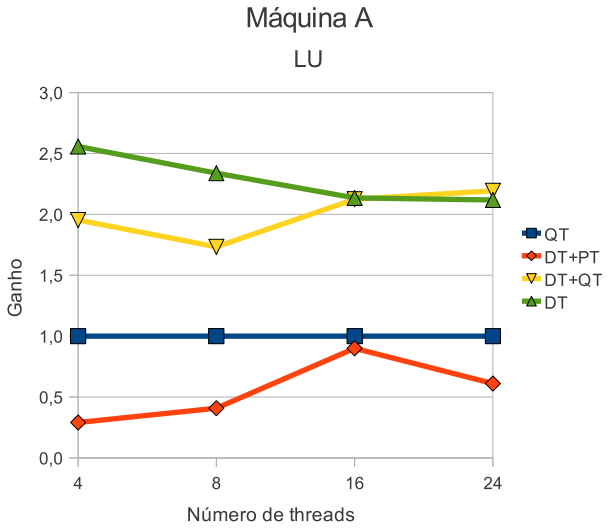
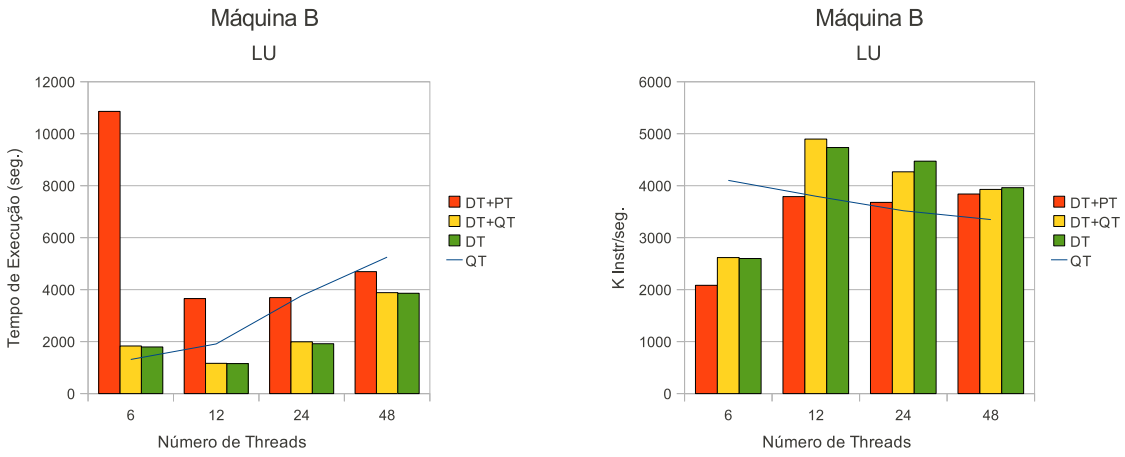


Figura 4.15: Desempenho do programa LU (Máquina A).

A figura 4.15 apresenta do gráfico de desempenho dos modelos de execução com as diferentes quantidades de *threads*. Os bons desempenhos são obtidos por DT+QT e DT, com destaque para DT que manteve desempenhos acima de 2,0× para todas as configurações. Ainda, com o auxílio das tabelas A.4 e A.5 verifica-se que, devido a necessidade de sincronização no roteador, os desempenhos de DT+QT e DT tendem a ficar próximos, à medida que a quantidade de *threads* do modelo aumentam. Observe que, na configuração do modelo com 24 *threads*, DT+QT registra o desempenho de 2,193×, que é superior ao desempenho de 2,118× de DT.

Os gráficos da figura 4.16 juntamente com as tabelas A.6 e A.7, apresentam os dados das simulações com o programa LU quando executadas na máquina B. A carga do problema foi reduzida para facilitar as simulações, que são mais lentas na máquina B. E a quantidade de *threads* utilizadas na paralelização dos testes, 6, 12, 24 e 48. Mostrando a capacidade dos novos modelos de SystemC paralelo de serem escaláveis.



(a) Tempos de Execução (b) Instruções/Seg.

Figura 4.16: Dados das simulações do programa LU (Máquina B).

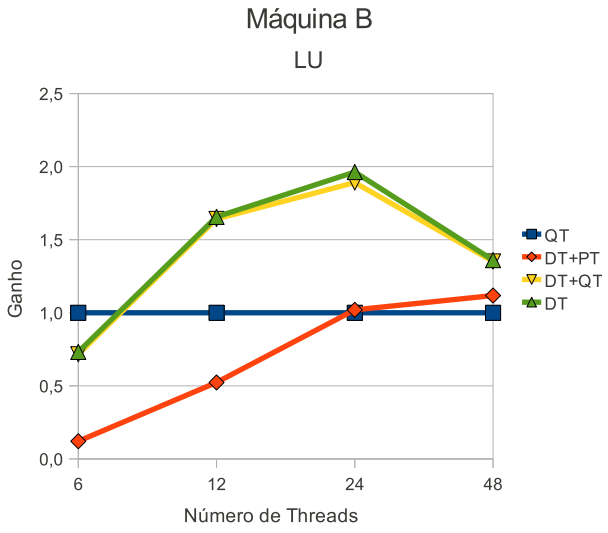


Figura 4.17: Desempenho do programa LU (Máquina B).

Com uma carga 2,5 vezes menor do que a utilizada na máquina A, ganhar desempenho em um ambiente onde o SystemC QT ainda seja eficiente é uma atividade consideravelmente complicada. Essa situação pode ser melhor visualizada na configuração com 6 *threads*. Nesse caso, nenhuma das novas versões paralelas do SystemC obteve desempenho superior à versão original. O custo de criar, manter e sincronizar as *threads*, com um problema pequeno de se calcular não é viável nesse modelo, diferentemente do que aconteceu na máquina A com 4 *threads*.

À medida que o número de *threads* utilizadas na execução das simulações é incrementado, as versões mais rápidas (DT+QT e DT) do SystemC *Multi-Threaded* tendem a ter desempenhos cada vez melhores a QT, principalmente, quanto maior o tamanho do problema de entrada.

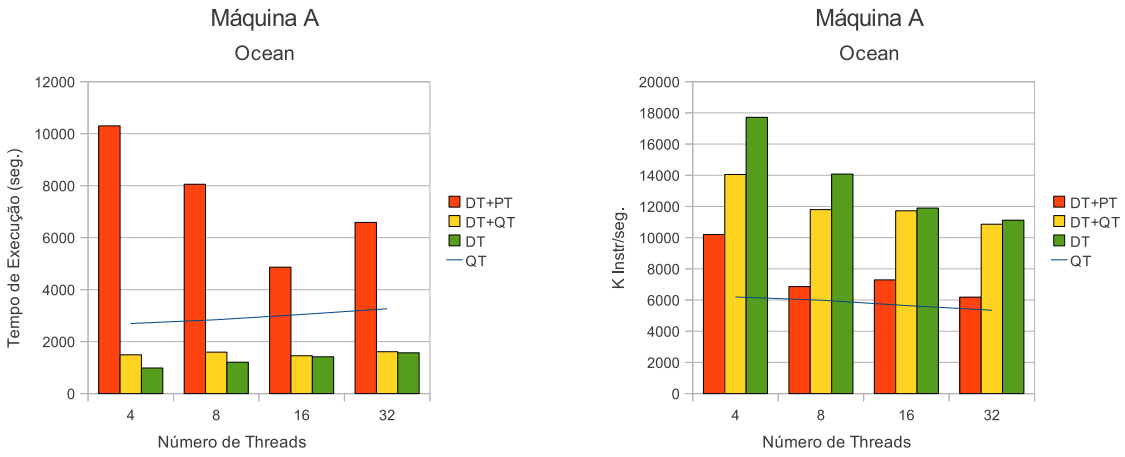
Um ponto a ser destacado no gráfico 4.16a e nos dados contidos na tabela A.6, são os tempos das simulações de LU com DT+QT e DT executando em 12 *threads*, 1.163,920 e 1.154,217 segundos, respectivamente, que foram inferiores a todos os demais tempos obtidos, inclusive dos modelos com 6 *threads*. Mas, os melhores desempenhos registrados por DT+QT e DT são vistos nas execuções com 24 *threads* (figura 4.17), sendo de 1,891× e 1,963 ×, respectivamente.

Os resultados apresentados, a seguir, na figura 4.18 e nas tabelas A.8 e A.9 do apêndice A, se referem a simulação do programa Ocean com uma matriz de entrada de 130x130, na máquina A. Esse programa exige que o número de *threads* utilizadas para processar a matriz de entrada seja uma potência de 2, sendo assim, o maior número de *threads* utilizadas nessa simulação foi de 32 *threads* ao invés de 24. Gerando uma carga maior nos núcleos de simulação da máquina A.

Primeiramente, atente que a simulação com a versão DT+PT não registra tempos de simulações menores do que os obtidos pela versão original QT do SystemC para processar a aplicação Ocean. Tanto que, o melhor desempenho dessa versão acontece na configuração com 16 *threads* e é de apenas 0,626×. Os desempenhos são mostrados no gráfico da figura 4.19. Mais uma vez, constata-se que essa versão paralelizada do SystemC, em certos casos, tende a executar um número maior de instruções do que as demais versões, para o mesmo problema.

A simulação com DT+QT, mostrou-se eficiente em todas as configurações, com média de desempenho de 1,927×. Sendo que o melhor desempenho obtido foi na configuração com 16 *threads* com 2,091× de ganho em relação à QT. Também sob essa mesma configuração que a versão DT+QT registrou seu melhor tempo de execução, de 1.455,221 segundos com 11.722,894K instruções executadas por segundos.

Os melhores desempenhos no geral foram registrados nas simulações com DT. Em média 2,330×. Com destaque para a configuração com 4 *threads* que obteve um desempenho de 2,737× e 17.716,372K instruções executadas por segundo. Para as demais



(a) Tempos de Execução (b) Instruções/Seg.

Figura 4.18: Dados das simulações do programa OCEAN (Máquina A).

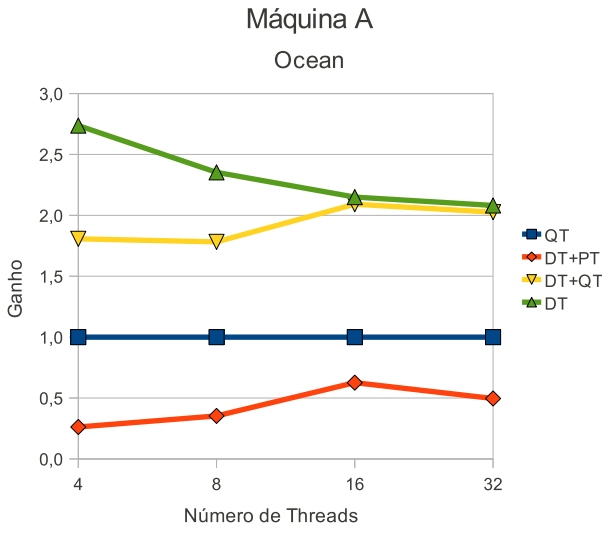
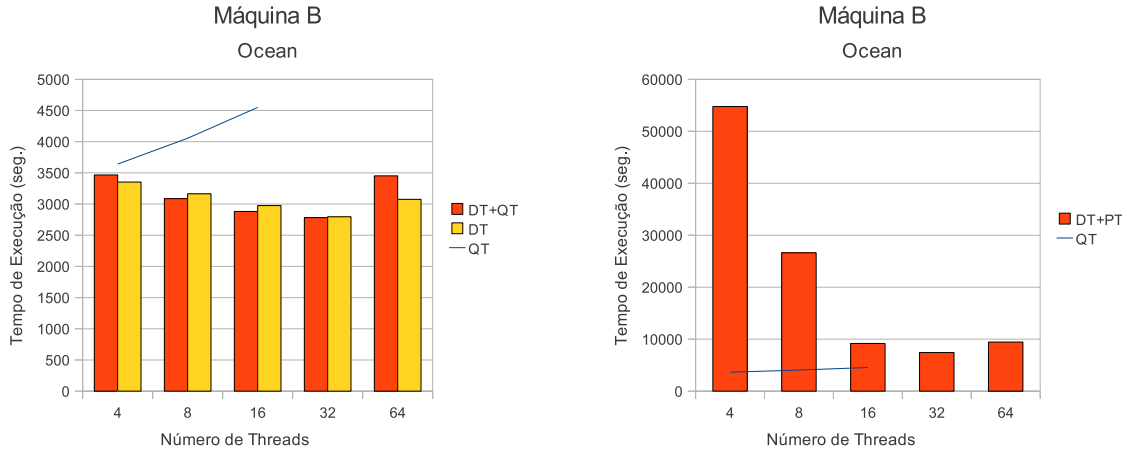


Figura 4.19: Desempenho do programa OCEAN (Máquina A).

configurações, com esse tamanho de matriz de entrada, o acréscimo de *threads* tende a deteriorar o desempenho, não compensando toda a sincronização envolvida no modelo de simulação.

Os gráficos das figuras 4.20 e 4.21 juntamente com as tabelas A.10 e A.11 apresentam os dados da simulação do programa Ocean na máquina B. Tanto os gráficos quanto a tabela A.11, as simulações com QT com 32 e 64 *threads* não contém dados. Isso porque, nessas configurações a simulação entrou em uma situação de execução infinita ou de término indeterminado, cujo o motivo não se tornou evidente no processo de depuração. Uma possível causa talvez seja uma condição de corrida presente no próprio programa Ocean, que ficou evidente somente sob essas configurações. Problemas semelhantes foram encontrados anteriormente, porém, devido ao tamanho e estrutura do código do programa foi possível resolver tal problema. Para programas mais complexos como Ocean, essa tarefa se torna complexa, principalmente em um ambiente paralelo de computação.

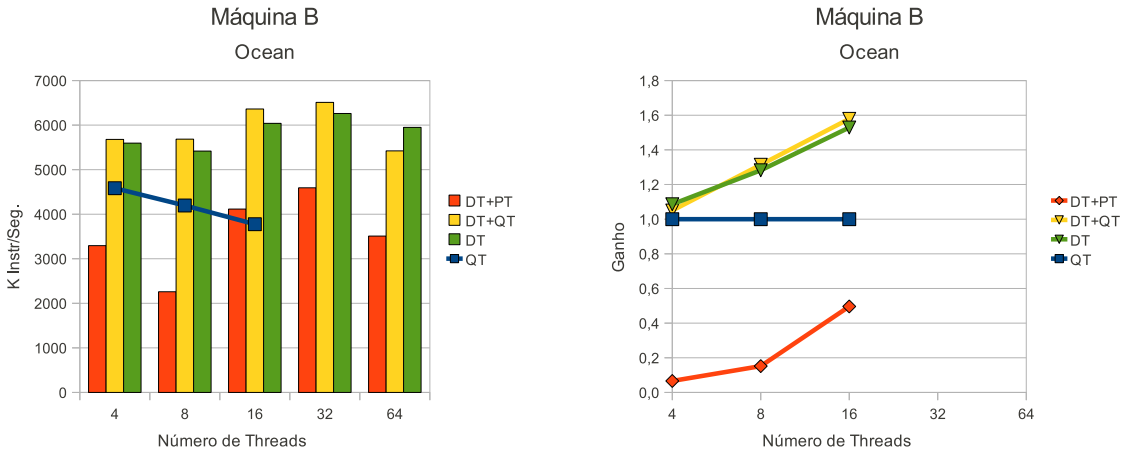


(a) Tempos de Execução

(b) Tempos de Execução de DPT

Figura 4.20: Dados das simulações do programa OCEAN (Máquina B).

Quanto aos demais resultados, diferentemente da versão QT, em que o tempo de simulação tende a aumentar com o número maior de *threads* (figura 4.20), as simulações de Ocean na máquina B, com as versões DT+QT e DT apresentaram ganhos de desempenho e tempos de simulação melhores com o acréscimo de mais *threads*, até a configuração com 32 *threads*. A figura 4.21b apresenta o gráfico de desempenho da simulação da plataforma de testes executando o programa Ocean. Apesar das configurações com 32 e 64 *threads* não possuírem os tempos de QT para cálculo de desempenho, é possível concluir que essas simulações executariam em torno 1,5 vezes mais lentamente que as versões DT+QT e DT.



(a) Instruções/Seg.

(b) Desempenho

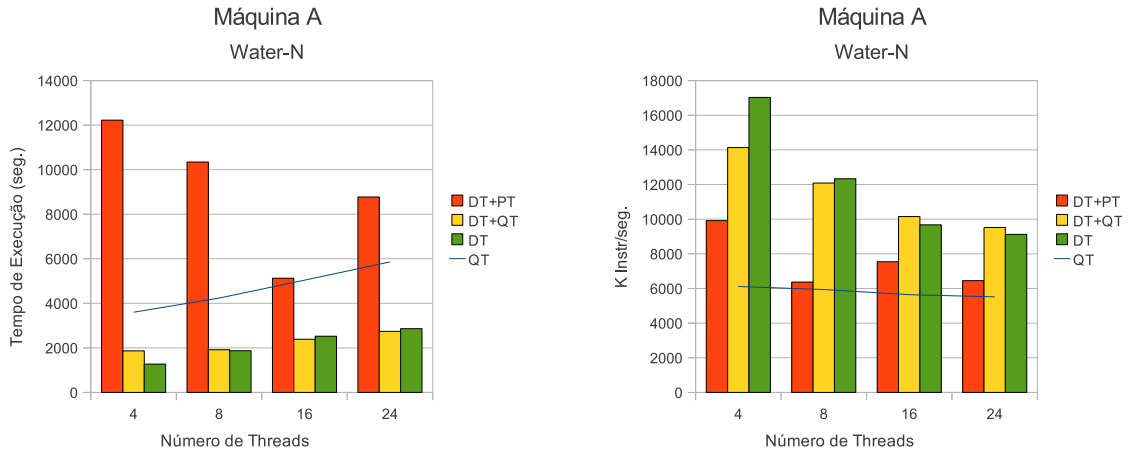
Figura 4.21: Dados das simulações do programa OCEAN (Máquina B).

Exceto pelo tempo e desempenho da simulação de Ocean com 4 threads, a versão DT+QT paralela do SystemC foi a que obteve os melhores resultados. Sendo que o melhor tempo de simulação, 2.782,435 segundos, com 6.510,733K Instr/s foi registrado na configuração com 32 *threads*. Mesmo que a versão DT+QT tenha sido a de melhor desempenho geral, as simulações com DT mantém desempenhos próximos, por exemplo, com as mesmas 32 *threads*, ela registrou o tempo de 2.795,487 segundos e 6.260,907K Instr/s.

Executar com a versão DT+PT continua a demonstrar que a implementação de processos mapeados como *pthreads* e escalonados pelo núcleo de simulação do SystemC é uma solução ineficaz, com forte influência no desempenho da simulação, mesmo com a presença das *dthreads*. Além de não haver melhoria de desempenho, seu comportamento é irregular, podendo variar pela carga de processamento na *thread* interna ao núcleo do SystemC ou pelo processamento de um número maior de instruções, como visto anteriormente.

Apesar dos desempenhos obtidos pela máquina A serem mais significativos do que os obtidos pela máquina B, as simulações mostraram que, as versões DT+QT e DT do SystemC são sempre mais eficientes nos ambientes com mais *threads* em execução e mais paralelismo.

Os resultados das simulações realizadas na plataforma de testes com o programa Water-NSquared na máquina A, são apresentados nos gráficos da figura 4.22 e nas tabelas A.12 e A.13 no apêndice A.



(a) Tempos de Execução

(b) Instruções/Seg.

Figura 4.22: Dados das simulações do programa Water-NSquared (Máquina A).

As simulações com Water-NSquared foram executadas com uma carga de 343 moléculas. Devido ao problema de entrada dessa aplicação exigir que ele seja um número inteiro cúbico. O número de *threads* utilizadas para as simulações foram de 4, 8, 16 e 24 *threads* para todas as versões aqui apresentadas do SystemC, exceto a versão original PT que, devido ao seu baixo desempenho, permaneceu fora das execuções.

Novamente, a versão paralela DT+PT do SystemC não têm ganhos de desempenho em relação à versão original QT. O mais próximo que ela ficou do tempo de execução de QT foi na configuração com 16 *threads*, com o tempo de 5.126,226 segundos contra 5.038,194 segundos de QT.

O gráfico da figura 4.23 mostra os desempenhos das versões paralelas do núcleo de simulação do SystemC em relação ao núcleo original QT. Os melhores desempenhos, no geral, foram alcançados pela versão totalmente paralela DT. A média de ganho de desempenho, em relação à QT, foi de 2,284×. Sendo o melhor desempenho de 2,831× e o pior desempenho de 1,997×, com 4 e 16 *threads*, respectivamente. Quanto à versão paralela DT+QT, apesar de seu melhor tempo de simulação ter sido de 1865,073 segundos com 4 *threads*, seu melhor desempenho em relação à QT foi de 2,209× com 8 *threads*. Destaca-se também, o melhor desempenho de DT+QT sobre DT nas configurações com 16 e 24 *threads*, 2,111× e 2,136× de DT+QT contra 1,997× e 2,046× de DT.

Os gráficos da figura 4.24 e as tabelas A.14 e A.15 no apêndice A, apresentam os dados com o programa Water-NSquared na máquina B. As simulações também foram feitas com uma carga de 343 moléculas, porém, o número de *threads* utilizadas foram de 6, 12, 24 e

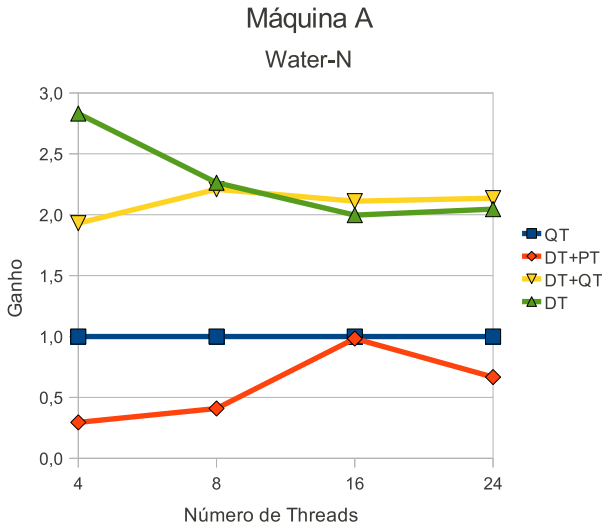
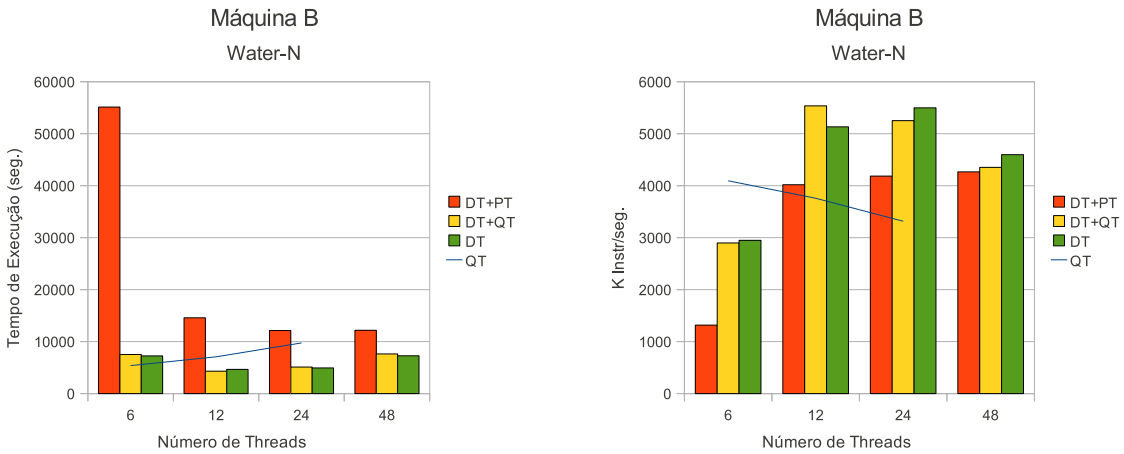


Figura 4.23: Desempenho do programa Water-NSquared (Máquina A).

48 *threads*.

Como o que aconteceu nas simulações na máquina A (figuras 4.22 e 4.23), as simulações de Water-NSquared com a versão DT+PT do SystemC paralelo, não registram ganhos de desempenho. Sendo que, na configuração com execução em 6 *threads* (figura 4.24a) o tempo de simulação de 55.127,334 segundos, foi aproximadamente 10 vezes maior que o tempo de simulação com o SystemC QT de 5.379,091 segundos. Das medidas de tempos possíveis de serem comparadas, o melhor desempenho de DT+PT em relação à QT foi de 0,803× quanto utilizadas 24 *threads* de execução (figura 4.25).

Curiosamente, as simulações com DT+QT e DT também não registram bons desempenhos na configuração com 6 *threads* na máquina B (figura 4.25). Possivelmente, devido às características de sincronização entre as *threads* internamente ao programa e o mecanismo de sincronização existente na própria plataforma de testes. Para as configurações com 12 e 24 *threads*, as versões DT+QT e DT apresentam melhores resultados em relação à QT. O melhor tempo de simulação foi registrado pela versão DT+QT executando com 12 *threads*, 4.303,722 segundos com desempenho de 1,643×. E o melhor desempenho foi registrado pela versão DT executando com 24 *threads*, de 1,973×. Mesmo que o tempo de simulação da versão QT não tenha sido registrado e sabendo que o tempo de simulação com essa versão tende a crescer com o acréscimo de *threads*, fica claro que as versões paralelas DT+QT e DT continuam a ter bons ganhos de desempenho.



(a) Tempos de Execução (b) Instruções/Seg.

Figura 4.24: Dados das simulações do programa Water-NSquared (Máquina B).

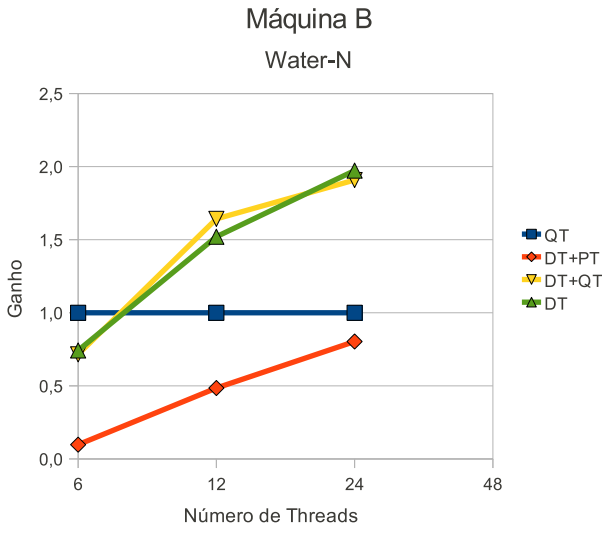
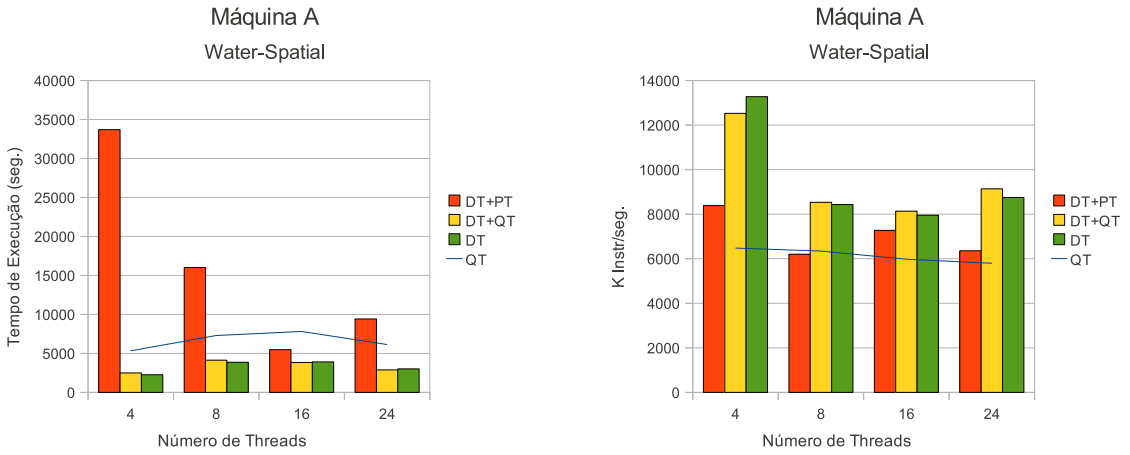


Figura 4.25: Desempenho do programa Water-NSquared (Máquina B).

Os resultados das simulações com o programa WATER-Spatial na máquina A, são apresentados nos gráficos da figura 4.26 e nas tabelas A.16 e A.17 no apêndice A. Esse programa, em especial, não foi simulado na máquina B.



(a) Tempos de Execução

(b) Instruções/Seg.

Figura 4.26: Dados das simulações do programa Water-Spatial (Máquina A).

Segundo [30], o programa Water-Spatial utiliza um algoritmo $O(n)$ mais eficiente que Water-NSquared $O(n^2)$ para um número maior de moléculas, o que não acontece aqui. Tanto que a carga ou quantidade de moléculas utilizadas para simulação foi reduzida de 343 para 216 moléculas, para facilitar as simulações.

Primeiramente, vale a pena destacar o comportamento da simulação de Water-Spatial em QT com 24 *threads* que, ao invés do tempo de simulação aumentar em relação ao tempo de simulação com 16 *threads*, diminuiu de 7812,003 segundos para 6.137,262 segundos. Mesmo comportamento apresentado por DT+QT e DT. Esses tempos de QT se mantiveram constantes durante todas as repetições da simulação.

Diferentemente do que aconteceu com Water-NSquared na máquina A, a simulação de Water-Spatial na versão DT+PT apresenta um bom caso de desempenho com 16 *threads*, de 1,425× em relação à QT, como mostra a figura 4.27. Para as demais quantidades de *threads*, QT permaneceu sem ganho de desempenho e, sem diminuir o tempo de simulação de 16 para 24 *threads*, como aconteceu nas demais versões.

Como apresentado em testes anteriores, simular com a versão DT+QT manteve um bom desempenho. A média de desempenho dessa versão executando Water-Spatial foi de 2,016× em relação à QT. Sendo seu melhor desempenho apresentado de 2,137× com 4 *threads*, e o pior desempenho de 1,766× com 8 *threads*. Para as configurações com 16 e 24

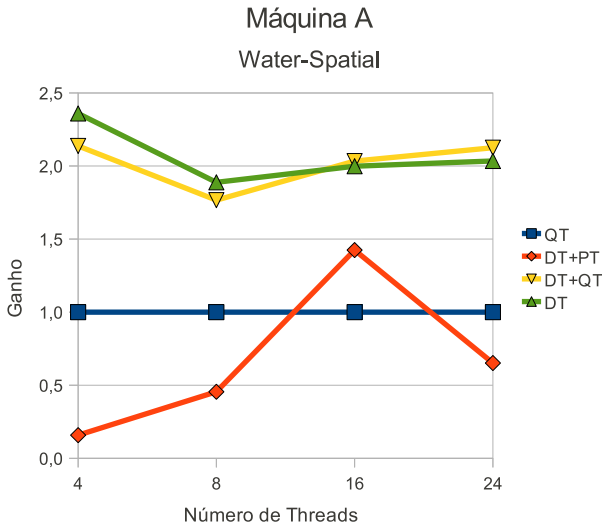


Figura 4.27: Desempenho do programa Water-Spatial (Máquina A).

threads, o desempenho de DT+QT foi superior ao desempenho apresentado pela versão DT. Que por sua vez, registrou o melhor desempenho geral de 2,360× com 4 *threads* e pior desempenho com 8 *threads* de 1,888×. Mesmo não superando o desempenho de DT+QT nas configurações com 16 e 24 *threads* ele se manteve próximo.

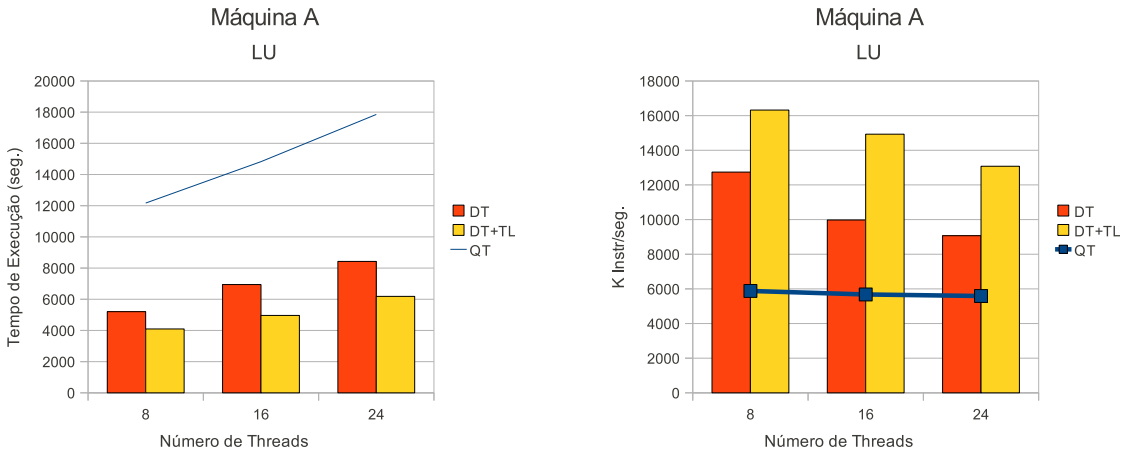
Na próxima seção, é apresentada uma análise mais aprofundada dos resultados registrados nessa seção e são apresentados alguns métodos para se obter um melhor desempenho do modelo *multi-threaded* sobre essa plataforma de testes em específico.

4.4 Modificações no Controle de Concorrência para Favorecer as Simulações *Multi-Threaded*

Nesta seção, são apresentadas algumas formas para se obter mais desempenho através de alterações no mecanismo de sincronização da plataforma de testes tornando-a favorável à simulação *Multi-Threaded*. Na seção anterior, vários testes com a plataforma proposta foram executados sob diversas configurações nas Máquinas A e B, variando o programa executado, a quantidade de *threads* e o modelo de execução do SystemC.

Mesmo com uma plataforma de simulação desfavorável à execução paralela, grande parte das simulações utilizando o SystemC *Multi-Threaded* registraram melhores desempenhos. O melhor ganho foi de 2,831× com a versão DT executando o programa Water-NSquared com 4 *threads* na máquina A. Valor este, abaixo do esperado.

Primeiramente, é alterada a implementação atual do *mutex* na interface TLM (apresentada na seção 3.3). O método *pthread_mutex_lock()*, internamente faz chamadas ao *futex* no núcleo do Linux e tem um *spinlock* de 300 tentativas antes de chamar o escalonador do Linux, o que pode acarretar em perda de desempenho. Portanto, ao invés de utilizar *pthread_mutex_lock()*, linha 20 da figura 3.10, esse método é substituído pelo laço *while (pthread_mutex_trylock()) sched_yield()*. Sendo assim, ao não conseguir o *lock* no *mutex* a *thread* é reescalonada e permite que outras *threads* executem. Isso pode melhorar o desempenho da simulação quando há uma grande quantidade de *threads*, pelo menos duas vezes maior que a quantidade de processadores, e uma carga razoável de trabalho a ser executada.



(a) Tempos de Execução

(b) Instruções/Seg.

Figura 4.28: Dados das simulações do programa LU com *TryLock* (Máquina A).

Os gráficos da figura 4.28, juntamente com as tabelas B.1 e B.2 no apêndice B, apresentam os resultados das execuções do programa LU na Máquina A com 8, 16 e 24 *threads* e tamanho do problema de entrada de 640 no modelo paralelo SystemC-DT. Para essas simulações, são repetidos os dados das simulações de LU na versão QT do SystemC, os dados das simulações obtidos anteriormente com a versão DT do SystemC paralelo e, são apresentados os dados da mesma versão DT do SystemC, porém, com a alteração na maneira como o *mutex* é utilizado na interface TLM, chamado de DT+TL (*DThread+TryLock*).

Todas as configurações de *threads* em DT+TL registraram desempenhos melhores que a versão DT em relação à versão original QT, como mostra a figura 4.29. Sendo que o melhor desempenho de 2,987× foi registrado pela configuração com 16 *threads*. Porém,

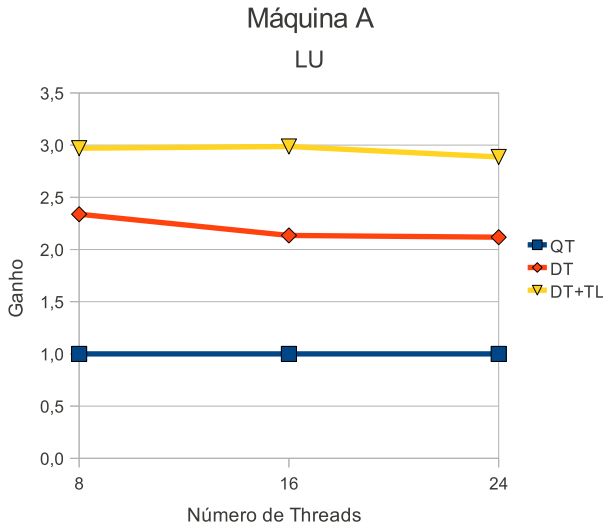
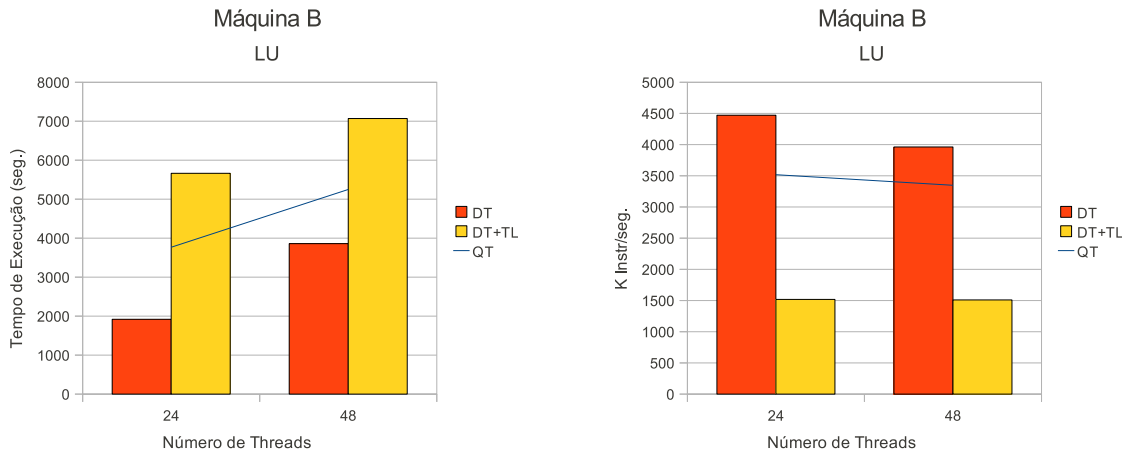


Figura 4.29: Desempenho do programa LU com *TryLock* (Máquina A).

essa alteração não resulta em melhor desempenho para todas as máquinas, como por exemplo, a máquina B. Os gráficos da figura 4.30 e a tabela B.3 (apêndice B) apresentam os resultados das simulações de LU com 24 e 48 *threads* na máquina B e tamanho do problema de entrada de 256.



(a) Tempos de Execução

(b) Instruções/Seg.

Figura 4.30: Dados das simulações do programa LU com *TryLock* (Máquina B).

Os resultados apresentados na figura 4.30 mostram que a mudança na forma como a operação de *lock* da interface TLM não trouxe benefícios às simulações na máquina B, podendo fazer com que estas passem a ter perdas de desempenho se comparadas a implementação anterior do *mutex* da interface. Por exemplo, na configuração com 24 *threads* o tempo de simulação aumentou em 50,3% em relação a versão não modificada da interface e, consequentemente, o desempenho foi de 0,665× (figura 4.31).

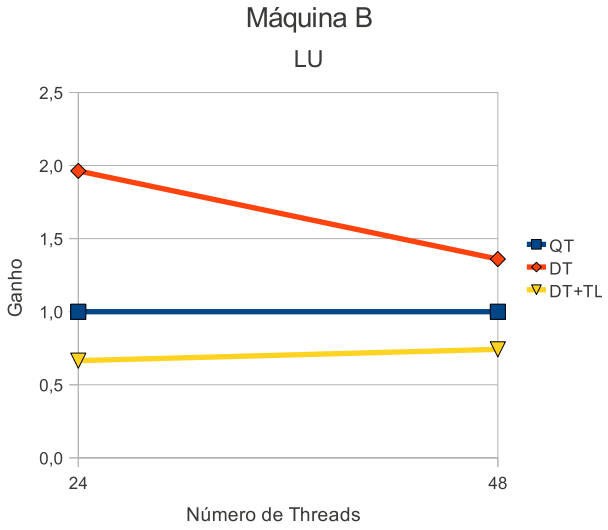


Figura 4.31: Desempenho do programa LU com *TryLock* (Máquina B).

Sendo o roteador da plataforma um ponto de grande contenção na execução paralela das simulações, em função da presença do *mutex* em sua interface do canal TLM. E que, se faça necessária a presença de um mecanismo de sincronização de acesso aos IP's e à memória, para manter a exatidão do resultado da simulação. Outra avaliação de desempenho está relacionada à retirada desse mecanismo de sincronização do roteador e sua distribuição em outros novos mecanismos de sincronização nos IP's e Memória conectados ao roteador. Dessa maneira, os processadores do modelo deixam de fazer a chamada ao método *transport_mutexed()* no roteador e esse passa a realizar essa chamada nas interfaces TLM dos canais que o conecta os IP's e memória. A nova plataforma de testes está representada na figura 4.32.

Os gráficos das figuras 4.33 e 4.34 e as tabelas B.4 e B.5 (apêndice B) apresentam dos dados das simulações de LU na máquina A com essa nova disposição do mecanismo de sincronização na plataforma de testes. A apresentação dos dados segue o padrão da figura anterior, sendo DT a versão do SystemC paralelo com o método *transport* da interface

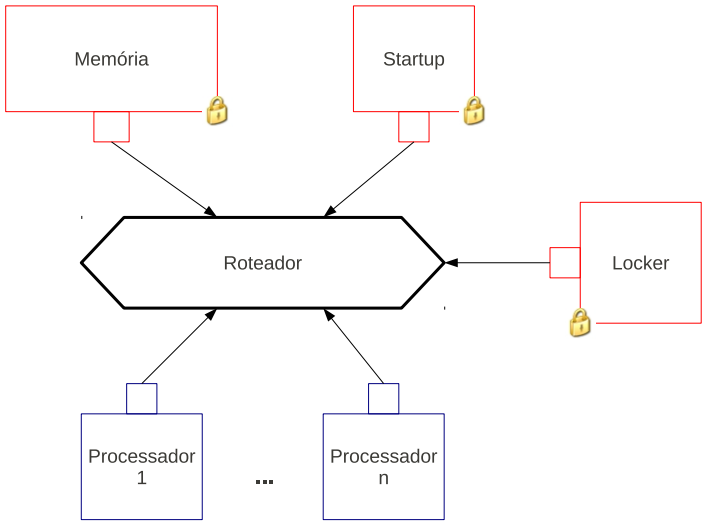
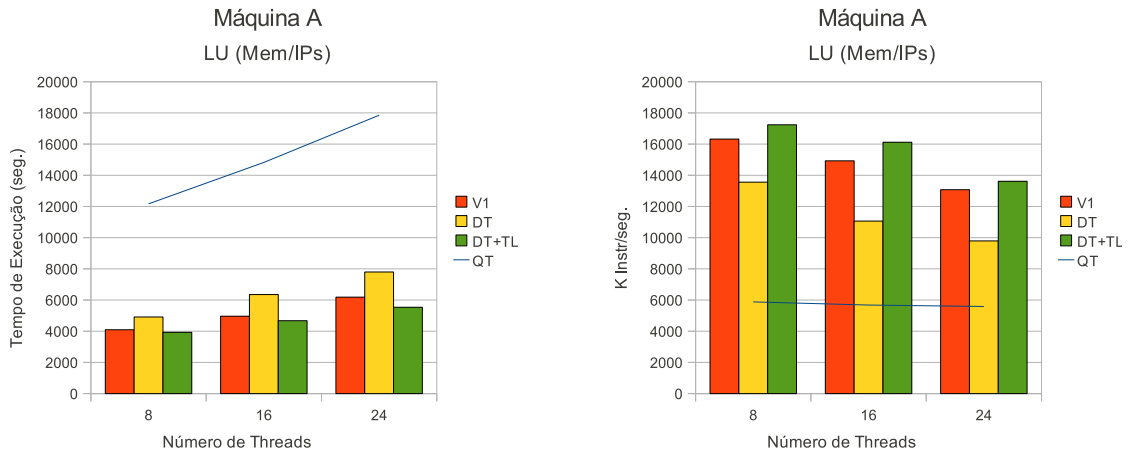


Figura 4.32: Plataforma de testes com sincronização individual para cada IP e memória.

TLM sem alteração no método de *lock* e DT+TL a versão do SystemC paralelo DT utilizando o mecanismo de *trylock*. Os dados de V1 diz respeito aos resultados obtidos com a versão DT+TL na otimização anterior, facilitando a visualização dos resultados.



(a) Tempos de Execução

(b) Instruções/Seg.

Figura 4.33: Dados das simulações do programa LU com *lock* nos dispositivos (Máquina A).

A simulação com a versão DT+TL foi a que atingiu, até o momento, os melhores resultados de desempenho na máquina A (figura 4.34). Com essa mudança de localização dos mecanismos de sincronização da plataforma, em média 3,164× em relação à versão QT do SystemC. Sendo que o melhor desempenho de 3,225× foi registrado na configuração com 24 *threads*. Também verifica-se a melhora de desempenho da versão DT, com destaque para a configuração com 8 *threads* que registrou o desempenho de 2,477× contra o desempenho de 2,338× da mesma configuração com toda sincronização no roteador.

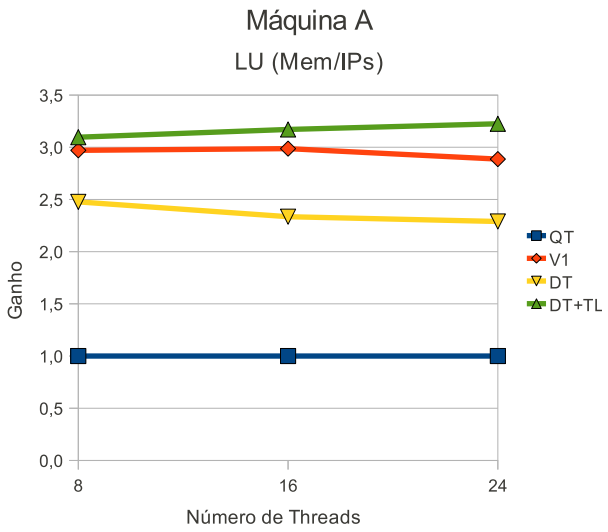
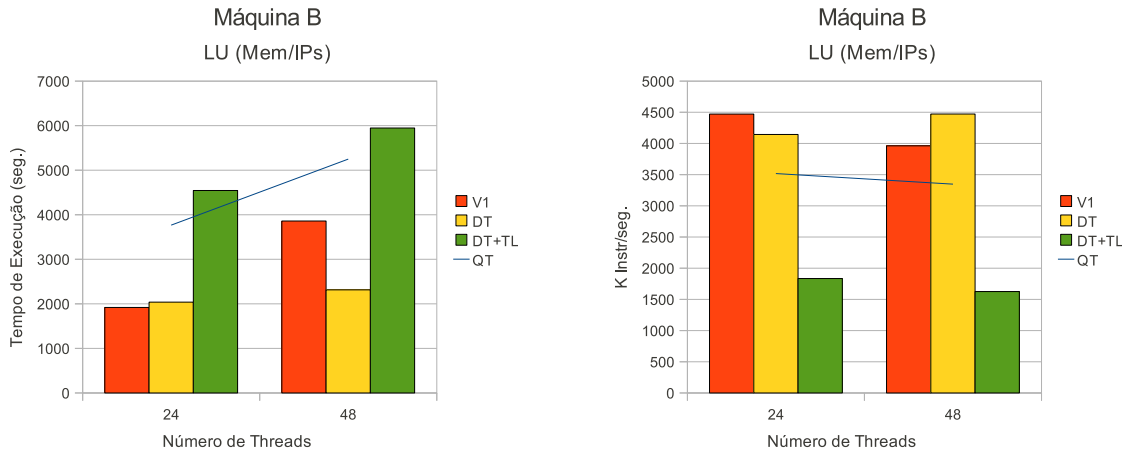


Figura 4.34: Desempenho do programa LU com *lock* nos dispositivos (Máquina A).

Os gráficos das figuras 4.35 e 4.36 e a tabela B.6 apresentam os resultados das simulações das versões DT e DT+TL com a nova configuração da plataforma de testes na máquina B. Os dados registrados pela versão DT, que teve o melhor desempenho na otimização anterior são apresentados como V1 nas legendas das figuras e tabela. Fica evidente a impossibilidade da versão DT+TL obter ganhos de desempenho na máquina B, mesmo que a contenção tenha sido removida do roteador. Porém, executar com a versão DT aumentou os ganhos de desempenhos, em relação aos obtidos na seção anterior. Por exemplo, na configuração com 48 *threads*, a simulação com DT executou com tempo de 2.314,631 segundos e ganho de desempenho de 2,268× em relação à versão QT. Nos gráficos 4.16a e 4.17, com mesma configuração, o tempo de simulação foi de 3859,648 segundos com ganho de desempenho de 1,360×, respectivamente.

Para esta plataforma de testes em específico, ainda é possível fazer mais uma alteração na disposição dos pontos de sincronização. Nessa nova alteração, representada na figura



(a) Tempos de Execução

(b) Instruções/Seg.

Figura 4.35: Dados das simulações do programa LU com *lock* nos dispositivos (Máquina B).

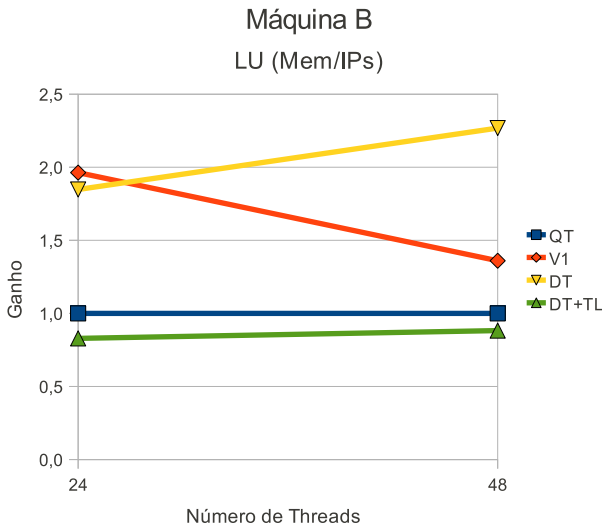


Figura 4.36: Desempenho do programa LU com *lock* nos dispositivos (Máquina B).

4.37, o módulo de memória não mais exclusivamente acessado por uma *thread*. Isso é possível porque os modelos de processadores escritos em ArchC não têm instruções que exijam acesso atômico à memória implementado. Os demais módulos IP's permanecem utilizando o canal de acesso exclusivo.

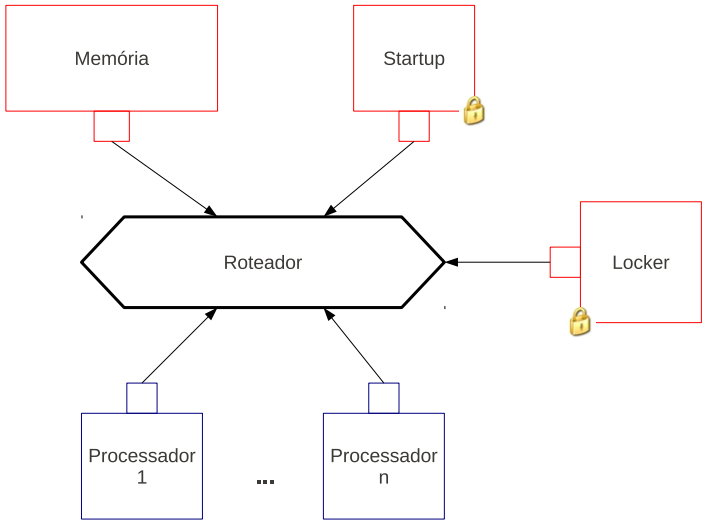
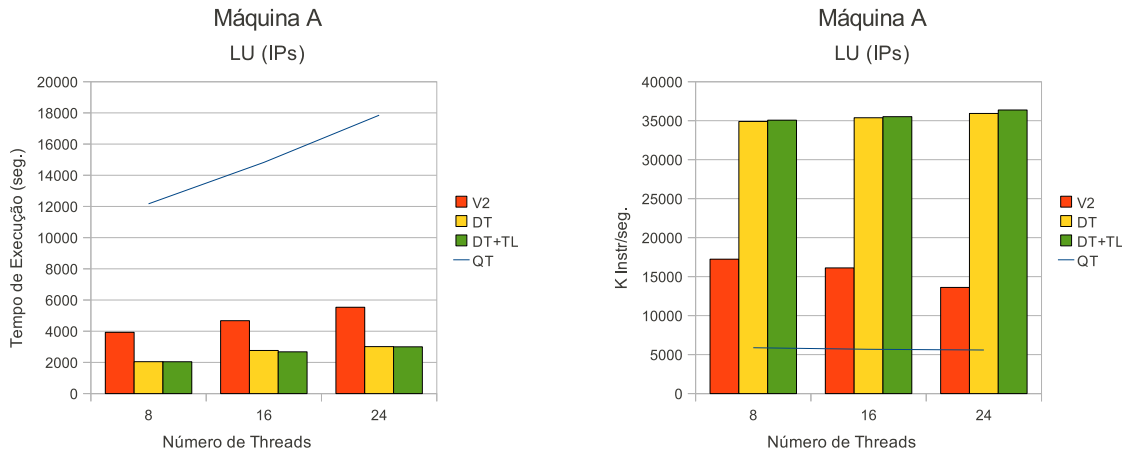


Figura 4.37: Plataforma de testes com sincronização somente nos IP's.

Os gráficos das figuras 4.38 e 4.39, juntamente com as tabelas B.7 e B.8 apresentam os dados das simulações de LU nesse novo esquema de comunicação sem o mecanismo de sincronização na interface TLM do módulo de memória na máquina A. A versão V2 do SystemC apresenta os melhores resultados registrados pela otimização anterior com DT+TL. Destas tabelas é possível verificar que, pelas características do programa LU, os processadores da plataforma realizam vários acessos à memória e a retirada do mecanismo de sincronização do canal de acesso a ela permitiu que o desempenho das simulações aumente consideravelmente. Apesar do desempenho de DT+TL ter sido melhor, tanto DT+TL e DT mantiveram ganhos de desempenho acima de 5,0×. Sendo o pior desempenho de 5,367× de DT na configuração com 16 *threads* e o melhor desempenho de 5,959× de DT+TL na configuração com 8 *threads*, como mostrado no gráfico da figura 4.39.

Os gráficos das figuras 4.40 e 4.41 e a tabela B.9, apresentam os resultados das simulações realizadas com a mesma plataforma sem sincronização no canal de comunicação com a memória na máquina B. A versão V2 do SystemC apresenta os melhores resultados registrados pela otimização anterior com a versão DT. Analisando o gráfico da figura 4.41 é possível verificar um fato interessante quanto à alternância de desempenho entre



(a) Tempos de Execução

(b) Instruções/Seg.

Figura 4.38: Dados das simulações do programa LU com *lock* nos IP's (Máquina A).

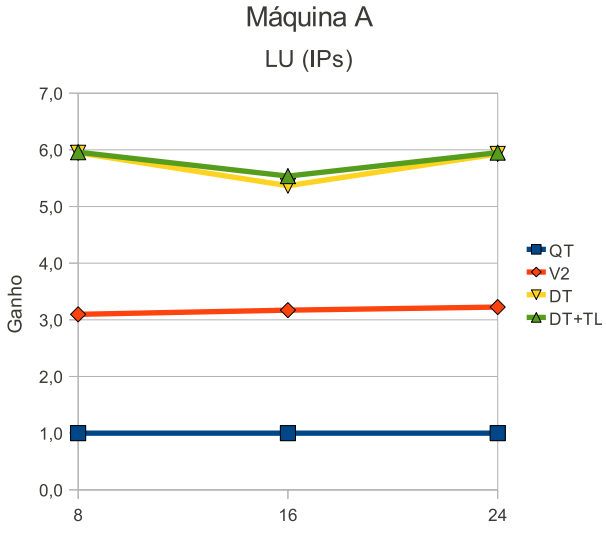
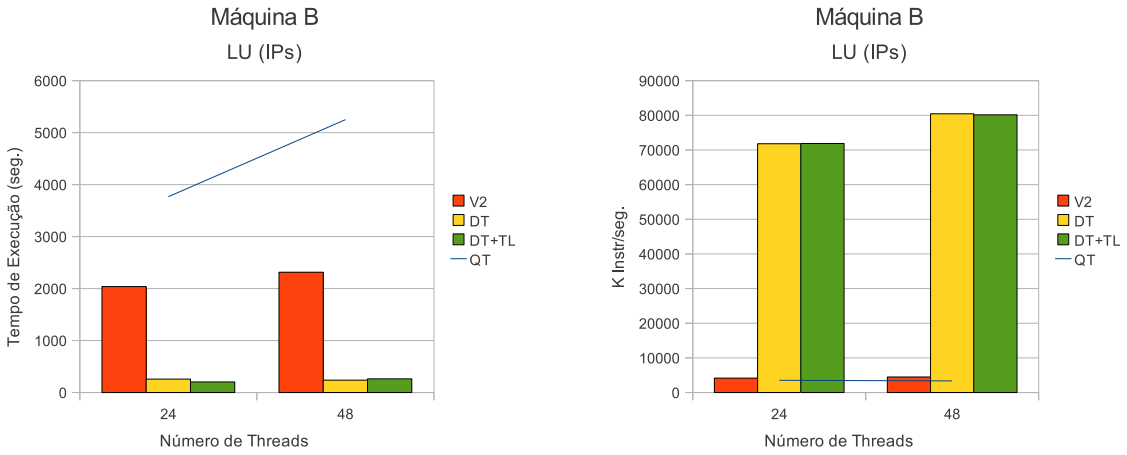


Figura 4.39: Desempenho do programa LU com *lock* nos IP's (Máquina A).

as versões DT e DT+TL. Apesar da versão DT+TL normalmente não ter um bom desempenho nessa máquina, na configuração com 24 *threads* seu desempenho foi melhor do que a versão DT, sendo de 18,556× e 14,590×, respectivamente. Desempenho, em parte, devido ao pouco acesso feito aos módulos IP's, que pouco influenciou no desempenho da simulação. Na configuração com 48 *threads* a versão DT obteve o valor máximo de desempenho em relação à QT de 22,029×, sendo que a versão DT registrou 19,961×.



(a) Tempos de Execução

(b) Instruções/Seg.

Figura 4.40: Dados das simulações do programa LU com *lock* nos IP's (Máquina B).

Dado que existem 12 *threads* executando em paralelo na máquina B, uma por núcleo de processamento da máquina, e somente uma delas adquire o *lock* do mutex, então ao utilizar o mecanismo de *while (pthread_mutex_trylock()) sched_yield()* 11 destas *threads* sofrerão as penalidades do reescalonamento e trocas de contexto do escalonador do Linux, tendo forte influência no desempenho da simulação. O que não acontece em máquinas com menos núcleos de processamento, como a máquina A, em que essa implementação é benéfica, quando o número de *threads* é superior ao número de núcleos do processador, possibilitando que mais *threads* possam executar. Tal comportamento é comprovado pelos resultados dos desempenhos das simulações desta seção.

Nesta seção, descreveu como pequenas alterações no modelo de sincronização das *threads* e melhorias na modelagem da plataforma de testes é possível alcançar melhores resultados de desempenho. Os picos de desempenho foram registrados quando optou-se por retirar o dispositivo de sincronização existente no canal de comunicação com a memória da plataforma de testes, com ganhos maiores que o número de processadores das máquinas, chamado de *superlinear speedup* [15]. Principalmente na máquina B, o

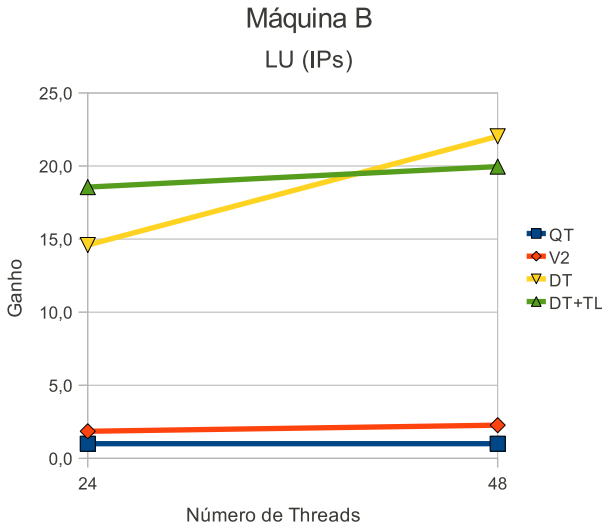


Figura 4.41: Desempenho do programa LU com *lock* nos IP's (Máquina B).

ganho do simulador ficou além do número de processadores, possivelmente, devido à hierarquia de memória da máquina. Na seção 4.3 foram apresentadas as descrições das máquinas, sendo que, na máquina B, as memórias cache L1 e L2 são individuais para cada núcleo de processamento. Portanto, se todos dados a serem processados por cada núcleo couberem nas suas respectivas caches, o tempo de acesso à memória é reduzido e, se poucos dados forem compartilhados entre os núcleos de processamento, o tempo de processamento tende a ser ainda menor.

O próximo capítulo, apresenta as conclusões deste projeto de pesquisa e os trabalhos que ainda podem ser realizados para se obter um melhor desempenho do SystemC *Multi-Threaded*, assim como das ferramentas diretamente envolvidas no processo de simulação.

Capítulo 5

Conclusões e Trabalhos Futuros

Esse trabalho foi responsável por diversas modificações no núcleo de simulação da linguagem SystemC, tornando-o devidamente adequado para executar em arquiteturas paralelas de hardware com memória compartilhada, introduzindo um novo tipo de processo, chamado *DThread* (*detached thread*).

O SystemC *Multi-Threaded* até agora tem se mostrado uma alternativa viável aos trabalhos previamente desenvolvidos, apresentados na seção 2.6, por não ter a necessidade de replicar o núcleo de simulação, criar um ambiente próprio de execução, ou até mesmo reescrever o código de toda a linguagem.

Desenvolver uma versão realmente paralela do SystemC não foi uma tarefa tão simples. A maior fonte de complicações e conflitos foi a intenção de se manter o máximo de compatibilidade do SystemC *Multi-Threaded* com a versão 2.2, fazendo com que o mínimo de alterações sejam necessárias para utilizar o novo modelo paralelo do núcleo de simulação. Detalhes como, introduzir as *dthreads* na linguagem, manter as chamadas de *wait* originais e deixar com que o núcleo de simulação defina como esse *wait* irá atuar na simulação e, permitir com que todos os tipos de processos possam coexistir na mesma plataforma de simulação, tornaram-se atividades complexas, principalmente pela necessidade de ainda garantir o desempenho das *dthreads*.

Para a validação do trabalho, o conjunto de testes foi escolhido pela quantidade de computação envolvida e pelo bom balanceamento da carga entre as *threads* do sistemas. Permitindo assim, uma melhor avaliação do desempenho do novo núcleo de simulação e a captura de pontos que precisam ser melhor avaliados e revistos para a continuação do trabalho. As versões DT+QT e DT do SystemC *Multi-Threaded* mostraram-se eficazes durante toda a bateria de testes com desempenhos muito próximos entre si e, portanto, ainda devem ser mantidas sob estudos e melhorias.

O melhor ganho de desempenho durante a primeira fase de testes foi de 2,831× com a versão DT executando o programa Water-NSquared com 4 *threads* na máquina A, porém,

após algumas mudanças no modelo de sincronização da interface TLM e redistribuição dos pontos de sincronização na plataforma de testes, foi possível obter ganhos superiores ao número de núcleos de processamento das máquinas utilizadas para testes. Sendo que na máquina A o melhor ganho de desempenho foi de 5,959× em relação à QT e na máquina B de 22,029× em relação à QT.

Após essas mudanças esse trabalho mostrou-se uma boa solução para obtenção de paralelismo para modelos escritos em SystemC, com ganhos de desempenho iguais ou superiores aos apresentados nos trabalhos da tabela 2.2. Como por exemplo, Ezudheen *et al.* [24], que registrou ganhos entre 5,3X e 8,0X em máquinas de 7 e 15 núcleos de processamento, respectivamente. Além do desempenho de modelos escritos em SystemC, esse trabalho, assim como Mello *et al.* [19], apresenta as primeiras etapas para a elaboração de um novo núcleo de simulação para o SystemC não baseado em eventos, executando uma plataforma mais complexa de simulação com a presença de processadores, memória, barramento e diversos outros dispositivos.

Essa etapa de obtenção de paralelismo está concluída, porém, alguns pontos para trabalhos futuros, que podem resultar em melhorias de desempenho são enumerados:

1. Eliminar a utilização de *mutexes* para a sincronização entre *threads*, utilizando um mecanismo de passagem de mensagens mais eficiente;

O controle de concorrência das *threads* no roteador é imprescindível para manter a execução correta do problema. Principalmente porque, a plataforma de testes utilizada compartilha vários recursos que são únicos e não comportam concorrência. Como visto na seção 4.4, o Posix *mutex* pode não ser a melhor opção, dependendo da máquina que esteja executando a simulação. Portanto, um mecanismo de sincronização mais eficiente e independente de plataforma será de grande contribuição no desempenho das simulações.

Apêndice A

Tabelas de Dados de Simulação

As tabelas para o programa FFT apresentam os tempos de execução em segundos e a quantidade de instruções executadas por segundo (K Instr/s). As tabelas são apresentas da seguinte forma: a primeira coluna é a compilação do SystemC utilizada, na segunda coluna é apresentado o tamanho da matriz calculada, as colunas seguintes representam o tempo real de execução (em segundos) e o número de instruções executadas por segundo (K Instr/seg) para as quantidades de *threads* utilizadas.

SystemC	**Carga**	**4**		**8**		**16**	
		Tempo	**K Instr/s**	**Tempo**	**K Instr/s**	**Tempo**	**K Instr/s**
QT	10	36,225	6043,163	62,641	6009,113	118,201	5849,949
PT	10	1675,483	130,671	2494,212	150,927	4185,097	165,228
DT+PT	10	47,579	9384,752	61,479	6618,482	72,287	5672,706
DT+QT	10	24,436	9669,062	39,380	7030,150	60,740	6055,575
DT	10	15,754	11441,758	33,278	7405,867	60,708	6010,184

Tabela A.1: Resultado das simulações do programa FFT (Máquina A).

SystemC	**Carga**	**4**		**8**		**16**	
		Tempo	**K Instr/s**	**Tempo**	**K Instr/s**	**Tempo**	**K Instr/s**
QT	10	51,067	4286,793	92,393	4074,014	183,397	3770,357
PT	10	2036,690	107,500	2993,656	125,746	5061,066	136,631
DT+PT	10	257,539	2853,833	164,132	2434,074	131,427	3646,937
DT+QT	10	53,021	3563,825	75,147	3236,513	124,337	3438,785
DT	10	74,971	2260,332	64,112	3689,093	124,891	3395,715

Tabela A.2: Resultado das simulações do programa FFT (Máquina B).

A tabela A.3 está disposta da seguinte maneira: a primeira coluna indica em qual máquina o desempenho está sendo calculado; a segunda coluna representa qual a versão

original do SystemC está sendo utilizada como a referência de cálculo de desempenho; a terceira coluna, é nova versão do SystemC a qual o desempenho está sendo medido; e por fim, as demais colunas indicam sob qual número de *threads* o desempenho está sendo calculado 4, 8 ou 16 *threads*.

Máquina	**Base**	**Versão**	**4**	**8**	**16**
A	QT	DT+PT	0,761	1,019	1,635
		DT+QT	1,482	1,591	1,946
		DT	2,299	1,882	1,947
	PT	DT+PT	35,215	40,570	57,896
		DT+QT	68,566	63,337	68,902
		DT	106,353	74,951	68,938
B	QT	DT+PT	0,198	0,563	1,395
		DT+QT	0,963	1,229	1,475
		DT	0,681	1,441	1,468
	PT	DT+PT	7,908	18,239	38,509
		DT+QT	38,413	39,837	40,704
		DT	27,166	46,694	40,524

Tabela A.3: Desempenho das simulações do programa FFT nas máquina A e B.

As próximas tabelas com os dados gerados pelas simulações de LU, OCEAN, Water-NSquared e Water-Spatial, além das colunas com a versão compilada do SystemC (SystemC), tamanho do problema de entrada do programa (Carga), tempo real de execução em segundos (Tempo) e instruções executadas por segundo (K Instr/s), agora, o desempenho (*Speedup*) de cada configuração de simulação, é inserido nas tabelas, uma vez que somente a versão original do SystemC QT é a referência de cálculo de desempenho. As tabelas com os dados obtidos pela simulação para uma máquina estão dividas em partes 1 e 2, devido à quantidade de dados apresentados, em que cada parte representa quantidades diferentes de *threads* utilizadas na execução do programa. Por exemplo, as simulações de LU na máquina A estão dividas entre as tabelas A.4 e A.5, sendo a primeira referente ao uso de 4 e 8 *threads* de execução e a segunda 16 e 24 *threads* de execução.

SystemC	**Carga**	**4**			**8**		
		Tempo	**K Instr/s**	**Ganho**	**Tempo**	**K Instr/s**	**Ganho**
QT	640	11044,038	5985,386	1,000	12168,615	5878,528	1,000
DT+PT	640	37928,925	9810,936	0,291	29846,903	7241,271	0,408
DT+QT	640	5653,985	14330,789	1,953	7020,246	10522,998	1,733
DT	640	4318,679	15622,210	2,557	5204,117	12742,044	2,338

Tabela A.4: Resultado das simulações do programa LU (Máquina A).

SystemC	Carga	16			24		
		Tempo	K Instr/s	Ganho	Tempo	K Instr/s	Ganho
QT	640	14824,615	5677,385	1,000	17851,161	5586,195	1,000
DT+PT	640	16476,407	7302,467	0,900	29238,990	6052,377	0,611
DT+QT	640	6978,588	10123,673	2,124	8139,890	9435,922	2,193
DT	640	6943,696	9973,194	2,135	8427,800	9068,176	2,118

Tabela A.5: Resultado das simulações do programa LU (Máquina A).

SystemC	Carga	6			12		
		Tempo	K Instr/s	Ganho	Tempo	K Instr/s	Ganho
QT	256	1311,671	4102,309	1,000	1910,953	3796,442	1,000
DT+PT	256	10858,857	2082,610	0,121	3654,295	3789,982	0,523
DT+QT	256	1831,771	2616,124	0,716	1163,920	4896,500	1,642
DT	256	1793,036	2598,683	0,732	1154,217	4733,520	1,656

Tabela A.6: Resultado das simulações do programa LU (Máquina B).

SystemC	Carga	24			48		
		Tempo	K Instr/s	Ganho	Tempo	K Instr/s	Ganho
QT	256	3766,935	3517,778	1,000	5249,129	3347,718	1,000
DT+PT	256	3693,640	3678,816	1,020	4693,274	3839,316	1,118
DT+QT	256	1992,451	4266,851	1,891	3884,663	3928,202	1,351
DT	256	1918,752	4471,465	1,963	3859,648	3961,871	1,360

Tabela A.7: Resultado das simulações do programa LU (Máquina B).

SystemC	Carga	4			8		
		Tempo	K Instr/s	Ganho	Tempo	K Instr/s	Ganho
QT	130	2694,548	6193,674	1,000	2839,907	5990,322	1,000
DT+PT	130	10302,396	10201,899	0,262	8055,183	6860,148	0,353
DT+QT	130	1490,921	14050,513	1,807	1593,884	11797,593	1,782
DT	130	984,628	17716,372	2,737	1207,148	14076,837	2,353

Tabela A.8: Resultado das simulações do programa OCEAN (Máquina A).

SystemC	Carga	16			32		
		Tempo	K Instr/s	Ganho	Tempo	K Instr/s	Ganho
QT	130	3043,335	5644,715	1,000	3263,631	5342,616	1,000
DT+PT	130	4862,495	7289,560	0,626	6585,730	6183,777	0,496
DT+QT	130	1455,221	11722,894	2,091	1611,089	10861,188	2,026
DT	130	1415,656	11896,238	2,150	1568,270	11118,361	2,081

Tabela A.9: Resultado das simulações do programa OCEAN (Máquina A).

SystemC	**Carga**	**4**			**8**		
		Tempo	**K Instr/s**	**Ganho**	**Tempo**	**K Instr/s**	**Ganho**
QT	130	3641,242	4583,316	1,000	4055,061	4195,361	1,000
DT+PT	130	54761,788	3295,039	0,066	27142,800	1729,437	0,149
DT+QT	130	3466,236	5679,671	1,050	3086,490	5685,396	1,314
DT	130	3353,196	5595,428	1,086	3163,692	5416,750	1,282

Tabela A.10: Resultado das simulações do programa OCEAN (Máquina B).

SystemC	**16**			**32**		**64**	
	Tempo	**K Instr/s**	**Ganho**	**Tempo**	**K Instr/s**	**Tempo**	**K Instr/s**
QT	4549,383	3776,082	1,000	-	-	-	-
DT+PT	9170,572	4115,209	0,496	7431,326	4592,912	9448,531	3509,767
DT+QT	2880,820	6361,442	1,579	2782,435	6510,733	3452,405	5421,885
DT	2975,548	6039,651	1,529	2795,487	6260,907	3075,440	5948,941

Tabela A.11: Resultado das simulações do programa OCEAN (Máquina B).

SystemC	**Carga**	**4**			**8**		
		Tempo	**K Instr/s**	**Ganho**	**Tempo**	**K Instr/s**	**Ganho**
QT	343	3599,532	6114,928	1,000	4234,673	5939,240	1,000
DT+PT	343	12222,130	9914,336	0,295	10342,387	6368,121	0,409
DT+QT	343	1865,073	14133,935	1,930	1917,043	12083,503	2,209
DT	343	1271,323	17028,099	2,831	1871,485	12333,650	2,263

Tabela A.12: Resultado das simulações do programa Water-NSquared (Máquina A).

SystemC	**Carga**	**16**			**24**		
		Tempo	**K Instr/s**	**Ganho**	**Tempo**	**K Instr/s**	**Ganho**
QT	343	5038,194	5641,247	1,000	5856,520	5516,367	1,000
DT+PT	343	5126,226	7543,362	0,983	8774,256	6452,440	0,667
DT+QT	343	2386,800	10151,388	2,111	2742,181	9519,167	2,136
DT	343	2522,713	9674,278	1,997	2861,733	9121,509	2,046

Tabela A.13: Resultado das simulações do programa Water-NSquared (Máquina A).

SystemC	**Carga**	**6**			**12**		
		Tempo	**K Instr/s**	**Ganho**	**Tempo**	**K Instr/s**	**Ganho**
QT	343	5379,091	4093,904	1,000	7068,987	3757,784	1,000
DT+PT	343	55127,334	1317,170	0,098	14577,770	4018,036	0,485
DT+QT	343	7516,271	2897,180	0,716	4303,722	5535,603	1,643
DT	343	7247,088	2947,851	0,742	4649,875	5132,257	1,520

Tabela A.14: Resultado das simulações do programa Water-NSquared (Máquina B).

SystemC	Carga	24			48	
		Tempo	K Instr/s	Ganho	Tempo	K Instr/s
QT	343	9741,442	3316,420	1,000		
DT+PT	343	12128,622	4184,092	0,803	12180,225	4265,918
DT+QT	343	5104,453	5251,797	1,908	7628,966	4352,612
DT	343	4937,473	5497,771	1,973	7272,595	4597,114

Tabela A.15: Resultado das simulações do programa Water-NSquared (Máquina B).

SystemC	Carga	4			8		
		Tempo	K Instr/s	Ganho	Tempo	K Instr/s	Ganho
QT	216	5331,022	6478,219	1,000	7292,892	6342,909	1,000
DT+PT	216	33699,884	8385,710	0,158	16021,766	6202,232	0,455
DT+QT	216	2494,642	12525,898	2,137	4128,856	8532,938	1,766
DT	216	2259,263	13274,499	2,360	3862,343	8431,901	1,888

Tabela A.16: Resultado das simulações do programa Water-Spatial (Máquina A).

SystemC	Carga	16			24		
		Tempo	K Instr/s	Ganho	Tempo	K Instr/s	Ganho
QT	216	7812,003	5977,468	1,000	6137,262	5796,794	1,000
DT+PT	216	5482,832	7274,327	1,425	9416,763	6357,236	0,652
DT+QT	216	3841,211	8135,737	2,034	2887,860	9136,473	2,125
DT	216	3908,773	7955,118	1,999	3016,856	8749,647	2,034

Tabela A.17: Resultado das simulações do programa Water-Spatial (Máquina A).

Apêndice B

Tabelas de Dados da Plataforma Modificada para Favorecer a Simulação *Multi-Threaded*

As tabelas apresentadas são referentes às simulações efetuadas na seção 4.4.

SystemC	**Carga**	**8**			**16**		
		Tempo	**K Instr/s**	**Ganho**	**Tempo**	**K Instr/s**	**Ganho**
QT	640	12168,615	5878,528	1,000	14824,615	5677,385	1,000
DT	640	5204,117	12742,044	2,338	6943,696	9973,194	2,135
DT+TL	640	4096,524	16321,408	2,970	4962,999	14929,473	2,987

Tabela B.1: Resultado das simulações do programa LU com *TryLock* (Máquina A).

SystemC	**Carga**	**24**		
		Tempo	**K Instr/s**	**Ganho**
QT	640	17851,161	5586,195	1,000
DT	640	8427,800	9068,176	2,118
DT+TL	640	6184,365	13076,370	2,886

Tabela B.2: Resultado das simulações do programa LU com *TryLock* (Máquina A).

SystemC	Carga	24			48		
		Tempo	K Instr/s	Ganho	Tempo	K Instr/s	Ganho
QT	256	3766,935	3517,778	1,000	5249,129	3347,718	1,000
DT	256	1918,752	4471,465	1,963	3859,648	3961,871	1,360
DT+TL	256	5663,436	1517,851	0,665	7069,031	1510,056	0,743

Tabela B.3: Resultado das simulações do programa LU com *TryLock* (Máquina B).

SystemC	Carga	8			16		
		Tempo	K Instr/s	Ganho	Tempo	K Instr/s	Ganho
QT	640	12168,615	5878,528	1,000	14824,615	5677,385	1,000
V1	640	4096,524	16321,408	2,970	4962,999	14929,473	2,987
DT	640	4913,069	13559,301	2,477	6349,269	11063,556	2,335
DT+TL	640	3929,804	17240,723	3,096	4675,776	16120,215	3,171

Tabela B.4: Resultado das simulações do programa LU com *lock* nos dispositivos (Máquina A).

SystemC	Carga	24		
		Tempo	K Instr/s	Ganho
QT	640	17851,161	5586,195	1,000
V1	640	6184,365	13076,370	2,886
DT	640	7798,148	9792,359	2,289
DT+TL	640	5535,135	13617,466	3,225

Tabela B.5: Resultado das simulações do programa LU com *lock* nos dispositivos (Máquina A).

SystemC	Carga	24			48		
		Tempo	K Instr/s	Ganho	Tempo	K Instr/s	Ganho
QT	256	3766,935	3517,778	1,000	5249,129	3347,718	1,000
V1	256	1918,752	4471,465	1,963	3859,648	3961,871	1,360
DT	256	2038,519	4144,519	1,848	2314,631	4472,279	2,268
DT+TL	256	4544,813	1834,191	0,829	5947,473	1626,047	0,883

Tabela B.6: Resultado das simulações do programa LU com *lock* nos dispositivos (Máquina B).

SystemC	Carga	8			16		
		Tempo	K Instr/s	Ganho	Tempo	K Instr/s	Ganho
QT	640	12168,615	5878,528	1,000	14824,615	5677,385	1,000
V2	640	3929,804	17240,723	3,096	4675,776	16120,215	3,171
DT	640	2045,954	34902,396	5,948	2762,357	35376,909	5,367
DT+TL	640	2042,099	35068,331	5,959	2677,748	35521,730	5,536

Tabela B.7: Resultado das simulações do programa LU com *lock* nos IP's (Máquina A).

SystemC	Carga	24		
		Tempo	K Instr/s	Ganho
QT	640	17851,161	5586,195	1,000
V2	640	5535,135	13617,466	3,225
DT	640	3011,795	35928,519	5,927
DT+TL	640	2999,467	36372,118	5,951

Tabela B.8: Resultado das simulações do programa LU com *lock* nos IP's (Máquina A).

SystemC	Carga	24			48		
		Tempo	K Instr/s	Ganho	Tempo	K Instr/s	Ganho
QT	256	3766,935	3517,778	1,000	5249,129	3347,718	1,000
V2	256	2038,519	4144,519	1,848	2314,631	4472,279	2,268
DT	256	258,186	71799,581	14,590	238,288	80438,500	22,029
DT+TL	256	203,000	71863,744	18,556	262,965	80146,780	19,961

Tabela B.9: Resultado das simulações do programa LU com *lock* nos IP's (Máquina B).

Referências Bibliográficas

[1] B.C. Albertini. Um framework de desenvolvimento de plataformas e um mecanismo de depuração baseado em reflexão computacional. Master's thesis, Universidade Estadual de Campinas, Campinas, SP, 2007.

[2] G. Araujo, S. Rigo, and R. Azevedo. *Processor Design with ArchC*, chapter 11, pages 275 – 294. Morgan Kaufmann, 2008.

[3] B. Bailey, G. Martin, and A. Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann Publishers, 1st edition, 2007.

[4] D.C. Black and J. Donovan. *SystemC From the Ground Up*. Springer ed, 1st edition, 2004.

[5] L. Cai and D. Gajski. Transaction level modeling: an overview. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24, New York, NY, USA, 2003. ACM.

[6] B. Chopard, P. Combes, and J. Zory. A conservative approach to systemc parallelization. In *Computational Science ICCS 2006*, volume 3994/2006 of Lecture Notes in Computer Science, pages 653–660, Berlim, Heidelberg, 2006.

[7] P. Combes, E. Caron, F. Desprez, B. Chopard, and J. Zory. Relaxing synchronization in a parallel systemc kernel. In *ISPA '08: Proceedings of the 2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 180–187, Washington, DC, USA, 2008. IEEE Computer Society.

[8] D.A.S Committee. *IEEE Std 1666-2005 IEEE standard SystemC language reference manual*, 2006.

[9] M. Damm, C. Grimm, J. Haas, A. Herrholz, and W. Nebel. Connecting systemc-ams models with osci tlm 2.0 models using temporal decoupling. In *Forum on Specification, Verification and Design Languages, 2008. FDL 2008.*, pages 25 – 30, 2008.

[10] A. Donlin. Transaction level modeling: flows and use models. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 75–80, New York, NY, USA, 2004. ACM.

[11] A. Douillet and G. Gao. Software-pipelining on multi-core architectures. In *Proceedings of 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 39–48, 2007.

[12] F. Ghenassia. *Transactional-Level Modeling with SystemC*. Springer ed, 1st edition, 2005.

[13] T. Grütker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Springer ed, 1st edition, 2002.

[14] N. Hatami, A. Ghofrani, P. Prinetto, and Z. Navabi. Tlm 2.0 simple sockets synthesis to rtl. In *4th International Conference on Design and Technology of Integrated Systems. DTIS '09.*, pages 3 – 8, 2009.

[15] D. P. Helmbold and C. E. McDowell. Modeling speedud (n) greater than n. *IEEE Trans. Parallel Distrib. Syst.*, 1(2):250–256, 1990.

[16] K. Huang, L. Bacivarov, F. Hugelshofer, and L. Thiele. Scalably distributed systemc simulation for embedded applications. In *International Symposium on Industrial Embedded Systems (SIES 2008)*, pages 271–274, June 2008.

[17] Yong-Qin Huang, Hong-Liang Li, Xiang-Hui Xie, Lei Qian, Zi-Yu Hao, Feng Guo, and Kun Zhang. Archsim: a system-level parallel simulation platform for the architecture design of high performance computer. *J. Comput. Sci. Technol.*, 24(5):901–912, 2009.

[18] Linux Kernel Organization Inc. The linux kernel. WebSite, Maio 2010. http://www.kernel.org/doc/.

[19] A. Mello, I. Maia, A. Greiner, and F. Pecheux. Parallel simulation of systemc tlm 2.0 compliant mpsoc on smp workstations. In *DATE '10: Proceedings of the conference on Design, Automation and Test in Europe*, pages 606–609, Dresden, Germany, 2010.

[20] Y. N. Naguib and R. S. Guindi. Speeding up systemc simulation through process splitting. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 111–116, San Jose, CA, USA, 2007. EDA Consortium.

[21] H. Nakamura, N. Sato, and N. Tabuchi. An efficient and portable scheduler for rtos simulation and its certified integration to systemc. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 1157–1158, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[22] OSCI. Systemc library. WebSite, Abril 2010. http://www.systemc.org.

[23] OSCI. Tlm-2.0 standard. WebSite, Setembro 2010. http://www.systemc.orghttp://www.systemc.org/downloads/standards/tlm20.

[24] Ezudheen P, P. Chandran, J. Chandra, B. P. Simon, and D. Ravi. Parallelizing systemc kernel for fast hardware simulation on smp machines. In *PADS '09: Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, pages 80–87, Washington, DC, USA, 2009. IEEE Computer Society.

[25] D. Pérez, G. Mouchard, and O. Temam. A new optimized implementation of the systemc engine using acyclic sheduling. In *DATE 04: Proceeding of the Conference on Design, Automation and Test in Europe Conference and Exhibition*, pages 552–557, Washington DC, USA, 2004.

[26] A. Rose, S. Swan, J. Pierce, and J.M. Fernandez. Transaction level modeling in systemc. Technical report, OSCI, 2005.

[27] T.M. Sigrist. Reestruturação de archc para integração a metodologias de projeto baseadas em tlm. Master's thesis, Instituto de Computação - UNICAMP, 2007.

[28] The ArchC Team. *The ArchC Architecture Description Language v2.0 Reference Manual*. Av.Albert Einstein, 1251 13084-971 PO Box 6176 - Campinas/SP - Brazil, 2007.

[29] W. Winieck and P. Bilski. Multi-core programming approack in the real-time virtual instrumentation. In *FMTC 2008 - IEEE International Instrumentation and Measurement Technology Conference*, May 2008.

[30] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995. ACM.

[31] Hao Ziyu, Qian Lei, Li Hongliang, Xie Xianghui, and Zhang Kun. A parallel systemc environment: Archsc. In *ICPADS '09: Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems*, pages 617–623, Washington, DC, USA, 2009. IEEE Computer Society.