Roberto Silva Cantanhede

Suporte a simulação distribuída em SystemC

Brasília DF

Roberto Silva Cantanhede

Suporte a simulação distribuída em SystemC

Orientador:
Ricardo Pezzuol Jacobi

Universidade de Brasília
Departamento de Ciência da Computação
Mestrado em Informática - Programa 2004
Início do programa: Março/2004
PNM: Agosto/2005
Sub-área no PNM: Modelagem de dispositivos

Brasília DF

Dissertação de Mestrado sob o título *"Suporte a simulação distribuída em SystemC"*, defendida por Roberto Silva Cantanhede e aprovada em 23 de maio de 2007, em Brasília, Distrito Federal, pela banca examinadora constituída pelos doutores:

Prof. Dr. Ricardo Pezzuol Jacobi
Orientador

Prof. Dr. Diógenes C. da Silva Júnior
UFMG

Prof. Dr. Marcos Vinicius Lamar
Universidade de Brasília

Resumo

A contínua evolução tecnológica da microeletrônica viabiliza a integração de sistemas cada vez mais complexos em dispositivos semicondutores. Os sistemas integrados monolíticos (*SoC - Systems on Chip*) atuais permitem a integração de processadores, memórias e módulos dedicados analógicos, digitais e de radio-freqüência em uma única pastilha de silício. A simulação de tais sistemas é uma etapa fundamental no desenvolvimento de um SoC, pois permite a verificação de sua funcionalidade antes do detalhamento de sua implementação. A disponibilização de modelos simuláveis dos elementos de processamento de um SoC já nas primeiras etapas do projeto é igualmente fundamental para acelerar o processo de desenvolvimento do software embarcado, permitindo que o código produzido possa ser executado e testado de forma concorrente ao projeto do hardware.

A redução do tempo de simulação afeta diretamente o ciclo de projeto do SoC, visto que impacta tanto no desenvolvimento do hardware quanto no do software embarcado. Um dos fatores limitantes na aceleração da simulação é a utilização de sistemas monoprocessados. Tipicamente, uma descrição de um SoC é compilada e executa em um computador monoprocessado que simula por software o paralelismo do hardware. Uma forma de se atingir o objetivo de acelerar a simulação de sistemas em silício é a executação concorrente dos módulos do sistema. Assim, em vez dos módulos serem simulados em um único processador, eles podem ser distribuídos entre nodos de um *cluster* de computadores, sendo simulados com paralelismo real.

O objetivo deste trabalho é o estudo da introdução de processamento concorrente em sistemas integrados descritos em SystemC. Essa linguagem atingiu, ao longo dos últimos anos, o *status* de padrão para descrições em nível de sistema. Baseia-se em C++, introduzindo conceitos de orientação a objetos na descrição do hardware. Neste trabalho é proposta a paralelização da simulação de sistemas descritos em SystemC pela distribuição de módulos entre processos de um sistema multiprocessado. A comunicação entre módulos SystemC se realiza através de filas não bloqueantes, sendo a troca de mensagens entre processos implementada através do protocolo TCP/IP.

Como estudo de caso para simulação concorrente foi estudado e descrito em SystemC um algoritmo de segmentação de imagens, que serve como base para métodos para deteção de movimento em seqüências de imagens a ser implementado em um SoC para redes de sensores em desenvolvimento no contexto do projeto NAMITEC. Apresenta-se o algoritmo de segmentação e os resultados de sua simulação em SystemC.

Abstract

The ever increasing evolution of microelectronics allows the integration of more and more complex systems in semiconductor devices. Present day System on Chip (SoC) may integrate processors, memories, analog, mixed-signals, digital and RF modules in a single chip. The simulation of a SoC is a fundamental step in system design, since it permits the verification of its funcionality before dwelling on the details of the hardware design. The availability of simulation models for the processing elements early in the design process is also important for the embedded software development, which may then occur concurrently to hardware design.

The reduction of simulation time have a direct impact on the design cycle time, affecting both the hardware and the software development. The use of monoprocessor platforms for simulation is a limiting factor in the search of simulation speed up. Tipically, the SoC description is compiled and executed in a single process, where the hardware paralelism is simulated by software. One possible way to reduce simulation time is the parallel execution of the hardware models. Instead of being simulated in a single process, the modules may be distributed among nodes of a cluster which execute them in parallel.

The goal of this work is to study the introduction of concurrent processing in the simulation of SoC described in SystemC. This languagem attained the status of a standard for system level modeling last years. It is based on C++, introducing object oriented concepts in the hardware modeling. The poposal of this work is to paralelize the simulation by distributing SystemC modules among different processes in a multiprocessor system. The communication among those modules is performmed through non-blocking fifos and is implemented over the TCP/IP protocol.

A case study was developed for verification purposes. It consists in the implementation of a image segmentation algorithm to be used as support for image detection in video sequences, as part of the research project NAMITEC which targets the development of a SoC for sensor networks. The image segmentation algorithm and the simulation results in SystemC are presented.

Sumário

Lista de Tabelas

Lista de Figuras

1 Introdução

1.1 Objetivo

O desenvolvimento de sistemas digitais é uma tarefa bastante complexa. é necessário elaborar a especificação do sistema, levantar os algoritmos que serão utilizados, decidir que plataforma utilizar para a implementação, elaborar a bancada de testes, implementar e testar cada parte, integrar o sistema, testar a integração até chegar a produção da primeira unidade. Esse processo é dispendioso e necessita de ferramentas especializadas para que cada etapa seja concluída com sucesso.

Para que um sistema digital seja modelado eficientemente, é possível utilizar uma hieraquia de modelos com diferentes níveis de abstração, onde as otimizações realizadas em cada nível proporcionam ganhos ao processo todo. Nos estágios iniciais é preferível que não se perca tempo definindo minúcias do hardware final, mas concentrar esforços na sua funcionalidade. Nesse momento a simulação de um modelo abstrato em alto nível de abstração traz benefícios para todo o projeto pois permite ter estimativas da complexidade e custo finais do sistema, bem como esboçar um método de teste.

Para tentar otimizar os passos de teste e integração, neste trabalho será apresentado um módulo que implementa comunicação entre dois pontos (comumente processos do sistema operacional) via protocolo TCP e permite que recursos de software e hardware em nós físicos, como computadores ou kits de prototipação, distintos sejam utilizados na simulação como um único sistema digital. A viabilidade dessa implementação depende da ferramenta usada como base para a execução de testes e integração e do tipo de comunicação envolvida entre os nós que se comunicam.

A validação será realizada através de um estudo de caso que permita executar os simuladores em paralelo e realizar a injeção de dados de processos externos dentro do sistema simulado.

1.2 Projeto de sistemas digitais

A qualidade das ferramentas utilizadas no processo de desenvolvimento de um sistema e sua facilidade de uso são fatores que podem ser decisivos no tempo de desenvolvimento de um produto. Se a ferramenta for excelente mas tiver uma curva de aprendizado pouco suave pode tornar proibitivo o tempo gasto no seu aprendizado. Quanto mais rápido o desenvolvedor se torna familiar ao seu ambiente de trabalho, melhor seu desempenho em propor soluções dentro do domínio dos elementos da linguagem.

Nesse contexto, considerando que as linguagens de programação estruturadas (e até as orientadas a objeto) de alto nível como Java, C, C++ e Pascal tem estruturas muito semelhantes e têm uma base de conhecimentos ampla e bem disseminada, aproveitar para a descrição do hardware todos os recursos que essas linguagens oferecem pode proporcionar ganhos não só para os desenvolvedores mas para todo o processo de desenvolvimento. Entretanto, essas linguagens, como se apresentam, não são adequadas para descrever sistemas em hardware.

Sistemas em software geralmente são escritos usando linguagens de alto nível descrevendo algoritmos seqüenciais. O desenvolvimento de algoritmos na forma seqüencial está vinculado ao paradigma de computação inerente ao modelo von Neuman: armazenar o código do programa numa memória, que é lido por um processador que o executa *seqüencialmente* e possivelmente escreve o resultado desse código de volta na mesma memória. Como conseqüência das escritas em memória um dispositivo que faz E/S por memória mapeada ou que funciona bisbilhotando o barramento de memória (como as *snoop caches*) pode apresentar um resultado em um outro dispositivo de saída ou sinalizar que o programa deve aguardar que um dispositivo esteja pronto. O desempenho de um sistema em software depende essencialmente do somatório dos tempos de suas instruções, tempo de espera por dispositivos, número de iterações (repetições) e número de interações. Existem otimizações arquiteturais que podem promover a execução paralela de instruções, mas este não é um recurso do sistema em software, é algo alcançado com suporte sofisticado do hardware.

Sistemas em hardware, por outro lado, podem definir suas saídas diretamente a partir das entradas (lógica combinacional) ou a partir de alguma combinação de entradas sucessivas e estados internos (lógica seqüencial). A computação é realizada executando um software (isto é, lendo, decodificando e executando instruções armazenadas em uma memória) ou implementando diretamente o algoritmo em hardware através da composi-

ção de módulos combinacionais e seqüenciais. No hardware, os sinais são alimentados de forma concorrente e podem armazenar resultados em estruturas intermediárias, como registradores e buffers. Para determinar o tempo gasto entre o estímulo do sistema e sua resposta é necessário medir o atraso físico dos sinais. O atraso dos sinais é influenciado por diversos fatores físicos como capacitância, indutância, temperatura, diferença de potencial e materiais utilizados para a construção do circuito. As saídas do sistema são calculadas com base em operações lógicas distribuídas no espaço físico da implementação. Embora mais eficientes do que as implementações em software, são inflexíveis pois uma vez que seu comportamento está definido, e o hardware é fabricado, ele não muda. A excessão são os sistemas configuráveis onde a exploração espacial dos operadores pode ser reorganizada por conexões programáveis, e logo esses dispositivos, situados entre os extremos dos sistemas em software e os sistemas em hardware, possuem parâmetros como tamanho de grânulo e número de conexões programáveis, bem como outras variáveis que os distinguem e proporcionam um terceiro eixo para uma possível comparação, que não abordamos neste trabalho.

Dadas as diferenças entre os sistemas de software e de hardware, as linguagens usadas para descrever um e outro são diferentes. Enquanto um software é geralmente escrito para uma arquitetura com um conjunto de instruções, um sistema de hardware descreve uma distribuição espacial física de componentes. Para chegar nesse sistema físico implementado de forma controlada e previsível é necessário construir modelos (ou protótipos) desse sistema e refiná-los conforme amadurecem ou ainda especificá-los e implementá-los sem prototipação [14]. Adotamos a abordagem da prototipação.

Um modelo de um sistema de hardware pode ser escrito em diferentes níveis de abstração, alguns com extrema riqueza de detalhes como o RTL e outros com características mais gerais, como o TLM. Esses modelos serão abordados oportunamente. Os níveis mais altos são muito semelhantes ao software, entretanto, cabe diferenciá-los. Um software é uma transcrição de um conjunto de algoritmos (como algoritmos de ordenamento, escalonamento e decisão) e de modelos computacionais (redes de Petri, autômatos, dataflows, eventos discretos) para execução em uma plataforma qualquer. Um modelo de hardware pode prover desde a plataforma para a execução do software até um sistema completamente discreto com um ou mais modelos computacionais implementados. Para o hardware, pode ser mais fácil ou mais complexo, transcrever os algoritmos em componentes discretos ou em algoritmos. Tudo depende do nível de detalhamento do hardware que se queira expressar no modelo. Além do nível de detalhamento, deve-se levar em conta outras variáveis como ciclo de vida do produto final, seu custo de produção e de gerações

futuras.

Assumindo que os modelos computacionais e linguagens de alto nível são elementos conhecidos, uma linguagem para descrição de hardware tem, em síntese e para diferenciá-la de linguagens comuns à geração de software, características como [7]:

- descrever o dispositivo em diferentes níveis de abstração;
- dar suporte ao paralelismo do hardware;
- introduzir a noção de sinal elétrico;
- modelagem da temporização;
- incorporar software da plataforma desenvolvida em diversos níveis;
- prover um modelo executável;
- permitir exploração de arquiteturas;
- ter um simulador eficiente e rápido e separar comunicação do sistema de processamento.

Em muitas linguagens de programação, usadas amplamente para os processadores de uso geral, falta suporte, por exemplo, para estruturas físicas básicas como fios e elementos mais elaborados como construções paralelas, típicas de qualquer sistema implementado em chips e característico de todos os níveis de abstração de hardware. Isso dificulta a simulação especialmente nos níveis mais baixos. Os sistemas implementados em silício são paralelos sempre, no sentido de que uma vez alimentados os transistores que implementam as operações lógicas ele sempre estará funcionando. Desativar individualmente um conjunto de transistores e deixá-los desativados durante o funcionamento do sistema é contraproducente, uma vez que os transistores ociosos consomem recursos de energia e demandaram capital para sua produção. Quanto melhor o uso do sistema como um todo, mais eficiente é a solução em termos de uso da área de silício [5].

Para explorar o paralelismo característico do hardware, é desejável que as linguagens utilizadas para sua construção proporcionem meios eficientes para modelar esse paralelismo. Em Java, temos uma classe que implementa a concorrência (`Thread`). Para outras linguagens temos bibliotecas responsáveis por essa abstração e até padrões para a implementação de processos paralelos, como o POSIX [10]. Ainda que a linguagem possa

expressar procedimentos paralelos, ela ainda precisa prover outras abstrações necessárias para modelar com precisão componentes de hardware.

Todo componente eletrônico tem atrasos de comunicação decorrentes da velocidade e de efeitos causados pelo movimento dos elétrons. Já citamos o campo magnético, *cross-talking*, dissipação de calor, variação na resistência entre outros. Cada um desses efeitos afeta de uma maneira ou de outra o tempo gasto no acionamento dos transistores constituintes das portas lógicas e consequentemente gerando atrasos nos sinais lógicos. Modelar esses atrasos é cada vez mais crítico dados os crescentes tamanho e complexidade dos sistemas modernos.

Além de permitir que os atrasos sejam modelados, as estruturas simples como fios e barramentos, curto-circuito e direção do sinal precisam também estar disponíveis para o projetista. Existem várias adaptações de linguagens de programação, utilizadas comumente apenas para geração de código executável para microprocessadores de uso geral, que implementam mecanismos adequados à descrição e implementação de componentes digitais. Esses mecanismos viabilizam a expressão de construções paralelas e entre essas linguagens podemos citar HandelC, VHDL, Verilog, System-Verilog e SystemC. Em cada uma dessas linguagens temos simuladores disponíveis que traduzem as indicações de atraso, mudança de valores e direção dos sinais em informações úteis que permitirão ao projetista decidir se o funcionamento do sistema é correto ou não e noções de como construir e integrar o sistema final.

Entre as diversas linguagens que podem ser utilizadas para a descrição de sistemas digitais, serão comparadas SystemC e VHDL. Para tanto, é necessário apresentar algumas informações preliminares sobre o SystemC.

1.3 SystemC

SystemC é uma biblioteca construída sobre o C++. C++ em seu início era também implementado através de estruturas da linguagem C que suportavam abstrações típicas das linguagens orientadas a objeto. No presente C++ tornou-se uma linguagem própria, mas mantendo compatibilidade com o legado do C. Um compilador C++ pode compilar um código C, mas a recíproca não é verdadeira. Os compiladores C já puderam compilar código C++ em seus primeiros estágios, entretanto, com a evolução da linguagem, isso não é mais verdadeiro. Tanto C como C++ podem utilizar bibliotecas escritas em uma ou outra linguagem desde que sejam fornecidas interfaces adequadas à linguagem C, além

da sintaxe dos operadores, comandos e funções comuns. Dessa forma, o SystemC é uma biblioteca com interface para o C++ e que implementa construções sem as quais seria muito trabalhoso descrever qualquer peça de hardware. Pode-se até questionar porque não usar C++, ou outra linguagem qualquer, diretamente, mas essa discussão remete a um problema semelhante a reinvenção da roda. É possível usar apenas C++ para implementar hardware, mas o esforço de modelagem é maior e certamente se sobrepõe a recursos já implementados, testados e documentados disponíveis no SystemC.

A base do SystemC é um núcleo de simulação dirigido por eventos. Esse núcleo reage aos eventos e faz a troca das tarefas para execução. O simulador é incapaz de diferenciar as finalidades das tarefas tratando-as de forma genérica. Os demais elementos do SystemC são portas (de entrada e saída, por exemplo), módulos para representar estruturas (que descrevem desde portas lógicas a procedimentos de alto nível, na forma de `SC_METHOD`s e `SC_THREAD`s), interfaces (assinaturas que serão utilizadas por outros módulos ou canais) e canais (`sc_fifo`, `sc_mutex`, `sc_signal`, etc) que provêem a abstração de comunicação. O simulador, as portas, módulos, interfaces e canais formam o núcleo da linguagem. Junto ao núcleo ainda existem tipos de dados definidos na biblioteca do SystemC para bits, vetores de bits, inteiros de precisão arbitrária e outros tipos. Além de alguns tipos comuns às linguagens de descrição de hardware ainda é possível definir tipos personalizados usando a sintaxe adequada no C++.

Sobre o núcleo do SystemC estão descritos canais de uso comum como filas, sinais, temporizadores, *mutexes*, semáforos e outros. Com esses recursos é possível ampliar as funcionalidades do SystemC descrevendo, por exemplo, canais novos como barramentos variados e com diferentes opções de arbitragem.

Para estender o SystemC, usa-se módulos. Cada módulo é um bloco que esconde detalhes de implementação facilitando o processo de manutenção do sistema. Os módulos comunicam-se por portas bem definidas de entrada, saída ou de entrada e saída, e internamente possuem vários processos (ou métodos, no jargão das linguagens orientadas a objeto) que se comunicam por canais que podem ser primitivos do tipo requisição e atualização, como `sc_signal` e `sc_fifo` ou hierárquicos, que possuem estados internos e funcionamento complexo que pode usar árbitros, gerar múltiplos eventos e tratar erros, por exemplo. Além das portas, possuem tipicamente variáveis internas que funcionam como registradores do estado do sistema. Em um módulo é possível também usar outros módulos em hierarquia.

Ainda sobre módulos é necessário enfatizar que os processos são implementados como

métodos C++ e que no construtor se indica o tipo de processo, se método (`SC_METHOD`) ou linha de execução (`SC_THREAD`). A partir deste ponto, é necessário deixar claro que métodos se referem a métodos do SystemC, sobrepondo-se a terminologia das linguagens orientadas a objeto. Isto porque métodos do C++ sozinhos não apontam diferenças importantes na semântica de simulação, isto é, um método (`SC_METHOD` do SystemC) pode possuir uma lista de sensitividade, ou seja, um conjunto de sinais que dispara sua execução sempre que qualquer um deles é modificado, atribuindo o comportamento de processo a funções. Observe-se que se um `SC_METHOD` não termina, logo não executando completamente, o simulador não chega num estado estável e não prossegue. Já um processo tipo linha de execução (`SC_THREAD`, ou simplesmente, *thread*) está sempre em execução sincronizada com o simulador, no sentido de que é possível inserir atrasos explícitos (através da função `wait()`) que instruem o simulador a avançar, interrompendo a *thread* e permitindo a execução de outros métodos e *thread*s, viabilizando a descrição de componentes que demoram vários ciclos de relógio para executar. Modelar o comportamento multi-ciclo usando um método é mais complexo quando comparado a facilidade proporcionada por uma *thread*.

Através das `SC_THREAD`s é possível inserir atrasos medidos em tempo de relógio *wall clock time* no tempo do simulador diretamente no código. Esse comportamento não é sintetizável mas traz um grande poder de expressão à linguagem. Embora *thread*s apresentem essa facilidade e executem sem interrupção entre funções `wait()` do SystemC, não é uma construção sintetizável. As construções sintetizáveis são usadas por ferramentas automáticas de síntese responsáveis por gerar a descrição da implementação física do hardware correspondente ao modelo. Métodos, quando marcados para execução serão acionados durante a simulação via chamada de função (como *request* e *request_ update*). O uso de uma estratégia distribuída, para executar métodos, isto é, vários processos do simulador, agrega ao tempo de simulação toda a sobrecarga associada a trocas de contexto e tempo de comunicação.

Ressalta-se também que o desempenho do simulador depende não apenas da linguagem, mas do suporte oferecido pela microarquitetura que executa o simulador.

As *threads* do SystemC executam do início ao fim e morrem quando sua execução termina por definição, não podendo ser reexecutadas após isso. Assim, se as *threads* não forem corretamente codificadas é possível que monopolizem o simulador, que não é preemptivo, comprometendo a simulação. Uma forma de resolver esta questão é introduzindo chamadas a função `wait()` do SystemC que permite que a *thread* seja escalonada, seja

para um instante de tempo determinado no futuro ou até que um determinado evento ou conjunto de eventos alternativos ou concorrentes ocorra.

A simulação no SystemC é baseada em eventos de diferentes tipos de acordo com o tipo de canal utilizado. Canais como FIFOs podem ter eventos como leitura, escrita e fila cheia. Canais do tipo sinal têm outros eventos como borda de subida, borda de descida, sinal em zero, sinal em um ou em estado indeterminado. Alguns objetos provêem eventos periódicos como o relógio (`sc_clock`) que dita uma cadência de eventos dentro do sistema. Os eventos é que determinam se um dado método ou *thread* será marcado como pronto para ser executado ou não. Uma vez algo esteja pronto para execução o simulador poderá executá-lo ou continuar uma execução suspensa anteriormente (via `wait()`, por exemplo).

Quando os canais são especificados utiliza-se chamadas a função `notify()` para que a fila de execução do simulador possa ser preenchida. A execução a partir dos eventos pode ser [7] imediata, atraso delta (delta-delay) ou de atraso não zero (non zero delay). Notificações imediatas sempre vão determinar que os processos que dependem dela sejam acionados antes das atualizações dos canais primitivos. Notificações com atraso delta colocarão os processos em execução após a etapa de atualização dos canais primitivos, isto é, verão todas as modificações feitas nos canais primitivos. É importante visualizar que notificações imediatas são entregues antes que os valores novos dos canais primitivos sejam visíveis. Por fim, atrasos não zero são elencados em uma lista de eventos ordenada no tempo. Quando o simulador chega ao instante correto o evento é disparado.

Os eventos que colocam um método ou *thread* a disposição para execução podem ser estáticos ou dinâmicos. Os eventos estáticos estão definidos já antes de se iniciar a simulação. Os eventos dinâmicos suspendem a lista de definição dos eventos estáticos e forçam um processo a esperar por um evento específico.

Os canais básicos e os módulos, que interagem como colocado anteriormente via eventos e notificações, são utilizados por instanciação, isto é, são declarados e inicializados com nomes e características que os tornam únicos e reaproveitáveis em módulos mais complexos. O simulador, em tempo de execução, verifica as ligações entre as instâncias de cada canal e módulo declaradas e então procede com a simulação. Assim, é possível construir módulos e nesses módulos instanciam-se outros módulos a fim de montar o sistema final. Um processador por exemplo, é um módulo composto por outros módulos menores que funcionam de acordo com um relógio e estão integrados no seu núcleo.

Para concluir a exposição do SystemC, é necessário esclarecer como funciona a avalia-

ção dos eventos. Um ciclo-delta (ou delta-cycle) é uma porção infinitesimal de tempo que permite ordenar os eventos no simulador. Embora um evento seja avaliado em um ciclo delta, o tempo de simulação não avança e essa avaliação ainda é feita em dois passos que são avaliação e atualização, permitindo que a semântica associada aos canais primitivos (como tempo de propagação de um sinal) seja corretamente reproduzida.

Abaixo citamos o funcionamento do escalonador do SystemC a partir de [7]:

1. Inicialização: cada processo é executado uma vez para `SC_METHOD`s ou até um ponto de sincronização (`wait()`) para os `SC_THREAD`s;

2. Avaliação: seleciona os processos que estão prontos para execução ou prontos para continuar, possivelmente executando notificações imediatas, com custo de ciclo-delta mas com atraso-zero (zero delay). Se as notificações imediatas forem mal utilizadas a simulação pode não sair deste ponto pois a avaliação se dá até que não existam mais processos prontos para continuar. Neste ponto são feitos pedidos de atualização;

3. Atualização: Os pedidos de atualização são executados até que não tenha mais pedido algum. Apenas o último pedido de atualização tem efeito caso tenha sido feito mais de uma vez para um mesmo canal;

4. Se há notificações agendadas para o tempo atual de simulação (delta delay), determinar que processos estão aguardando e voltar a etapa de avaliação;

5. Se não há mais notificações agendadas para qualquer instante, o simulador termina;

6. Se há outras notificações para o futuro, avança-se o tempo no simulador para o próximo instante, que é o da notificação agendada para mais cedo;

7. Determinar os processos que estão prontos para execução por eventos que tenham notificações pendentes no tempo atual e repetir a partir do estágio de Avaliação.

Para se comparar as duas linguagens, visualize o VHDL, e mesmo o Verilog, como uma linguagem desenvolvida especificamente para descrever sistemas digitais com um simulador semelhante ao do SystemC. Construções como fios, atrasos, atribuições paralelas e métodos paralelos são fáceis de representar nessas linguagens, entretanto, utilizar qualquer coisa fora do domínio dos modelos do hardware sendo projetado é difícil e compromete a execução do simulador. Gerar saída no console do simulador, fazer com que essa saída seja oriunda das respostas do sistema e realimentá-la é tarefa complexa. Por

exemplo, transformar informações matematicamente exige, no VHDL, um pacote matemático específico e a ausência desse pacote pode proibir a execução da simulação por falta de resultado. Mas o pacote matemático é um requisito que pode não ser necessário no sistema final pois uma tabela com os valores resolveria. Como o projetista perdeu recursos para prover o pacote matemático e apenas posteriormente descobriu uma solução barata, seu concorrente resolveu adotar SystemC e utilizar o processador matemático da CPU que executava o simulador, o que ele não conseguia fazer usando Verilog.

Da mesma maneira que o processador matemático da plataforma utilizada para executar o simulador VHDL não pode ser utilizado, os outros recursos computacionais como a interface de rede, o monitor de vídeo e outros dispositivos de entrada e saída também. Não há pacotes de fácil acesso que viabilizem tal coisa, mas, usando o SystemC e as interfaces disponíveis no ambiente pode-se utilizar todos os recursos disponíveis na plataforma do simulador de forma independente do SystemC ou do compilador C++ empregado utilizando bibliotecas de acesso do sistema operacional e dos dispositivos que são comumente escritos em C e tem amplo suporte da comunidade via Internet.

Visto que SystemC suporta estruturas que permitem modelar sistemas de hardware e que possui suporte para bibliotecas do sistema operacional e outras via bibliotecas do C, propõe-se que é possível integrar instâncias diferentes do simulador em máquinas diferentes como um sistema distribuído. Além de instâncias diferentes é possível encapsular outros sistemas e viabilizar a geração de eventos destes de forma visível ao simulador enriquecendo a experiência de simulação e diminuindo o tempo de implementação dos componentes do sistema final. Conforme expusemos, o simulador do SystemC não prevê que eventos gerados por outro simulador, ou programa ou hardware externo, executando em paralelo sejam levados em consideração durante a simulação, entretanto, é possível, através dos módulos adequados, expor os eventos gerados por um canal de comunicação em um simulador A e aguardar por estes eventos em um outro processo em um simulador B.

Uma vez implementado um canal de comunicação entre simuladores paralelos podemos distribuir o esforço computacional entre nós diferentes possivelmente aumentando a eficiência do sistema, seja otimizando a execução, seja usando recursos que de outra forma não estariam disponíveis. Neste trabalho focamos no segundo caso. Apresentaremos um módulo externo ao simulador que funciona de forma independente mas que consegue gerar eventos que se tornam compreensíveis para o simulador do SystemC.

Usando essa abordagem é possível diminuir o tempo necessário para se ter as primeiras

impressões do sistema funcionando visto que todos os recursos para a plataforma do simulador podem estar prontos, são encapsuláveis (como a rede e outros dispositivos de entrada e saída) e vão permitir um diagnóstico precoce dos elementos a otimizar, gargalos e custos diversos.

Outro uso prático do canal de comunicação entre simuladores paralelos seria sobrepor a execução de dois simuladores arranjados para execução de verificação funcional. No cenário indicado em [1] é apresentada uma bancada de testes com auto-verificação baseada em um modelo de referência. A dificuldade de executar dois simuladores paralelos, ou mesmo a complexidade de se integrar o modelo de referência e o dispositivo sendo testado no mesmo simulador são postos como dificultades. Tendo um canal de comunicação eficiente, capaz de gerar eventos e notificações entre os simuladores pode-se paralelizar a execução da verificação funcional, integrando suas diversas etapas. Sem a comunicação entre simuladores, é necessário guardar os estímulos e as respostas do *design* em arquivos e em uma etapa seguinte de alimentar os estímulos no modelo de referência e comparar suas respostas com os resultados do arquivo de saída gerado na fase anterior, processos sem sobreposição alguma.

1.4 Justificativa

Durante o desenvolvimento de um sistema em silício são importantes as diversas etapas de verificação e refinamento dos modelos. Ver o comportamento do sistema em campo ou ter o desempenho do sistema final durante as etapas de simulação e desenvolvimento pode trazer efeitos positivos sobre possíveis investimentos e ainda chamar a atenção de que é possível fazer e que um sistema é eficiente, além de viabilizar medidas concretas do esforço necessário para se ter o sistema final em funcionamento.

A verificação da funcionalidade do sistema em níveis abstratos é uma etapa fundamental no projeto de sistemas em silício. Quanto mais cedo for a deteção de erros de projeto menor será o custo de sua correção [1]. No caso de sistemas em silício, existe um custo extremamente alto na sua fabricação física. A descoberta de erros de projeto depois da implementação física incorre em custos extremamente altos. Um exemplo clássico disso é a falha no processador Pentium da Intel em 1994, que só foi corrigida depois da fabricação e distribuição do processador. O custo monetário foi enorme, sem levar em conta o prejuízo a imagem da empresa. Por esses motivo, um esforço muito grande é feito durante a concepção do sistema para que ele execute corretamente já na primeira rodada

de fabricação.

A motivação para montar um estudo de caso vem do projeto Namitec [9], especificamente do grupo de *Desenvolvimento de um SoC reconfigurável, com sensores integrados e capacidade de comunicação sem fio*. Nesse grupo existe uma proposta de utilizar um SoC reconfigurável no rastreamento de objetos em campo, entre SoCs para outras finalidades.

Para tentar demonstrar que é possível um sistema de rastreamento em desenvolvimento com comportamento de uma implementação final e expor seus problemas em estágio inicial, tenta-se aqui implementar um segmentador de imagens, primeiro passo na constituição desse tipo de sistema, que constitua uma plataforma básica para o desenvolvimento desse SoC de rastreamento. Para isso é necessário implementar uma interface de captura de imagens, uma de segmentação e uma que disponibilize o resultado dessa segmentação para análise.

Conforme vemos em [17], o problema da segmentação de imagens é um passo básico para diferentes processos no campo das aplicações de processamento de informações visuais como:

- compressão de imagens baseadas em objetos;
- reconhecimento e rastreamento de objetos em sistemas de transporte inteligentes;
- visão computacional (para robôs);
- descrição de conteúdo multimidia como componentes espaço temporais;
- detecção de movimento.

Dos algortimos propostos para a segmentação de imagens, pode-se classificá-los a grosso modo em:

- Temporais: diferenças entre imagens consecutivas ou entre uma imagem e o fundo, com a desvantagem anunciada de que não é adequado para aplicações que necessitem obter tanto informações de objetos parados quanto em movimento;
- Espaciais (active contour, snake, sobel), baseadas em:
  - pixel;
  - bordas;
  - regiões;

  - modelos.

- Espaço-temporais.

Algoritmos temporais dependem da existência de pelo menos duas imagens que possam ser usadas como referência para o processamento. Algoritmos espaciais baseados em pixels e bordas demandam etapas de deteção de bordas e segmentação, além de que utilizam operações dispendiosas computacionalmente como produtos de matrizes e gradientes, o que desrespeita a autonomia, muito limitada, dos sistemas sem fio. Por isso decidiu-se utilizar a detecção de segmentos espaciais porposta em [20] e [8]. Esse algoritmo apresenta vantagens como pouca sensibilidade a ruídos, independência entre as imagens, processamento de pixels em paralelo, processamento de imagens coloridas ou monocromáticas e alto desempenho a baixo custo computacional.

O texto da dissertação é estruturado como segue. No capítulo 2 é realizada uma revisão sobre o desenvolvimento de modelos em SystmeC. No capítulo 3 apresenta-se conceitos relacionados a simulação distribuída e descreve-se a solução adotada neste trabalho para paralelizar a simulação de descrições SystemC. No capítulo 4 descreve- se o estudo de caso utilizado para analisar a solução proposta. Finalmente, as conclusões deste trabalho são discutidas no capítulo 5.

2 *Modelagem em SystemC de sistemas em silício*

2.1 Caracterização dos modelos mais comuns

Quando se descreve um sistema em silício é comum, nas diversas linguagens que podem ser utilizadas, dar detalhamentos diferentes para as diversas partes deste sistema. Existem algumas nomenclaturas, sem convenção definitiva ainda, associadas a alguma característica específica daquele nível de detalhamento. Quanto mais funcionalidade e menos detalhes de implementação o modelo expressa, diz-se que o modelo é de mais alto nível. Um modelo que tenha todos os detalhes possíveis é chamado um modelo de baixo nível. Pode ficar a sensação de que um modelo de baixo nível é ruim, entretanto, o modelo utilizado para a fabricação final do sistema é gerado com o auxílio de ferramentas de síntese é o modelo mais próximo da implementação final (física). A fabricação a partir de um modelo de alto nível sem restrições na codificação ainda não é possível pois não contempla muitos detalhes que a tecnologia contemporânea para implementação em silício requer.

Como existem diversas nomenclaturas para os níveis de descrição de hardware, descreve-se a utilizada neste trabalho a seguir. Uma especificação executável é o modelo de mais alto nível que consideraremos. Nele, não há qualquer referência a temporização, módulos ou modelo de comunicação, apenas é possível validar as entradas e as saídas do sistema como um todo, uma caixa preta responsável por abstrair todos os detalhes de implementação e decisões de projeto associadas. A especificação executável é de interesse de validação funcional e para futura depuração do sistema final. Ela pode ser apenas um oráculo com todas as respostas codificadas sem que as respostas sejam geradas por algoritmos e usada como base de comparação para as respostas produzidas através de decisões algorítmicas tomadas sobre as entradas dos modelos menos abstratos.

Considerando um desenvolvimento incremental, a partir da especificação executável

pode-se dar detalhes em diferentes níveis às partes do sistema. Cada detalhamento pode ser agrupado nas características: comunicação entre módulos, tempo de relógio (wall clock time), precisão de interfaces de comunicação em termos de pinos, precisão da execução em ciclos de relógio e precisão temporal nos atrasos dos sinais da implementação do sistema. O detalhamento da comunicação entre módulos pode ser feito de diversas maneiras, entre elas cita-se variáveis, métodos, funções, sinais, pinos ou portas e procedimentos. A comunicação por variáveis compartilhadas é uma das formas mais rudimentares que se pode usar para comunicar módulos distintos. Métodos, funções e procedimentos com suas respectivas assinaturas (que nada mais são que listas de parâmetros) podem agrupar funcionalidades que posteriormente podem ser implementadas como partes de software ou componentes de hardware com parâmetros bem definidos; através deles módulos diferentes podem usar e disponibilizar funcionalidades entre si. Sinais são utilizados para comunicar módulos com uma semântica de atualização definida (requisição de atualização e atualização efetiva), ou seja, mais elaborada que a atualização imediata característica das variáveis compartilhadas. Pinos são fronteiras de ligação típicas de módulos de hardware, podendo ser de entrada e/ou saída explicitando a direção da comunicação.

Uma vez que uma especificação apresente o detalhamento de comunicação entre seus módulos (módulos abstratos ou não) ela deixa de ser uma especificação funcional típica e pode ser detalhada como um modelo em nível de transações ou TLM de *transaction level model*. Esse modelo tem em si o detalhamento suficiente das informações trocadas entre os módulos do sistema, além da delimitação desses módulos (`SC_MODULE`). Essa separação e definição das fronteiras de comunicação, mesmo que em nível abstrato permite que o modelo seguinte, ou modelo funcional temporizado, possa evidenciar o tempo de simulação. Não faremos distinção entre o modelo funcional não temporizado e o modelo TLM, como feito em [7].

Um modelo TLM tem como característica o foco nos problemas da aplicação como um sistema que pode ser implementado em hardware. Como resolver o problema deve ser tratado na especificação executável, por exemplo. Idealmente um modelo TLM é assíncrono, ou seja, para cada intervalo de tempo do simulador uma operação tem todos os dados que precisa para avançar e gerar uma resposta. Cada transação é auto contida, assim, uma transação n+1 não deve depender das entradas da transação n. Embora pareça com um modelo dataflow, uma transação pode interferir no tratamento de outra transação seguinte, mas não pode depender disso. Um exemplo é uma transação de transmissão de um dado. Se há necessidade de *handshake* ele deve estar na mesma transação que o dado de outra forma não seria transmitido por violação de protocolo. Esse mesmo dado

pode ainda instruir o canal a atrasar mensagens posteriores. É possível também que uma transação grande possua diversas transações menores, tantas quantas forem necessárias à funcionalidade. Cita-se como exemplo o barramento USB que precisa de várias mensagens com sincronização e sinalização de final, que podem ser encapsuladas em transações, com a finalidade de comunicar dados sendo que a seqüência das mensagens altera estados internos do sistema não necessariamente transmitindo o dado na primeira mensagem.

Num simulador de sistema de hardware, é importante ter em mente que temos três divisões temporais: o tempo de execução, o tempo de simulação (tempo do simulador) e o tempo do ciclo. Cada contador desses tempos avança de forma independente. O tempo de execução é medido pelo relógio cronológico (wall clock time) e varia em função da complexidade do sistema, do nível de detalhamento do modelo e dos recursos computacionais disponíveis. O tempo de simulação avança de acordo com a semântica do simulador e em intervalo definido durante a execução (pelo `sc_set_time_resolution` ou na falta deste, no intervalo definido por `sc_clock` para o SystemC, em outros simuladores o tempo avançará depois de processados todos os eventos gerados nos delta-delays). Não havendo eventos agendados para um instante dentro do intervalo do simulador, ele incrementará um tempo de ciclo. O tempo de ciclo dispara um evento regular recebido pelo sistema e que sincroniza sua execução. O modelo que leva em conta o tempo do simulador é o modelo funcional.

Através do modelo funcional é possível aumentar ou diminuir o tempo necessário para a emissão da resposta de cada módulo, no tempo do simulador, permitindo fazer testes de sincronização entre os módulos e fazer uma exploração do espaço de arquitetura e determinar limites relativos para os períodos de execução. No modelo TLM as funções do sistema são executadas conforme a disponibilidade dos dados. No modelo funcional, os atrasos codificados pelo programador interferem diretamente nessa disponibilidade.

A seguir, é necessário explicitar as fronteiras de comunicação do sistema em pinos de entrada e ou saída, que serão as interfaces com o exterior do circuito após a fabricação e que determinam os limites dos valores binários que podem ser comunicados e possivelmente sinais de controle. Uma descrição que tem todos os pinos correspondentes do circuito é chamado modelo comportamental ou *behavioral model*. Neste modelo o comportamento do sistema e seus módulos estão todos definidos com razoável precisão. Houve ferramentas de síntese (como foi o caso do behavioral compiler de [15]) que já conseguiram sintetizar o sistema físico final a partir de modelos comportamentais, desde que o código do modelo obedecesse a certas restrições e regras de codificação.

Para viabilizar uma síntese de melhor qualidade e mais eficiente, deve-se detalhar mais ainda o modelo do sistema, explicitando detalhes internos aos módulos e outros detalhes de comunicação. O modelo que detalha os módulos internamente é chamado de RTL, o *register transfer level* ou nível de transferência de registradores. Neste modelo o número de vezes que um módulo aparece no sistema e o seu compartilhamento ficam evidentes. Os protocolos de comunicação e multiplexação dos fios de comunicação devem ser bem definidos e geralmente obedecem orientações das ferramentas usadas para gerar automaticamente o modelo físico. Além disso, a temporização desse modelo é mais rigorosa e já não funciona apenas com a disponibilidade dos dados. Utiliza-se esse modelo para descrever circuitos síncronos, definido em termos de blocos de lógica combinacional e registradores. A transferência de informações é sincronizada por um relógio, e as tarefas tem duração definida em termos do número de ciclos de relógio.

Na especificação executável, no modelo TLM, no funcional e até mesmo no comportamental, a execução é orientada pela disponibilidade dos dados, caracterizando a execução como dataflow: o fluxo de dados determina um passo assíncrono para o sistema. Nos modelos RTL e GateLevel, a sincronização é feita pelo relógio do sistema. Os eventos são esperados e detalhados com base nos pulsos deste relógio e não mais com base na disponibilidade dos dados, o tempo não é mais dado de forma abstrata, representando um esforço maior por conta do codificador realizar qualquer modificação. Nesse nível de detalhamento, o número de ligações entre os componentes é maior e exige mais atenção no momento de acrescentar, remover, conectar e desconectar os pares de módulos.

No modelo físico, ou *gate level*, são descritos todos os transistores, fios e ligações entre os componentes eletrônicos do circuito. Esse modelo geralmente é feito automaticamente pelas ferramentas de síntese a partir de modelos RTL e comportamentais e demandam o máximo de esforço de implementação pois devem considerar a posição física de cada parte do sistema e os atrasos e efeitos das demais partes do sistema sobre si mesmo.

A tabela a seguir resume as principais diferenças e características dos modelos apresentados. Veja que deve ficar claro que os modelos de baixo nível têm necessariamente as características funcionais dos modelos de alto nível sempre com um detalhamento maior.

Em [3], vemos que são colocados apenas três níveis de detalhamento: comportamental, arquitetural e implementação. Isso é um leve contraste com [7] que apresenta os *modelos* em que nos baseamos. Assim, o nível comportamental compreenderia qualquer modelo sem informação de tempo (a especificação executável e o TLM). Para o nível arquitetural são separados os componentes que computam e os componentes que armazenam resulta-

	GateLevel	RTL	Behavioral	FM	TLM	Especif. Executável
Esforço de codificação	←	Maior			Menor	→
Precisão de atrasos	x	-	-	-	-	-
Precisão de ciclos	x	x	-	-	-	-
Precisão de pinos	x	x	x	-	-	-
Tempo	x	x	x	x	-	-
Comunicação	x	x	x	x	x	-
Funcionalidade	x	x	x	x	x	x
Sincronização	precisa	relógio	dataflow	dataflow	dataflow	dataflow
Comunicação	sinais	sinais	sinais/filas	filas	interfaces	(*)
Tempo de simulação	←	Maior			Menor	→
Foco nos tempos de	atraso	ciclo	simulador	simulador	execução	execução

Tabela 1: Tabela comparativa dos modelos de descrição de hardware (*) variáveis compartilhadas, métodos ou funções

dos, além dos fios, barramentos, portas de entrada e saída e informação abstrata de tempo (compreendendo o modelo funcional e comportamental). Para o nível de implementação, os componentes que computam resultados são descritos em termos de transferências de informações entre seus componentes por ciclo de relógio ou em termos de seqüências de instruções.

2.2 Implementação de alguns modelos em SystemC

Nesta seção apresentamos exemplos que caracterizam alguns modelos de interesse.

Uma especificação executável de uma ULA, unidade lógico aritmética, com 5 operações: multiplicar, dividir, somar, "e" lógico e "não" lógico, com indicação de erro para divisão por zero e sem *overflows* ou *underflows* pode ser da seguinte maneira:

```
// Definicoes comuns
#include <stdio.h>
#define _MUL 0
#define _DIV 1
#define _ADD 2
#define _oAND 3
#define _NOT 4

int ula(int a, int b, int op, int *err) {
```

```
    int r = 0;
    *err = 0;
    switch(op) {
      case _MUL: r = a * b; break;
      case _DIV: if(b == 0) *err = 1; else r = a / b; break;
      case _ADD: r = a + b;  break;
      case _oAND: r = a & b; break;
      case _NOT: r = !a; break;
    }
    return r;
}
int main(int argc, char* argv[]) {
    // Parse de parametros
    int err, a, b, op;
    if(argc < 3) return 0;
    op = argv[1][0]-'0';
    a = argv[2][0]-'0';
    b = argv[3][0]-'0';
    printf("Op: %u A: %u B: %u \n", op, a, b );

    //
    printf("Resultado: %u", ula( a, b, op, &err ) );
    printf("Erro: %u \n ", err);
    return 1;
}
```

Observe que nos modelos de especificação executável, a computação ocorre ainda de forma seqüencial, sem indicação do tempo gasto pelas operações e que basta um compilador C para que funcionem. Refinando o modelo percebe-se que o resultado da ULA estava implícito na chamada de função e temos então a noção exata dos valores que podem ser comunicados pela ULA.

```
//Definicoes comuns
...
typedef struct ula_t {
       int err;
```

```
        int a;
        int b;
        int op;
        int r;
     } ula_dummy;
void ula(struct ula_t *v_ula) {
    v_ula->err = 0;
    switch(v_ula->op) {
      case _MUL: v_ula->r = v_ula->a * v_ula->b;  break;
      case _DIV:
          if(v_ula->b == 0) v_ula->err = 1;
          else v_ula->r = v_ula->a / v_ula->b;  break;
      case _ADD: v_ula->r = v_ula->a + v_ula->b;  break;
      case _oAND: v_ula->r = v_ula->a & v_ula->b; break;
      case _NOT: v_ula->r = !(v_ula->a);     break;
    }
}
int main(int argc, char* argv[]) {
  int err, a, b, op;
  // Parse de parametros
  ...
  //
  struct ula_t ula_s;
  ula_s.err=0;
  ula_s.a=a;
  ula_s.b=b;
  ula_s.op=op;

  ula( &ula_s );
  printf("Resultado: %u", ula_s.r );
  printf("Erro: %u \n ", ula_s.err);
  return 1;
}
```

De acordo com os documentos do consórcio do SystemC [16], num modelo TLM toda comunicação deve ser feita via chamada de funções. Assim, precisamos modificar nova-

mente a ULA para que ela receba e devolva seus operadores através de funções padronizadas e passe então a efetivamente requerer o SystemC. Quando da escrita desse documento a função padrão utilizada para os modelos TLM tem semântica de interface idêntica à da classe `sc_fifo`, a qual utilizaremos por nos parecer de mais fácil compreensão, sem entrar no mérito da implementação da interface.

```
// Definicoes comuns
...
#include "systemc.h"
#include <stdlib.h>
typedef struct ula_ti {
  int a;
  int b;
  int op;
} ula_dummyi;
typedef struct ula_to {
  int err;
  int r;
} ula_dummyo;

SC_MODULE(ula) {
  sc_fifo_in <struct ula_ti *> inputs;
  sc_fifo_out<struct ula_to *> outputs;

  void operate(void) {
    struct ula_to r;
    struct ula_ti *vi_ula;
    while(true) {
      vi_ula = inputs.read();
      r.err = 0;
      switch(vi_ula->op) {
        case _MUL: r.r = vi_ula->a * vi_ula->b; break;
        case _DIV: if(vi_ula->b == 0) r.err = 1;
                   else r.r = vi_ula->a / vi_ula->b; break;
        case _ADD: r.r = vi_ula->a + vi_ula->b; break;
        case _oAND: r.r = vi_ula->a & vi_ula->b; break;
```

```
        case _NOT: r.r = !(vi_ula->a); break;
      }
      outputs.write(&r);
    }
  }

  SC_CTOR(ula) {
    SC_THREAD(operate);
  }
};

int sc_main(int argc, char* argv[]) {
  // Parse de parametros
  ...
  //
  ula sc_ula("sc_ula"); // Instanciar a ula;
  sc_fifo <struct ula_ti *> ula_si; // // Instanciar canal
  sc_fifo <struct ula_to *> ula_so; // // Instanciar canal
  struct ula_ti ula_vi; // Valores
  struct ula_to *ula_vo; // Valores
  sc_ula.inputs(ula_si); // Ligacoes
  sc_ula.outputs(ula_so);
  // Valores de entrada
  ula_vi.a=a;
  ula_vi.b=b;
  ula_vi.op=op;
  ula_si.write(&ula_vi);
  // Simulacao
  cout << "Ready to start!"<< endl;
  sc_start(2);
  cout << "Simulation is over!" << endl;
  // Valores de saida
  ula_vo = ula_so.read();
  printf("Resultado: %u ", ula_vo->r );
  printf("Erro: %u \n ", ula_vo->err);
```

```
  return 1;
};
```

O último modelo que apresentaremos exemplo é o funcional temporizado. Neste modelo, as diferentes operações levam tempos diferentes para concluir. Esse tempo diferente é estimado e colocado na função `wait()` para permitir expressar o custo computacional, e conseqüentemente explorar o espaço de arquitetura. Para tanto, exibimos a seguir os trechos de código modificados:

```
...
      switch(vi_ula->op) {
        case _MUL:
            r.r = vi_ula->a * vi_ula->b; wait(45, SC_NS); break;
        case _DIV:
            if(vi_ula->b == 0) r.err = 1;
            else r.r = vi_ula->a / vi_ula->b; wait(35, SC_NS); break;
        case _ADD:
            r.r = vi_ula->a + vi_ula->b; wait(10, SC_NS); break;
        case _oAND:
            r.r = vi_ula->a & vi_ula->b; wait(10, SC_NS); break;
        case _NOT: r.r = !(vi_ula->a); wait(5, SC_NS); break;
      }
      outputs.write(&r);
      cout << "Tempo do simulador " << sc_time_stamp() << endl;
...
int sc_main(int argc, char* argv[]) {
  // Parse de parametros
  ...
  //
  sc_set_time_resolution(1.0, SC_NS);
  ula sc_ula("sc_ula"); // Instanciar a ula;
  ...
  // Simulacao
  cout << "Ready to start!"<< endl;
  sc_start(10); cout << "10 ns" << endl;
  sc_start(10); cout << "20 ns" << endl;
```

```
  sc_start(10); cout << "30 ns" << endl;
  sc_start(10); cout << "40 ns" << endl;
  sc_start(10); cout << "50 ns" << endl;
  cout << "Simulation is over!" << endl;
...
```

Ao executarmos o modelo com as modificações propostas temos os resultados a seguir:

Parâmetros	t0	t1	t2	t3	t4	t5	Resultado	Erro
Op: 0 A: 5 B: 5	10 ns	20 ns	30 ns	40 ns	45 ns	50 ns	25	0
Op: 1 A: 5 B: 0	10 ns	20 ns	30 ns	35 ns	40 ns	50 ns	0	1
Op: 2 A: 5 B: 3	10 ns	10 ns	20 ns	30 ns	40 ns	50 ns	8	0
Op: 4 A: 5 B: 3	5 ns	10 ns	20 ns	30 ns	40 ns	50 ns	0	0

Tabela 2: Resultados do modelo funcional temporizado

Observe os tempos abordados anteriormente. O tempo de execução do código é muito curto, praticamente instantâneo. O tempo do simulador avança em passos irregulares dados pelos atrasos da ULA e o tempo de ciclo é de 10ns, e avança de forma manual. Para gerar um relógio com pulsos de comprimento 20 ns bastaria adicionar um sinal e mudar o seu valor entre cada `sc_start(10)`, mas é possível gerar eventos periódicos usando uma linha de relógio `sc_clock` do SystemC produzindo eventos periódicos automaticamente e executar o simulador por tempo indefinido, chamando-se `sc_clock()` sem parâmetros.

Os demais modelos não serão apresentados pois entram em detalhes como número de bits das entradas e saídas e a propagação da linha com o relógio (*clock*) e fogem do escopo deste trabalho. Nos modelos apresentados, a comunicação é feita de forma *explícita* mas não *detalhada* em grânulo fino: não é possível dizer quantos ciclos de relógio são necessários para multiplicar ou que algoritmo é usado nem quantos pinos teria um chip com a ULA implementada. No modelo TLM e no modelo funcional temporizado a idéia é estabelecer uma ordem de atualização dos sinais e verificar o seu comportamento em situação de concorrência e não determinar quantos ciclos exatos toma a execução de cada componente; essa tarefa está no nível do modelo RTL, após a definição da precisão dos tipos que estão sendo operados. Combinar ULAs como a descrita já responde questões sobre como o sistema final responderá caso a temporização das multiplicações em relação às somas não seja respeitada.

Cabe ainda dizer que no modelo TLM apresentado foi utilizada uma interface padrão do SystemC, a `sc_fifo`. Entretanto, o poder de expressão dessa linguagem é bem maior. Através de *event finder*s, *event*s definidos pelo usuário e da chamada `notify()` é possível

definir canais complexos com diversas fases e estados internos com interações complexas ao longo do código ainda assim garantindo a reusabilidade. Detalhes sobre esse tipo de implementação podem ser encontrados em [7]. Para implementar o modelo proposto de simulação distribuída será mantido o uso do `sc_fifo` por simplicidade.

Espera-se que com modelos abstrato escondendo detalhes particulares de sincronização entre os módulos através de interfaces diminua-se o ciclo de desenvolvimento e torne possível visualizar a execução de camadas superiores da aplicação sem que as camadas inferiores estejam completamente detalhadas. Definidas as funcionalidades, propomos diminuir o tempo de execução da simulação e viabilizá-la permitindo o uso de recursos de coexistência inviável com a biblioteca do SystemC.

3 Simulação distribuída

Ao modelar sistemas complexos, os recursos de um único computador podem não ser suficientes para gerar resultados em tempo hábil. Quanto mais detalhes modelados mais tempo o simulador levará executando um mesmo cenário. Nem todos os recursos necessários para a execução do sistema podem estar disponíveis em um mesmo nó. Para diminuir o impacto desses problemas em sistemas modelados em SystemC pode-se distribuir o simulador em diversas máquinas, num *cluster* heterogêneo. Para possibilitar essa distribuição, é necessário rever alguns conceitos e acompanhar alguns exemplos.

Num modelo TLM, mesmo com a descrição do hardware em estágio inicial, é possível que o custo de execução do sistema seja alto. Um exemplo disso é o modelo de verificação utilizado em projetos como o BrazilIP [2] [18] e citado em [1] como um possível arranjo para a verificação funcional. No caso do dispositivo USB, implementado pela Universidade de Brasília, foram gastas mais de 24 horas executando a simulação com uso de CPU em 100%. No caso do decodificador mpeg, implementado pela UFCG, um ganho razoável pôde ser observado quando comparada a execução da simulação em um computador de 32bits com um processador e um computador com dois processadores de 64 bits, utilizado para diminuir o tempo de simulação.

O modelo utilizado para fazer a verificação desses dispositivos consistia em utilizar um modelo TLM externo que fornecesse os estímulos idênticos (*source*) a um modelo de referência, preferencialmente uma especificação executável, e ao dispositivo sendo testado (*Design Under Verification* ou apenas DUV) e os resultados fornecidos para um comparador (*checker*) responsável por verificar se as respostas do modelo de referência são equivalentes às respostas do DUV. Os dispositivos sendo testados eram modelos RTL usados posteriormente como entrada para ferramentas de síntese automática. Para que as transações de alto nível pudessem ser traduzidas do source para o modelo RTL e de volta para o *checker* foram utilizados *drivers* responsáveis por implementar os protocolos de comunicação e injetar os sinais apropriadamente no DUV, e monitores, responsáveis por coletar os resultados e repassá-los ao *checker*. Veja a ilustração do modelo de verificação,

apelidado de “U invertido”, na figura 1

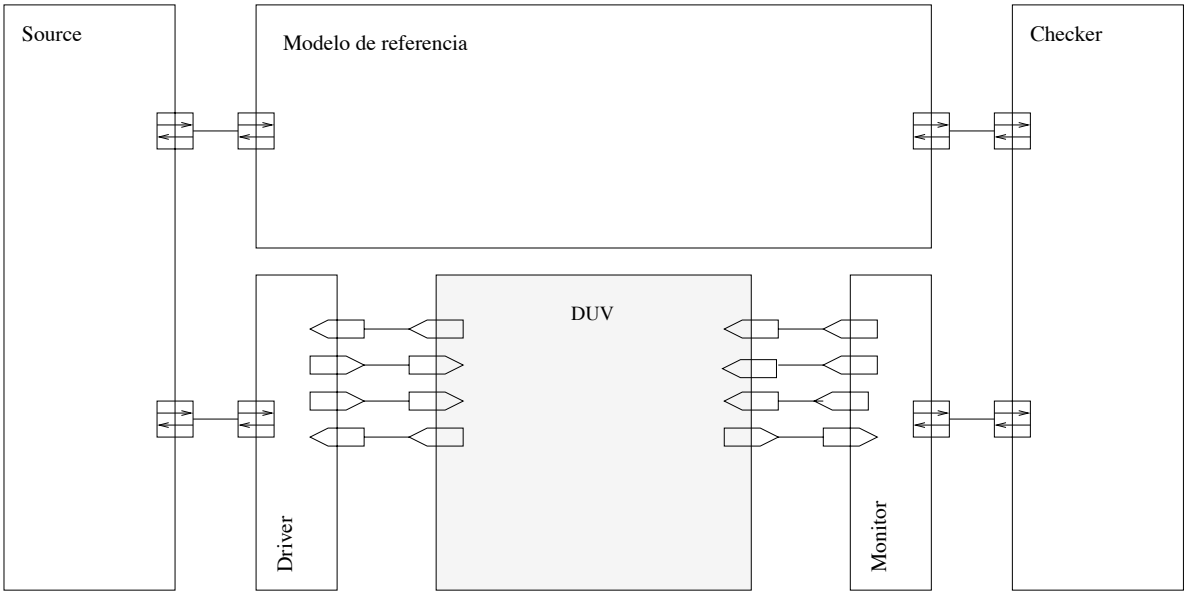

Figura 1: Modelo do “U” invertido utilizado no BrazilIP

Os ganhos no tempo de simulação obtidos no caso do decodificador mpeg foram oriundos dos recursos disponibilizados pelo sistema operacional para o simulador do SystemC. O sistema operacional tinha a disposição dois processadores de melhor desempenho e arquitetura de 64bits podendo dividir os processos do simulador e do sistema operacional. Mesmo que os dois processadores não fossem utilizados pelo simulador os processos de responsabilidade do SO são executados em paralelo. Para criar uma visão única do sistema, disponibilizando os dois processadores para o simulador é necessário rever como o simulador do SystemC funciona.

No capítulo anterior citamos que o simulador do SystemC executa os `SC_METHOD`s seqüencialmente, com cada `SC_METHOD` executando em tempo infinitesimal (um *delta_ cycle*) sem avançar o tempo do simulador, desde que eles estejam prontos para execução, comportamento que se repete para os `SC_THREAD`s até que seja encontrada alguma primitiva bloqueante ou `wait()`. As threads e métodos do SystemC (versão 2.0.1) são postos em execução através da *framework QuickThreads* [12], que é uma conjunto de funções e mecanismo de encapsulamento para implementar threads, ou seja, processos leves dentro de um processo do sistema operacional.

A vantagem de se utilizar uma thread é que o tempo extra gasto pelo sistema operacional para guardar e recuperar o contexto do processo é reduzido. No caso das quickthreads, elas não são preemptivas, ou seja, sempre vão executar até devolverem o controle ao processo principal, ou thread do escalonador. Uma pequena desvantagem da implementação

do SystemC é que as threads executam sobre um único processo e depende do sistema operacional para que seja alocada em mais de um processador. Daí concluímos que se o sistema operacional não puder oferecer múltiplos processadores a um processo não poderemos obter ganhos em tempo de simulação para uma mesma arquitetura onde se possa variar o número de processadores.

Uma solução barata para forçar o uso de vários processadores poderia ser utilizar uma primitiva como "fork" e obter dois processos idênticos executando a simulação e comunicando-se usando algum mecanismo de IPC (*interprocess communication*). Ainda assim resta o problema de comunicar os simuladores em processadores funcionando em paralelo. Esse problema de comunicação quando resolvido permite, além de possivelmente tornar a execução mais rápida, que processos alheios ao simulador, como dispositivos de entrada e saída quaisquer, possam se comunicar com os dispositivos sendo simulados, ampliando as possibilidades da linguagem.

Computadores podem ser classificados usando a classificação de Flynn [6], em arquiteturas do tipo SISD (um fluxo de instruções e um fluxo de dados), SIMD (um fluxo de instruções e vários fluxos de dados), MISD (vários fluxos de instruções e um fluxo de dados) e MIMD (multiprocessador, vários fluxos de instruções e vários fluxos de dados). Arquiteturas SISD e SIMD são mais comuns e não oferecem paralelismo em nível de processos, isto é, processos paralelos. Arquiteturas MISD não possuem implementação representativa. As arquiteturas MIMD (multicomputadores, *clusters* ou *Network of Workstations*, várias CPUs ligadas por uma rede de alto desempenho, uma máquina paralela) são interessantes por permitir que um mesmo problema possa ser abordado por diferentes processadores em diferentes processos, sejam através de múltiplos programas ou de um mesmo programa executando em nós diferentes.

A implementação da comunicação entre os processadores, isto é, entre os processos que executam nesses processadores pode ser feita em diferentes níveis. Os diversos níveis que uma máquina paralela possui em sua arquitetura, são citados a seguir, do mais alto ao mais baixo [4]:

1. Modelo de programação: esconde a ordem de execução atrás da ordem sequencial do programa;

   (a) Multiprogramação (*CAD*): não há coordenação ou comunicação entre as tarefas;

   (b) Endereçamento compartilhado (*shared memory*): todas as tarefas podem ver o

que as outras tarefas concorrentes estão fazendo e tomar decisões;

(c) Troca de mensagens (*message passing*): a comunicação é iniciada explicitamente entre os pontos envolvidos (por exemplo: sockets, pipes, sistemas de filas);

(d) Dados paralelos (*data parallel*): os dados são processados ao mesmo tempo com um evento de sincronização a cada passo; as arquiteturas *data parallel* compreendem os processadores vetoriais, os vetores processadores e registradores vetoriais para *Load* e *Store*;

(e) Dataflow: um pipeline circular simples que explora o casamento de *tokens* com as operações básicas de *tag match*, alocar *buffer*, transferir dados e postar eventos;

(f) Sistólico: os dados se movem em um vetor de elementos processadores.

2. Abstração de comunicação, limite entre o usuário e o sistema: estabelece um compromisso entre o software e o hardware de forma a trabalhar corretamente sem proibir a exploração tecnológica;

   (a) Software/Multiprogramação;

   (b) Bibliotecas (MPI, PVM);

   (c) Sistema operacional.

3. Hardware de comunicação;

4. Meio de comunicação.

O sistema operacional pode disponibilizar suporte do hardware do processador, fornecer opções de entrada e saída, o escalonador ou dispositivos de rede (para sistemas multicomputadores). Quando se utiliza bibliotecas que expõem os recursos do sistema operacional de forma padrão em diversas plataformas, diz-se que essas bibliotecas são responsáveis por criar uma visão única do sistema em que o programador não precise se preocupar com a distribuição dos processos e número de nós, por exemplo.

Dentro das camadas de abstração que apresentamos, nenhum dos mecanismos faz parte do SystemC. Iremos utilizar na camada mais alta troca de mensagens para comunicar os processos. Essa comunicação é implementada em nível de biblioteca do sistema operacional que abstrai o hardware e o meio de comunicação. Adotaremos troca de mensagens por ser de uso mais intuitivo. Os nós serão endereçados no início da execução e a

comunicação entre eles se dará por primitivas *send* e *receive*, principalmente para evitar problemas de concorrência com as escritas, leituras e semáforos associados ao modelo de memória compartilhada e por não haver disponibilidade de máquinas sistólicas, dataflow ou data parallel para esse estudo.

3.1 Processo de paralelização

O processo de paralelização envolve quatro etapas:

1. decomposição do cômputo em tarefas, expondo a concorrência
2. atribuição de tarefas aos processos, agrupamento de tarefas e balanceamento de carga
3. coordenação dos acessos a dados, comunicação, sincronização e redução de comunicação (escolha de primitivas bloqueantes, não bloqueantes ou síncronas)
4. delegação dos processos aos processadores, exploração do princípio da localidade (dados privados)

No processo de paralelização, questões importantes são levantadas e que geralmente interferem em maior ou menor proporção em diferentes etapas, uma vez que: no processo de coordenação certas tarefas podem ser atribuídas a outros processos; podem expor maior ou menor concorrência; ser estáticas ou dinâmicas; ter variado seu tamanho de grânulo e ainda há a questão do balanceamento de carga que incorre em lógica extra de gerenciamento, serialização e mesmo em custo de sincronização.

A atribuição de tarefas pode ser estática ou dinâmica, sendo que o dinamismo na atribuição de tarefas abre um leque de opções com vantagens bem peculiares. Algumas formas de atribuição de tarefas: semi-estática, fixada via *profiles*, tarefas dinâmicas, auto-escalonadas com ou sem guia, filas centralizadas ou distribuídas com ou sem roubo de tarefas (*task stealing*).

O balanceamento de carga pode fazer uso de diversos modelos conhecidos como o paralelismo de dados, o paralelismo de funções, controle, tarefas e a técnica de pipelines. Os efeitos colaterais da descentralização são a serialização e as opções de sincronização de desempenho, além de dificuldades de uso variadas como barreiras, sincronização ponto a ponto ou por grupos, bem como o uso de mecanismos de trava ou *locks*: individuais,

pequenos de seção crítica, semáforos ou grupos de travas em vetores ou tabelas de hash. Outras questões a resolver no balanceamento são a distribuição dos dados e do código e computações redundantes.

3.2 Aplicação do processo de paralelização

De acordo com o processo de paralelização colocado em [4] e citado na seção anterior, o cômputo precisa ser decomposto em tarefas, expondo a concorrência. Cada tarefa em um projeto SystemC pode ser expressa em diferentes níveis de detalhe. Utilizando o exemplo da ULA, podemos ver que transmitir uma transação inteira, com os operandos e o operador seria mais barato que transmitir separadamente cada operando e o operador separadamente, dada uma latência positiva e todo o *overhead* de comunicação. Veja a figura 2

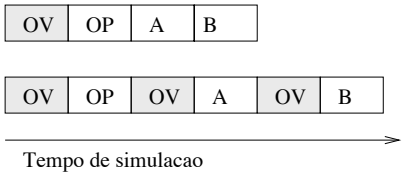

Figura 2: Exemplo de acumulação do *overhead* de comunicação no tempo total de simulação

Deve-se visualizar que cada vez que um dado é comunicado e que se aguarda uma resposta agregamos ao tempo de execução duas vezes a latência de comunicação (figura 3). Assim, precisa-se levar em consideração o que se comunica em cada um dos modelos possíveis de se descrever na linguagem e quanto ele agrega em tempo final de simulação.

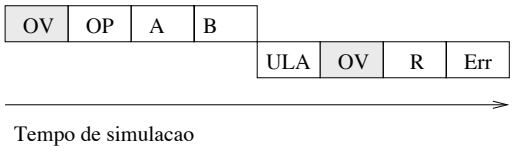

Figura 3: Exemplo de acumulação do *overhead* de comunicação bidirecional no tempo total de simulação

Conforme citamos no capítulo 2, os modelos gate level, RTL e comportamental apresentam os detalhes de comunicação em nível de pinos e logo não são candidatos à comunicação entre processos a menos que um conjunto de valores representados pelos pinos possa ser agrupado em uma ***transação*** passível de interpretação por um par *driver*/monitor imitando um modelo de mais alto nível. Modelos funcionais e TLM já tem um grão de

comunicação maior, bom candidato a ser transmitido para processamento em outro nó. A especificação executável, por sua natureza, pode ser paralelizada usando qualquer estratégia tradicional pois não tem compromisso com as fronteiras de comunicação da mesma forma que os modelos do sistema. A partir deste ponto os modelos funcionais e TLM serão referenciados apenas como TLM.

3.2.0.1 Decomposição em tarefas

Nos modelos TLM que estão no escopo deste trabalho, deve-se separar claramente *o tempo de comunicação, o tempo de execução da simulação e o tempo do simulador*.

Para explicitar o *tempo de comunicação*, precisamos definir o tamanho do grão que se comunica e seu tempo de execução. O grão que se considera é um componente arquitetural, que entende-se por um módulo `SC_MODULE`, hierárquico ou não, isto é, que pode ou não instanciar outros `SC_MODULE`s e que tem pelo menos uma `SC_THREAD` responsável por receber as *transações* que serão tratadas pelo módulo arquitetural em questão ou um canal adequado. Neste trabalho utilizamos um canal padrão do SystemC que é o `sc_fifo`. Daremos mais detalhes sobre esta escolha adiante.

Os diversos componentes arquiteturais que se referem aqui formam o sistema que está sendo modelado. Cada um deles pode ter detalhamentos diferenciados e certamente terão tempos de execução distintos tanto pela sua natureza quanto para cada transação recebida. Lembre-se que uma transação contém todas as informações necessárias para que o componente arquitetural possa desempenhar seu papel. No modelo do "U invertido" pode-se distinguir que o modelo de referência é um componente arquitetural e que o DUV, encapsulado pelo respectivo *driver* e monitor, formam outro componente. Pode ser que internamente o DUV ou o modelo de referência possam ser subdivididos em outros componentes, entretanto, o critério relevante para a etapa de decomposição de tarefas é *o tempo de execução* do componente, além é claro, dos requisitos funcionais do mesmo.

Os requisitos funcionais se apresentam sob a forma de demanda física do componente. Um gerador de imagens para um processador de imagens digitais pode não estar disponível no computador de melhor desempenho para a execução da simulação. Outro caso seria o da simulação de um dispositivo de saída, que pode não estar disponível em todos os nós do *cluster* usado para simulação. Também pode ser observado que o modelo de referência poderá consumir recursos significativos de CPU e que obviamente não depende do DUV para ser executado - são fortes candidatos a serem paralelizados.

Logo que os requisitos funcionais e a demanda de CPU com os tempos de execução dos componentes estejam claros, pode-se separar as tarefas em relação à eficácia da rede de conexões entre os simuladores. Não é vantajoso paralelizar a execução da simulação de um componente cuja transação tome tanto ou mais tempo de comunicação que o tempo gasto em seu processamento. Confira a ilustração na figura 4. É importante observar que nessa figura consideramos que a interface de rede não é capaz de realizar comunicação sem interromper o processador principal, impedindo que processamento possa ser realizado em paralelo com a comunicação. Interfaces modernas e dispendiosas podem contornar essa situação utilizando buffers grandes e transferindo para o sistema operacional vários pacotes em uma interrupção. Para o caso (a) não há paralelização; para o caso (b), temos dois nós processando o mesmo conjunto de dados com o tempo de comunicação é igual ao tempo de processamento sem ganho algum; para o caso (c) com tempo de comunicação sendo a metade do tempo de processamento observa-se que ainda sobra uma janela capaz de comunicar dados para uma próxima rodada de computação e o caso (d) mostra o caso em que o tempo de comunicação é maior que o tempo de processamento, sem ganho algum.

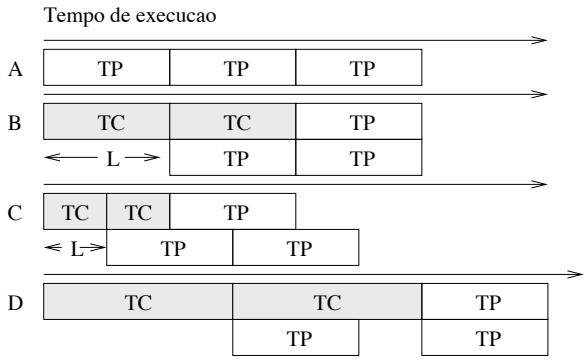

Figura 4: (A) Tempo de Processamento TP de 3 transações; (B) TP igual ao Tempo de Comunicação TC com L indicando a latência; (C) TC menor que TP, reduz o tempo de execução e (D) TC maior que TP

Existe espaço para definir metodologias de obtenção do desempenho de módulos e como dividi-los, tanto *bottom up* (do alto nível para o baixo) quanto *top down* (de modelos de baixo nível encapsulando-os até o alto nível). Uma delas pode ser encontrada em [3].

3.2.0.2 Atribuição de tarefas aos processos, coordenação e delegação

Neste ponto temos os componentes arquiteturais, as tarefas, divididos pelo tempo de execução e podemos agrupá-las conforme o desempenho da rede de comunicação que liga os nós que participarão da simulação distribuída. Cada processo agrupa tarefas de acordo

com o seu padrão de comunicação, que é determinado pelo projetista.

Definidos e agrupados os módulos em processos é necessário fazer o escalonamento das operações e decidir como os dados serão transmitidos, como e quando os resultados serão coletados. Neste ponto são definidas se serão utilizadas primitivas bloqueantes, não bloqueantes ou síncronas. As primitivas síncronas como barreiras e *join*s não serão abordadas neste trabalho. No caso das primitivas bloqueantes ou não bloqueantes depende-se de como são implementadas. No caso das filas que bloqueiam a execução para leitura quando vazias, pode-se implementar a leitura não bloqueante verificando-se o tamanho da fila antes de lê-la efetivamente. Para a escrita, de forma semelhante, pode-se verificar se há escritas pendentes antes de bloquear por falta de espaço em buffer ou mesmo implementar um buffer intermediário entre a fila final e sua fonte.

A exploração do princípio da localidade é oriundo do reaproveitamento de dados que não precisariam ser retransmitidos. Para o caso das transações recebidas pelos modelos esse princípio pode ser explorado para diminuir o uso da rede, iniciando processos geradores e consumidores no mesmo nó, por exemplo. Uma explicação ampla sobre o processo de paralelização pode ser encontrado em [4].

3.3 Opções de implementação

De acordo com a proposta de modelo TLM apresentada em [13], existem interfaces e canais padronizados no SystemC como uma biblioteca à parte que podem ser utilizados para poupar esforço de redefinir padrões para o modelo. Essas interfaces são padronizadas (todas herdam devem implementar `sc_interface`) sem foco em como serão implementadas. Entretanto associados a essas interfaces vêm algumas restrições de codificação que permitem a um sistema que as respeita ser analisado por ferramentas automáticas e oferece em contrapartida segurança em seu uso.

As interfaces propostas podem ser bloqueantes, não bloqueantes, unidirecionais e bidirecionais. As interfaces bloqueantes não podem ser utilizadas em `SC_METHOD`s por conterem primitivas `wait()`. As unidirecionais bloqueantes têm definidos os métodos `put`, `get` e `peek`. As não bloqueantes `nb_get`, `nb_can_get`, `nb_peek`, `nb_can_peek`, `ok_to_peek`, `nb_put`, `nb_can_put` e `ok_to_put`, além de uma interface bidirecional bloqueante `transport`.

Outras interfaces especificadas são de canais TLM, um canal `tlm_req_rsp_channel`, `tlm_fifo` e `tlm_transport_channel`. Em [13] apresenta-se a recomendação de que o

usuário deve implementar algumas ou todas essas interfaces ou implementá-las diretamente via `sc_export`.

Os módulos no SystemC, conforme colocado no capítulo 1 podem se comunicar por interfaces (separam declaração da implementação), canais primitivos (que possuem uma semântica de atribuição `request_update` e `update`, implementam a interface `sc_prim_channel`) e canais hierárquicos (implementam `sc_channel` e podem possuir portas, módulos ou outros canais).

Uma terceira opção é implementar um módulo comum (`SC_MODULE`) que se comporte como parte do sistema e realize a comunicação entre os processos usando canais primitivos da própria linguagem.

3.4 Implementação

A rigor, não é a proposta deste trabalho criar um canal que possa ser utilizado diretamente por um IP de forma generalizada e nem mesmo propor mais uma forma de implementar canais que possam ser usados para modelar um problema específico. Para conseguir fazê-lo com algum êxito seria necessário criar uma nova interface, fazer uma classe que implementasse essa interface e forçar o projetista a usar essa classe como se fosse um barramento de seu sistema, o que pode não ser o caso depois de paralelizado o sistema. Além disso, teria de implementar uma forma de serializar seus dados respeitando um protocolo que só seria útil na distribuição da simulação, que possivelmente poderia ter um retorno negativo, por forçá-lo a utilizar um canal específico. Assim, para simplificar o uso, assumimos que um `SC_MODULE` com uma porta de entrada ou saída do tipo `sc_fifo`, comunicando um tipo de dado plano seria uma abordagem mais direta do problema. É claro que é possível definir uma interface nos moldes propostos pela especificação do TLM, apresentada rapidamente na seção anterior, definindo uma interface de comunicação formal, e uma classe de implementação mas isso não será feito. Faremos apenas uma classe de implementação que permita uma generalização posterior.

3.4.1 Recursos utilizados

Quase todos sistemas operacionais modernos implementam um mecanismo de comunicação que permite que uma mensagem de um computador seja entregue a outro. Esse mecanismo de comunicação, característico das arquiteturas MIMD pode ser implementado em diferentes níveis, conforme explicado anteriormente. É comum montar *clusters*

e redes de estações de trabalho, executando esses sistema operacionais, usando componentes de prateleira, e esses componentes, depois do advento da Internet, comumente suportam os protocolos de comunicação da família IP, suportando tanto o protocolo TCP quanto o protocolo UDP. Esses protocolos implementam, segundo o modelo OSI, a camada de transporte e tem características de uso comuns, mas comportamentos diferentes. O protocolo TCP implementa transferência de dados em ordem, retransmissão de pacotes perdidos, descarte de pacotes duplicados, controle de fluxo e provê algum controle de erros, além disso, é um protocolo orientado a conexão. O protocolo UDP, por sua vez, garante apenas que os datagramas, se entregues, estarão inteiros. O UDP não garante que os dados não estejam corrompidos, não garante a ordem e não troca nenhuma mensagem adicional por pacote enviado, por isso, é considerado um protocolo leve. Para enviar pacotes TCP ou UDP, são usados endereços IP (que é o protocolo do nível de rede) com 4 octetos e números de porta entre 0 e 65535 sendo que as portas de 1 a 1024 são reservadas.

A rede utilizada neste trabalho usa tecnologia Ethernet nas camadas física e de enlace. Numa rede ethernet, é possível ligar vários computadores entre si e isolar ou não a comunicação em *broadcast/multicast* de acordo com o dispositivo que implementa a rede. Caso a rede seja implementada em um *switch*, é pouco provável que todos os computadores escutem todas as transmissões (microsegmentação), isso diminui o número de colisões e tempo de espera até a retransmissão e aumenta a eficiência da rede, mas impede que uma abstração do barramento, onde um nó fala e todos escutam, seja implementada com facilidade. Mesmo assim, usando um *hub*, todas os pontos da rede podem escutar todas as comunicações.

Sobre a família de protocolos TCP/IPv4 e a tecnologia Ethernet de 10 megabits por segundo, é importante ainda destacar os tempos gastos para se comunicar um byte útil. Conforme colocado anteriormente, se o tempo gasto para processar uma transação de n bytes for maior que n*(tempo para comunicar um byte útil) paralelizar a transação não valerá a pena.

tamanho do quadro	quadros	*overhead*	Bytes úteis/quadro	Bytes úteis/s	Tempo/byte útil
mínimo 84 bytes	14880	60*	24	357120	2,800μs
máximo 1538 bytes	812	60*	1478	1200136	0,833μs

Tabela 3: Tempos de transmissão por carga útil. *38 bytes do ethernet, 20 do TCP e 12 do IP

Os módulos implementados para se poder aproveitar a rede TCP/IP externa têm as características comuns abaixo:

1. É responsabilidade do programador atribuir os endereços e portas para cada processo do simulador, definindo quais são as portas de comunicação que serão utilizadas e, para o emissor, que endereços IP terão processos de recepção ativos;

2. Internamente implementam `SC_THREAD`s com diversos `wait()`: `SC_METHOD`s sempre executam do início ao fim e são disparados apenas por eventos do SystemC;

3. Usam primitivas não bloqueantes do sistema operacional: usar primitivas bloqueantes do sistema operacional bloqueia todo o processo do simulador impedindo que o emissor e o receptor estejam no mesmo processo, por exemplo;

4. Usam primitivas bloqueantes do SystemC: garantir a semântica dataflow para a execução da simulação e ainda é possível encapsular as primitivas bloqueantes em primitivas não bloqueantes;

5. São implementados em pares emissor/receptor: cada parte do par constituirá uma parte de um canal unidirecional facilmente e combinando dois pares pode-se obter canais bidirecionais;

6. Conexão, desconexão e reconexão são feitas na primeira leitura da fila de entrada, com um time out explícito no código do módulo e quando um envio falha respectivamente;

7. Quebra a transação, ou seja, os dados fornecidos na `sc_fifo` automaticamente em quantos pacotes de rede forem necessários;

8. A linha de relógio força uma definição por parte do programador do passo do simulador que se não for através de um `sc_clock`, deve utilizar a função `sc_set_time_resolution()` ou terá um erro de *runtime*;

9. Usam uma estrutura de dados como

```
struct fifo_data {
  unsigned long len;    // assumed 32 bits
  unsigned char * data; // assumed 8 bit char array sized "len"
};
```

   para comunicar tanto emissor quanto receptor.

10. Durante a transmissão, são acrescentados ainda 7 bytes de cabeçalho onde 3 bytes indicam o tamanho total do pacote, limitando seu tamanho a 17 megabytes, 2 bytes com um sequencial e 2 bytes com um identificador arbitrário.

3.4.2 Comunicador UDP

O primeiro teste de implementação foi feito com o protocolo UDP com o intuito de criar uma abstração do barramento ethernet para um barramento em SystemC com um computador enviando dados para todos os computadores da rede. A declaração simplificada dos módulo de envio e recepção:

```
SC_MODULE(udpfifo_out) {
  sc_fifo_in<struct udpfifo_data *> data;
  sc_in<int> port;
  sc_in<char *> ip;
  void data_send(void);
  SC_CTOR(udpfifo_out) {
    SC_THREAD(data_send);
  }
};

SC_MODULE(udpfifo_in) {
  sc_fifo_out<struct udpfifo_data *> data;
  sc_in<int> port;
  sc_in<bool> clock;
  void write(void);
  SC_CTOR(udpfifo_in) {
    SC_THREAD(write);
    sensitive << clock;
  }
};
```

As fronteiras de comunicação para o projetista são filas `sc_fifo` e portas que indicam o endereço ip e a porta utilizados. No módulo de envio (`udpfifo_out`) há uma fila responsável por captar os dados que serão enviados para o computador endereçado nas portas de entrada do módulo e no módulo de recebimento (`udpfifo_in`), uma fila responsável por ser a fonte dos dados recebidos e um canal primitivo que diz em que porta serão recebidos os dados.

Além de depender de biblioteca de captura de pacotes de rede e do dispositivo que implementa a rede, tendo como requisito que seja um *hub*, foi observado também que

pacotes de dados maiores que 65k costumam ser descartados antes de chegar à rede, possivelmente pelo driver de dispositivo ou *kernel* do sistema operacional. Assim, para que a transmissão de pacotes maiores fosse possível, seria necessário implementar uma nova máquina de estados de comunicação, provavelmente um super set dos recursos que são utilizados pelo protocolo TCP, inviabilizando o tempo de implementação deste trabalho.

Dessa forma, optou-se por abandonar a idéia de criação de um barramento por uma implementação ponto a ponto que fosse capaz de entregar mensagens completas da origem ao destino e que permitisse clonar as mensagens para quaisquer nós que necessitem de cópia dos pacotes transmitidos.

3.4.3 Comunicador TCP

O `SC_MODULE` que implementa o comunicador TCP tem uma interface semelhante ao UDP. Abaixo a sua declaração simplificada:

```
SC_MODULE(tcpfifo_out) {
  sc_fifo_in<struct tcpfifo_data *> data;
  sc_in<bool> clock;
  sc_in<int> port;
  sc_in<char *> ip;
  void data_send(void);
  SC_CTOR(tcpfifo_out) {
    SC_THREAD(data_send);
    sensitive << clock;
  }
};


SC_MODULE(tcpfifo_in) {
  sc_fifo_out<struct tcpfifo_data *> data;
  sc_in<int> port;
  sc_in<bool> clock;
  void write(void);
  SC_CTOR(tcpfifo_in) {
    SC_THREAD(write);
    sensitive << clock;
```

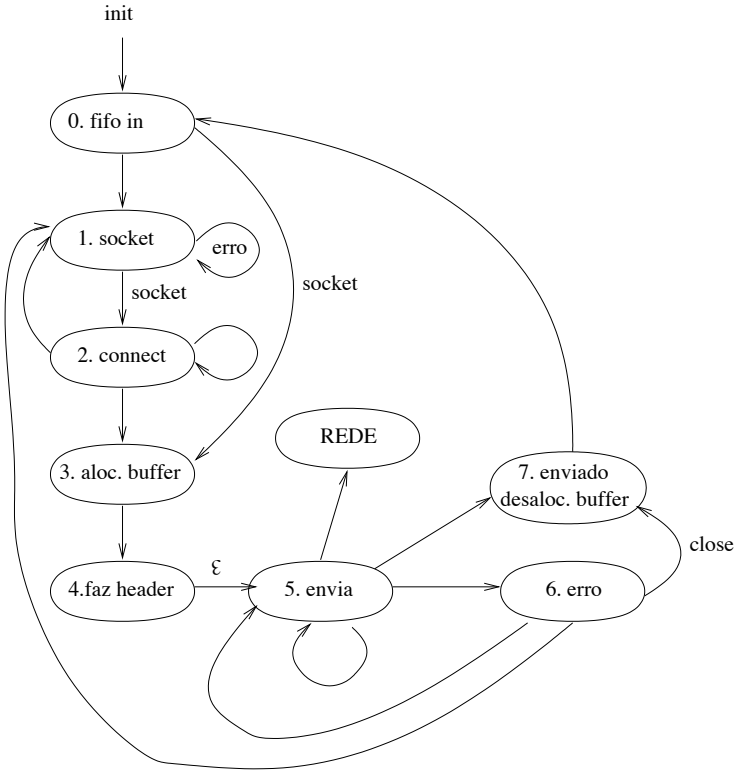

Figura 5: Máquina de estados simplificada do módulo TCPFIFO_OUT

```
  }
};
```

A diferença do comunicador TCP para o UDP está na confiança deste na ordem e garantias de entrega do protocolo TCP. Além de ter as características comuns listadas anteriormente, neste módulo são implementadas a transmissão do dado passado para a fila do módulo, a identificação de início e fim dos dados da fila e a remontagem, a partir dos pacotes da rede, da fila original. Essa identificação e remontagem é necessária para que os dados enviados pela `sc_fifo` não sejam limitados em tamanho.

Nas figuras 6 e 5 têm-se os modelos simplificados das máquinas de estados implementadas em C que estão encapsuladas dentro dos `SC_MODULE`s. Essas máquinas de estado são responsáveis por verificar a temporização da transmissão, fazer o tratamento de erros e, principalmente, dividir e recompor os dados transmitidos pela `sc_fifo`. Ocorre que uma vez enviados, os dados são serializados no canal de comunicação TCP/IP, gerando um ônus a mais para o módulo tcpfifo_in que precisa detectar os cabeçalhos e dividir as mensagens enviadas pelo canal em mensagens que façam sentido quando repassadas para a `sc_fifo`.

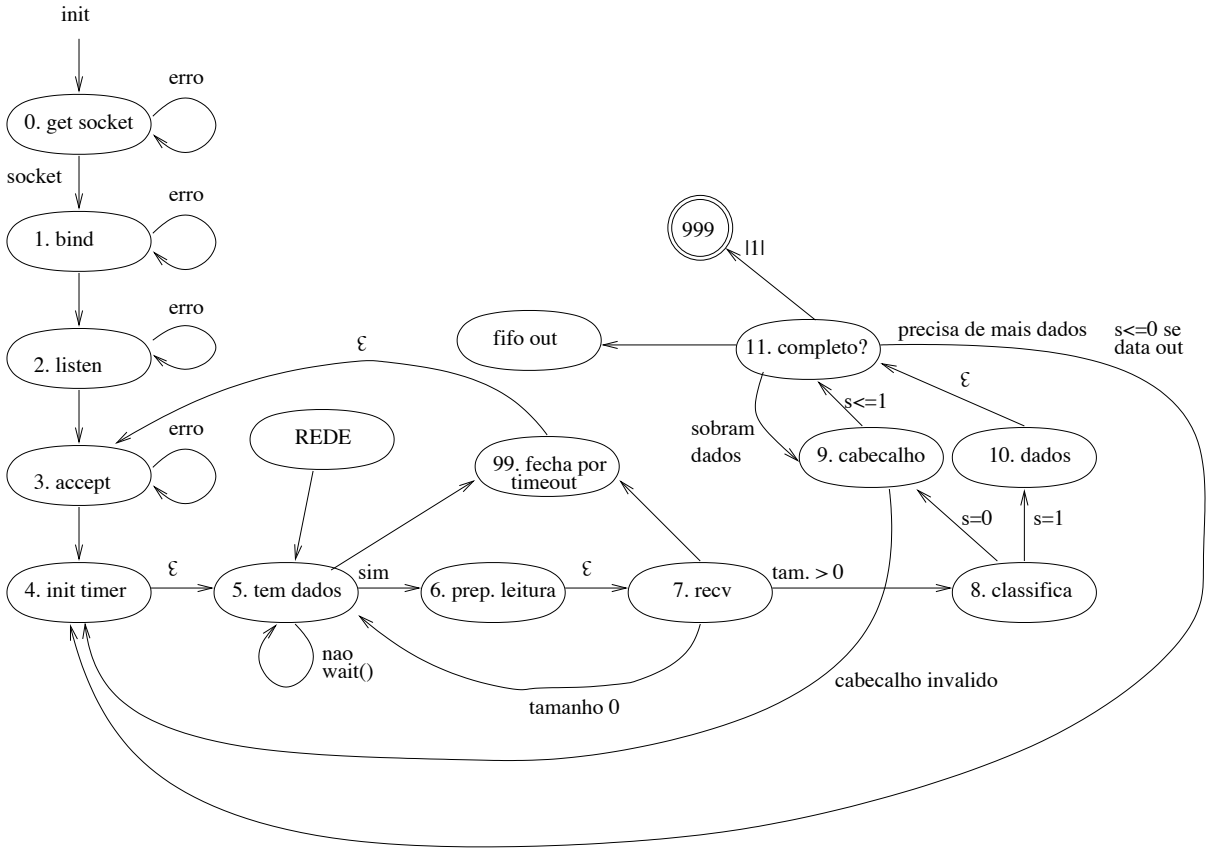

Figura 6: Máquina de estados simplificada do módulo TCPFIFO_IN

A temporização do módulo de envio do `tcpfifo_out` é muito simples comparada a do `tcpfifo_in`, apenas acrescenta-se, para cada transição da máquina de estados, uma chamada `wait()`. Isso justifica-se por se tratar de um módulo que está encapsulado e que pode estar no mesmo processo que vai receber sua transmissão. Como a recepção e a transmissão são feitos em blocos garante-se que os processos avançarão de forma progressiva. Assim, se os `wait()` não estiverem intercalados, a conexão não se fecha por conta da falha na inicialização dos *endpoints* ou mesmo por *timeout* de envio ou recepção por quaisquer dos módulos. A temporização do módulo `tcpfifo_in` precisa de ajustes para garantir que o desempenho desejado possa ser alcançado por conta da fragmentação dos dados nos diversos pacotes transmitidos. Em cada transmissão pode ser recebido mais de uma transação que precisa tomar lugar na simulação. Se não houver uma primitiva `wait()` no `tcpfifo_in` a `SC_THREAD` não permitirá que outros `SC_METHOD`s ou mesmo `SC_THREAD`s executem. Se houver `wait()`s demais, o desempenho da simulação fica comprometido, gerando *starvation* para os processos que precisam ler das filas ou disponibilidade excessiva de dados nas filas com conseqüente redução de desempenho do sistema no simulador.

3.4.4 Módulos de apoio

Para viabilizar a transmissão de um mesmo dado para mais de um nó, foram criados módulos de apoio que permitem:

1. receber 4 conexões e serializá-las em uma (demux);
2. receber uma conexão e distribuir para 4 (mux);
3. receber uma conexão e copiá-la para dois outros nós (split);
4. comprimir e descomprimir uma *stream*;
5. receber parâmetros e modificá-los no simulador em tempo de execução.

O número de quatro conexões foi escolhido em função do estudo de caso apresentado a seguir, mas podem ser codificados facilmente para um número maior ou menor, arbitrariamente.

Um exemplo de uso está ilustrado na figura 7, onde é possível ver a comunicação direta entre dois processos e uma conexão com o próprio processo em *loopback* e o possível uso pelo próprio processo que recebe o dado originalmente.

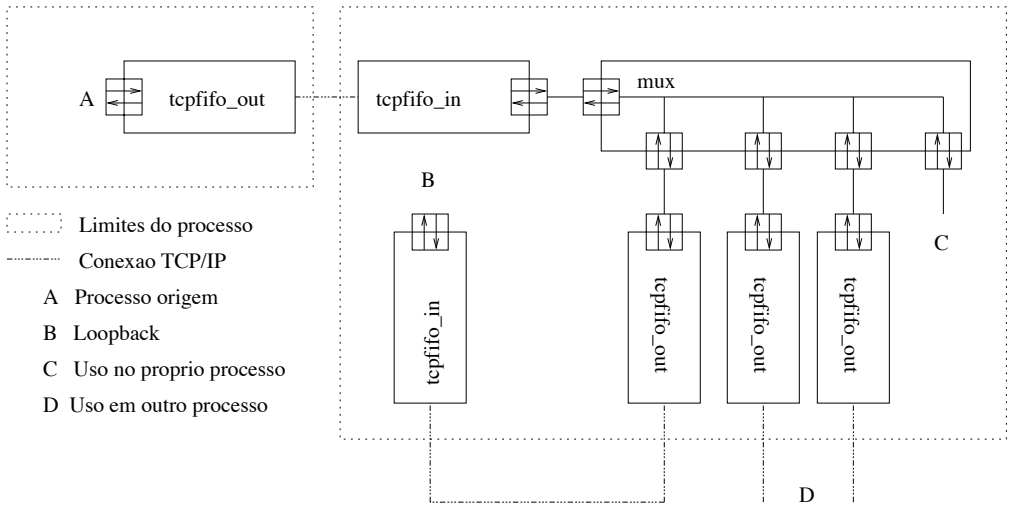

Figura 7: Ilustração de uso do comunicador TCP

Além dessas opções foram elaborados módulos que combinam outros módulos, sejam eles de apoio, comunicadores ou de dispositivo. Esses módulos foram usados nos testes de desempenho apresentados no estudo de caso no capítulo a seguir.

3.4.5 Módulos *dispositivo*

Da mesma forma em que é possível se comunicar dois processos para permitir a simulação paralela, também é possível comunicar o processo de um simulador sem um recurso específico com um simulador que dispõe desse recurso ou mesmo com um dispositivo em teste. Para validar essa idéia implementou-se um canvas de um servidor X que pode desenhar uma imagem no monitor de vídeo ilustrado na figura 8.

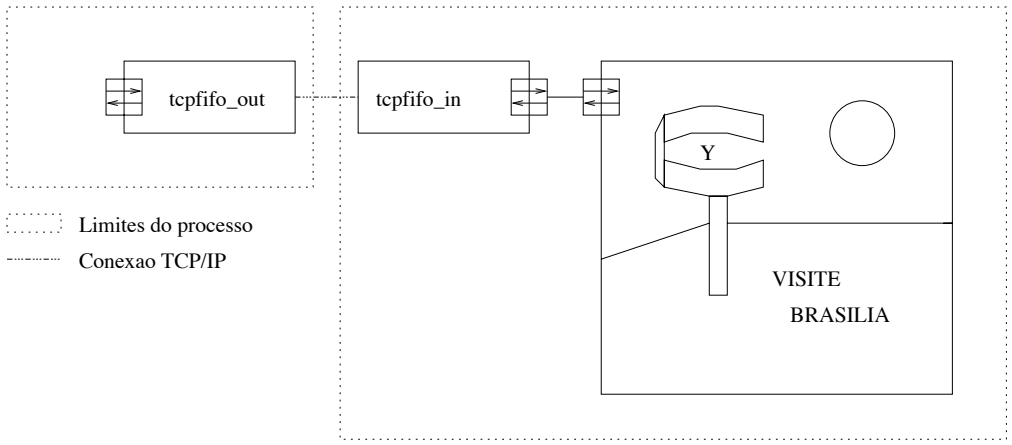

Figura 8: Ilustração de uso do comunicador TCP em uso como dispositivo, integrando simuladores

3.4.6 Módulos não-SystemC

Dadas algumas tentativas frustradas de conseguir obter imagens utilizando a interface video for Linux [19] como um IP dentro do SystemC, através do comunicador TCP é possível fazer um programa externo, sem QuickThreads [12] ou threads escalonadoras internas que injete a imagem na conexão TCP/IP estabelecida respeitando o cabeçalho adotado no comunicador TCP, viabilizando inclusive que uma câmera remota possa enviar as mensagens para processamento no simulador.

Além de enviar imagens esse módulo é responsável pela temporização das imagens em quadros por segundo. Como o escalonador do sistema operacional é imprevisível, foram adotados temporizadores capazes de fazer o processo dormir por tempo variado, maior quando um quadro era capturado com antecedência e menor quando houvesse atraso tentando manter o número de quadros dentro de uma média.

Outra possibilidade também é se utilizar kits de prototipação com FPGAs com interface Ethernet e integrá-los à simulação (figura 9)

Outro módulo auxiliar foi desenvolvido para permitir comunicar parâmetros em tempo

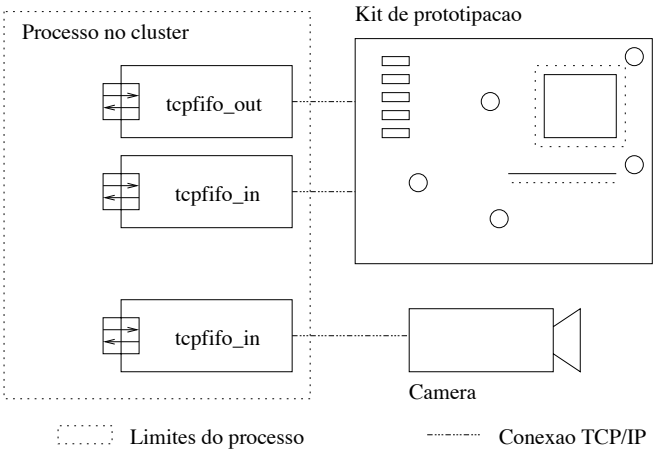

Figura 9: Ilustração de uso do comunicador TCP integrando dispositivos

de execução. Ele recebe como entrada dois valores diretamente pela linha de comando no shell do sistema operacional encaminhando esses valores para a simulação. Com isso é possível alterar valores diretamente aos sinais (`sc_signal`) do SystemC sem interromper o simulador e sem que um console para entrada de valores precise ser implementado.

4 *Estudo de caso: Modelagem e simulação de um segmentador de imagens*

Segmentadores de imagens tendem a ser de grande valia no futuro com a popularização da computação ubíqua. Com a presença de computadores em todos os lugares, é cada vez mais certo que sensores óticos precisarão reconhecer padrões no ambiente. A partir dos segmentadores, veículos equipados com câmeras especiais podem detectar animais ou obstáculos em seu caminho e diminuir a velocidade automaticamente, ou mesmo reconhecer traços de seus ocupantes e evitar partidas bruscas ou mesmo acionar acessórios do veículo, além de sistemas de vigilância automatizados, compressão de vídeo com estimativa de movimento, previsão de movimentos e outras aplicações. Todas essas aplicações tem como bloco fundamental a segmentação eficiente de imagens, além, é claro, da motivação do projeto Namitec e o SoC reconfigurável.

Assim, selecionou-se para o estudo de caso um algoritmo de segmentação de imagens que permitisse o paralelismo em nível de tarefa, onde cada nó pode processar uma imagem inteira de forma independente.

O algoritmo é baseado nos trabalhos de [20] e [8], onde é apresentado o algoritmo de segmentação por crescimento de regiões.

4.1 Arquitetura

Para executar o segmentador de imagens o problema foi decomposto, como pode ser visto na figura 10, em:

- Geração da imagem (camera)
- Segmentação (segmentador)

- Exibição de resultados (vgafifo)

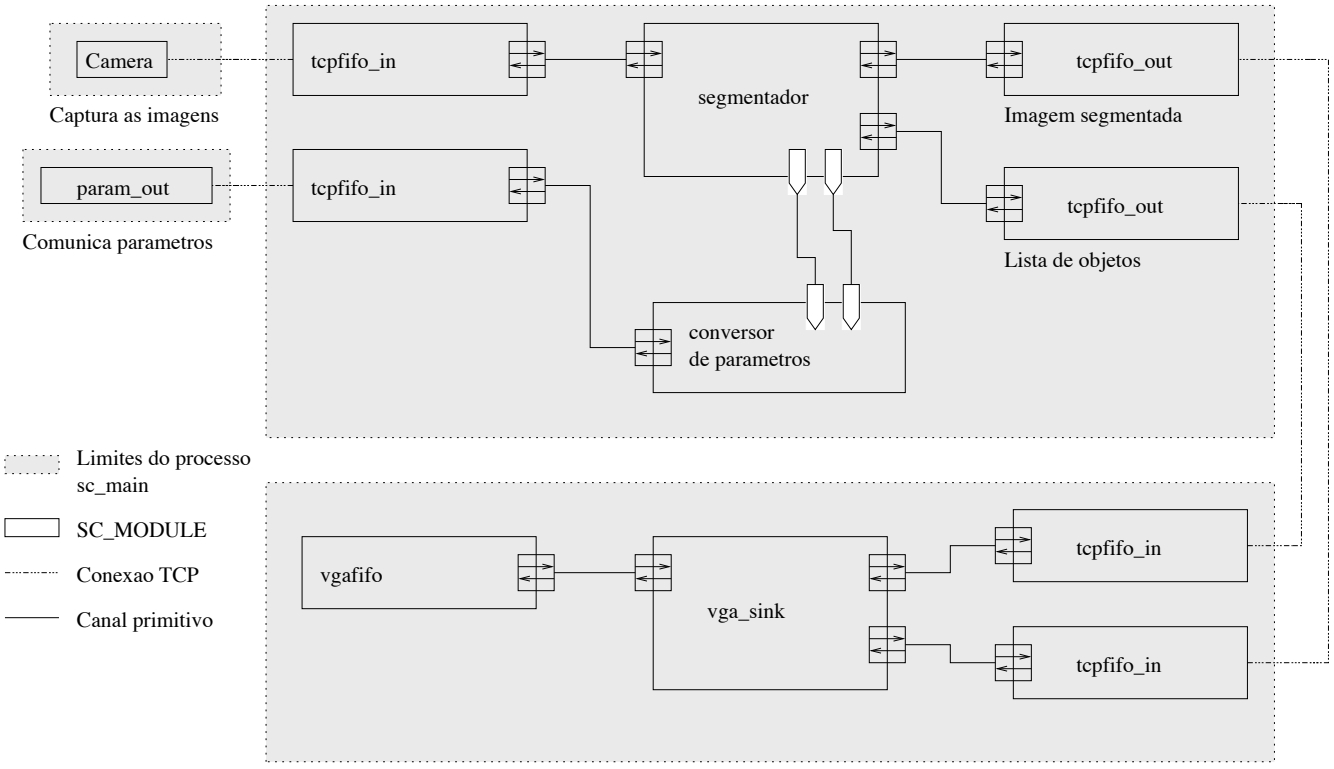

Figura 10: Arquitetura mínima do segmentador, com paralelismo entre tarefas

Assim, pelo menos essas três etapas, poderiam ser executadas em processos do simulador diferentes. Cada processo pode executar em máquinas separadas interconectadas por uma rede TCP/IP. A estrutura de dados, a que chamaremos pacote, utilizada para comunicar os módulos é a utilizada pelo comunicador TCP descrito anteriormente, com a indicação de tamanho e o conteúdo em bytes. Como é responsabilidade do projetista conectar os módulos e como a comunicação dos processos é ponto a ponto, o formato dos dados comunicados varia para cada par de módulos e são independentes. Aqui adotou-se comunicar sempre um formato padrão, com um cabeçalho limitado a 7 bytes (3 bytes tamanho e 4 bytes reservados).

Para gerar as imagens é utilizada a interface padrão de vídeo do Linux conhecida com V4L (*Video for Linux*) [19]. Essa interface permite acessar de forma padrão diversos dispositivos capazes de gerar imagens e também dispositivos de áudio e TV. Uma primeira tentativa foi feita de gerar as imagens dentro do módulo SystemC, entretanto, o *driver* de comunicação do Linux responsável por transferir as imagens para o *buffer* do processo do simulador congelou a estação de trabalho diversas vezes. Então, abordou-se o problema de gerar as imagens em um processo separado, utilizando o módulo de apoio apresentado na seção 3.4.6, completamente desacoplado do SystemC. Existem algumas particularidades

na configuração do V4L dependentes do dispositivo como alocar *buffers* fora do espaço de endereçamento físico disponível para os processos e outras dependências como o desempenho do *driver*, que pode ser muito diferente de uma configuração para outra. Um exemplo é o *driver* da Pinnacle DC10+ (PCI), capaz de gerar mais de 40 quadros por segundo e o *driver* genérico para o *chipset* OV511 sobre USB1, com a captura limitada a 15 quadros por segundo.

Capturada a imagem ela é colocada no formato *ppm* [11], que não tem perdas e permite ler os valores dos *pixels* diretamente em ordem sequencial e sem bytes intermediários a não ser o diminuto cabeçalho. Embora esse formato não seja adequado para tráfego em rede, é muito prático para processar.

Para a segmentação das imagens geradas pelo dispositivo de vídeo, foi implementado o algoritmo descrito em [17], explicado na seção adiante.

Como saída do segmentador é gerada uma lista de objetos encontrados contendo as características: área, coordenadas esquerda, direita, superior e inferior, cor (atribuída arbitrariamente), cor dos *pixels* das extremidades do objeto representada em 16 bits (por componente: 5 bits/vermelho, 6 bits/verde e 5 bits/azul) e a soma das características como palavra de verificação. Outras informações provenientes do algoritmo de segmentação também poderiam ser extraídas como o centróide do objeto, cor média, *pixel* de onde partiu a segmentação, extremos e conjuntos de *pixel* do contorno, por exemplo.

Da lista de objetos gerada como saída do segmentador alimenta-se o terceiro módulo responsável por exibir, de forma primitiva, os objetos encontrados. Esse módulo de exibição foi implementado no sistema de janela X de forma a abrir uma janela e nela exibir o conteúdo de cada imagem no formato ppm recebida na nessa mesma janela e, além disso, receber a lista de objetos e criar quadrados ao redor dos objetos encontrados.

4.2 Descrição do algoritmo

O algoritmo [17], originalmente concebido para execução paralela por *pixel* em hardware, permite quatro estados por *pixel*: auto excitável, não excitado, excitado e inibido. Também são definidas seis etapas: inicialização, detecção de um *pixel* auto excitável (*pixel* líder), auto excitação do *pixel* líder, detecção dos *pixels* excitáveis dependentes, excitação dos *pixels* dependentes (crescimento de região) e inibição dos *pixels* excitados logo depois de cada processo de crescimento de região. O algoritmo segue repetindo a partir da detecção de um novo *pixel* líder até encontrar o final da imagem.

Na fase de inicialização são calculados os pesos de conexão dos *pixels* a partir da informação de luminância ou dos componentes RGB da imagem de acordo com a equação abaixo, usando os valores $I_{max}(R)$ que é o maior valor da componente R da imagem, e os valores da componente R do *pixel* ij e kl, onde $I_{ij}(R)$ é o *pixel* central e $I_{kl}(R)$ são os *pixels* vizinhos de I_{ij}, ou seja, I_{i-1j-1}, I_{ij-1}, I_{i+1j-1}, I_{i-1j}, I_{i+1j}, I_{i-1j+1}, $I_{,ij+1}$ e I_{i+1j+1}, para cada *pixel* ij da imagem. A equação e o valor I_{max} tem correspondentes em cada um dos componentes da imagem, isto é, R (dado no exemplo), G e B.

$$W_{ij;kl} = \frac{I_{max}(R)}{1+ \mid I_{ij}(R) - I_{kl}(R) \mid} \tag{4.1}$$

Caso a imagem tenha apenas um componente, $W_{ij;kl}$ pode ser usado para segmentação. De outra forma,

$$W_{ij;kl} = min\{W_{ij;kl}(R), W_{ij;kl}(G), W_{ij;kl}(B)\} \tag{4.2}$$

e logo é feita a obtenção do componente W_{ij} somando-se os valores de $W_{ij;kl}$ para kl igual a $i-1j-1, ij-1, i+1j-1, i-1j, i+1j, i-1j+1, ij+1$ e $i+1j+1$. Uma vez obtidos os valores de W_{ij} pode-se determinar os *pixels* líder marcado-os sempre que forem maiores que um valor ϕ_p. Os autores do algoritmo destacam que essa é uma característica importante e responsável por suprimir *pixels* com ruído impedindo que se tornem segmentos separados, além de garantir a robustez contra variações de contraste já que são levadas em conta apenas as diferenças entre os *pixels* vizinhos. Observe que quanto maior o corte ϕ_p, menos *pixels* alcançarão a soma dos valores mínima e conseqüentemente menos objetos serão detectados. Se o corte ϕ_p for muito baixo, muitos detalhes serão considerados objeto e logo teremos uma quantidade muito grande de objetos pouco significativos, fenômeno conhecido como supersegmentação.

Cada objeto é separado conforme faça fronteira com um *pixel* líder (auto excitado) e de acordo os valores mínimos de cada componente RGB de seus *pixels* vizinhos ($W_{ij;kl}$) e de um ponto de corte de excitação (ϕ_z), que definirá que *pixels* fazem parte do objeto. Tem-se então a fase principal de segmentação. A imagem é varrida em busca do *pixel* líder da esquerda para a direita da primeira até a última linha (em [17]) essa busca é feita através de um *token* que garantirá que toda a imagem será varrida e que apenas um objeto estará crescendo por vez). Selecionado o primeiro *pixel* líder, ele é excitado e a seguir o crescimento da região é realizado em passos. Cada *pixel* vizinho ao *pixel* excitado tem seu peso de conexão $W_{ij;kl}$ comparado com um ponto de corte ϕ_z. Na implementação de [17] todos os *pixels* vizinhos a um *pixel* excitado são excitados em paralelo em um passo de crescimento de região. Esses passos se sucedem, a partir dos *pixels* excitados no

passo anterior, realizando passos de excitação semelhantes até que não haja mais *pixel* excitável, quando então, a região é etiquetada e os *pixels* são inibidos. Depois disso busca-se o segundo *pixel* líder e não inibido para que o processo se repita até que não se encontre mais *pixels* líder.

Os *pixels* restantes ou são ruídos ou segmentos sem *pixel* líder que podem ser tratados em um segundo passo que os agrupe por luminância ou cor.

4.3 Implementação do algoritmo

A implementação, em feita inicialmente em linguagem C, foi encapsulada segundo a sintaxe do SystemC em um `SC_MODULE` com interface transcrita abaixo:

```
SC_MODULE(segfifo) {
  sc_fifo_in <struct tcpfifo_data*> data_in;
  sc_fifo_out <struct tcpfifo_data*> segments;
  sc_fifo_out <struct tcpfifo_data*> oobject_list;
  sc_in <int> LeaderPixelt; // phi p
  sc_in <int> ExcitePixelt; // phi z
  sc_in <bool> clock;
  void segmenter(void);
  SC_CTOR(segfifo) {
    SC_THREAD(segmenter);
    sensitive << clock;
  } // SC_CTOR
};
```

De acordo com a descrição do algoritmo, os estados dos *pixels* ditos auto-excitável, não excitado, excitado e inibido foram mapeados em uma estrutura de dados que guarda apenas os estados excitado e inibido. A determinação dos *pixels* auto-excitáveis é feita a partir de busca sequencial nos *pixels* da imagem da esquerda para a direita a partir da primeira linha, saltando alguns *pixels* já inibidos e/ou excitados. A inibição é vista quando um objeto é atribuído ao *pixel*.

Os valores $W_{ij;kl}$ são armazenados por *pixel* em uma estrutura de dados chamada *complex_pixel*, que aparece abaixo, com até dois ponteiros para cada *pixel* vizinho, os da direita apontando para o *pixel* vizinho propriamente e o outro com o valor $W_{ij;kl}$,

deixando espaço para otimizações no uso da memória. Para os *pixels* das bordas são acrescentados *pixels* nulos nas diagonais, acima, abaixo, a direita ou a esquerda. Pixels nulos correspondem a *pixels* com valor 0 nas componentes RGB e já inibidos.

```
typedef struct complex_pixel { // 40x
  int W; // Wij
  int x,y;
  char excited;
  unsigned char* rgb; // ^pixel
  unsigned char Wne, Wn, Wnw; // Wij,kl
  unsigned char We,      Ww;
  unsigned char Wse, Ws, Wsw;
  struct object * self; // inibidor
  struct complex_pixel *next; // independe de linha
  struct complex_pixel *next_to_excite; // atalho
  struct complex_pixel *NWpixel;
  struct complex_pixel *Wpixel;
  struct complex_pixel *SWpixel;
} s_cpixel;
```

Durante a busca por *pixels* excitados, cada *pixel* vizinho e não excitado do novo *pixel* excitado é colocado numa pilha para que possa ser avaliado e eventualmente iniciar um novo passo de crescimento.

O algoritmo foi implementado nos passos:

1. ler a fila de entrada *data_ in* e validar a imagem;
2. percorrer a imagem, *pixel* a *pixel*, buscando os maiores valores (I_{max}) de cada componente R, G e B de *pixel* da imagem;
3. acrescentar uma borda nula ao redor da imagem, inicializando as estruturas respectivas ao que seriam os *pixels* dessa borda;
4. inicializar as estruturas de percorrimento da imagem;
5. calcular as componentes $W_{ij;kl}$ para cada *pixel* com kl apontando para as direções norte ($W_{ij;i-1j}$), sul ($W_{ij;i+1j}$), leste ($W_{ij;ij-1}$), oeste ($W_{ij;ij+1}$), nordeste ($W_{ij;i-1j-1}$),

noroeste ($W_{ij;i-1j+1}$), sudeste ($W_{ij;i+1j-1}$) e sudoeste ($W_{ij;i+1j+1}$) para as componentes R, G e B aplicar a equação 4.2;

6. calcular a componente W_{ij} por *pixel* que é a soma de $W_{ij;kl}$ em todas as direções;

7. definir os valores de *pixel* líder ϕ_p e de corte ϕ_z para *pixel* do mesmo objeto. Utilizou-se um ajuste automático baseado em uma lista de Ws para identificar a faixa de valores de W mais frequente (mais objetos). A eficiência desse algoritmo está na escolha correta desses parâmetros;

8. Para cada *pixel* não excitado da imagem, buscar por *pixels* líder verificando se o valor do W_{ij} é suficiente para que ele seja um *pixel* líder ($W_{ij} > \phi_p$) e quando encontrá-lo fazer os passos de crescimento de região:

   - excitar o *pixel* ij e empilhar os 8 *pixels* vizinhos de W_{ij} sempre que $W_{ij;kl} > \phi_z$ e que eles não estejam excitados;
   - repetir o processo enquanto houver elementos na pilha;
   - quando a pilha estiver vazia, acrescentar objeto à lista de objetos encontrados;

9. selecionar os objetos com área maior que um valor determinado;

10. escrever a lista de objetos na fila de saída e a imagem com os segmentos coloridos.

As figuras 11, 12 ilustram uma cena típica de dormitório com objetos identificados através do algoritmo. A imagem 11 é a cena original, a imagem 12 destaca os objetos da forma como o segmentador os vê e onde os *pixels* em regiões sem *pixel* líder também identificados.

Figura 11: Cena de dormitório

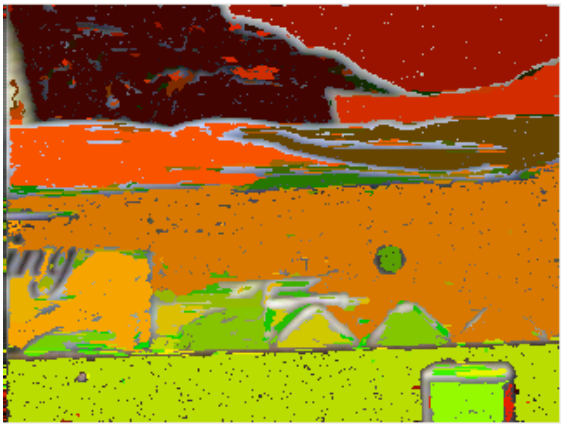

Figura 12: Cena de dormitório em segmentos, exemplificando os segmentos da imagem

4.3.1 Resultados

Completando a arquitetura que executa o algoritmo, foram utilizados os módulos adicionais já descritos anteriormente: o que alimentam o segmentador com imagens obtidas de uma fonte em tempo real, como uma câmera web e o módulo que exibe as imagens ou respectivos segmentos. Esse último foi adaptado também para obter uma medida de desempenho da rede sem o segmentador.

As configurações dos equipamentos utilizaram foram:

- (D)esktop: AthlonXP 2600+ 1.91GHz (Barton), 512Mb RAM, placa mãe A7N8X-X, rede nVidia nForce Network Controller (onboard FastEthernet), Pinnacle Studio DC10+ PCI;

- (N)otebook: Turion64 Mobile ML37 até 2GHz, 1Gb RAM, HP Pavilon série dv5000, rede Realtek família RTL8139 (PCI FastEthernet).

O sistema operacional utilizado foi o Linux, distribuição Ubuntu 6.06 LTS com kernel versão 2.6-15-27.386 da distribuição sem ajustes para a estação N e, para a estação D o parâmetro mem=490M foi utilizado na carga do *kernel* a fim de disponibilizar *buffer* de tamanho suficiente para a capturadora de vídeo.

Para conectar as duas máquinas foram utilizadas as portas IEEE 802.3u FastEthernet de um roteador Linksys WRT54GC LinkSys (firmware 1.02.5).

Nos dois primeiros cenários foram utilizadas imagens de tamanho 128 x 96 *pixels* com componentes R, G e B, transmitidas no formato ppm que é constituído por um identificador "P6", largura, altura, número de bits por *pixel* e dados da imagem, resultando em transações de aproximadamente 37kbytes por imagem. De acordo com os tempos apre-

sentados anteriormente para o barramento ethernet, para imagens deste tamanho temos um limite superior teórico de 32 quadros por segundo, calculados assumindo que todos os pacotes TCP/IP seguem com o tamanho máximo, sem contar com o *overhead* devido a perda de pacotes e pacotes de sincronização na rede, e que a transmissão de cada byte leva 0,833μs. Veja que a transmissão de uma única imagem toma cerca de 31ms.

A figura 13 mostra o cenário utilizado para tentar determinar a vazão máxima da rede, envolveu colocar o módulo de exibição nas duas estações de trabalho (D, N) utilizadas e enviar seqüências de imagens limitadas a 30 quadros por segundo nas interfaces de *loopback*, apenas na estação que possui a camera, na interface *ethernet* da máquina com a câmera e para a interface *ethernet* na estação N.

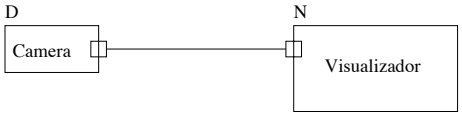

Figura 13: Cenário 1: interação entre um processo e um simulador

Interface	Estação origem	Estação destino	Quadros (média)	Quadros (pico)
loopback	D	D	23.81	27
ethernet	D	D	24.39	29
ethernet	D	N	24.39	26

Tabela 4: Resultados do cenário envolvendo o módulo de exibição e de transmissão de imagens

Para este cenário foi observado que a interface de loopback provida pelo kernel do Linux utilizado, ao contrário do esperado, teve um número de quadros de pico menor que quando testada contra a interface de rede.

O cenário 2 (ver figura 14), utiliza um dos canais de distribuição do multiplexador (um de quatro disponíveis) para alimentar a visualização, sem passar por um segmentador, e usando os demais para alimentar segmentadores. As imagens são fornecidas pelo módulo de câmera, entregues ao multiplexador que fica na mesma estação da câmera (D) e entrega de 3 a 0 filas de imagens para os segmentadores da própria estação ou da estação N.

Esse cenário foi concebido dessa forma para tentar diminuir o número de imagens comunicadas que, por utilizarem um formato inadequado, demonstraram comprometer o tempo de execução. Assim, os segmentos apresentados sobre a imagem enviada refletiam aproximadamente os objetos exibidos pela quarta imagem capturada de cada seqüência de 4 quadros. Para executar esse cenário o segmentador calibrado fixando-se os parâmetros ϕ_p e ϕ_z pulando a detecção automática de parâmetros de *pixel* líder e de corte para objeto na

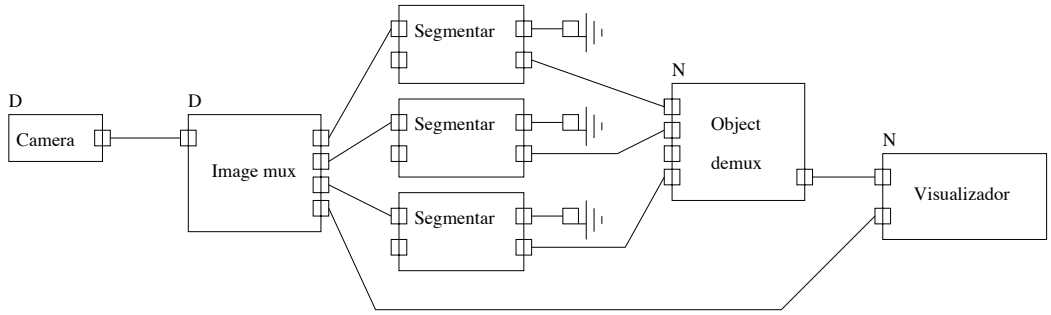

Figura 14: Cenário 2: interação entre 7 simuladores

tentativa de manter o número de objetos aproximadamente constante e conseqüentemente com desempenho do algoritmo de segmentação também constante.

Os módulos de apoio que permitem, nesse cenário, a paralelização trivial do algoritmo distribuindo as imagens entre os possíveis nós arranjando três segmentadores dois a dois, usando um total de 7 processos do simulador independentes. A saída do multiplexador é selecionada usando o algoritmo *round-robin* para cada elemento colocado na fila de entrada e o demultiplexador que escolhia as entradas usando a política *first in first out*. Através dos contadores nos módulos de apoio foi possível verificar a vazão do sistema em quadros por segundo e quadros de pico, medidos no demultiplexador.

Segmentadores na estação D	Segmentadores na estação N	Quadros por segundo (média)	Quadros por segundo (pico)
3	0	8.45, 9.05	10,11
2	1	8.65, 9.41	10,11
1	2	8.98, 9.50	10,11
0	3	9.88, 10.03	11,12

Tabela 5: Resultados do cenário envolvendo a arquitetura completa apontando mínimos e máximos obtidos.

Num terceiro cenário, utilizando uma rede FastEthernet, de 100Mbits, e imagens de 230k (320x240) verificou-se o desempenho do segmentador em um módulo combinado (Tabela 6) de recepção da imagem, segmentação e exibição, com e sem compressão pela biblioteca Zlib [21]. As imagens são sempre geradas na estação D e foram transmitidas primeiro para a própria estação D e no segundo caso da estação D para a estação N. Para este cenário tem-se apenas 2 processos por vez, o módulo combinado SystemC e o processo não SystemC que gera as imagens.

Um quarto cenário (Tabela 7), também testado numa rede FastEthernet, de um lado temos um processo que tem um modelo em SystemC que comunica imagens como conteúdo de suas transações que

Quadros por segundo	Enviados	(média)	(pico)
D	5	4.76	5
N	6	5.68	7
N (z)	6	5.73	7

Tabela 6: Resultados para segmentação e exibição em um único processo

- recebe duas imagens de tamanho 320x240 (tcpfifo_in);
- segmenta uma gerando uma imagem segmentada e uma lista de objetos (segfifo);
- encaminha a imagem segmentada e lista de objetos (tcpfifo_out);
- encaminha a outra (mux, tcpfifo_out).

e de outro lado um outro processo que implementa as atividades a seguir em módulos SystemC:

- recebe uma imagem segmentada com sua respectiva lista de objetos (tcpfifo_in);
- recebe uma imagem não segmentada (tcpfifo_in);
- gera uma imagem segmentada e uma lista de objetos a partir da imagem não segmentada recebida (seg_fifo);
- exibe a imagem segmentada gerada e a recebida com seus respectivos objetos uma após a outra (mux e vga).

além do processo que gera as imagens.

Para esse cenário foram testados o uso de imagens não comprimidas e de imagens comprimidas usando a biblioteca Zlib. Para o caso das imagens comprimidas, elas são geradas, comprimidas, comunicadas, descomprimidas e processadas ou encaminhadas comprimidas para outro processo do SystemC. Foram obtidos os resultados abaixo, que reiteram um pequeno ganho ao se paralelizar a simulação e mostram uma perda em virtude da sobrecarga imposta pela compressão:

4.3.2 Críticas da implementação

Abaixo alguns pontos que consideramos críticos nessa implementação:

Estações	Tamanho comprimido	Tamanho regular	Quadros enviados	quadros (média)	quadros (pico)
D/N	-	230.4k	7	6.12	8
D/N(z)	175k	230.4k	5	5.46	7

Tabela 7: Resultados para segmentação e exibição em dois processos em computadores distintos

- o formato de imagens usado para trafegar na rede é ruim, o que compromete o desempenho total do segmentador. Melhor seria utilizar um outro padrão de compactação ou mesmo compressão sem perdas;

- o algoritmo de seleção do demultiplexador faz com que a ordem dos quadros se perca, o que é inadequado mas não compromete, inicialmente, o segmentador mas inviabiliza o uso do mesmo para detecção de movimento;

- um problema foi observado no módulo que exibe os quadros e seus objetos pois para cada três listas de objetos temos apenas uma imagem e ambos chegam em tempos diferentes; ao final não havia imagens disponíveis para exibir sob as listas de objetos que se sobrepuseram, o que foi evitado nos cenários posteriores;

- a paralelização do processamento foi precipitada; ainda seria possível dividir a imagem em duas ou quatro partes com uma pequena sobreposição e acrescentar uma etapa de consolidação de objetos com pontos comuns na área de sobreposição em um único objeto e assim quebrar o efeito *pipeline* da arquitetura utilizada;

- usando um algoritmo de compressão e descompressão como módulos extras entre os processos não é uma opção trivial que resulte em diminuir o tempo gasto com comunicação, os ganhos verificados na compressão foi de 20% em relação ao tamanho original em detrimento do tempo de processamento e da latência final do sistema; o tempo de processamento aumenta em certa de 12%;

- primitivas de sincronização não foram abordadas apesar de no uso prático do algoritmo para estimativa de movimento, por exemplo, a ordem dos quadros se torna importante;

- a rede, embora utilize componentes fast ethernet funcionou em alguns casos como uma rede ethernet de 10mbits, o que foi constatado após várias tentativas com resultados incoerentes. Esse problema foi resolvido desligando-se a autonegociação de velocidade das interfaces de rede à véspera da conclusão do trabalho.

5 Conclusão

Relembrando que o objetivo deste trabalho era de oferecer condições a execução de um modelo utilizando recursos em nós diferentes, funcionando em modelos SystemC em alto nível os resultados são consideráveis. A partir do estudo de caso apresentado vemos que foram utilizados nós, no caso estações de trabalho, diferentes em arquitetura e com dispositivos diferentes. A estação D utiliza um processador de 32bits e a estação N é de 64bits funcionando com um sistema operacional de 32bits. A estação N, enquanto utilizada apenas para receber os objetos e imagens a cada rodada e compor a exibição final, revelou o desempenho do sistema menor que quando comparada a situação oposta, onde foi utilizada a estação N como segmentadora para 3 *streams*.

Percebe-se que, embora o ganho da paralelização para este caso tenha sido pequeno (cerca de 10%), a utilização componentes externos ao simulador para obter ganhos na execução da simulação foi possível. É interessante observar que para o caso do segmentador revela-se o quanto a modelagem de alto nível pode ser útil. A escolha de uma rede de 10Mbits/s pode representar, em tempo execução, um gargalo devido sua baixa capacidade de comunicação. A utilização uma rede de 100Mbits/s ou 1Gbits/s minimizaria as eventuais perdas em desempenho na simulação devido ao custo da comunicação.

Ao longo do desenvolvimento desse trabalho ficou claro que a temporização da recepção de dados da rede e sua injeção no simulador poderia ser melhor trabalhada. Deve-se lembrar que cada processo do simulador SystemC executa seus métodos e *threads* de forma sequencial no tempo de relógio até que seja possível avançar no tempo do simulador. Para cada transação que é colocada dentro do processo do simulador são iniciadas outras tarefas, que podem ser demoradas, e que eventualmente ocasionam perda de desempenho em virtude do protocolo de reconexão ou ainda, no outro extremo ter o caso de monopolização do processo pela *thread* de comunicação fornecendo dados a uma taxa que o simulador não consegue consumir. Tudo isso devido a impossibilidade de recepção e execução realmente paralelas, ou pelo menos preemptiva, dentro do mesmo processo do SystemC (versão 2.0.1).

Como proposta de trabalhos futuros propõe-se avaliar outras interfaces de comunicação para arquiteturas paralelas como o PVM e o MPI e a utilização de implementações dedicadas do segmentador em protótipos reconfiguráveis, em tempo de simulação, para aproximar ainda mais o ambiente de desenvolvimento do ambiente de produção do dispositivo. Outras possibilidades de implementação giram em torno de introduzir balanceamento entre os `SC_THREAD`s de comunicação e os do sistema sendo simulado para diminuir o custo de sincronização, bem como explorar diversos algoritmos de balanceamento de carga para os processos (conjunto de `SC_THREAD`s balanceadas) e o uso de nós heterogêneos.

Referências

[1] Janick Bergeron. *Writing Testbenches: Functional verification of HDL models - Second Edition*, pages 1–512. Kluwer Academic Publishers, 2003.

[2] Brazilip. http://www.brazilip.org.br/, 2007. Acessado em 14 de maio de 2007.

[3] L. Cai, D. Gajski, P. Kritzinger, and M. Olivares. Top-down system level design methodology using specc, vcc and systemc. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 1137, Washington, DC, USA, 2002. IEEE Computer Society.

[4] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A hardware/software approach*, pages 1–1025. Morgan Kaufmann Publishers, Inc., 1999.

[5] Andre' DeHon. Reconfigurable architectures for general-purpose computing. Technical Report AITR-1586, MIT, 1996. Acessado em 14 de maio de 2007.

[6] M. Flynn. Some computer organizations and their effectiveness. *Transactions on Computing*, C-21:948–960, 1972.

[7] Thorsten Grotker, Stan Liao, Grant Martin, and Suart Swan. *System Design with SystemC*, pages 1–209. Kluwer Academic Publishers, 2002.

[8] Takashi Morimoto, Osamu Kiriyama, Youmei Harada, Hidekazu Adachi, Tetsushi Koide, and Hans Jürgen Mattausch. Object tracking in video pictures based on image segmentation and pattern matching. *2005 IEEE International Symposium on Circuits and Systems*, pages 3215–3218, 2005.

[9] Tecnologias de micro e nanoeletrônica para sistemas integrados inteligentes - namitec. http://www.ccs.unicamp.br/namitec/, 2005. Acessado em 14 de maio de 2007.

[10] Posix - portable operating system interface. http://standards.ieee.org/regauth/posix/. Acessado em 14 de maio de 2007.

[11] Formato gráfico ppm. http://netpbm.sourceforge.net/doc/ppm.html, 2003. Acessado em 14 de maio de 2007.

[12] Compressed report about quickthreads framework. ftp://ftp. cs. washington. edu /tr /1993 /05 /UW-CSE-93-05-06.PS.Z, 2007. Acessado em 14 de maio de 2007.

[13] Adam Rose, Stuart Swam, John Pierce, and Jean-Michel Ferandez. Transaction level modeling in systemc. 2005. Acessado em 14 de maio de 2007.

[14] Ian Summerville. *Software engineering*, pages 1–742. Addison-Wesley, 1996.

[15] Synopsys. http://www.synopsys.com/, 2007. Acessado em 14 de maio de 2007.

[16] Systemc. http://www.systemc.org/. Acessado em 14 de maio de 2007.

[17] T. Koide T. Morimoto, Y. Harada and H. J. Mattausch. Pixel-parallel digital cmos implementation of image segmentation by region growing. *Proceedings of the 2006 Conference on Asia South Pacific Design Automation: ASP-DAC 2006, Yokohama, Japan, January 24-27, 2006*, pages 176–181, 2006.

[18] Laboratório de arquiteturas dedicadas. http://lad.dsc.ufcg.edu.br/pmwiki.php?n=Lad.Ip, 2005. Acessado em 14 de maio de 2007.

[19] The video4linux wiki. http://linuxtv.org/v4lwiki, 2003. Acessado em 14 de maio de 2007.

[20] K. Yamaoka, T. Morimoto, H. Adachi, T. Koide, and H. J. Mattausch. Image segmentation and pattern matching based fpga/asic implementation architecture of real-time object tracking. *Proceedings of the 2006 Conference on Asia South Pacific Design Automation: ASP-DAC 2006, Yokohama, Japan, January 24-27, 2006*, pages 176–181, 2006.

[21] Zlib. http://www.gzip.org/zlib, 2007. Acessado em 18 de maio de 2007.