Modelagem e Simulação de Sistemas Dinamicamente Reconfiguráveis em Granularidades Diversas

Tese de Doutorado

Alisson Vasconcelos de Brito
Candidato

Prof. Elmar Uwe Kurt Melcher Dr.
Orientador

Campina Grande, Paraíba, Brasil
Março de 2008

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

B862m
2008 Brito, Alisson Vasconcelos de.
Modelagem e simulação de sistemas dinamicamente reconfiguráveis em granularidades diversas / Alisson Vasconcelos de Brito. — Campina Grande, 2008.
113f. : il.

Tese (Doutorado em Engenharia Elétrica) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
Referências.
Orientador: Prof. Dr. Elmar Uwe Kurt Melcher.

1. Sistemas Reconfiguráveis. 2. Simulação. 3. Reconfiguração Dinâmica. I. Título.

CDU004.274(043)

Modelagem e Simulação de Sistemas Dinamicamente Reconfiguráveis em Granularidades Diversas

Alisson Vasconcelos de Brito

Tese de Doutorado apresentada em março de 2008

Elmar Uwe Kurt Melcher, Dr.
Orientador

Raimundo Carlos Silvério Freire, Dr.
Presidente da Comissão

Edna Natividade da Silva Barros, Dr.
Componente da Banca

Marius Strum, Dr.
Componente da Banca

Guido Costa Souza de Araújo, Dr.
Componente da Banca

Rodolfo Jardim de Azevedo, Dr.
Componente da Banca

Campina Grande, Paraíba, Brasil
Fevereiro de 2008

“A imaginação é mais
importante que a ciência,
porque a ciência é limitada,
ao passo que a imaginação
abrange o mundo inteiro.”
Albert Einstein

Dedicatória

Dedico este trabalho à minha
esposa, Ana Laura, eterna
companheira.

Agradecimentos

Agradeço primeiramente a Deus, pela saúde, serenidade e auxílio nos momentos difíceis.

Ao meu orientador, Prof. Elmar Melcher, pelo apoio técnico e moral, sempre nas horas exatas.

Aos meus pais, Sérgio e Graça, pela educação que me deram e ainda dão e pelo apoio emocional sempre bem-vindo.

Aos meus irmãos, Gracielle e Sérgio, pelo amor e carinho que sempre me deram.

À minha esposa, Ana Laura, que me inspirou e incentivou, abrindo mão de suas ambições pessoais em prol das minhas. Sou eternamente grato por isso.

Aos meus sogros, Hélio e Lêda Rosas e aos meus cunhados, Wilson e Rodolfo, pelo acolhimento e apoio moral e técnico.

Aos meus amigos que me acompanharam desde os tempos de graduação, Alexandre, Petrônio, Eloi, Osório e todos os demais.

Aos professores Jürgen Becker e Michael Hübner que me acolheram muito bem na Alemanha e me ajudaram a seguir os caminhos certos nesse trabalho.

Aos meus amigos alemães Matthias Kühnle e Florian Thoma que trabalharam comigo e acabaram me ensinando bastante.

E aos demais amigos e familiares meu fraterno obrigado!

Resumo

Uma metodologia inovativa para modelagem e simulação de sistemas parcial e dinamicamente reconfiguráveis é apresentada neste trabalho. Como a reconfiguração dinâmica pode ser vista como o processo de remoção e inserção de módulos num sistema, a metodologia apresentada é baseada no bloqueio da execução de módulos não configurados durante a simulação, sem que o restante do sistema pare sua atividade normal. Uma vez provida a possibilidade de remover, inserir e trocar módulos durante a simulação, todos sistemas modelados utilizando este simulador podem se beneficiar das reconfigurações dinâmicas.

Com o objetivo de provar os conceitos definidos, modificações no núcleo do SystemC foram realizadas, adicionando novas instruções para desconfigurar e reconfigurar módulos em tempo de simulação, permitindo que o simulador seja utilizado tanto em nível de transações (TLM), como no nível de transferência entre registradores (RTL). No nível TLM ele permite a modelagem de sistemas de hardware num nível maior de abstração, assim como sua integração com softwares embarcados, enquanto que no nível RTL, o comportamento dinâmico do sistema pode ser observado no nível de sinais. Ao mesmo tempo em que todos os níveis de abstração podem ser simulados, todas possíveis granularidade podem ser consideradas. De forma geral, todo sistema capaz de ser simulado utilizando SystemC pode também ter seu comportamento modificado em tempo de execução. O conjunto de instruções desenvolvidas reduz o tempo de ciclo de projeto. Comparado a estratégias tradicionais, informações sobre o comportamento adaptativo e dinâmico dos sistemas estarão disponíveis nos estágios mais iniciais do desenvolvimento.

Três aplicações diferentes foram desenvolvidas utilizando esta metodologia em diferentes níveis de abstração e granularidade. Considerações foram feitas a respeito da decisão sobre como aplicar a reconfiguração dinâmica da melhor forma possível. Os resultados adquiridos auxiliam os projetistas na escolha da melhor relação custo/benefício em termos de área de chip ocupada e atraso necessário para reconfiguração.

Abstract

An innovative methodology to model and simulate partial and dynamic reconfiguration is presented in this work. As dynamic reconfiguration can be seen as the remove and reinsertion of modules into the system, the presented methodology is based on the execution blocking of not configured modules during the simulation, without interfere on the normal system activity. Once the simulator provides the possibility to remove, insert and exchange modules during simulation, all systems modeled on this simulator can have the benefit of the dynamic reconfigurations.

In order to prove the concept, modifications on the SystemC kernel were developed, adding new instructions to remove and reconfigure modules at simulation time, enabling the simulator to be used either at transaction level (TLM) or at register transfer level (RTL). At TLM it allows the modeling and simulation of higher-level hardware and embedded software, while at RTL the dynamic system behavior can be observed at signals level. At the same time all the abstraction levels can be modeled and simulated, all system granularity can also be considered. At the end, every system able to be simulated using SystemC can also has your behavior changed on run-time. The provided set of instructions decreases the design cycle time. Compared with traditional strategies, information about dynamic and adaptive behavior will be available at earlier stages.

Three different applications were developed using the methodology at different abstract levels and granularities. Considerations about the decision on how to apply dynamic reconfiguration in the better way are also made. The acquired results assist the designers on choosing the best cost/benefit tradeoff in terms of chip area and reconfiguration delay.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1. Motivação

Arquiteturas Reconfiguráveis [1, 2] têm sido largamente utilizadas nos últimos anos na área de projeto de circuitos digitais, não só como ferramenta de prototipagem, mas também como produto final [9]. Estas arquiteturas são representadas principalmente pelas FPGAs (*Field-Programmable Gate Arrays*), micro chip formado por um arranjo de portas lógicas que podem ser combinadas para formar sistemas digitais diversos.

Com o advento de novas FPGAs capazes de serem reconfiguradas em tempo de execução (*Dynamically Reconfigurable* – FPGA ou DR-FPGA), surgiu o conceito de Arquiteturas Dinamicamente Reconfiguráveis e Computação Dinamicamente Reconfigurável [4, 5, 80]. Estes DR-FPGAs, como os da Família Virtex da Xilinx (www.xilinx.com) permitem, não apenas que configurações completas sejam feitas em tempo de execução, mas permitem também que algumas partes do sistema sejam removidas enquanto outras continuem trabalhando normalmente. A esta funcionalidade dá-se o nome de Reconfiguração Dinâmica Parcial.

Juntamente com o surgimento das DR-FPGAs surgiram também outras formas de circuitos dinâmica e parcialmente reconfiguráveis, circuitos considerados de granularidade grossa, ou *Coarse-grained* [56, 59, 67, 84]. Nestes, os elementos reconfigurados em tempo de execução não são as portas lógicas, modificando o circuito na escala de bits, mas as Unidades Lógicas e Aritméticas (ULA), modificando o circuito na escala de bytes ou de palavras.

As DR-FPGAs abriram caminhos para o surgimento de novas linhas de

pesquisa que têm se dedicado a estudar a Reconfiguração Dinâmica de forma geral e a explorar como os sistemas atuais podem se beneficiar destes novos conceitos. Existem atualmente livros [78, 79], eventos internacionais e outras publicações especializadas no assunto. Eventos como o FPL (*International Conference on Field Programmable Logic and Applications*) e o RAW (*Reconfigurable Architectures Workshop*) são os eventos que mais têm divulgado trabalhos científicos na área da Reconfiguração Dinâmica. Este último tem a reconfiguração em tempo de execução como principal foco.

Como não é necessário todo sistema estar configurado ao mesmo tempo num FPGA, há claramente uma economia de área de chip utilizada e, conseqüentemente, uma redução no consumo de energia. Outros benefícios como aumento no desempenho, facilidade de adaptação a mudanças e melhor aproveitamento de área de chip também já foram alcançados em projetos diversos [6, 10, 11, 12]. Pesquisas na linha de computação orgânica [55], eletrônica automotiva [82, 83] e de segurança [3] também têm visto na reconfiguração dinâmica uma oportunidade para o desenvolvimento de sistemas de hardware antes inviáveis. Novos sistemas adaptativos são capazes de desenvolver, por exemplo, sistemas de hardware tolerantes a falhas que corrigem as falhas em tempo de execução através de novas configurações, sistemas orgânicos, que se inspiram no comportamento de células orgânicas para executar tarefas de extrema adaptabilidade, dentre outras aplicações que requerem mudanças na estrutura do hardware.

DeHon [66] faz uma extensa revisão bibliográfica de arquiteturas reconfiguráveis, classificando os propósitos mais comuns onde sistemas reconfiguráveis têm sido utilizados nos últimos anos. Neste trabalho é feita uma classificação das técnicas de reconfiguração dinâmica pela granularidade do particionamento do sistema em partes menores. Arquiteturas de granularidade fina (*Fine Grain*), representadas pelas FPGAs, são geralmente mais flexíveis no desenvolvimento de algoritmos em hardware, mas o atraso decorrente das sucessivas reconfigurações é maior. Os trabalhos que utilizam granularidade grossa (*Coarse-Grain*) perdem menos tempo com as reconfigurações e necessitam de menos área de chip para a lógica de roteamento. Como suas atividades de roteamento são mais

simples, normalmente estas arquiteturas consomem menos energia e necessitam de menos área de chip. Há também arquiteturas que se beneficiam dos dois tipos de granularidade, consideradas arquiteturas de granularidade intermediária. Estas arquiteturas geralmente são formadas por agrupamento de FPGAs.

Arquiteturas mistas, que mesclam processadores convencionais (geralmente processadores RISC) com processadores dinamicamente reconfiguráveis, são uma excelente alternativa para contrapor as fraquezas, principalmente no atraso, das sucessivas reconfigurações geralmente necessárias. Essas são chamadas em inglês de *Dynamically Reconfigurable Systems-on-Chip* (DR-SoCs) e vêm se tornando foco de projetos de pesquisa pelo mundo [76, 77, 81].

1.2. Problemas

Este trabalho parte do pressuposto de que nenhuma abordagem seja ela de granularidade mais fina, ou de granularidade grossa, é ideal para todas as aplicações. Cada aplicação possui uma "granularidade ideal", com a qual é encontrada a relação área versus tempo ideal para ela. Encontrar esta granularidade pode ser o ponto chave para viabilização da reconfiguração dinâmica em aplicações específicas. A tarefa de descobrir esta granularidade ideal é complicada pela ausência de técnicas e ferramentas específicas [16, 17, 56, 57].

Os DR-FPGAs disponíveis no mercado possuem muitas limitações, principalmente no que se refere ao desempenho de suas reconfigurações dinâmicas. O tempo de reconfiguração tem se tornado o maior gargalo no uso das reconfigurações em tempo de execução. A escolha de uma granularidade mais grossa, ou de arquiteturas mistas (DR-SoCs) pode fazer com que esse gargalo seja diminuído.

No que diz respeito à reconfiguração dinâmica, a falta de ferramentas de projeto, particionamento, simulação e síntese flexíveis, capaz de prever o desempenho de sistemas em diversas granularidades é o problema central que será atacado por este trabalho, mais especificamente a modelagem e simulação.

Figura 1-1: fluxo de projeto baseado em FPGAs

A Figura 1-1 apresenta um típico fluxo de projetos que possuem um sistema funcionando em FPGA como alvo. Basicamente duas grandes atividades são desenvolvidas. O projeto do sistema em si e as atividades de simulação e verificação funcional. A simulação de alto nível de abstração é utilizada para realizar testes com a especificação (quando uma especificação executável é utilizada) e com o projeto de alto nível desenvolvido. Ultimamente, linguagens como SystemC (www.systemc.org), SystemVerilog (www.systemverilog.org) ou, até mesmo, linguagem C/C++ (ou variações), permitem a especificação executável de sistemas num alto nível de abstração. O objetivo é projetar a arquitetura geral do sistema, implementar funcionalidades básicas, mecanismos de sincronização, interfaces entre blocos e obter uma especificação executável do mesmo nas fases iniciais do projeto.

Uma vez definidas as características básicas do sistema, segue-se para o projeto em nível de transferência entre registradores, ou RTL (*Register Transfer Level*). Neste nível de abstração são caracterizados os detalhes das

funcionalidades dos blocos e das interfaces de comunicação, definindo os pinos de cada barramento e a lógica relacionada a eles. Dependendo da estratégia de projeto, pode-se dividir a simulação em nível RTL em duas fases. Inicialmente, pode-se optar por simular o sistema RTL num nível mais alto de abstração, aproveitando as linguagens e ferramentas utilizadas na fase anterior (SystemC, SystemVerilog, C/C++). Para ser sintetizado o código deve ser descrito utilizando uma linguagem de descrição de hardware (*HDL – Hardware Description Language*). Neste caso, ele deve ser migrado para uma linguagem como Verilog ou VHDL (linguagens HDL), para então ser sintetizado. Outra opção é escrever o código diretamente em HDL a partir da especificação escrita em TLM, sem utilização de tradução de código. As ferramentas utilizadas para tal migração também podem ser utilizadas para simular o resultado do particionamento do sistema em hardware e software.

A partir de uma descrição do sistema em HDL, o mesmo pode ser sintetizado. Durante a síntese a especificação do sistema é mapeada para as características e restrições do hardware alvo, no caso, o FPGA escolhido. Durante a fase de "*Place and Route*", cada módulo é mapeado para uma posição específica do FPGA e as trilhas de roteamento entre esses módulos são mapeadas. Ferramentas de projeto dos próprios fabricantes dos FPGAs são utilizadas para a simulação nesta fase. Neste momento valores mais precisos podem ser obtidos, como energia dissipada e área de silício necessária para configuração no FPGA.

Tratando-se de reconfiguração dinâmica, outros problemas são adicionados que devem ser tratados. Com a reconfiguração dinâmica, vários aspectos do projeto são modificados. Novas variáveis são adicionadas e comportamentos não previstos durante os ciclos de projeto convencionais são agora reais. Como prever, por exemplo, o que ocorreria com um determinado sistema, se um de seus módulos fosse substituído por outro com funcionalidade completamente diferente? Ou se simplesmente fosse removido do sistema? Tais comportamentos vão repercutir durante todas as fases do fluxo de projeto. Desta forma, se faz necessário algum suporte ferramental capaz também de modelar e simular tais comportamentos, das fases iniciais, até os níveis mais baixos de abstração.

A carência de um suporte ferramental para reconfiguração dinâmica é o foco principal deste trabalho. Para tanto, foi escolhida a linguagem SystemC como base principalmente, por esta poder ser utilizada durante as principais fases de simulação do sistema (simulação em nível TLM e nível RTL). Para os demais níveis de simulação, geralmente ferramentas dedicadas e fornecidas pelos próprios fabricantes de FPGA são utilizadas e mais recomendadas.

1.3. Objetivos

Este trabalho tem como objetivo geral:

- Desenvolvimento de uma metodologia de modelagem e simulação capaz de auxiliar na avaliação da utilização da reconfiguração dinâmica em sistemas digitais de granularidades diversas.

Algumas tarefas deverão ser executadas para que este objetivo seja alcançado, sendo os objetivos específicos deste trabalho:

- O desenvolvimento de um mecanismo para a simulação e o auxílio na análise de sistemas dinamicamente reconfiguráveis;
- Desenvolvimento de um exemplo que reforce o conceito defendido pela metodologia, de ser capaz de modelar e simular os principais casos de reconfiguração dinâmica definidos pela literatura;
- Desenvolvimento de estudos de caso que mostrem a viabilidade da metodologia de simulação.

1.4. Metodologia

Inicialmente foi realizada uma revisão bibliográfica sobre os principais trabalhos publicados nos meios de divulgação de maior relevância da área. Analisando tanto a reconfiguração dinâmica em si, quanto os mecanismos de simulação utilizados. Simuladores baseados em eventos discretos as operações do sistema são representadas como uma seqüência cronológica de eventos, onde cada evento ocorre num instante no tempo e marca uma mudança de estado no sistema [90]. Mais especificamente em simulação de sistemas de hardware, para cada processo pertencente ao sistema, há uma lista de sensibilidade a determinados eventos. Uma vez que certo evento

ocorre, os módulos sensíveis a ele são executados. Por exemplo, um processo sensível ao sinal positivo de *clock* do sistema, será executado a cada ciclo de *clock*, sempre que esse passar do valor baixo para o alto (de zero para um).

Baseado nesse conhecimento, uma metodologia para simular reconfiguração dinâmica foi desenvolvida. A estratégia adota foi a de anular a execução de determinados módulos do sistema simulando assim a desconfiguração do mesmo do sistema. Mesmo quando eventos ao quais eles são sensíveis ocorrem no sistema. Da mesma forma, o processo deve ser revertido, fazendo com que módulos antes desconfigurados, voltem a executar normalmente, simulando então a reconfiguração desses módulos no sistema.

Para que os objetivos gerais e específicos sejam alcançados, uma ferramenta de simulação de reconfiguração dinâmica foi desenvolvida aplicando-se modificações no código-fonte do núcleo do SystemC (que possui código aberto e livre). Novos métodos foram adicionados permitindo que qualquer módulo especificado seja bloqueado durante a simulação. Este bloqueio faz com que o sistema não sofra mais qualquer influência dos módulos bloqueados.

Uma variedade de definições de reconfiguração dinâmica foi encontrada na literatura especializada. A partir delas, foram caracterizados quais requisitos um simulador de reconfiguração dinâmica deve contemplar para ser considerado de completo, segundo a literatura atual. A partir daí, é mostrado que os métodos desenvolvidos em SystemC são suficientes para modelar e simular todos os diferentes tipos de sistemas dinamicamente reconfiguráveis citados pela literatura.

Estudos de caso foram desenvolvidos mostrando a como a ferramenta desenvolvida em SystemC pode ser utilizada de forma simples e prática, produzindo resultados de qualidade capazes de apoiar o projetista em decisões referentes à reconfiguração dinâmica, tais como, economia de área de chip e comportamento do sistema frente as sucessivas reconfigurações de parte dele (ou dele por completo) em tempo de execução.

1.5. Organização do Documento

Este documento está organizado em três partes:

- Introdução ao problema e a proposta de Tese (Introdução);
- Revisão da literatura e estado da arte (Capítulos 2 e 3);
- Apresentação da metodologia de simulação de sistemas dinamicamente reconfiguráveis em granularidades diversas, ressaltando o modelo de simulação baseado em eventos e como ele foi desenvolvido utilizando SystemC (Capítulo 4);
- Resultados obtidos em estudos de caso práticos que demonstram a aplicação e benefícios da metodologia (Capítulo 5);
- Análise dos resultados obtidos, ressaltando a contribuição da metodologia para o estado da arte, ao apresentar benefícios que sistemas reais podem obter ao utilizá-la no projeto de sistemas dinamicamente reconfiguráveis (Capítulo 6);
- Projeto de pesquisa que explora capacidades de ferramentas para modelagem e simulação de reconfiguração dinâmica (Apêndice A);
- Projeto de pesquisa que apresenta experimentos com SystemC que monitora o comportamento de módulos simulados com o objetivo de detectar possíveis aplicações de reconfiguração dinâmica (Apêndice B);

Procurou-se objetivar este documento referenciando as técnicas e métodos já consagrados na literatura ao invés de apresentá-los no documento. Assume-se neste documento o conhecimento prévio sobre técnicas de projeto e simulação de circuitos digitais e circuitos integrados, sobre arquiteturas reconfiguráveis e SystemC.

Capítulo 2

Sistemas Dinamicamente Reconfiguráveis

2.1. Introdução

O FPGA é um dispositivo programável pelo usuário que pode ser utilizado em um produto final, ou para fins de prototipagem. Ele tem sido uma alternativa aos ASIC devido ao baixo custo no desenvolvimento e por permitir uma diminuição no tempo de introdução de novos produtos no mercado. Ele também vem sendo uma alternativa ao uso de processadores especializados como DSP, por exemplo, devido ao elevado desempenho computacional.

Internamente um FPGA possui os seguintes elementos básicos, como pode ser visto na Figura 2-2:

- Blocos Lógicos de Configuração (ou CLB – *Configuration Logic Block*): nestes elementos são implementadas as funções lógicas do sistema;
- Blocos de Entrada e Saída (ou IOB – *Input/Output Block*): estão ligadas aos pinos do circuito integrado (CI) e quando programadas podem adquirir o comportamento de entrada ou saída, ou ambos;
- Recursos de interconexões: eles permitem interligações entre os blocos lógicos e as células de E/S;
- Estrutura de configuração: ela permite fixar o comportamento dos blocos lógicos e o caminho das interconexões entre eles, sejam eles blocos lógicos ou células de E/S.

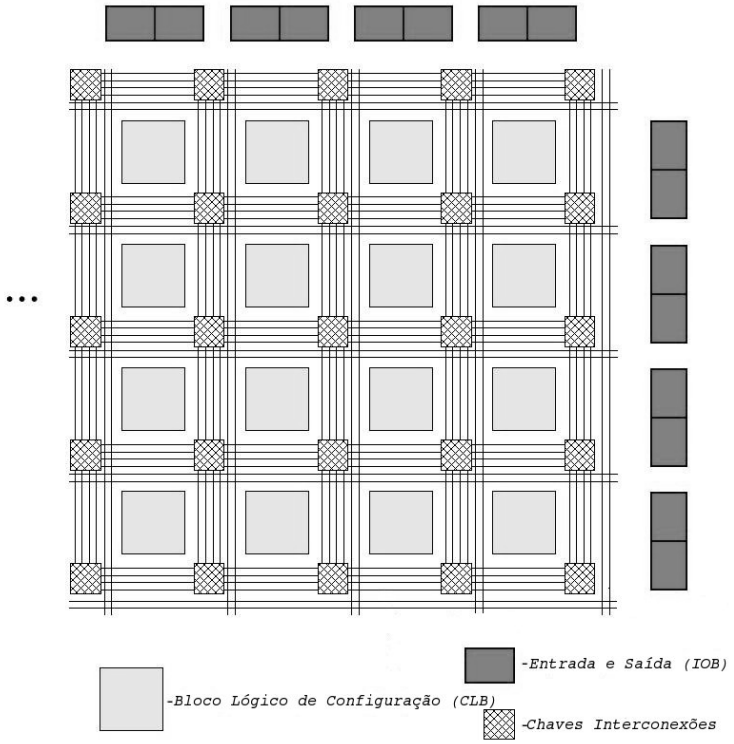


Figura 2-2: estrutura básica de um FPGA.

O tipo mais comum de memória de configuração utilizada pelos fabricantes de FPGA é o *Static Random-Access* Memory (SRAM) [18], que pode ter sua configuração definida apenas durante o ciclo de iniciação ou ser reconfigurado durante sua execução.

Lysaght [5] propôs em 1993 uma nomenclatura que engloba os vários tipos de reconfiguração dos diversos FPGA, que iremos seguir neste trabalho:

- Dinamicamente reconfigurável: nestes FPGA pode-se reconfigurar qualquer elemento lógico ou de interconexão enquanto o restante do FPGA continua operando normalmente;
- Parcialmente reconfigurável: permite uma reconfiguração seletiva enquanto o restante do dispositivo permanece inativo e retendo a sua informação de configuração;
- Reconfigurável: nesta categoria são incluídos todos os tipos de FPGA, exceto os FPGA que são do tipo "one-time programming" (FPGA pode ser configurado uma só vez).

- Programável: abrange todos os tipos de FPGA.

Uma das formas de controlar o processo de configuração do DR-FPGA consiste na elaboração de uma arquitetura formada de um microcontrolador que forneça o arquivo de configuração ao FPGA no instante adequado [7, 8, 9]. Nesta abordagem o FPGA executa as tarefas que precisam de desempenho em hardware, satisfazendo assim as especificações do projeto.

Um dos problemas encontrados no campo da computação reconfigurável é a falta de ferramentas de síntese de alto nível (comportamental) que façam uso do conceito de reconfiguração dinâmica, pois as atuais ferramentas foram projetadas para arquiteturas fixas ou estáticas encontradas nos ASIC [13].

Suprir esta falta é um dos objetivos deste trabalho. Atualmente, projetos de sistemas dinamicamente reconfiguráveis são feitos utilizando a metodologia tradicional, utilizando ferramentas de síntese lógica (nível RTL – *Register Transfer Level*) [12] e sem suporte a reconfiguração dinâmica. Sem um suporte ferramental, os cenários de configuração são escritos separadamente como se fossem sistemas independentes. Os efeitos das trocas entre eles são previstos manualmente através de inferências matemáticas e testados apenas em hardware.

Em sistemas dinamicamente reconfiguráveis, algumas operações básicas são sempre executadas, gerando um grande conjunto de possíveis aplicações que pode usufruir de tais funcionalidades, tais como:

- A remoção de um módulo do FPGA, aumentando a área reconfigurável livre e diminuindo o consumo de energia;
- A inserção de um módulo, antes não configurado, numa área livre do FPGA;
- A troca de dois ou mais módulos dentro de uma mesma área, onde o módulo maior vai determinar o tamanho da área alocada. Os módulos em questão não podem existir ao mesmo tempo. Neste caso, a estrutura de roteamento deve ser redefinida junto com a reconfiguração, evitando inconsistências.

Um ambiente de desenvolvimento adequado deve suportar as tais operações básicas, a fim de ser capaz de englobar os casos mais comuns de reconfiguração dinâmica.

A simulação da lógica dinamicamente reconfigurável (*Dynamically Reconfigurable Logic – DRL*) tem sido desenvolvida de diversas formas, como as descritas a seguir, ou por variações destas.

A simulação do DR-FPGA através de software [19, 84] consiste na simulação de modelos de DR-FPGA especificados em alguma linguagem de descrição de hardware. Estes simuladores são geralmente desenvolvidos juntamente com o hardware da própria DR-FPGA. São oferecidos como ferramenta para dar suporte ao desenvolvimento com a DR-FPGA. Tais ferramentas auxiliam na preparação de cada cenário e na síntese do código deles para o hardware. A simulação é feita em cada cenário individualmente e o tempo necessário para montar todas possíveis configurações é alto. Como cada cenário é desenvolvido separadamente e supondo cenários de complexidades iguais, pode-se dizer que o tempo necessário para desenvolver um módulo com três cenários diferentes é três vezes maior do que o tempo necessário para desenvolver um sistema com apenas um cenário. Nossa metodologia não visa sobrepor esta, mas trabalhar em conjunto. Uma vez que o sistema é simulado em níveis mais altos de abstração utilizando SystemC, se faz necessária a simulação do sistema em hardware. Para tanto ferramentas de simulação do funcionamento do DR-FPGA são necessárias para a obtenção de resultados mais precisos, como a ocupação de área e o tempo de reconfiguração. Tais dados podem ser utilizados para a retroalimentação do modelo desenvolvido utilizando nossa técnica, tornando-o cada vez mais consistente com o hardware alvo, possibilitando que novos cenários sejam criados e simulados mais rapidamente.

A técnica denominada *Clock Morphing* baseia-se na criação de um novo sinal lógico que indica que módulos estão desconfigurados. Quando um módulo é removido do sistema (por desconfiguração), ele emite sempre um sinal especial aos outros módulos que, a partir de então, ignoram pelos

outros módulos [14].

Outra estratégia é a utilização de pares casados de blocos de controle (ex.: multiplexador e demultiplexador) para chavear grupos de tarefas dinâmicas e mutuamente exclusivas [20]. Sinais de reconfiguração são enviados para esses blocos de controle que, baseado nisso, roteiam (ou param de rotear) todos os sinais para os blocos configurados (ou não configurados, respectivamente).

O Chaveamento Dinâmico de Circuitos (*Dynamic Circuit Switching* - DCS) utiliza chaves de isolamento para modelar a reconfiguração dinâmica. As tarefas dinâmicas são agrupadas em conjuntos ativos e inativos, os quais abrigam tarefas mutuamente exclusivas. Elas são conectadas às tarefas estáticas por meio de chaves de isolamento que permitem colocar um conjunto de tarefas em um dos seguintes estados: inativa, ativa ou transição [21, 22, 23].

As três últimas técnicas citadas, *Clock Morphing*, Multiplexador/Demultiplexador e DCS, podem ser utilizados em linguagens de alto nível, como SystemC e SystemVerilog, porém desenvolvem estratégias que necessitam de elementos lógicos adicionais para a simulação da reconfiguração dinâmica, o que pode tornar essa tarefa bastante trabalhosa, principalmente no que se refere a sistemas mais complexos, com vários cenários de configuração. No trabalho de Leandro Kojima [91] é apresentada uma metodologia para simulação de sistemas dinamicamente reconfiguráveis através de DCS utilizando SystemC. Neste trabalho um estudo de caso da Banda Base de um Controlador Bluetooth foi desenvolvido e dois cenários foram testados em nove diferentes DR-FPGAs.

Nossa abordagem, de anular a execução de módulos desconfigurados pelo núcleo do simulador (do SystemC mais especificamente), permite que a reconfiguração dinâmica seja simulada de forma simples, sem adicionar elementos lógicos extras, simplesmente chamando métodos especiais acrescentados à linguagem, que bloqueiam a execução dos mesmos a partir de então. Outra vantagem importante é que, uma vez que o simulador é modificado, todo modelo de simulação desenvolvido naquela linguagem pode

ser utilizado para simular as reconfigurações dinâmicas, sejam eles modelos em alto nível, como em TLM (*Transaction Level Model*), ou em nível mais baixo, em RTL (*Register Transfer Level*).

Todas as técnicas de DRL trabalham em nível lógico, o que as torna dependentes de arquiteturas. A falta de ferramentas completas que operam em nível de sistema, leva os projetistas de hardware a:

- Dividir manualmente o sistema em diferentes cenários que serão configurados seqüencialmente;
- Desenvolver, utilizando uma linguagem de descrição de hardware (*Hardware Description Language* - HDL), sintetizar, fazer verificação funcional do projeto, depurá-lo e gerar o arquivo de configuração para cada um dos cenários, levando em conta os espaços liberados e disponíveis do DR-FPGA.
- Fazer o teste de integração físico dos vários cenários manualmente.

Dentre as técnicas desenvolvidas que se beneficiam dos recursos das reconfigurações dinâmicas que utilizam granularidade fina destaca-se o conceito de Figuras de Execução. Este conceito foi defendido por Luiz Brunelli [16,17] neste mesmo grupo de pesquisa em sua defesa de tese de doutorado. As Figuras de Execução combinam os conceitos de computação reconfigurável e FPGA dinamicamente reconfigurável, mas não estão restritos a eles. Estes conceitos são utilizados de modo a eliminar as interconexões entre neurônios em Redes Neuronais. Isso é obtido concentrando-se a atenção nos dados e não nas transformações sobre eles, a semelhança com o que ocorreu durante a transição da Programação Modular para a Programação Orientada a Objetos. Os dados são fixos, não são transportados por barramentos ou ligações e, uma vez criados, não se movem pela arquitetura; e as operações aritméticas são as responsáveis pela dinâmica do processamento de dados. Os elementos de processamento trabalham sob demanda, quando necessário, eles são configurados no momento exato ao lado dos dados que servirão de entrada para os mesmos, e assim que terminam seu processamento, são retirados para dar lugar a

novos elementos. Desta forma, apenas os elementos necessários no momento estarão configurados no circuito.

2.2. Granularidade de Sistemas

As arquiteturas reconfiguráveis, seja em tempo de execução ou não, podem ser classificadas pela sua granularidade. A granularidade de uma arquitetura reconfigurável é definida pelo tamanho da menor unidade funcional (*Configurable Logic Blocks* - CLB) endereçável pelas ferramentas de mapeamento. Arquiteturas são ditas possuírem granularidade fina (*Fine Grain*) quando suas unidades funcionais possuem apenas 1 bit. Estes sistemas são representados principalmente pelos FPGAs em geral. Arquiteturas com CLBs mais largos são ditos de possuírem granularidade grossa (*Coarse Grain*) [56].

Arquiteturas de granularidade fina são mais precisas na programação de novas operações, mas são menos eficientes no que diz respeito ao roteamento de sinais entre os elementos lógicos. Como seus elementos lógicos são menores e são necessários em maiores quantidades, eles acabam gerando um grande *overhead* de área utilizada para o roteamento e possuem uma capacidade limitada de executar essas operações, além de serem menos eficientes no consumo de energia [56, 58, 67]. São vários os trabalhos encontrados na literatura que optam por utilizar granularidade fina para implementação de reconfiguração dinâmica [5, 6, 13, 19, 21, 80]. Lysaght [70] desenvolveu redes neurais utilizando FPGA dinamicamente reconfigurável de granularidade fina, obtendo em suas análises tempos de reconfiguração equivalentes a 42,23% do tempo de execução total. Hadley et al. [10], também trabalhando em granularidade fina, obteve o tempo de reconfiguração representando 53,5% do tempo total de execução do sistema. FPGAs da Atmel [80] com suporte a reconfiguração dinâmica possuem uma freqüência máxima de reconfiguração de aproximadamente 33MHz, e cada palavra de reconfiguração leva aproximadamente 1000ns para ser lida.

Já as arquiteturas de granularidade grossa, por utilizarem ciclos de dados (*data path*) mais largos (por exemplo, 8, 16 ou 32 bits) possuem menos flexibilidade em suas implementações, em compensação, possuem

um roteamento mais simplificado, por utilizarem menos elementos configuráveis (porém maiores individualmente) e necessitarem de menos rotas para interligá-los, aproveitam melhor a área de chip disponível, ocupando-o com mais elementos lógicos e menos linhas de roteamento. É mais simples alocar esses elementos no sistema e encaminhar os dados entre eles, além de também haver uma queda no tamanho necessário da memória de configuração, fazendo com que as configurações sejam mais rápidas [56, 67]. São exemplos destas arquiteturas: KressArray [59], Colt [60], MATRIX [61], Datapath FPGA [62], Garp [63], RAW [64], DReAM [65], entre outras. Algumas destas arquiteturas são também denominadas de granularidade "intermediária" por serem formadas pelo agrupamento de FPGAs formando uma arquitetura de granularidade um pouco mais grossa [56, 57].

O projeto de arquiteturas em granularidades diversas tem demonstrado a dificuldade na escolha de um grau de granularidade "ideal" para sistemas dinamicamente reconfiguráveis específicos. Considerando que o principal objetivo das reconfigurações dinâmicas é o ganho de área de chip, e a maior dificuldade são os atrasos gerados pelas sucessivas configurações, pode-se dizer que a granularidade ideal é aquela na qual é alcançada a melhor relação custo/benefício entre o atraso inserido ao sistema pelas sucessivas reconfigurações, e o aproveitamento de área obtido.

2.3. Sistemas de Granularidades Mistas (DR-SoC)

Sistemas de arquitetura mistas, aqui considerados, são aqueles formados por um processador RISC trabalhando na mesma pastilha de um processador dinamicamente reconfigurável, recebendo originalmente a denominação de *Dynamically Reconfigurable Systems on Chips*. Classificamo-nas como mistas por possuírem tanto processadores dinamicamente reconfiguráveis, quanto processadores convencionais, geralmente RISC.

A *Atmel Corporation* dispõe de um DR-SoC no mercado chamado FPSLIC, que combina arquiteturas de FPGAs da família AT40K, de 5 a 40 mil portas lógicas, com processadores RISC de 8 bits e 20 MIPS (milhões de

instruções por segundo) [80].

Projetos científicos financiados pela União Européia já desenvolvem produtos que integram diferentes processadores dinamicamente reconfiguráveis em granularidades diversas.

Por exemplo, o consórcio *Smart Chips for Smart Surroundings*, ou apenas 4S (www.smart-chips.net) [81], parte do pressuposto de que não há tecnologia capaz de preencher todos os pré-requisitos das aplicações. Por isso, o consórcio 4S propõe uma arquitetura de hardware heterogênea formada por diferentes partes, com sistema operacional e ferramentas, que permitem atribuir dinamicamente aplicações e tarefas para o dispositivo de hardware que mais "se encaixa" em cada caso. Essa atribuição é baseada nas propriedades de cada dispositivo e na necessidade dinâmica atual da aplicação. Esse conceito visa aperfeiçoar o projeto do sistema de hardware e software em tempo de compilação, e também estender a funcionalidade do dispositivo também durante a execução.

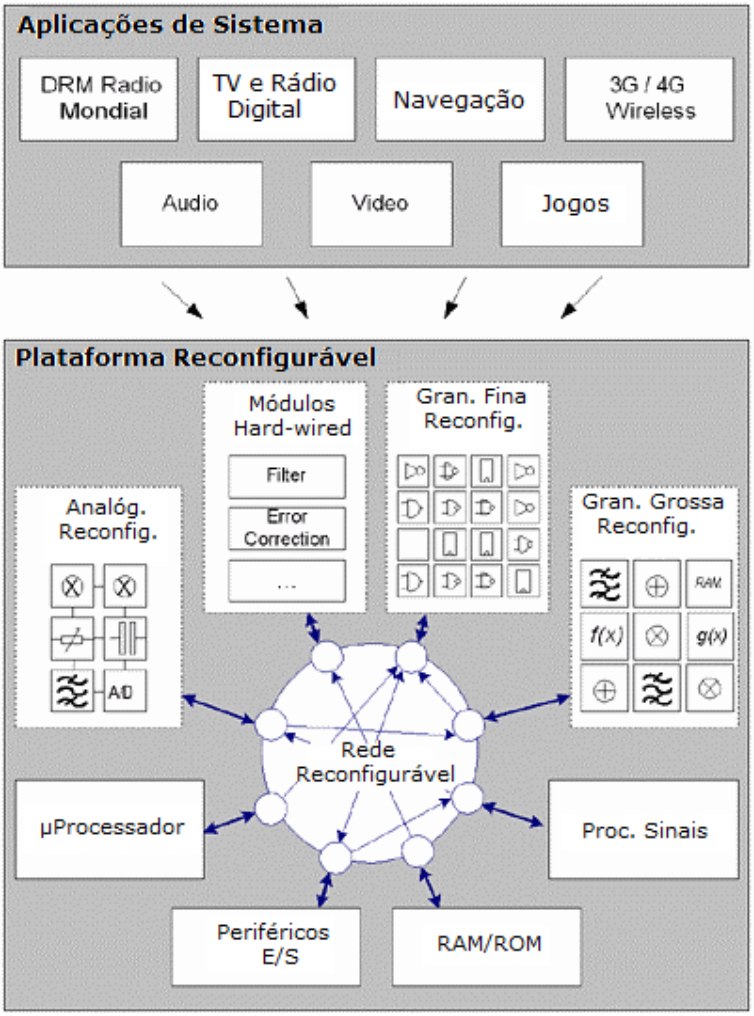


Figura 2-3: visão geral do projeto 4S

A Figura 2-3 mostra a visão geral da plataforma desenvolvida durante o projeto 4S. Nesta ilustração um conjunto vasto de aplicações pode ser executado pela plataforma, que possui diferentes tipos de dispositivos reconfiguráveis capazes de executá-las. Durante a execução, os dispositivos podem ser reconfigurados para melhor atender os requisitos das aplicações, ou para melhorar o desempenho e o consumo de energia geral da plataforma. Aplicações, uma vez alocadas para um determinado dispositivo, podem ser realocadas a outro, assim como dispositivos podem ser reconfigurados em tempo de execução para melhor suprir as necessidades atuais das aplicações em execução.

Com meta semelhante a do Projeto 4S, o Projeto MORPHEUS (www.morpheus.arces.unibo.it) visa desenvolver uma metodologia e uma

plataforma inovadoras no que diz respeito a sistemas dinamicamente reconfiguráveis. Diferente do Projeto 4S, o baixo consumo de energia não é um dos principais focos da plataforma, mas sim, objetiva o alto desempenho e a flexibilidade.

A Figura 2-4 apresenta um diagrama com o esquemático de tal plataforma. O processador ARM é um processador RISC convencional e é responsável por executar os programas (software) do usuário. Já os processadores PiCoGA, eFPGA e XPP são dinamicamente reconfiguráveis e são responsáveis por executar as tarefas de hardware. O DNA (*Direct Network Access*) é responsável por controlar a troca de dados entre os processadores na rede do chip (*Network-on-Chip - NoC*). A comunicação entre os elementos configuráveis e o processador ARM também é feita através da rede NoC.

Dois barramentos AMBA são utilizados para comunicação entre os componentes do sistema. O barramento de dados (no topo da figura), para troca de dados e instruções do processador ARM para o controlador da rede NoC, e o barramento de configuração (na parte inferior da figura) para a troca dados referentes à configuração dinâmica dos processadores. O Gerente de Configuração (*Configuration Manager* - CM) é responsável por receber requisições de configuração do processador ARM e repassá-las para os respectivos elementos reconfiguráveis através do barramento de configuração. Desta forma o tráfego de dados não compete com o tráfego de configuração no sistema.

Um mecanismo de interrupção é utilizado para sincronização entre os elementos configuráveis e o processador ARM. Sempre que um processamento é realizado, os elementos reconfiguráveis enviam um sinal de interrupção para o processador ARM que irá decidir que nova tarefa será realizada por ele. O Controlador de Interrupção (IC) recebe todos pedidos de interrupção e repassa-os para o processador ARM.

O projeto MORPHEUS traz a tona alguns desafios abordados pela arquitetura de computadores moderna. Tais como, o controle e reconfiguração dinâmica, a modularidade e heterogeneidade da plataforma,

integração de arquiteturas de granularidades finas e grossas, estrutura eficiente de interconexão, hierarquia de memórias e a integração do sistema como um todo.

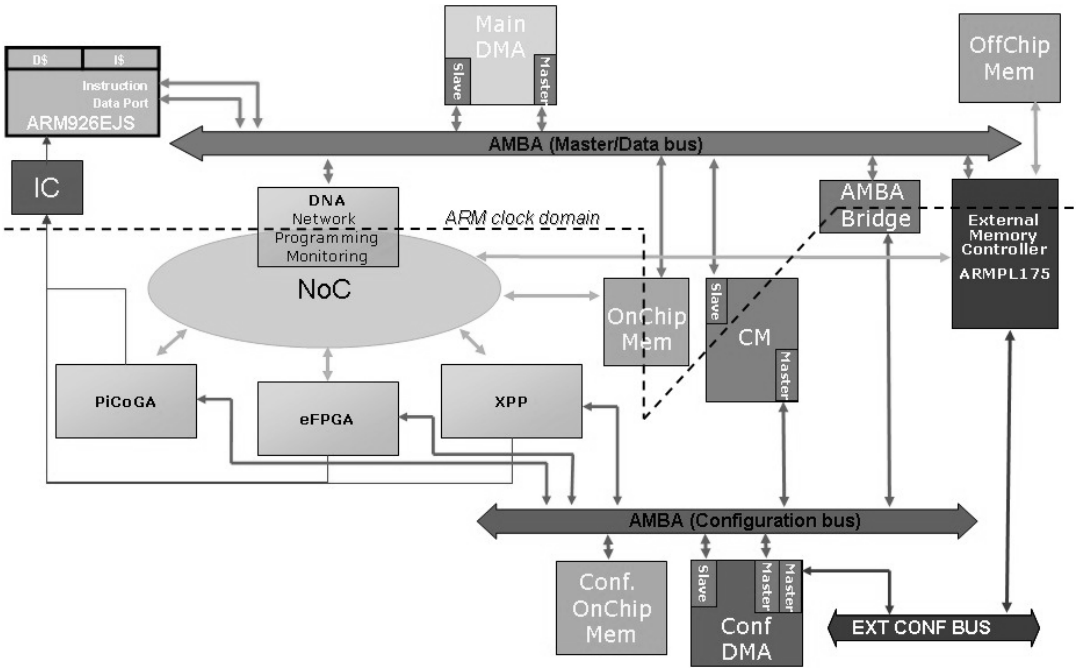


Figura 2-4: arquitetura da Plataforma MORPHEUS

Capítulo 3

Modelagem e Simulação de Sistemas Dinamicamente Reconfiguráveis

3.1. Introdução

Um dos pontos chaves para a viabilização da Reconfiguração Dinâmica é a disponibilidade de ferramentas flexíveis de simulação que permitam os projetistas estimar os pros e contras das reconfigurações dinâmicas antes de implementarem um protótipo. A flexibilidade aqui se refere à possibilidade da ferramenta ser utilizada para várias arquiteturas de hardware dinamicamente reconfiguráveis, independente de fabricante.

A independência de arquitetura está diretamente ligada ao nível de implementação utilizada pela ferramenta de simulação. Ferramentas de simulação, que trabalham no detalhe de funcionalidades de hardware, ditas ferramentas de baixo nível de abstração, são dependentes de arquitetura, e devem evoluir juntamente com o hardware alvo. Estas ferramentas geralmente são proprietárias e caras, por exemplo, as ferramentas da Xilinx [73] e da Atmel [80]. O trabalho de Ferrandi, Santambrogio e Sciuto [24] mostra como implementar reconfiguração dinâmica usando DR-FPGA e ferramentas proprietárias da Xilinx.

Possibilitar uma simulação de sistemas dinamicamente reconfiguráveis em alto nível implica no uso de alguma linguagem de simulação de hardware que suporte a reconfiguração de seus elementos em tempo de simulação.

Alguns conceitos da reconfiguração dinâmica são claramente implementados pelas linguagens orientadas a objetos. Estas linguagens permitem [32]:

- Modelos abstratos de dados (classes), que podem ser instanciados

dinamicamente em tempo de execução;

- Reuso de classes e objetos (herança);
- Mudança de comportamento em tempo de execução (polimorfismo).

Uma linguagem que poderia se adequar perfeitamente ao propósito de simular reconfiguração dinâmica é o IEEE 1800 SystemVerilog [27,28], que suporta a modelagem e a verificação de hardware baseados em transações. SystemVerilog também é orientado a objetos e possui suporte a modelagem e verificação em alto nível de abstração, permite chamada de funções C, C++ e SystemC e co-simulação com blocos SystemC. Simular reconfiguração dinâmica pode requerer a modificação de funcionalidades internas da linguagem ou pelo menos, o bom entendimento delas. Por não possuir uma implementação de código fonte aberto e disponível para experimentos (até o momento da elaboração deste trabalho), não foi possível que a utilizássemos para aplicar nossa metodologia.

A limitação de todas estas linguagens de modelagem de hardware *(Hardware Description Language – HDL)* é a falta de suporte à criação e destruição de objetos durante a simulação. Isto limita a possibilidade de utilizá-las para simular sistemas dinamicamente reconfiguráveis num nível de abstração mais elevado do que o nível estrutural, baseado em *netlist*.

SystemC [25,26], como uma linguagem de descrição de hardware baseada na linguagem C++ orientada a objetos, pode também ser utilizada para o propósito de modelagem e simulação de sistemas dinamicamente reconfiguráveis. Apesar de utilizar uma linguagem de alto nível, SystemC pode ser utilizado também para projetos sintetizáveis. Ele é organizado em dois níveis de abstração, o nível RTL (*Register Transfer Level*) foi desenvolvido especialmente para ser interpretado por ferramentas de síntese.

SystemC é a opção, juntamente com o SystemVerilog, mais recomendada para a aplicação em reconfiguração dinâmica. Como o SystemC é gratuito, de código aberto, por possuir um número maior de

publicações disponíveis, e por considerarmos o grupo de desenvolvedores SystemC mais numeroso e acessível, decidimos por não utilizar o SystemVerilog.

O projeto JHDL [29] suporta reconfiguração dinâmica. Ele descreve uma linguagem de descrição de *hardware* baseada em Java (www.sun.com) que possibilita o usuário projetar a estrutura e o leiaute de um circuito, depurá-lo na simulação, gerar o *netlist* e montar as interfaces para os bits de síntese. Há outros trabalhos na literatura que tratam da descrição de hardware utilizando linguagem Java [30, 31]. JHDL suporta reconfiguração dinâmica e síntese, além de possuir uma integração com o ambiente do Ptolemy, mas alguns autores [17] indicam falhas estruturais no JHDL que dificultam sua utilização em grandes projetos. Não há muitos trabalhos científicos que fazem referência ao JHDL (o último data de 1999, encontrado na seção de referências do próprio site do projeto – www.jhdl.org), o que pode significar uma descontinuidade no projeto. Esses fatores levaram a não optar por sua utilização.

Ptolemy [68] é um projeto que estuda modelagem, simulação e projeto de sistemas concorrentes, embarcados e de tempo real, baseado na montagem de componentes concorrentes. Utiliza Java como linguagem de desenvolvimento e permite a instanciação e a destruição de elementos (atores) em tempo de simulação, técnica denominada Mutação (*Mutation* [69]). O Ptolemy, apesar de permitir a mutação de cenários em tempo de simulação, não permite a síntese de seus elementos. No Apêndice A podem ser encontrados os detalhes sobre experimentos elaborados com as Mutações do Ptolemy.

3.2. Simulação de Sistemas Dinamicamente Reconfiguráveis Utilizando SystemC

Atualmente a linguagem de descrição de hardware que melhor reúne qualidades para especificação e simulação de sistemas dinamicamente reconfiguráveis é a SystemC. Ela é uma linguagem de código fonte aberto (*open-source*), gratuita e possibilita a modelagem e a simulação em nível de sistema utilizando os conceitos da orientação a objetos [25].

SystemC não permite a criação, nem a destruição de objetos durante a simulação, mas o fato de ter seu código fonte aberto, permite-nos estudar a possibilidade de estender suas funcionalidades para permitir a simulação de sistemas dinamicamente reconfiguráveis.

O projeto Adriatic [33] apresenta uma técnica de modelagem de sistemas dinamicamente reconfiguráveis em nível de sistema voltada à escolha de candidatos à reconfiguração. É utilizado especificamente para uma arquitetura proprietária formada por uma DR-FPGA e um co-processador encarregado de gerenciar as reconfigurações. A modelagem em nível de sistemas é implementada utilizando SystemC, mas tal técnica não visa à simulação dos comportamentos modelados, o que faz desta técnica não interessante para os propósitos deste trabalho.

Já o projeto OSSS+R [34] visa utilizar os conceitos da orientação a objetos, tais como herança e polimorfismo, para a simulação de reconfiguração parcial e dinâmica. Implementa uma extensão da linguagem SystemC, adicionando comandos para troca, durante a simulação, de módulos que são descendentes de uma mesma classe pai. Não permite que módulos sejam removidos de sistemas, nem que novos módulos sejam inseridos sem substituir módulos já existentes. Uma limitação desta técnica é sua abordagem *top-down*. Aqui novos comandos foram adicionados no topo da pilha de abstração do SystemC. Apenas sistemas descritos utilizando o nível de modelagem de transações (TLM) são capazes de serem simulados. Tais sistemas, uma vez migrados para o nível de transferência de registradores (RTL), não podem mais utilizar as funções da biblioteca desenvolvida para o nível TLM.

3.3. Escolha da linguagem a ser utilizada neste trabalho

Com o intuito de modelar e simular sistemas dinamicamente reconfiguráveis, a linguagem SystemC foi escolhida para ser utilizada neste trabalho. Para que isso pudesse ser concretizado, numa abordagem *bottom-up*, o código fonte do SystemC foi modificado para permitir que módulos sejam removidos e inseridos em sistemas durante a simulação, sem que a simulação seja paralisada para tal.

Nesta técnica inédita e inovadora, novas funções foram adicionadas ao núcleo de simulação do SystemC, fazendo com que possam ser utilizados por qualquer nível de abstração. Tal técnica desenvolvida é capaz de simular todos e quaisquer sistemas dinamicamente reconfiguráveis, em qualquer nível de abstração do SystemC. No Capítulo 4, detalhes do desenvolvimento desta nova técnica, principal foco desta tese, são apresentados.

No Apêndice B deste documento, são apresentados os primeiros experimentos que nos levaram a busca pela modificação do código-fonte do SystemC. Neste projeto de pesquisa, foi investigado como seria possível monitorar a execução de cada módulo simulado, auxiliando na definição de que módulos, e quando, poderiam ser desconfigurados para dar espaço para outros módulos no sistema, poupando área. Isso é possibilitado pelo desenvolvimento de um tipo especial de porta que deve utilizada por todos módulo sob monitoramento. Tal porta registra toda e qualquer passagem de dados. Com a modificação do código fonte do simulador, essas portas não são mais necessárias, já que todas as execuções de todos módulos do sistema são registradas em arquivo de histórico (*log*) para análise do comportamento das reconfigurações em tempo de simulação. Esse arquivo de *log* pode tanto ser utilizado antes, como após a implantação da reconfiguração dinâmica nos sistemas simulados. Antes ele pode ser utilizado para verificar potenciais utilizações, e após, para visualização do comportamento do sistema frente às sucessivas reconfigurações.

Capítulo 4

Simulação de Sistemas Dinamicamente Reconfiguráveis em Granularidades Diversas utilizando SystemC

4.1. Introdução

Atualmente a reconfiguração dinâmica total e a reconfiguração parcial são realidades [67]. Diversas indústrias investem e fornecem soluções tanto em granularidade fina (ex: FPGAs [82]) quanto grossa (ex: XPP [9]). Com esta capacidade, é possível reduzir a área de configuração necessária e desenvolver sistemas a custos mais baixos e mais eficientes no consumo de energia, onde o tempo é o parâmetro mais crítico.

A principal contribuição deste trabalho é possibilitar os engenheiros descobrirem ainda mais cedo, no seu processo de desenvolvimento, qual a relação custo-benefício entre o tempo gasto com as sucessivas configurações e a economia de área que se atinge com elas. Tal relação, geralmente, é obtida apenas depois da fase de prototipagem, durante os testes de sistemas complexos. Uma vez sendo possível a simulação de sistemas de forma simples, os benefícios concretos da reconfiguração em tempo de execução podem ser avaliados de forma mais confiável e tranqüila durante as fases iniciais do desenvolvimento do projeto.

A técnica inovadora apresentada aqui permite a modelagem e a simulação de tais sistemas através da disponibilização de rotinas básicas de SystemC. Ela permite a previsão do comportamento dinâmico de sistemas antes de sua síntese para a arquitetura alvo e sua configuração no FPGA. Mais além, é possível a avaliação dos sistemas ainda antes deles serem codificados em linguagem de descrição de hardware (HDL). Três artigos [83,

84, 85] publicados em eventos internacionais resumem o funcionamento, aplicação e resultados adquiridos utilizando a técnica apresentada neste capítulo.

4.2. Metodologia para Simulação de Reconfiguração Dinâmica

A metodologia utilizada para a simulação de reconfiguração dinâmica foi a de alterar o mecanismo de simulação de simuladores baseados em eventos. O simulador deve, para cada módulo do sistema, antes de executá-los, verificar se eles foram desconfigurados. Em caso afirmativo, não devem ser executados.

A Figura 4-5 apresenta um modelo geral de um simulador baseado em eventos e organizado em módulos e processos, utilizados principalmente em simulação de sistemas digitais de hardware. Cada módulo pode implementar um ou mais processos, que executam as tarefas propriamente ditas. Cada processo possui uma lista de sensibilidade, indicando as quais eventos o processo é sensível. Um processo deve ser executado num determinado ciclo de simulação se um dos eventos de sua lista de sensibilidade ocorrer naquele ciclo. No exemplo apresentado na Figura 4-5, o evento E3 pode representar alguma mudança no sinal de *clock*, e como todos os processos representados (processos 1, 2 e 3) são sensíveis a ele, sempre que houver uma mudança no sinal de *clock*, todos serão executados.

O escalonador, que faz parte do núcleo do simulador, decide qual a seqüência de execução para cada ciclo. Na Figura 4-5, o evento E1 é escalonado, e pesquisado nas listas de sensibilidade dos processos, sendo encontrado nas listas dos processos 1 e 3, pertencentes aos módulos A e B, respectivamente. Neste caso, apenas esses processos são executados neste ciclo.

O tempo simulado é formado por uma seqüência de ciclos de simulação. A cada ciclo, um ou mais eventos podem ocorrer. Caso nenhum evento ocorra naquele ciclo, nenhum ciclo é realmente criado, fazendo com que o número total de ciclos cresça mais rapidamente e a simulação ocorra num tempo mais curto. Isso falando do ponto de vista da máquina que executa a simulação, pois com relação ao tempo simulado, ele não é

dependente da quantidade de eventos de cada ciclo, porém, depende unicamente do tempo simulado relativo ao último evento disparado. Não faz diferença para o tempo simulado, quantos eventos serão executados em cada ciclo.

Quanto ao desempenho da simulação, as listas de sensibilidade devem também ser levadas em consideração. Quanto maior o número de eventos nas listas de sensibilidade, maior a probabilidade de um processo ser executado custando processamento da máquina (ou máquinas) que executa o simulador.

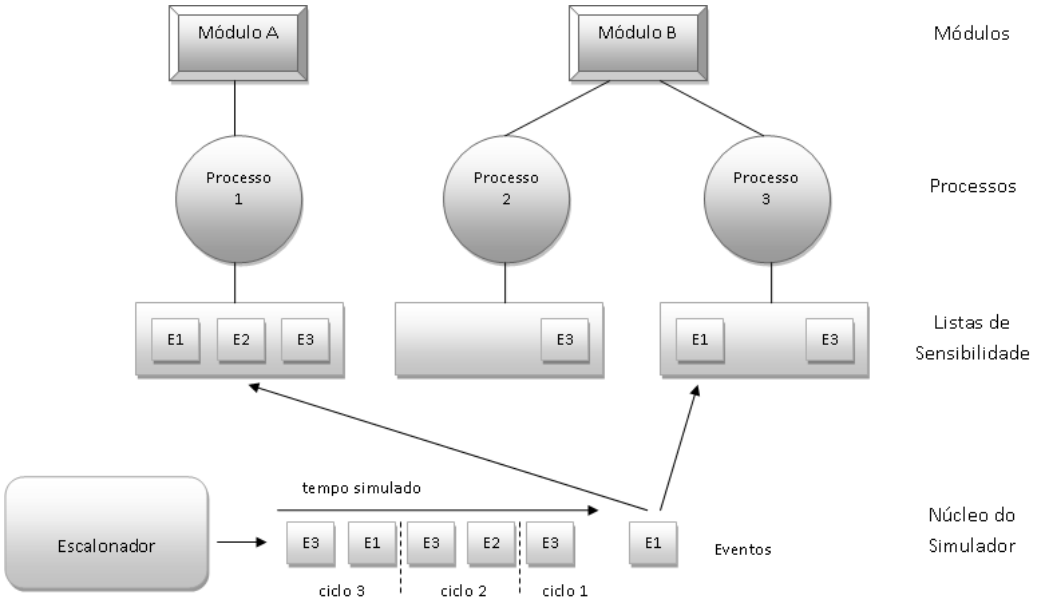


Figura 4-5: mecanismo de execução de simuladores de sistemas de hardware digitais baseados em eventos.

Tratando-se de simulação, na reconfiguração dinâmica, um módulo não configurado pode ser definido como um módulo que não é executado, independentemente dos eventos ocorridos, ou de sua lista de sensibilidade. De forma análoga, módulos não configurados, devem poder ser reconfigurados durante a simulação, implicando que eles devem voltar a serem executados normalmente, como qualquer outro módulo.

A metodologia apresentada neste trabalho baseia-se na interceptação das mensagens de execução geradas para os módulos, fazendo com que módulos não configurados não sejam executados. A adoção do bloqueio de

módulos, ao invés do bloqueio de processos deve-se a uma razão de caráter conceitual, mas nada impede que outras implementações adotem outra postura.

Essa razão conceitual baseia-se no significado de um módulo para um sistema. Geralmente um módulo determina uma responsabilidade específica de parte do sistema. Por exemplo, um módulo que implementa Transformações Discretas de Cossenos (*Discrete Cosine Transform* – DCT) deve executar apenas essa operação, que pode ser separada em vários processos (somas, multiplicações, cossenos etc.), seriais e/ou seqüenciais, que unidos desenvolvem a função principal da DCT como um todo. Apesar de tecnicamente possível, não é comum que algumas desses processos necessitem ser modificados em tempo de execução. O mais lógico é que um módulo inteiro (o DCT, por exemplo) seja desconfigurado do sistema para dar espaço a outros módulos.

A Figura 4-6 apresenta as modificações que devem ser efetuadas no simulador a fim de interceptar os sinais de execução para módulos não mais configurados. Como mencionado anteriormente, a estratégia é bloquear módulos, fazendo-se necessário manter uma lista dos módulos bloqueados (ou desconfigurados) e que não devem ser executados pelo sistema. Ao invés de executar diretamente os processos relacionados ao evento disparado num determinado instante, propomos que a execução seja interceptada para checagem da lista de módulos bloqueados. Se o módulo não existir nessa lista, a execução é feita normalmente, caso contrário, os processos não são executados e a simulação comporta-se como se o evento não tivesse ocorrido para aquele módulo, mas ocorrido normalmente para os outros módulos sensíveis ao evento e não pertencentes à lista. Isso é importante ser ressaltado. Apenas os módulos na lista de bloqueados devem ter seus processos impedidos de executar. Assim como apresentado na Figura 4-6, na checagem foi detectado que o Módulo B estava registrado na lista de bloqueados, por isso, o Processo 3 não pôde ser executado, mesmo sendo sensível ao evento E1.

Tendo a metodologia sido implementada num simulador, a aplicação da

reconfiguração dinâmica fica resumida apenas a manutenção da lista de módulos bloqueados, inserindo alguma referência aos módulos que devem ser desconfigurados do sistema, e removendo-as para os mesmos voltem a ser reconfigurados.

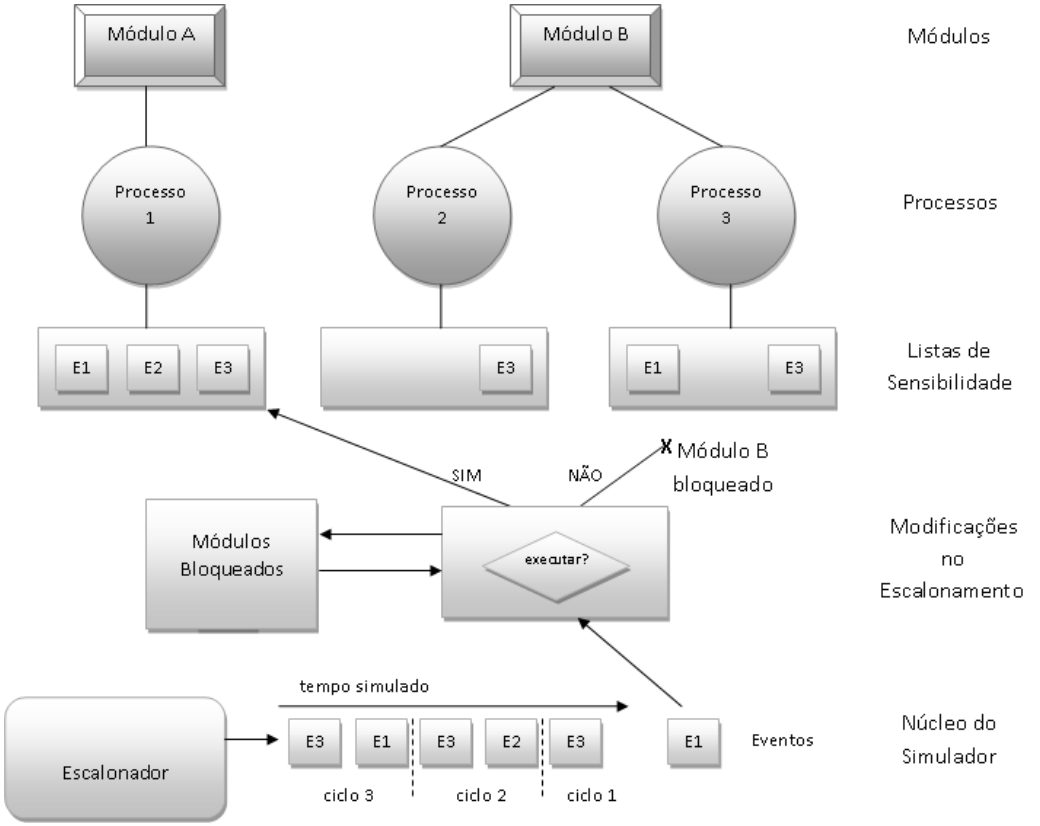


Figura 4-6: mecanismo de execução de simuladores modificado para bloqueio de módulos, representando a desconfiguração deles do sistema.

A estratégia para implementação da metodologia depende de cada desenvolvedor e das limitações do simulador utilizado, mas de forma geral, a informação dos módulos bloqueados deve ser mantida de alguma forma. Seja ela centralizada numa, como distribuída através de propriedades dos módulos.

Uma estratégia aparentemente simples seria a limpeza da Lista de Sensibilidade dos módulos bloqueados. Uma vez realizada a limpeza, nenhuma modificação extra na execução dos processos pelo simulador precisa ser realizada. Desse modo, um módulo bloqueado não seria sensível a nenhum evento, não executaria até sua lista de sensibilidade fosse restaurada. Isso aumentaria o desempenho do simulador, já que o mesmo

não faria nenhuma checagem adicional para bloquear os processos. Por outro lado, a logística para limpar as Listas de Sensibilidade e restaurá-las pode complicar o processo. Limpar a lista implica em remover todo seu conteúdo, mas é necessário que esse conteúdo seja armazenado para ser restaurado quando reconfiguração do módulo for solicitada. Por isso, deve-se haver o controle dos módulos desconfigurados, armazenando suas respectivas listas de sensibilidade. O ganho com essa estratégia está, portanto no desempenho do simulador, mas não na complexidade de implementação e manutenção, já que ambos deverão manter uma referência aos módulos bloqueados, sendo que na limpeza das listas, essas referências serão mais complexas, devendo conter várias estruturas, uma para cada processo do módulo em questão.

Outra estratégia seria o controle dos módulos desconfigurados através de propriedades adicionadas a eles. Uma simples variável poderia indicar se um determinado módulo estaria, ou não configurado naquele momento. Essa estratégia eliminaria a necessidade da lista de módulos bloqueados no simulador. O desempenho do simulador também aumentaria, já que a checagem de uma propriedade de um objeto (de um módulo, no caso) é computacionalmente mais simples do que uma varredura numa lista. A questão novamente seria, qual o esforço necessário para implementar essas modificações nos módulos da simulação. Isso pode variar de simulador para simulador. No SystemC, objeto de estudo desse trabalho, a modificação da propriedade do módulo não seria complicada, mas como utilizá-la na prática sim. Por exemplo, caso um determinado módulo, chamemos de módulo gerente, necessite desconfigurar um determinado módulo A da simulação, seria necessário que o mesmo mantivesse uma referência ao objeto do módulo A. A única classe que mantém referência a todos módulos da simulação é a classe principal, que cria declara os objetos, cria as instâncias, os conecta, configura e inicia a simulação. O módulo gerente de configuração precisaria manter uma lista com as referências a todos os outros módulos do sistema para que a desconfiguração e reconfiguração de todos eles sejam possíveis a qualquer momento durante a simulação. Outros simuladores que utilizem mecanismos diferentes de instanciação de objetos podem ter essa

estratégia aplicada de modo mais simplificado que o SystemC.

Como nosso foco é mostrar a viabilidade da simulação da reconfiguração dinâmica através do simples bloqueio da execução de módulos não configurados (e claro, da restauração de módulos antes bloqueados, simbolizando a reconfiguração dos mesmos), adotamos pela estratégia mais simples, a de bloqueio de módulos auxiliada pela checagem de uma lista de módulos bloqueados, mesmo que isso não resulte na opção de melhor desempenho. Números práticos sobre o desempenho do simulador modificado serão apresentados posteriormente nesse capítulo.

A aplicação dessa metodologia utilizando uma das alternativas de desenvolvimento citadas acima, ou outras idéias que venham a surgir devem ser consideradas e encorajadas a serem comparadas com a implementação apresentada nesse trabalho.

4.3. Ciclo de Projeto de Sistemas Dinamicamente Reconfiguráveis com SystemC

O desenvolvimento da metodologia apresentada utilizando o SystemC resultou na implementação de duas novas rotinas capazes de simular a reconfiguração dinâmica de sistemas. Uma vez que o simulador sempre precisa checar a lista de módulos bloqueados, a estratégia adotada foi a de oferecer aos desenvolvedores maneiras de inserir e remover elementos dessa lista.

Essas rotinas acrescentam ao SystemC a possibilidade de modelagem e simulação de sistemas dinamicamente reconfiguráveis em diversas granularidades e, simultaneamente, permitem que, tanto sistemas desenvolvidos no nível mais alto de abstração (TLM), quanto no nível de descrição de registradores (RTL) possam ser modelados e simulados.

Outro aspecto importante é como uma técnica como essa pode ser integrada aos processos de desenvolvimento atuais. Esta integração deve ser idealmente de uma maneira leve, com uma curva de aprendizagem suave e sem maiores impactos ao processo. Para isso, a linguagem de descrição de hardware, SystemC, foi utilizada. Um conjunto de rotinas simples de utilizar foi desenvolvido, que podem ser aplicadas em qualquer sistema

descrito em SystemC.

A Figura 4-7 apresenta um ciclo de projeto básico utilizando SystemC como principal ambiente de desenvolvimento. Normalmente o mesmo ciclo de projeto é utilizado, indiferente se o sistema utilizará, ou não, os recursos da reconfiguração dinâmica. Esta falta de técnicas e ferramentas específicas torna o desenvolvimento de sistemas dinamicamente reconfiguráveis uma tarefa árdua e onerosa.

Geralmente, após a especificação do sistema, ele é descrito no mais alto nível de abstração, utilizando os recursos do SystemC TLM (Nível de Transações – *Transaction Level Modeling*). Neste caso serão especificadas as funcionalidades gerais dos módulos, as interfaces e os protocolos de comunicação entre eles. Os detalhes dos pinos bit a bit, serão descritos após um refinamento (normalmente, num processo gradual, manual ou semi-automático) do sistema do modelo TLM para o modelo RTL (Nível de Transferência entre Registradores – *Register Transfer Level).* Em ambos os níveis, TLM e RTL, o sistema é simulado e verificado, podendo o projeto ser revisto em ambos os níveis, caso necessário. Após o término deste ciclo, o sistema pode ser traduzido para uma linguagem de descrição de hardware (HDL – *Hardware Description Language*), que depois de sintetizado, poderá ser utilizado para programar o FPGA. Um de nossos objetivos com nossa técnica de simulação de sistemas dinamicamente reconfiguráveis é justamente reduzir o tempo gasto neste ciclo, entre a verificação e a re-implementação do sistema em TLM e RTL.

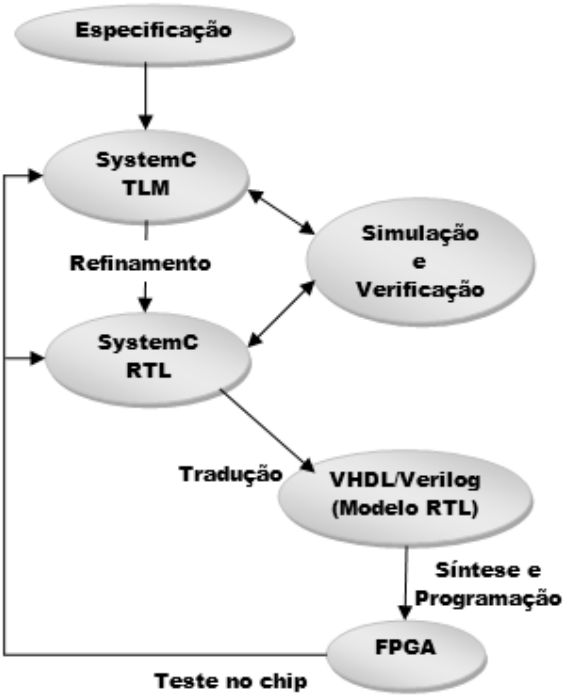


Figura 4-7: ciclo de projeto baseado em SystemC

Com este intuito, propomos uma extensão no modelo do ciclo de projeto apresentado na Figura 4-7. A modelagem e a simulação da reconfiguração dinâmica e parcial são agregadas ao ciclo de projeto, como mostra a Figura 4-8.

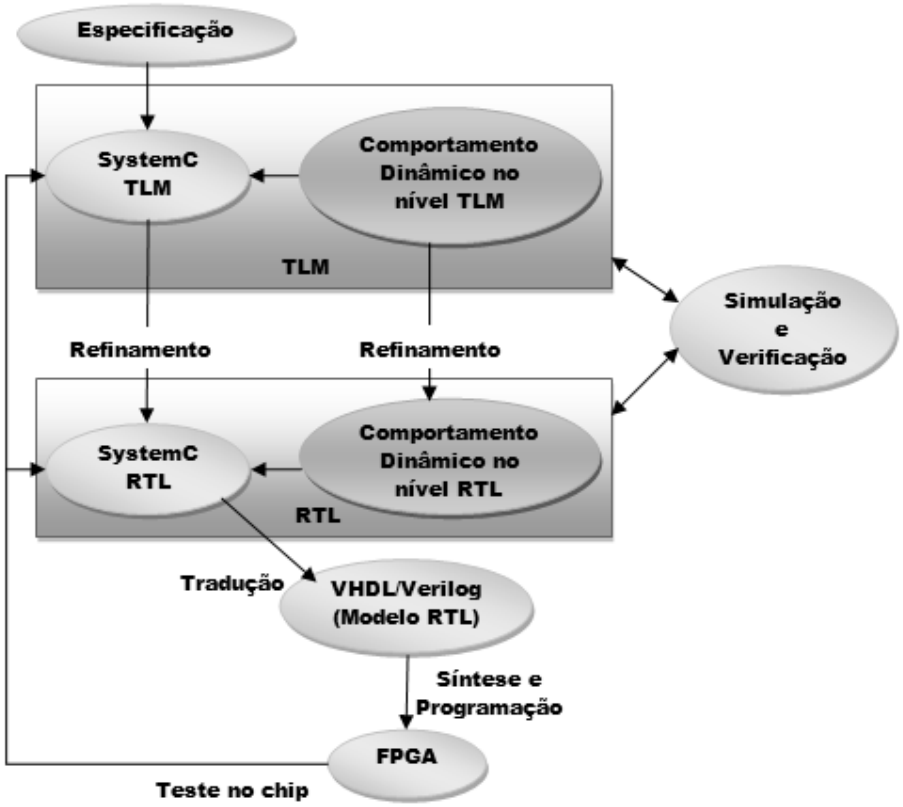


Figura 4-8: ciclo de projeto baseado em SystemC com suporte à reconfiguração dinâmica.

O comportamento dinâmico nos níveis TLM e RTL é realizado pela utilização de rotinas específicas. O projetista decide a melhor lógica para utilizá-las. Uma forma interessante é o desenvolvimento em um ou mais módulos especializados na reconfiguração dinâmica, centralizando tais decisões. Tais módulos, em hardware poderão ser gerenciadores de configuração dedicados.

Inicialmente durante as simulações de reconfigurações dinâmicas a quantidade de informações sobre o funcionamento dos elementos do hardware é limitada. Geralmente ainda não são conhecidas a área dos elementos, nem o tempo de reconfiguração para cada um deles. Nesse caso, deve-se trabalhar com estimativas e refinamento. Num primeiro ciclo, o sistema deve ser simulado com as informações e inferências que estiverem disponíveis. Aqui números sobre os módulos que sejam fáceis de serem alcançados pelas tecnologias atuais podem ser inferidos para seus módulos. Isso fará com que a escolha da tecnologia não seja o foco da simulação

deste ciclo, mas a possibilidade do sistema ser particionado no tempo e por conseqüência ter partes desconfiguradas em tempo de execução. Há sistemas que não possuem módulos que não funcionam o tempo todo, isso os impossibilita de serem removidos do sistema em qualquer instante.

No próximo ciclo, sabendo que o sistema pode ser simulado utilizando reconfiguração dinâmica, os resultados sobre os módulos em hardware obtidos após a síntese em FPGA poderão ser reutilizados para realimentar o sistema e simulá-lo novamente, agora com números mais precisos.

Estando o sistema de simulação refinado, vários novos cenários podem ser criados e simulados rapidamente, sem a necessidade de gerá-los novamente em FPGA. Utilizando apenas a simulação em TLM os vários cenários dinâmicos podem ser testados e números reais podem ser obtidos.

4.4. Simulação de Reconfiguração Dinâmica utilizando SystemC

Afim de que se possa demonstrar a generalidade da metodologia defendida neste trabalho, devemos primeiramente classificar exatamente o que um simulador de reconfiguração dinâmica (e parcial) deve ser capaz de modelar. Segundo a literatura, a operação básica na reconfiguração dinâmica e parcial é a substituição, ou troca, de módulos, tarefas ou funcionalidades ativas no sistema por outros implementados, mas não configurados no chip.

No artigo [5], Lysaght et. al. descreve que a reconfiguração parcial é realizada através da execução de uma seqüência de tarefas por módulos em hardware que são escalonadas no tempo.

O trabalho de Zhang et. al. [13] afirma que para simular a operação de um FPGA dinamicamente reconfigurável, um simulador deve ser capaz de modelar qualquer circuito estático que está ativo, ao mesmo tempo em que modela a troca de circuitos dinâmicos por novos circuitos. Ou seja, a troca de módulos de hardware que estão configurados no chip, por outros módulos ainda não configurados.

No artigo em que a empresa Xilinx, maior fabricante de FPGAs parcialmente reconfiguráveis, apresenta como utilizar sua ferramenta de desenvolvimento PlanAhead para modelagem de sistemas dinamicamente

reconfiguráveis [73], é recomendado que para desenvolver um sistema utilizando reconfiguração parcial é necessário definir macros de barramento entre os módulos que necessitam ser substituídos. No processo de substituição se desativa um modulo para em seguida ativar outro. Isso significa que, segundo a Xilinx, só a desativação e ativação de módulos são suficientes para simular a reconfiguração dinâmica.

Já segundo Pleis et. al. [89], um sistema dinamicamente reconfigurável é formado por diferentes funcionalidades e, quando uma funcionalidade, ou configuração, é trocada por outra, os manipuladores de interrupção, ou serviços de interrupção específicos devem também ser carregados. Ou seja, um sistema dinamicamente reconfigurável é formado por diferentes funcionalidades que são substituídas no tempo.

De acordo com o RECONF 2, projeto financiado pela Comissão Européia através do Programa IST (Contrato IST-2001-34016), que objetiva o desenvolvimento de um ambiente de desenvolvimento capaz de utilizar as FPGAs Dinamicamente Reconfiguráveis (DR-FPGA) de forma eficiente, três tipos de reconfiguração dinâmica podem ser identificados. O documento "Deliverable 2.2", que especifica o ambiente de desenvolvimento proposto simulação de reconfiguração dinâmica, publicado em janeiro de 2003 [88] faz a seguinte citação:

> "Do ponto de vista da aplicação, três tipos de reconfiguração dinâmica podem ser identificados.
>
> O primeiro tipo consiste na troca entre dois ou mais módulos reconfiguráveis (denominados D_Modules) com a mesma interface e na mesma área da FPGA. Se os tamanhos dos módulos forem diferentes, o maior determina o tamanho da área envolvida no processo de reconfiguração. Os módulos diferentes são ligados funcionalmente por suas interfaces, já que não podem ser apresentados ao mesmo tempo no FPGA.
>
> O segundo tipo de reconfiguração dinâmica

> consiste na limpeza de alguns D_Modules do FPGA e usar a área associada a eles para implementar outros D_Modules com ou sem funcionalidades relacionadas.
>
> O terceiro tipo de reconfiguração corresponde ao particionamento da funcionalidade de um módulo em vários sub-módulos, fazendo a mesma computação, mas em tempos compartilhados. Este particionamento pode ser feito automaticamente, sem nenhuma ligação com o uso do escalonamento do módulo dentro da aplicação, mas não vão prover uma implementação com precisão de ciclo de relógio.
>
> O primeiro tipo é um subconjunto do segundo, porém permitindo projetista criar explicitamente uma ligação entre dois ou mais módulos para facilitar o particionamento e otimizar a eficiência do FPGA."

Essa classificação aprofunda o que os demais trabalhos acima citados definem sobre o que é simular sistemas dinâmica e parcialmente reconfiguráveis. Se uma ferramenta de simulação for capaz de modelar essas três situações, ela será capaz de simular qualquer sistema dinamicamente e parcialmente reconfigurável, segundo as definições dos trabalhos aqui citados.

A Figura 4-9 apresenta o primeiro tipo de reconfiguração dinâmica, quando um módulo (D_Module C), no primeiro cenário, é substituído por outro (D_Module D) na mesma área.

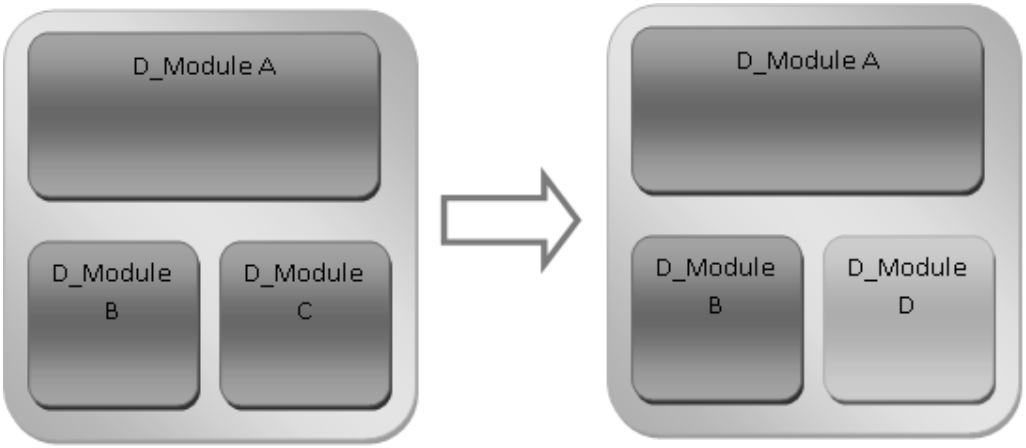


Figura 4-9: primeiro tipo de reconfiguração dinâmica. Substituição de módulo configurável.

Já na Figura 4-10, o segundo tipo de reconfiguração é apresentado, onde um módulo reconfigurável (D_Module C) é removido para liberação de área.

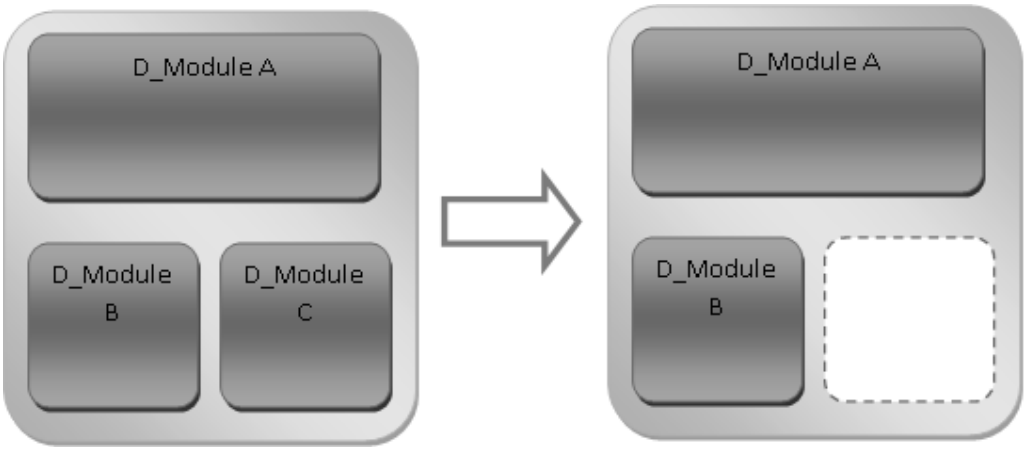


Figura 4-10: segundo tipo de reconfiguração dinâmica. Remoção de módulo configurável.

No terceiro tipo de reconfiguração, apresentado na Figura 4-11, um módulo (D_Module A) é particionado em módulos menores executando a mesma tarefa, mas agora em tempos compartilhados.

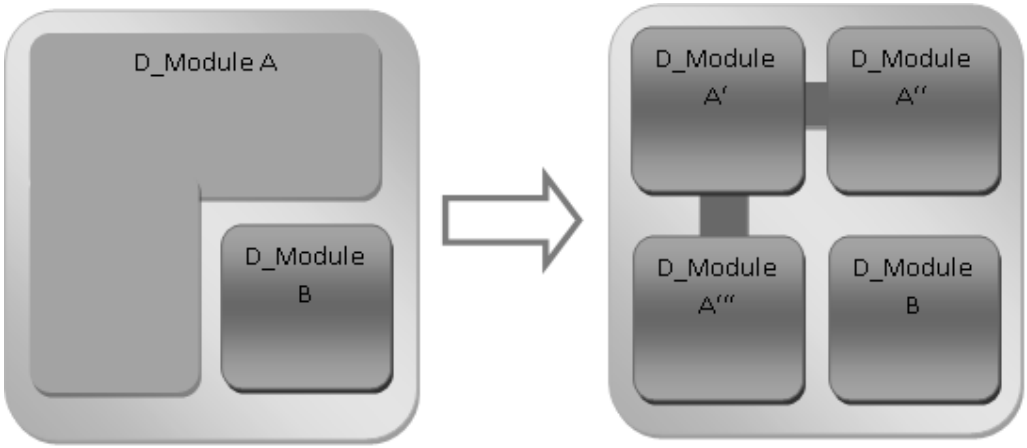


Figura 4-11: terceiro tipo de reconfiguração dinâmica. Particionamento de módulo configurável.

A estratégia utilizada neste trabalho é prover operações básicas que permitam ao SystemC simular tais situações. Para isso duas rotinas especiais foram desenvolvidas, uma para ativar, e outra para desativar módulos durante a simulação, chamadas *dr_sc_turn_on* e *dr_sc_turn_off*, respectivamente. Elas foram implementadas modificando o código fonte interno do núcleo de simulação do SystemC. A Figura 4-12 apresenta o cabeçalho das três principais rotinas adicionadas ao SystemC implementadas na classe *sc_simcontext* (no arquivo *sc_simcontext.h*). As rotinas *dr_add_constraint* são utilizadas para que os atributos dos módulos sejam armazenados, como a área de chip utilizada e o atraso de reconfiguração necessário. Elas possuem duas implementações, uma com, e outra sem, o atraso de reconfiguração como parâmetro. A declaração *extern* indica que as rotinas podem ser chamadas fora da classe *sc_context*, no caso, nos sistemas de simulação desenvolvidos. Sempre que uma nova rotina precisar ser adicionada ao SystemC e disponibilizada para os desenvolvedores, a mesma técnica deve ser utilizada.

```
extern void dr_sc_turn_off(std::string module_name);
extern void dr_add_constraint(std::string module_name, int area);
extern void dr_add_constraint(std::string module_name, int area, sc_time reconfDelay);
extern void dr_sc_turn_on(std::string module_name);
```

Figura 4-12: principais funções adicionadas à biblioteca do SystemC. Do arquivo sc_simcontext.h

Os nomes dados às rotinas fazem alusão à expressão em inglês, utilizada para ligar (*turn on*) e desligar (*turn off*) equipamentos eletrônicos.

Como será mostrado a seguir, na técnica mostrada aqui, módulos da simulação SystemC comportam-se como se realmente estivessem sendo ligados, ou desligados. O prefixo *sc* representa a abreviação de SystemC, sempre utilizado para designar código fonte adicionado pelo grupo e não código nativo da linguagem C++. Já o prefixo *dr* representa a abreviação de *Dynamic Reconfiguration* (Reconfiguração Dinâmica).

A Figura 4-13 apresenta a implementação das rotinas apresentadas na Figura 4-12, implementadas no arquivo *sc_simcontext.cpp*. Todas elas executam outras funções internas da mesma classe (sem o uso da palavra reservada *extern*).

```
1. void dr_sc_turn_off(std::string module_name){
2.     sc_get_curr_simcontext()->dr_add_config(module_name);
3. }

4. void dr_sc_turn_on(std::string module_name){
5.     sc_get_curr_simcontext()->dr_remove_config(module_name);
6. }

7. void dr_add_constraint(std::string module_name, int area){
8.     sc_get_curr_simcontext()->dr_addConstraint(module_name,area);
9. }

10. void dr_add_constraint(std::string module_name, int area, sc_time delay){
11.     sc_get_curr_simcontext()->dr_addConstraint(module_name,area,delay);
12. }
```

Figura 4-13: implementação das rotinas apresentadas na Figura 4-12 (arquivo sc_simcontext.cpp)

Quando a rotina *dr_sc_turn_off* é executada (recebendo sempre o nome do módulo em questão como parâmetro) os métodos SystemC (*sc_method*) deste módulo não irão executar até que a rotina inversa, *dr_sc_turn_on* seja executada (com o mesmo nome do módulo desativado anteriormente). Um diagrama de seqüência mostra na Figura 4-14 o que ocorre quando a função *dr_sc_turn_off* é executada.

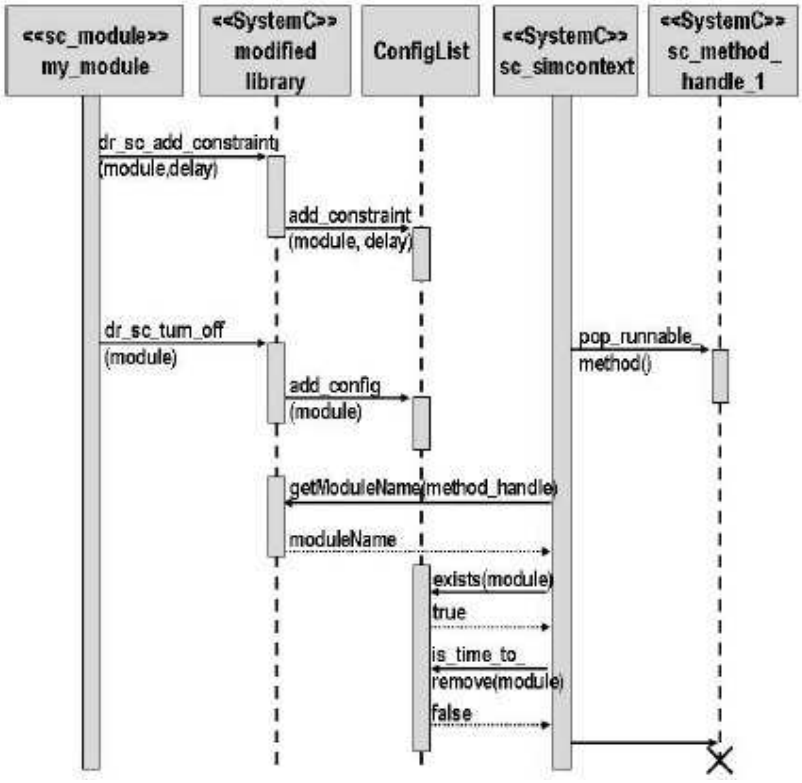


Figura 4-14: diagrama de seqüência que mostra passos para a desativação de um módulo.

Inicialmente os valores de área dos módulos e o atraso necessário para configurá-los são passadas para o simulador através da rotina *dr_sc_add_constraint*. A área refere-se à quantidade de área ocupada por cada módulo no sistema. Esta informação é utilizada para gerar relatórios automaticamente sobre o uso das áreas de chip.

É importante ressaltar que a gerência de área deve ser realizada pelo projetista da simulação, e não pelo simulador. Um módulo gerente de configurações deve ser desenvolvido e manter o controle da área utilizada por cada módulo e da área total ocupada por eles a cada instante de simulação. Essa lógica precisa ser implementada na simulação, da mesma forma que precisará ser implementada no sistema final na FPGA.

Já o atraso de configuração modela o tempo necessário para reescrever as tabelas lógicas do FPGA. Cada módulo pode ter um ou mais métodos SystemC (*sc_method*). Durante cada ciclo, a tabela de escalonamento é checada pelo objeto *sc_simulation_context* utilizando a rotina

pop_runnable_method. Se houver um método que deve ser executado naquele ciclo, um objeto *sc_method_handle* é retornado. Este é responsável por executar o respectivo *sc_method*. Nossa técnica intervém neste estágio, evitando a execução dos métodos. Um objeto chamado *ConfigList* armazena o nome dos módulos (e seus atributos) que não podem ser executados. A rotina *dr_sc_turn_off* dispara a rotina *add_config*, que adiciona o nome do módulo à lista. Uma vez na lista, nenhum método do módulo será executado.

Durante cada ciclo de simulação, antes de executar os métodos dos módulos, o nome do respectivo módulo é buscado na lista encadeada (*ConfigList*). Se ele estiver na lista, sua execução é abortada e a simulação continua normalmente. Do ponto de vista do restante do sistema de simulação, a não execução dos métodos é transparente. Os métodos forçados a não executar comportam-se da mesma forma que outros métodos que não possuem tarefas a realizar.

A seqüência de execução para a rotina *dr_sc_turn_on* é mostrada na Figura 4-15. Inicialmente a rotina *request_remove* da classe *ConfigList* é disparada. Ela sinaliza que o módulo pode ser executado novamente. Entretanto, ela será removida da lista apenas depois do atraso de configuração passado anteriormente pela rotina *dr_sc_add_constraint*. O tempo de execução é checado com a consulta à rotina *is_time_to_remove*. Ela retorna "verdadeiro” se o tempo entre a chamada da rotina *request_remove* e *is_time_to_remove* for maior, ou igual ao atraso de configuração do módulo.

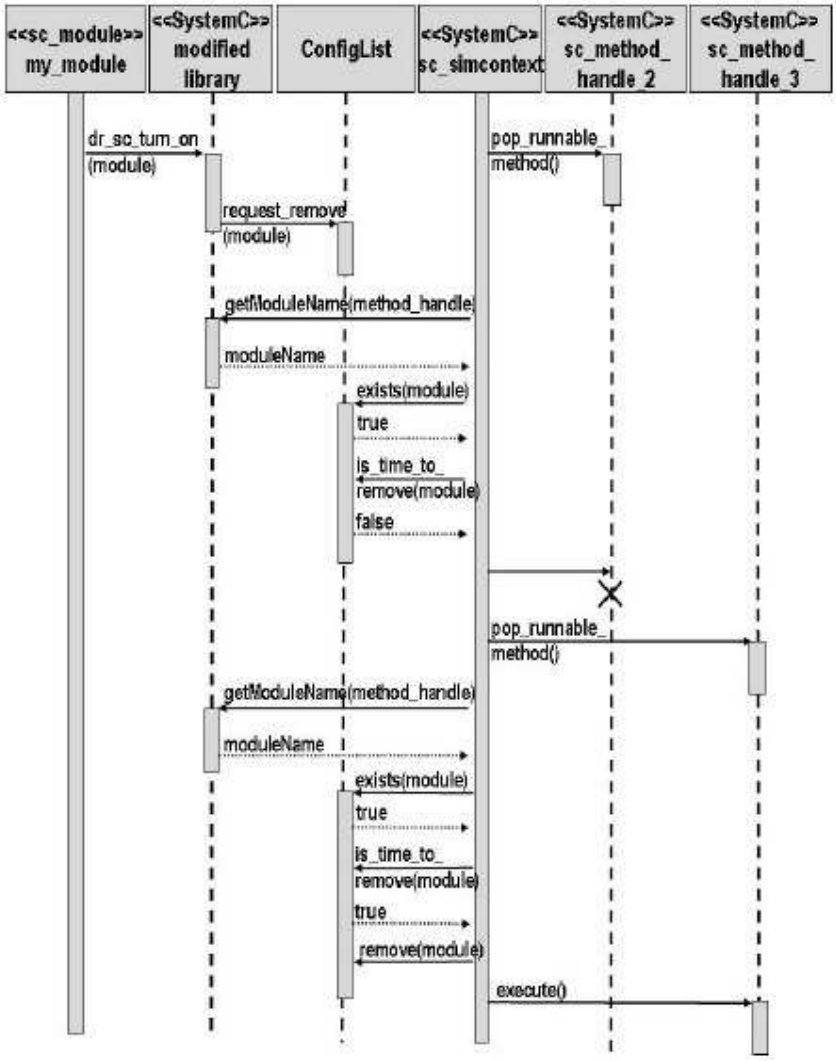


Figura 4-15: diagrama de seqüência que mostra passos para a ativação de um módulo.

As funções apresentadas são detalhadas no código-fonte da Figura 4-16. Como visto anteriormente, elas são chamadas pelas rotinas mostradas na Figura 4-13. A rotina *dr_add_config* recebe o nome do módulo como parâmetro e adiciona-o, juntamente com o tempo atual de simulação (*m_curr_time*) e o atraso de reconfiguração, à lista *ConfigList*. Como mencionado, a partir deste instante, o módulo referenciado não terá mais seus métodos executados pelo escalonador do SystemC.

```
1. void sc_simcontext::dr_add_config(std::string module_name){
2.    configList->addConfig(module_name,m_curr_time,0,constraintList->getReconfDelay(module_name));
3. }

4. void sc_simcontext::dr_remove_config(std::string module_name){
5.    sc_time delay = configList->getReconfDelay(module_name);
6.    if(delay > sc_time(0,SC_NS)){
7.       configList->setActionTime(module_name,delay + m_curr_time);
8.       configList->request_remove(module_name, true);
9.    }
10.    else
11.      configList->removeConfig(module_name);
12.   }

13. void sc_simcontext::dr_addConstraint(std::string module_name, int area){
14.    constraintList->addConfig(module_name,m_curr_time,area);
15. }

16. void sc_simcontext::dr_addConstraint(std::string module_name, int area, sc_time reconfDelay){
17.    constraintList->addConfig(module_name,m_curr_time,area,reconfDelay);
18. }
```

Figura 4-16: implementação das rotinas internas mostradas na Figura 4-13.

As implementações da rotina *dr_add_constraint* vistas nas linhas 13 e 16, relacionam os atributos área e atraso de configuração ao nome do módulo. Tais parâmetros são adicionados à outra lista, chamada *ConstraintList*, não apresentada nos diagramas de seqüência por motivo de simplificação dos mesmos. A rotina *dr_remove_config* é mostrada na linha 4. Aqui também são apresentados mais detalhes que foram omitidos dos diagramas de seqüência. Na linha 5, é recebido da lista *ConfigList* qual o atraso de reconfiguração para o respectivo módulo (variável *delay*). Logo em seguida, seu valor é testado, caso seja nulo, significa que não há atraso, e o nome do módulo é removido imediatamente da lista na linha 11. Caso contrário, na linha 7, é armazenado na lista *ConfigList*, a partir de que momento o módulo poderá voltar a executar (rotina *setActionTime*). Tal momento é calculo somando-se o tempo de simulação atual (*m_curr_time*) com o atraso de reconfiguração (*delay*). Na linha 8, a rotina *request_remove* marca com *true* que o módulo poderá ser removido da lista após o tempo decorrido. Nos diagramas de seqüência este processo é retomado apenas pela rotina *request_remove* que faria o papel de ambas.

Após montada toda a infra-estrutura necessária para armazenar os dados dos módulos que não devem mais executar, só resta modificar a estrutura interna de execução dos métodos SystemC. Esta estrutura é implementada também na classe *sc_simcontext*, mais especificamente na

rotina *crunch*. O trecho modificado desta função é apresentado na Figura 4-17. Inicialmente, na linha 3, a rotina *pop_runnable_method* retorna o tratador de método (*sc_method_handle*) do próximo método a ser executado na simulação. Esta rotina encapsula a lógica do escalonador para escolha dos métodos que devem ser executados. Nos códigos modificados dois objetivos são almejados, evitar a execução de métodos da lista *ConfigList*, e armazenar num arquivo de histórico, quando e quais módulos executam ou deixam de executar. O objeto *fout* é ligado diretamente ao arquivo de histórico, que imprime diretamente para ele. Quando um método é bloqueado, um "X" é impresso (linhas 18 e 20), e quando ele é executado, a área do módulo respectivo é impressa (linhas 15 e 24).

Os três comandos condicionais alinhados das linhas 10, 11 e 12 fazem a verificação se o módulo deve ou não ser executado. Primeiramente, é verificado se o nome do módulo pertence à lista *ConfigList* (linha 10). Em seguida é verificado se o método *request_remove* foi executado para o módulo em questão (linha 11) e, finalmente, é verificado se já foi passado o tempo do atraso de reconfiguração do módulo (linha 12). Caso essas três condições sejam satisfeitas, o módulo é removido da lista *ConfigList* (linha 13), o método é realmente executado pelo comando execute (linha 14), e a área do módulo é armazenada no arquivo de histórico (linha 15). Caso uma das condições não seja verdadeira, um "X" é impresso no arquivo de histórico.

```
1.  while( true ) {

2.    // execute method processes
3.    sc_method_handle method_h = pop_runnable_method();
4.    while( method_h != 0 ) {
5.       try {
6.          if(m_curr_time > sc_time(0,SC_NS)){
7.             str += get_method_name(method_h);
8.             str += ";";
9.          }

10.         if(configList->exists(get_method_name(method_h))){
11.            if(configList->isOff(get_method_name(method_h)))
12.               if(configList->getActionTime(get_method_name(method_h)) <= m_curr_time){
13.                 configList->removeConfig(get_method_name(method_h));
14.                 method_h->execute();
15.                 fout << constraintList->getArea(get_method_name(method_h)) << ";";
16.              }
17.              else
18.                fout << "X;";
19.           else
20.             fout << "X;";
21.           }
22.        else{
23.           method_h->execute();
24.           fout << constraintList->getArea(get_method_name(method_h)) << ";";
25.        }
26.    }
27.    catch( const sc_report& ex ) {
28.       ::std::cout << "\n" << ex.what() << ::std::endl;
29.       m_error = true;
30.       return;
31.    }
32.    method_h = pop_runnable_method();
33. }
```

Figura 4-17: rotina *crunch*, responsável por executar todos os métodos das simulações.

4.5. Estimativa de Área e Atraso de Reconfiguração

Tanto os valores de área, quanto o atraso de reconfiguração para cada módulo devem ser calculados de antemão e passados para o simulador. Esses só podem ser obtidos com precisão através da programação do sistema em hardware. Depois disso, esses valores podem ser utilizados para realimentar o simulador, fazendo com que o mesmo se torne mais calibrado e cada vez mais próximo da realidade.

Em casos de sistemas que estejam sendo simulados pela primeira vez, sem nenhum número sobre sua implementação no chip, apenas poucas conclusões podem ser tiradas da simulação. Como não há dados sobre o sistema real, pode-se num primeiro momento adotar que as áreas e os atrasos de reconfiguração são os mínimos possíveis, fazendo que todos os módulos possam ser configurados ao mesmo tempo e sem atraso de reconfiguração. Dessa forma pode-se primeiramente entender como o

sistema se comporta em termos de funcionalidade caso de alguns de seus módulos sejam desconfigurados do sistema. Se houver conclusões que indiquem que partes do sistema podem realmente ser particionadas no tempo, o ciclo de projeto pode prosseguir para o desenvolvimento com reconfiguração dinâmica.

De forma análoga, os valores iniciais podem ser adotados de forma hipotética pensando no pior caso, onde as áreas e os atrasos de reconfiguração dos módulos são muito grandes, ou máximos. Dessa forma pode-se também os resultados obtidos mostrarão os requisitos de hardware máximos que o sistema exigirá para ser desenvolvido de forma satisfatória.

Uma vez que o desenvolvimento do sistema prossegue até o hardware e dados reais podem ser obtidos (mesmo que a reconfiguração dinâmica não tenha sido ainda implementada), o simulador pode ser realimentado e as simulações podem ser repetidas agora com dados mais concretos, mostrando limitações no uso de área dos módulos e no uso de sucessivas reconfigurações no tempo. Os dados podem ser comparados e a arquitetura final do sistema pode ser selecionada.

A granularidade do sistema pode finalmente ser estimada. Que granularidade melhor se adéqua a um determinado sistema? Esses dados sobre a execução do sistema em hardware podem auxiliar nessa escolha. Como as simulações podem ser facilmente modificadas e rapidamente simuladas, resultados diversos podem ser obtidos considerando características de diversos processadores em diversas granularidades.

Supondo que se deseja desenvolver um sistema utilizando reconfiguração dinâmica e não se sabe em que granularidade isso deve ser feito. O primeiro passo a ser tomado é desenvolver o ambiente de simulação do mesmo. O que deve ser feito em todo caso, mesmo que as reconfigurações dinâmicas não sejam consideradas. O segundo passo seria a determinação de que módulos poderiam ser desconfigurados do sistema em tempo de execução. Isso pode ser obtido mesmo desconhecendo quanto de área cada módulo ocupa, ou qual o tempo necessário para reconfigurá-los. A seguir será apresentado nesse capítulo como isso pode ser feito

através do armazenamento dos históricos de execução de cada módulo do sistema. Conhecendo os módulos que podem ser desconfigurado, o terceiro passo é saber quanto de área ocupa e qual o tempo de reconfiguração necessário para cada módulo em cada hardware candidato a utilização. Esses valores podem ser obtidos pela implementação do sistema no hardware alvo, ou através de estimativas baseadas em implementações anteriores de módulos iguais ou semelhantes aos módulos estudados. Com os valores em mãos, o último passo seria a realimentação do simulador para novas estimativas e comparações entre as diversas arquiteturas estudadas.

4.6. Prova do Conceito

Como prova do conceito, um simples exemplo é apresentado na Figura 4-19 e Figura 4-20, explicando o uso prático das rotinas implementadas e apresentadas na seção anterior. Ele ilustra o controle de configuração de um sistema simples implementado no nível RTL do SystemC e contendo apenas dois módulos (*moduleA* e *moduleB*). Ver Figura 4-18.

```
#include "systemc.h"

class moduleA : public sc_module {
public:
     sc_in_clk clock;
     sc_out<bool> out;
     sc_in<bool> in;

  SC_HAS_PROCESS(moduleA);

     moduleA(sc_module_name name) : sc_module(name){

     SC_METHOD(main_action);
     sensitive << clock;
  }

     void main_action();
};
```

```
#include "moduleA.h"

void moduleA::main_action(){
     out = !in;
}
```

a)

```
#include "systemc.h"

class moduleB : sc_module{
public:
     sc_in_clk clock;
     sc_out<bool> out;
     sc_in<bool> in;

     SC_HAS_PROCESS(moduleB);

     moduleB(sc_module_name name) : sc_module(name){
          SC_METHOD(main_action);
          sensitive_pos << clock;
     }
     void main_action();
private:
     bool internal_value;
};
```

```
#include "moduleB.h"

void moduleB::main_action(){
     out = internal_value;

     if(internal_value)
          internal_value = false;
     else
          internal_value = true;

     cout << "B.in:"<< internal_value << endl ;

}
```

b)

Figura 4-18: código fonte do módulo A (a) e do módulo B (b) na prova do conceito.

Utilizando as rotinas *dr_sc_turn_on* e *dr_sc_turn_off,* a idéia é fazer com

que os módulos trabalhem sempre em momentos distintos, seqüenciais e não paralelos. Primeiramente, apenas o módulo “A” é configurado e depois apenas o “B”. No contexto das simulações, apenas um módulo estará presente no sistema em cada instante.

```
#include <systemc.h>
SC_MODULE (ConfigurationManager){
 sc_in_clk<bool> input_clock;
 SC_CTOR(ConfigurationManager){
  SC_METHOD(config_action);
  sensitive << input_clock;
  dr_sc_add_constraint("moduleA",12, sc_time(6,SC_NS));
  dr_sc_add_constraint("moduleB",15, sc_time(10,SC_NS));
 }
 void config_action(){
  sc_time current = sc_time_stamp();
   if(current == sc_time(50,SC_NS)){
          dr_sc_turn_off("moduleB");
  }
   if(current == sc_time(100,SC_NS)){
          dr_sc_turn_off("moduleA");
            dr_sc_turn_on("moduleB");
  }
  if(current == sc_time(150,SC_NS)){
          dr_sc_turn_off("moduleB");
          dr_sc_turn_on("moduleA");
   } } };
```

Figura 4-19: código-fonte de *ConfigurationManager* na prova do conceito.

As características dos módulos A e B são passadas no construtor da classe por parâmetro. Eles ocupam, respectivamente, 12 e 15 unidades de área e necessitam de 6 e 10 nanossegundos para serem reconfigurados (atraso de reconfiguração). Esses números são hipotéticos e sua determinação depende da programação na FPGA. O método *config_action* é chamado a cada ciclo de relógio (a cada 5 nanossegundos). No instante 50ns, o módulo B é desativado, permanecendo apenas o módulo A. No instante 100ns, ocorre o inverso, o módulo A é desativado e o módulo B é reconfigurado. Tal comportamento pode tanto ser observado na Figura 4-19, quanto na Figura 4-20, onde os sinais de onda gerados na saída da simulação.

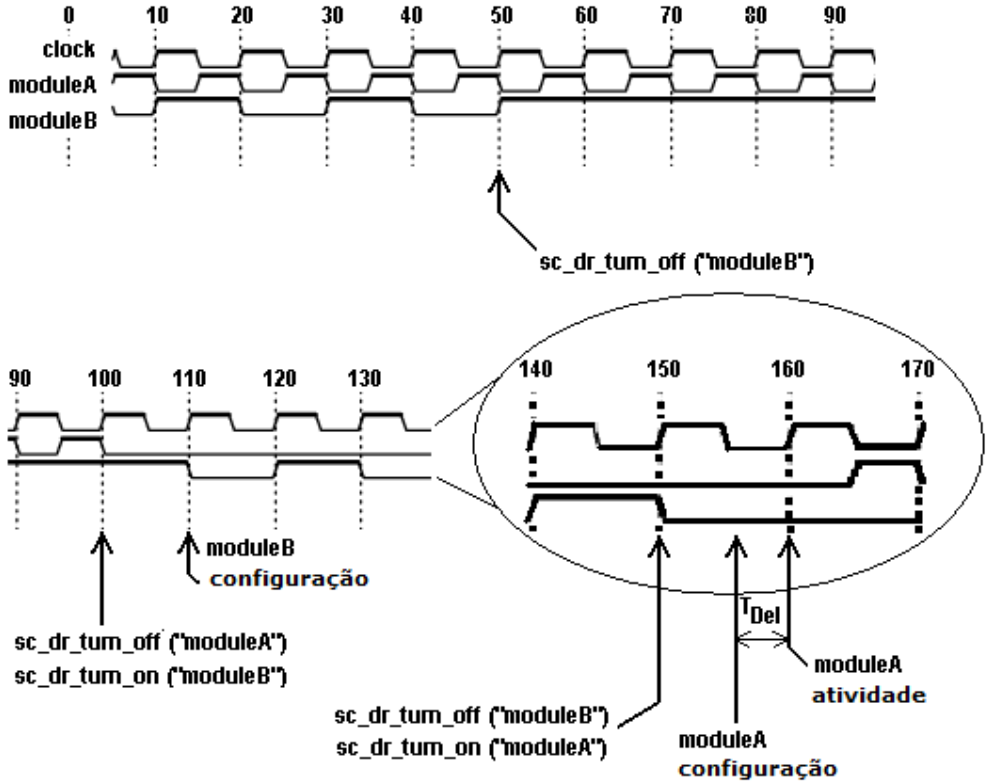


Figura 4-20: forma de onda do exemplo de prova do conceito.

Na Figura 4-20, são apresentados detalhes das reconfigurações em tempo de execução. Observa-se que não há mudança no sinal de saída do módulo B a partir do instante 50ns, nem no sinal do módulo A, a partir de 100ns. Com isso o processo de desconfiguração é imediato. Já o processo de configuração depende do atraso de configuração de cada módulo. Para o módulo B, o sinal de configuração foi dado no instante 100ns, mas ele apenas volta à sua atividade normal no instante 110ns, já que seu atraso de reconfiguração é de 10ns. Já o módulo A, que possui um atraso de reconfiguração de 6ns, o comportamento é menos direto. O sinal de reconfiguração foi passado no instante 150ns, mas sua atividade foi presenciada apenas no instante 160ns. A partir do instante 156ns, o módulo A estaria habilitado a processar dados da entrada, mas como o período de relógio é de 10ns, ele pôde processar os dados da entrada apenas no instante 160ns, após a queda do sinal de relógio. Estes 4ns de diferença entre o tempo o instante de configuração e o tempo de atividade podem confundir o projetista do sistema, mas não devem ser considerados como imprecisão do mecanismo de reconfiguração dinâmica.

Este exemplo apresenta dois importantes papéis que merecem ser comentados. Ele prova que realmente os módulos comportam-se como se estivessem sendo configurados em tempo de execução (ou de simulação). Isso pode ser validado especialmente através da visualização das formas de onda geradas durante a simulação. Na seção seguir, uma forma mais simples de análise das reconfigurações através dos históricos das execuções dos módulos será apresentada. Outro aspecto importante é observar que ele também prova que é possível simular a reconfiguração dinâmica no mais baixo nível de abstração do SystemC, o RTL (Nível de Transferência entre Registradores – *Register Transfer Level*). Outras técnicas similares de simulação de sistemas dinamicamente reconfiguráveis, como o OSSS+R [34] (dos projetos em andamento, este é o que possui maior número de publicações nos últimos anos), são capazes de simular sistemas dinamicamente reconfiguráveis utilizando SystemC (ou outros simuladores em linguagens de alto nível), mas apenas aqueles sistemas desenvolvidos utilizando níveis mais altos de abstração, como o SystemC TLM.

Nos capítulos seguintes, estudos de caso serão apresentados onde nossa técnica foi utilizada para simular sistemas dinamicamente reconfiguráveis, tanto em nível TLM, quanto em RTL.

4.7. Histórico das Execuções

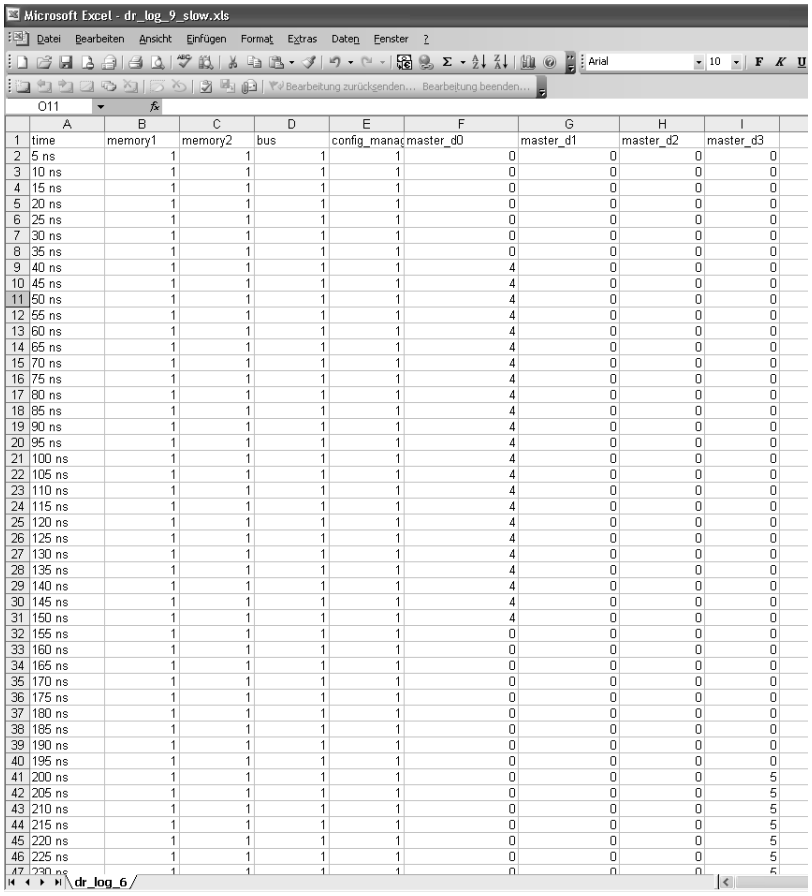
Como detalhado acima na seção “Simulação de Reconfiguração Dinâmica utilizando SystemC”, sempre que uma simulação é executada com a versão modificada do SystemC apresentada aqui, um arquivo de histórico é criado. Um exemplo do arquivo é apresentado na Figura 4-21. Em cada linha são armazenados os tempos de simulação, a seqüência de valores dos módulos e depois a seqüência de nome dos respectivos módulos, tudo em formato CSV (*Comma Separate Value*), ou valores separados por vírgulas. Isso facilita a interpretação automatizada das informações. Planilhas eletrônicas, como o Excel da Microsoft, possuem suporte a tal formato de arquivos.

```
'Log of Dynamically Reconfigured Modules in Simulation'

Time;Module Name;Status

0 s;1;1;1;X;X;X;X;1;1;
5 ns;1;1;1;X;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.m
10 ns;1;1;X;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3;
15 ns;1;1;1;X;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.
20 ns;1;1;X;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3;
25 ns;1;1;1;X;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.
30 ns;1;1;X;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3;
35 ns;1;1;1;X;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.
40 ns;1;1;4;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3;
45 ns;1;1;1;4;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.
50 ns;1;1;4;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3;
55 ns;1;1;1;4;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.
60 ns;1;1;4;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3;
65 ns;1;1;1;4;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.
70 ns;1;1;4;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3;
75 ns;1;1;1;4;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.
80 ns;1;1;4;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3;
85 ns;1;1;1;4;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.
90 ns;1;1;4;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3;
95 ns;1;1;1;4;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.
100 ns;1;1;4;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3
105 ns;1;1;1;4;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top
110 ns;1;1;4;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3
115 ns;1;1;1;4;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top
120 ns;1;1;4;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3
125 ns;1;1;1;4;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top
130 ns;1;1;4;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3
135 ns;1;1;1;4;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top
140 ns;1;1;4;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3
145 ns;1;1;1;4;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top
150 ns;1;1;4;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3
155 ns;1;1;1;X;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top
160 ns;1;1;X;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3
165 ns;1;1;1;X;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top
170 ns;1;1;X;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3
175 ns;1;1;1;X;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top
180 ns;1;1;X;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3
185 ns;1;1;1;X;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top
190 ns;1;1;X;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3
195 ns;1;1;1;X;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top
200 ns;1;1;X;X;X;5;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3
205 ns;1;1;1;X;X;X;5;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top
210 ns;1;1;X;X;X;5;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3
215 ns;1;1;1;X;X;X;5;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top
220 ns;1;1;X;X;X;5;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3
225 ns;1;1;1;X;X;X;5;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top
230 ns;1;1;X;X;X;5;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3
235 ns;1;1;1;X;X;X;5;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top
240 ns;1;1;X;X;X;5;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3
245 ns;1;1;1;X;X;X;5;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top
250 ns;1;1;X;X;X;5;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.master_d2;top.master_d3
255 ns;1;1;1;X;X;X;5;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;top.master_d2;top
```

Figura 4-21: exemplo de um arquivo de histórico de execução.

Um programa Java foi desenvolvido para fazer uma filtragem nos arquivos de histórico de execução e organizá-los para que os mesmos possam ser lidos por planilhas eletrônicas e apresentados de uma forma mais legível, como pode ser visto na Figura 4-22. A filtragem dos dados substitui todas as letras “X” por zero. Também completa os valores dos métodos que não tiveram nenhuma atividade em determinado instante, evitando que a tabela final fique com campos em branco. A tabela resultado pode ser vista como a representação do uso do sistema em termos de área. Os instantes com valor zero para algum módulo representam que o mesmo não estava presente no momento, significando que certa quantidade de área foi

poupada.



Microsoft Excel - dr_log_9_slow.xls

	A	B	C	D	E	F	G	H	I
1	time	memory1	memory2	bus	config_mana	master_d0	master_d1	master_d2	master_d3
2	5 ns	1	1	1	1	0	0	0	0
3	10 ns	1	1	1	1	0	0	0	0
4	15 ns	1	1	1	1	0	0	0	0
5	20 ns	1	1	1	1	0	0	0	0
6	25 ns	1	1	1	1	0	0	0	0
7	30 ns	1	1	1	1	0	0	0	0
8	35 ns	1	1	1	1	0	0	0	0
9	40 ns	1	1	1	1	4	0	0	0
10	45 ns	1	1	1	1	4	0	0	0
11	50 ns	1	1	1	1	4	0	0	0
12	55 ns	1	1	1	1	4	0	0	0
13	60 ns	1	1	1	1	4	0	0	0
14	65 ns	1	1	1	1	4	0	0	0
15	70 ns	1	1	1	1	4	0	0	0
16	75 ns	1	1	1	1	4	0	0	0
17	80 ns	1	1	1	1	4	0	0	0
18	85 ns	1	1	1	1	4	0	0	0
19	90 ns	1	1	1	1	4	0	0	0
20	95 ns	1	1	1	1	4	0	0	0
21	100 ns	1	1	1	1	4	0	0	0
22	105 ns	1	1	1	1	4	0	0	0
23	110 ns	1	1	1	1	4	0	0	0
24	115 ns	1	1	1	1	4	0	0	0
25	120 ns	1	1	1	1	4	0	0	0
26	125 ns	1	1	1	1	4	0	0	0
27	130 ns	1	1	1	1	4	0	0	0
28	135 ns	1	1	1	1	4	0	0	0
29	140 ns	1	1	1	1	4	0	0	0
30	145 ns	1	1	1	1	4	0	0	0
31	150 ns	1	1	1	1	4	0	0	0
32	155 ns	1	1	1	1	0	0	0	0
33	160 ns	1	1	1	1	0	0	0	0
34	165 ns	1	1	1	1	0	0	0	0
35	170 ns	1	1	1	1	0	0	0	0
36	175 ns	1	1	1	1	0	0	0	0
37	180 ns	1	1	1	1	0	0	0	0
38	185 ns	1	1	1	1	0	0	0	0
39	190 ns	1	1	1	1	0	0	0	0
40	195 ns	1	1	1	1	0	0	0	0
41	200 ns	1	1	1	1	0	0	0	5
42	205 ns	1	1	1	1	0	0	0	5
43	210 ns	1	1	1	1	0	0	0	5
44	215 ns	1	1	1	1	0	0	0	5
45	220 ns	1	1	1	1	0	0	0	5
46	225 ns	1	1	1	1	0	0	0	5

Figura 4-22: informações do arquivo de histórico apresentado no Microsoft Excel depois de filtragem e organização.

Com os arquivos filtrados e formatados, as informações podem ser vistas na forma de tabela, o que facilita a interpretação e a geração de gráficos. A Figura 4-23 é um exemplo de um gráfico de uso de área de chip que foi gerado diretamente da tabela gerada de um arquivo de histórico qualquer. Além da análise direta do uso de área de chip, outras informações podem ser tiradas de um gráfico como este, como por exemplo, o nível de paralelismo entre os módulos executados e o impacto dos atrasos de reconfiguração no desempenho geral do sistema. Utilizando o gráfico da Figura 4-23 como exemplo, quanto mais longos forem os espaços vazios entre os módulos no gráfico, menor serão as atividades dos módulos, podendo significar maiores atrasos de reconfiguração, e menos módulos

executados haverá no mesmo intervalo de tempo, significando uma perda na eficiência de utilização de área pelo sistema.

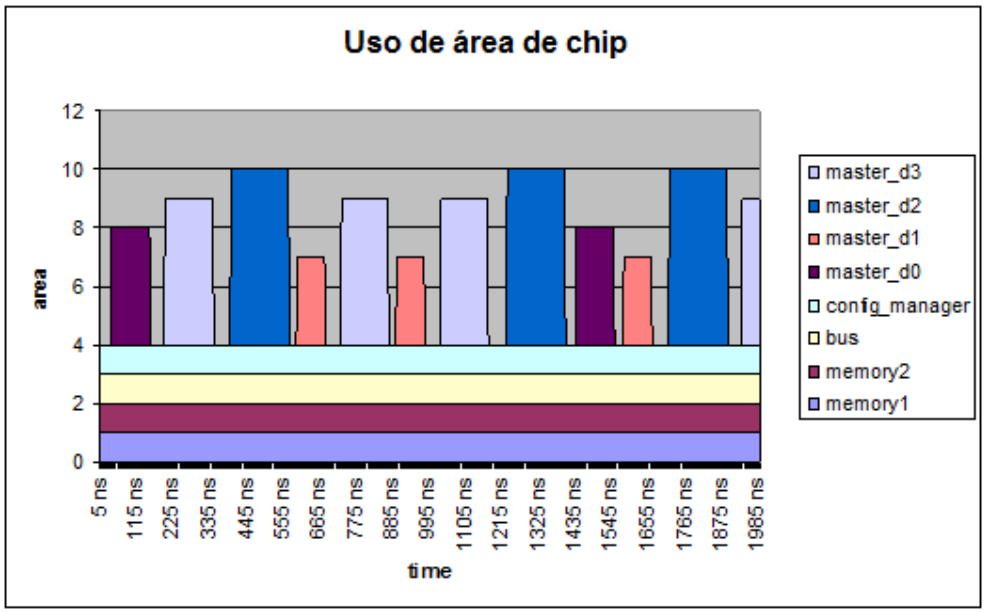


Figura 4-23: gráfico de uso de área de chip gerado a partir de uma tabela de histórico de execução.

Sistemas tradicionais que não utilizam reconfiguração dinâmica podem ter seus históricos de execução analisados e possíveis módulos candidatos a serem removidos do sistema podem ser encontrados. Isso torna a análise dos históricos da execução o ponto inicial para qualquer desenvolvedor que pense em utilizar reconfiguração dinâmica em seus sistemas.

4.8. Desempenho do Simulador

O desempenho do simulador foi testado através da simulação do decodificador MPEG-4 desenvolvido pelo grupo do Laboratório de Arquiteturas Dedicadas (LAD) da Universidade Federal de Campina Grande (UFCG) [91]. Ele foi desenvolvido completamente no laboratório iniciando pelo modelo RTL em SystemC e concluindo com o projeto final em microchip. O decodificador implementa o *Simple Profile Level 0* do padrão MPEG-4. A arquitetura do decodificador compreende o projeto de hardware personalizado para a decodificação do fluxo de bits (bitstream), *Variable Length Code* (VLC), decodificação de textura, compensação de movimento e conversão do espaço de cores. Os experimentos em hardware demonstraram que o decodificador atinge 30 quadros por segundo [91].

No simulador modificado para reconfiguração dinâmica, a cada ciclo de simulação a lista de módulos desconfigurados (*Config_List*) é analisada. Os módulos que estiverem nessa lista são bloqueados e não executados. Isso causa um impacto no tempo de simulação, já que a cada ciclo, toda a lista deve ser percorrida. O objetivo deste experimento foi analisar esse impacto.

Para analisar o desempenho do simulador, um vídeo de 16 quadros foi decodificado em dois momentos distintos. No primeiro momento o SystemC tradicional sem qualquer modificação na versão 2.1.1 foi utilizado. No segundo momento, nosso simulador modificado para simular reconfiguração dinâmica foi utilizado. Em ambos os casos, não houve qualquer utilização das funcionalidades de reconfiguração dinâmica na simulação.

Os resultados constataram que no ambiente simulado, utilizando o MPEG-4 em RTL, o simulador modificado apresentou um desempenho três vezes inferior ao desempenho do simulador tradicional sem modificações. Ver Tabela 4-1. Esse resultado demonstra que as verificações realizadas na lista de módulos desconfigurados a cada ciclo de simulação prejudicam o desempenho do simulador.

Tabela 4-1: Desempenho do SystemC modificado para reconfiguração dinâmica para sistemas em RTL.

Simulador	Tempo de simulação
SystemC tradicional	12m56.630s
SystemC modificado	36m34.334s

Acreditamos que o tempo de simulação elevado é compensado pelo ganho no tempo de projeto, já que a metodologia de simulação que apresentada é capaz de prever o comportamento das reconfigurações dinâmicas de forma antecipada, antes mesmo de o sistema existir em hardware (nesse caso, com restrições de resultados devido à ausência de números reais sobre o sistema funcionando em hardware).

Utilizando Chaveamento Dinâmico de Circuitos (*Dynamic Circuit Switching* - DCS) [21, 22, 23, 91] possa ser que haja uma melhoria no desempenho das simulações, porém na literatura não se encontra referências a isso.

4.9. Prova de Generalidade da Ferramenta

Com o intuito de demonstrar que todos os três tipos de reconfiguração dinâmica definidos na seção 4.2 podem ser modelados e simulados com a ferramenta desenvolvida neste trabalho, as três abordagens correspondentes para o simulador SystemC proposto são apresentadas.

No geral, a metodologia utilizada é sempre a de implementar, declarar e conectar todos módulos SystemC envolvidos na simulação. A Figura 4-24 mostra como deve ser feita a substituição de um módulo C (*sc_module C*) por outro módulo D (*sc_module D*). Neste caso, ambos são instanciados e conectados inicialmente no primeiro cenário, porém o módulo D é desconfigurado assim que a simulação é iniciada. Em seguida, o módulo C é desativado utilizando a rotina de desconfiguração (*dr_sc_turn_off("moduleC")*) e o módulo D é configurado (*dr_sc_turn_on("moduleD")*).

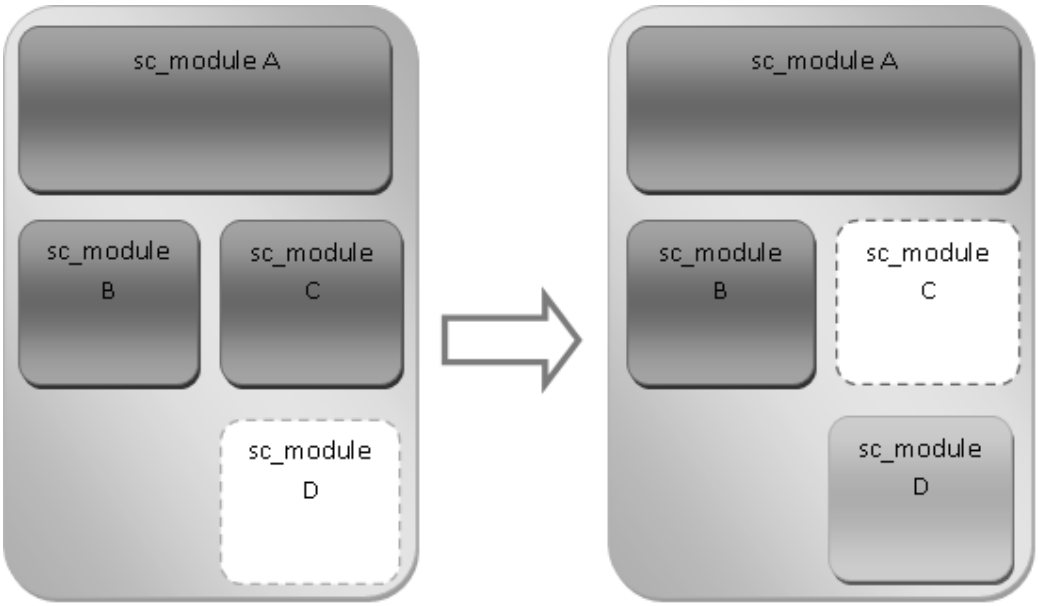


Figura 4-24: implementação do primeiro tipo de reconfiguração dinâmica.

No segundo tipo de reconfiguração dinâmica, um módulo deve ser removido do sistema. Como pode ser visto na Figura 4-25, todos módulos são instanciados e conectados inicialmente. No segundo cenário, o módulo C é desconfigurado (*dr_sc_turn_off("moduleC")*).

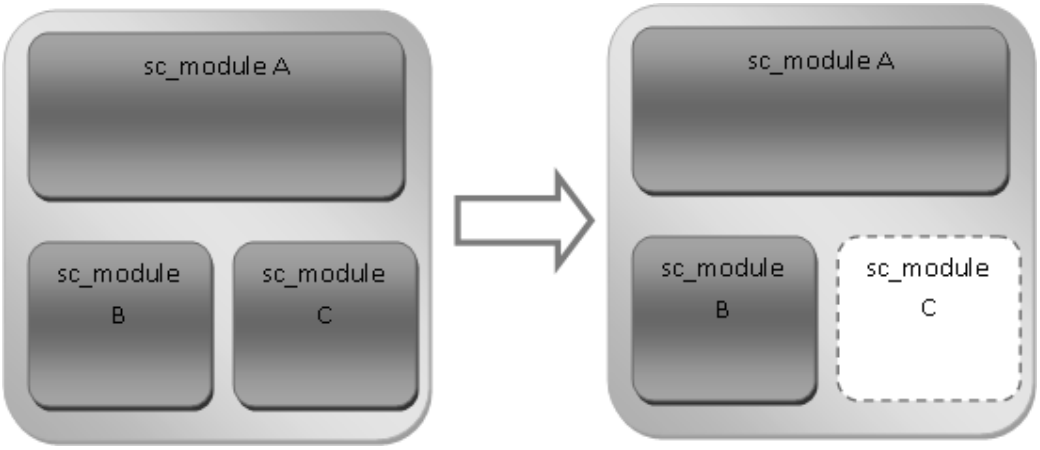


Figura 4-25: implementação do segundo tipo de reconfiguração dinâmica.

Para que o terceiro tipo de reconfiguração dinâmica (ver Figura 4-26) possa ser implementado, é necessário instanciar inicialmente o módulo A completo (*sc_module A*), assim como instanciar e conectar os módulos resultantes de seu particionamento (módulos A‘, A‘‘ e A‘‘‘). Esses são desconfigurados assim que a simulação é iniciada. No segundo cenário os módulos são invertidos. O módulo A completo é desconfigurado e os módulos particionados são configurados. Como eles já foram conectados no primeiro cenário, nada mais precisa ser feito para que eles trabalhem corretamente.

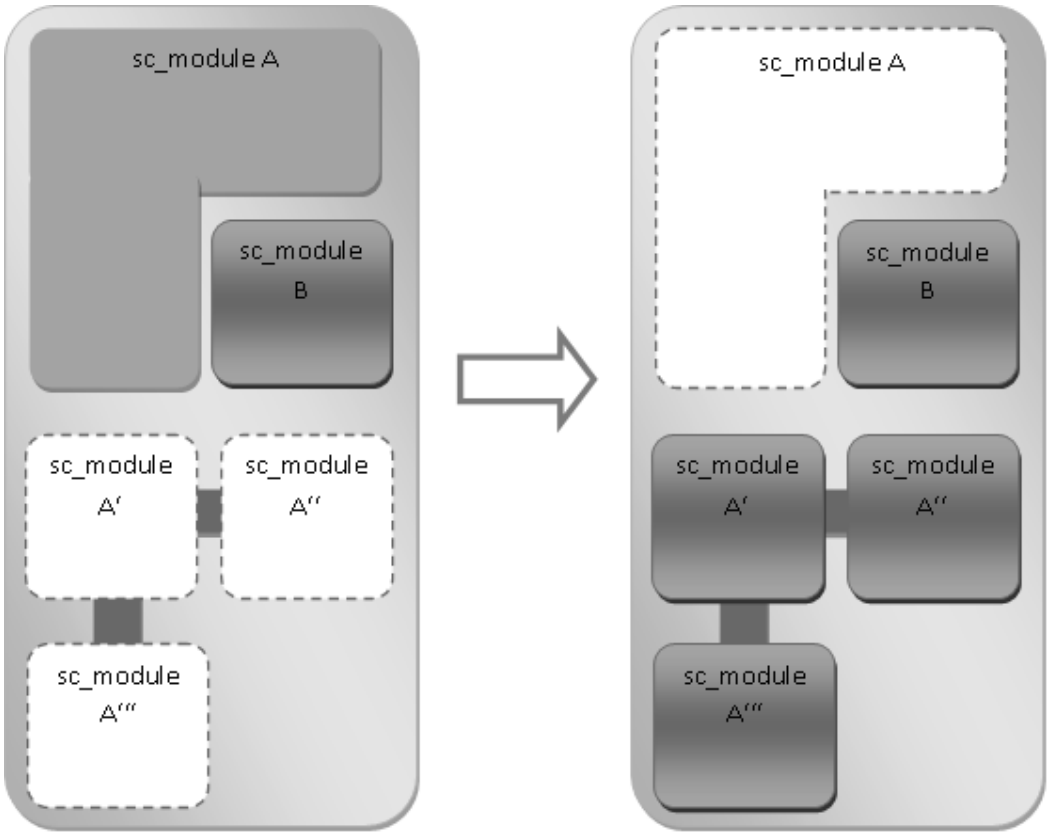


Figura 4-26: implementação do terceiro tipo de reconfiguração dinâmica.

4.10. Mudança de Versão

Com as mudanças de versão no SystemC, novas alterações no código-fonte são necessárias para que sistemas dinamicamente reconfiguráveis possam ser modelados e simulados. As seções a seguir apresentam as diferenças entre as últimas versões do SystemC.

As adaptações que mudam a cada versão são basicamente nos métodos *crunch* e no *get_method_name* responsáveis pela execução dos módulos e pela captura dos nomes dos mesmos, respectivamente (para maiores informações sobre o código-fonte adaptado do SystemC, vide o Anexo C). Nas últimas mudanças de versão, o mecanismo para captura dos nomes dos módulos foi alterado, exigindo que o mesmo fosse modificado. As adaptações de versão foram, em média, desenvolvidas em 2 horas de trabalho, onde maior parte do tempo foi dedicada ao entendimento de que mudanças deveriam ser desempenhadas e de como desempenhá-las.

A Figura 4-27 apresenta como o método *get_method_name* foi

implementado na versão 2.0.1 do SystemC. Neste caso os nomes dos módulos podem ser obtidos através do *handle->module->name* que é chamado num laço que percorre todo o arranjo de métodos executáveis da simulação. Nas versões seguintes esse arranjo foi modificado no SystemC e, conseqüentemente, o laço utilizado para percorrê-lo (ver Figura 4-29 e Figura 4-31).

```
sc_string sc_simcontext::get_method_name(sc_method_handle
handle){
    // prepare all thread processes for simulation
    const sc_method_vec& thread_vec = m_process_table->
method_vec();
    for( int i = thread_vec.size() - 1; i >= 0; -- i ) {
     if(thread_vec[i] == handle){
          return handle->module->name();
     }
    }
    return sc_string("");
}
```

Figura 4-27: método *get_method_name* na versão 2.0.1 do SystemC.

No método *crunch*, a rotina *method_h->execute()* executa as ações de cada módulo do sistema, como pode ser visto na Figura 4-28. As condições foram adicionadas para que as ações executem apenas quando estiverem na lista de módulos não configurados. Como na versão 2.1.1 não houve modificações significantes na forma como os métodos são executados, não foram necessárias grandes modificações no método *crunch*, como pode ser visto na Figura 4-30.

```
void
sc_simcontext::crunch()
{
      ...
    sc_method_handle method_h = pop_runnable_method();
      while( method_h != 0 ) {
        try {
         if(m_curr_time > sc_time(0,SC_NS)){
           str += get_method_name(method_h) + ";";
         }
         if(configList->exists(get_method_name(method_h))){
           if(configList->isOff(get_method_name(method_h)))
             if(configList->getActionTime(get_method_name
method_h)) <= m_curr_time){
              configList->removeConfig(get_method_name
(method_h));
              method_h->execute();
              fout << constraintList->getArea(get_method_name
(method_h)) << ";";

            }
            else
              fout << "0;";
           else
             fout << "0;";
         }
          else{
             method_h->execute();
             fout << constraintList->getArea(get_method_name
(method_h)) << ";";
           }
      }
      ...
}
```

Figura 4-28: método *crunch* na versão 2.0.1 do SystemC

Na versão 2.1.1 do SystemC o nome dos módulos são obtidos também através do método *handle->host->name()* como pode ser visto na Figura 4-29. Como mencionado anteriormente, modificações no laço foram necessárias para que os métodos dos módulos da simulação possam ser encontrados na tabela de processos.

```
std::string sc_simcontext::get_method_name(sc_method_handle handle){
    // prepare all thread processes for simulation
    sc_method_handle method_p;
    for ( method_p = m_process_table->method_q_head(); method_p; method_p
= method_p->next_exist() )
      if(method_p == handle){
            const char* ret;
            ret = handle->host->name();
            return ret;
      }
    return std::string("");
}
```

Figura 4-29: método *get_method_name* na versão 2.1.1 do SystemC.

No método *crunch*, a rotina *method_h->execute()* executa as ações de

cada módulo do sistema, como pode ser visto na Figura 4-30, assim como na versão anterior.

```
inline void
sc_simcontext::crunch(){
      ...
    while( true ) {
      sc_method_handle method_h = pop_runnable_method();
      while( method_h != 0 ) {
        try {
          if(m_curr_time > sc_time(0,SC_NS)){
            str += get_method_name(method_h);
            str += ";";
          }
          if(configList->exists(get_method_name(method_h))){
            if(configList->isOff(get_method_name(method_h)))
              if(configList->getActionTime(get_method_name(method_h))
<= m_curr_time){
                configList->removeConfig(get_method_name(method_h));
                method_h->execute();
                fout << constraintList-
>getArea(get_method_name(method_h)) << ";";
              }
                else
                  fout << "X;";
            else
                fout << "X;";
          }
          else{
            method_h->execute();
            fout << constraintList-
>getArea(get_method_name(method_h)) << ";";
          }
      ...
      }
      ...
}
```

Figura 4-30: método *crunch* na versão 2.1.1 do SystemC.

Já na versão 2.2 do SystemC, os nomes dos módulos são obtidos através do método *handle->name*, diferentemente da versão anterior. A modificação efetuada no método *get_method_name* pode ser vista na Figura 4-31 abaixo.

```
std::string sc_simcontext::get_method_name(sc_method_handle handle){
    // prepare all thread processes for simulation
    sc_method_handle method_p;
    for (method_p = m_process_table->method_q_head(); method_p; method_p
= method_p->next_exist() )
      if(method_p == handle){
            const char* ret;
            ret = handle->name();
            return ret;
      }
    return std::string("");

}
```

Figura 4-31: método *get_method_name* na versão 2.2 do SystemC.

No método *crunch*, a rotina que executa as ações de cada módulo do

sistema foi alterada para *method_h->semantics,* como pode ser visto na Figura 4-32 abaixo.

```
inline void
sc_simcontext::crunch(){
      ...
sc_method_handle method_h = pop_runnable_method();
  while( method_h != 0 ) {
    try {
      if(m_curr_time > sc_time(0,SC_NS)){
        str += get_method_name(method_h);
        str += ";";
      }
        if(configList->exists(get_method_name(method_h))){
          if(configList->isOff(get_method_name(method_h)))
            if(configList->getActionTime(get_method_name(method_h))
<= m_curr_time){
              configList->removeConfig(get_method_name(method_h));
              method_h->semantics();
              fout << constraintList-
>getArea(get_method_name(method_h)) << ";";

            }
            else
              fout << "X;";
         else
                        fout << "X;";
      }
      else{
        method_h->semantics();
        fout << constraintList->getArea(get_method_name(method_h)) <<
";";
      }
   ...
}
```

Figura 4-32: método *crunch* na versão 2.2 do SystemC.

4.11. Resumo do Capítulo

A simulação da reconfiguração dinâmica através da simples modificação de poucas instruções do SystemC mostra como numa abordagem *bottom-up* uma pequena modificação pode resultar em grandes possibilidades. O mesmo problema é atacado em outras abordagens [34] pelas camadas superiores do SystemC, implementando novas classes base variantes dos módulos (*sc_module*) e interfaces TLM para comunicação entre módulos. Isso se torna uma limitação, já que possibilita a simulação apenas de sistemas TLM (sistemas RTL não utilizam interfaces TLM para comunicação entre os módulos).

Outra importante ferramenta apresentada é a de Histórico de Execuções. Inicialmente em nossas pesquisas procuramos primeiramente monitorar as execuções dos módulos, a fim de detectar potenciais aplicações da reconfiguração dinâmica (ver Apêndice B). Monitorando simulações desta

forma nos permite não apenas aplicar a reconfiguração dinâmica em módulos de execução menos freqüente, mas também nos permite um melhor conhecimento de nosso próprio sistema, possibilitando-nos encontrar falhas de projeto (ou modelagem) e inferir consumo de energia (quando provido dos valores de consumo por operação de cada módulo).

Capítulo 5

Resultados

5.1. Introdução

Utilizando a técnica de modelagem e simulação de reconfiguração dinâmica, dois estudos de caso foram desenvolvidos e são apresentados aqui. O primeiro estudo de caso apresenta uma aplicação real criada na Universidade de *Karlsruhe* (Alemanha) para a empresa Daimler-Crysler. Este estudo de caso é ideal para provar que a ferramenta é bastante adequada para simular sistemas fim, de granularidade grossa e implementados utilizando o nível de abstração TLM do SystemC. Os resultados deste trabalho foram publicados no *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'2007)* em Porto Alegre [83]

O segundo estudo de caso apresenta um sistema meio, onde a técnica de simulação de reconfiguração dinâmica foi utilizada para o desenvolvimento de um simulador de processadores dinamicamente reconfiguráveis. Tal simulador pode ser configurado para emular processadores quaisquer e ser integrado com outros simuladores, a fim de formar uma plataforma de simulação completa. Este exemplo também demonstra a simulação de sistemas utilizando o nível de abstração TLM do SystemC. Os detalhes do desenvolvimento e os resultados adquiridos neste trabalho foram descritos num artigo científico apresentado no *Reconfigurable Architectures Workshop* em *Long Beach* na Califórnia [84]

Como apresentou o exemplo para prova de conceito no Capítulo 4, a técnica para simulação e modelagem de reconfiguração dinâmica também é viável para sistemas de granularidade fina e utilizando o nível RTL de abstração do SystemC.

5.2. Estudos de caso 1: aplicação automotiva

O advento de sistemas dinamicamente e parcialmente reconfiguráveis permite que sistemas de hardware tenham suas configurações modificadas em tempo de execução e, no caso dos sistemas parcialmente reconfiguráveis, isso pode ocorrer sem que as partes não afetadas do sistema parem sua execução normal. A reconfiguração dinâmica, genericamente falando, permite o desenvolvimento de sistemas de hardware ainda mais flexíveis, heterogêneos, eficientes e econômicos. Em contrapartida, modificar configurações em tempo de execução requer um processamento relativamente alto, chamado tempo de reconfiguração. Altos tempos de reconfiguração podem anular os benefícios adquiridos em termos de aproveitamento de área de chip e energia poupada. Conhecer a relação entre o custo das reconfigurações e seus benefícios é uma tarefa que pode não ser trivial, principalmente em sistemas complexos. O objetivo desse trabalho é adicionar a sistemas SystemC a capacidade de modelar e simular sistemas dinamicamente reconfiguráveis. Com isso, o comportamento de tais sistemas pode ser analisado com mais antecedência, agilizando o fluxo de desenvolvimento e aumentando a predição do sistema.

5.2.1. Sistema alvo modelado e simulado

Com o objetivo de validar a técnica desenvolvida, uma aplicação real automotiva já funcionando em hardware foi modelada e simulada [2,3] (ver Figura 5-33). O sistema automotivo é dinamicamente a parcialmente reconfigurável. Ele possibilita alocar funções internas da cabine do automóvel sob demanda, tais como, controle de abertura das janelas, posição das poltronas e posição do espelho central.

No sistema automotivo, quatro aplicações da cabine interna podem ser configuradas, controle de abertura das janelas (quatro instâncias), controle da posição das poltronas do motorista e passageiro dianteiro (duas instâncias), acendimento dos faróis (uma instância) e posicionamento do espelho central (uma instância), o que resulta num total de 8 possíveis instâncias. Se o usuário requisita certo serviço, a unidade de hardware correspondente é reconfigurada e iniciada numa posição desocupada da área dinamicamente

reconfigurável do sistema de FPGA.

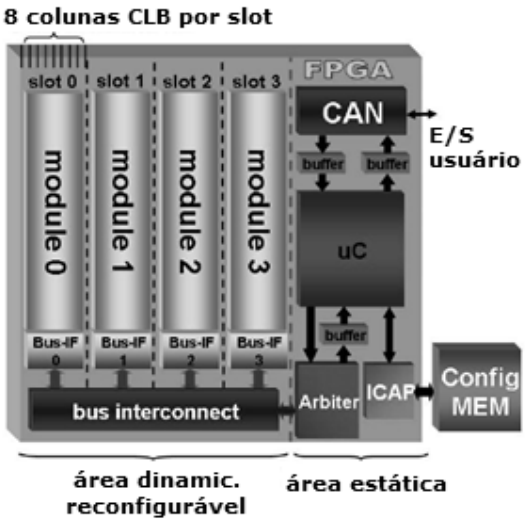


Figura 5-33: arquitetura da aplicação automotiva modelada.

5.2.2. Simulação do sistema

Com o objetivo de avaliar o desempenho, o sistema automotivo foi simulado utilizando diferentes quantidades de área física dentro da parte dinamicamente reconfigurável. O sistema atualmente funciona em hardware com 32 colunas CLBs, como cada espaço (módulo), onde cada aplicação é localizada ocupa 8 colunas (ver Tabela 5-2), até no máximo 4 instâncias podem ser executadas em paralelo, necessitando de reconfiguração dinâmica para reconfigurar as outras instâncias quando solicitadas. Tamanhos hipotéticos de FPGA foram assumidos com objetivo de comparar diferentes opções de projeto que poderiam ter sido tomadas, tais como 8, 16, 24, 32, 40 e 64 colunas CLBs. Com 8 colunas apenas uma instância de aplicação pode ser executada, com 16 colunas, duas instâncias, com 24, três instâncias e assim sucessivamente. Objetivando uma comparação, o sistema foi simulado com 64 colunas, situação em que todas as instâncias poderiam ser configuradas simultaneamente e a reconfiguração dinâmica não seria mais necessária.

Todos os números utilizados para esse experimento, mencionados no texto e apresentados na Tabela 5-2 foram obtidos através de testes reais do sistema funcionando em silício (hardware). Nessa o número de colunas (igual para todas aplicações) representa quando de área de FPGA cada aplicação

ocupa. O tempo de operação determina quanto tempo, no máximo, cada aplicação pode ficar em execução até o cumprimento de sua tarefa. Por exemplo, a abertura completa de uma janela pode durar no máximo 5 segundos, isso se a mesma estiver completamente fechada. Já o retrovisor e os faróis possuem tempos de operação menores. Durante a simulação esses tempos foram selecionados randomicamente respeitando o intervalo estipulado para cada aplicação. Já o atraso de reconfiguração representa o tempo necessário para configurar uma instância de cada aplicação disponível. Esse tempo é igual para todas aplicações, já que a área ocupada pelas mesmas também é a mesma.

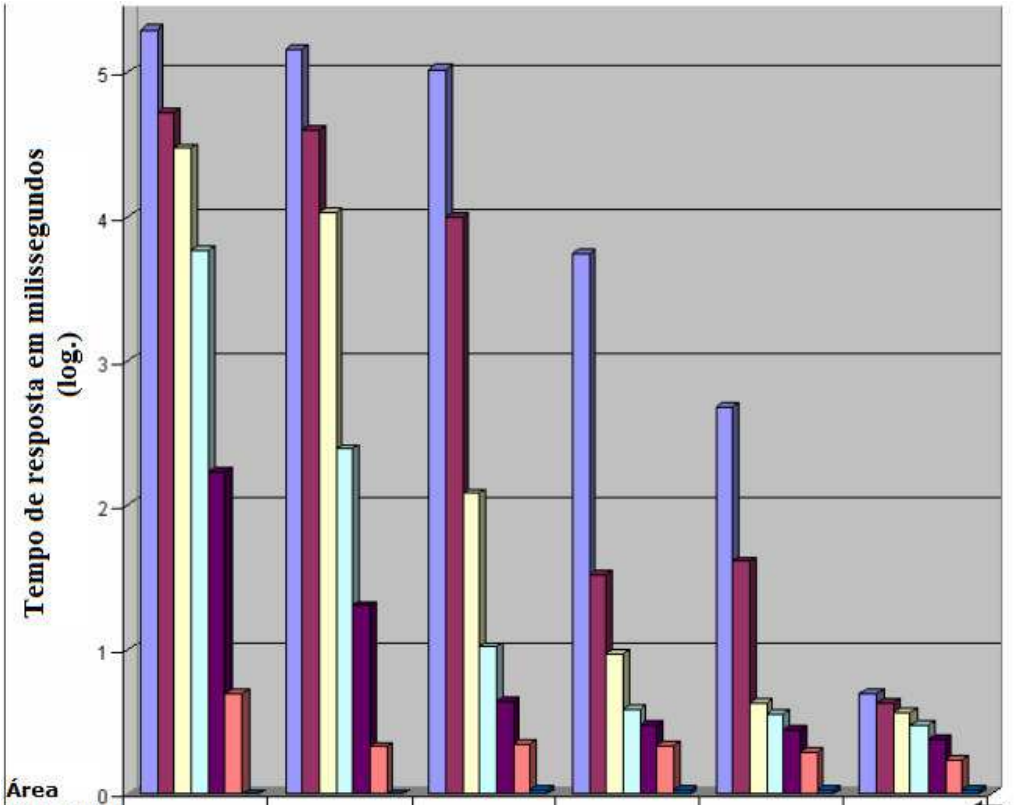
Tabela 5-2: aplicações da cabine interna do automóvel.

Aplicação	Número de colunas (área)	Tempo de operação (seg.)	Atraso de configuração
Janelas	8	0<t<5	2,96ms
Assentos	8	0<t<5	2,96ms
Retrovisor	8	0<t<3	2,96ms
Faróis	8	0<t<1	2,96ms

Ambos, o melhor e o pior caso, podem ser realizados com o objetivo de se obter o consumo de área e desempenho. A idéia é analisar como o sistema responde a sucessivas requisições do usuário quando diferentes áreas configuráveis são disponíveis. Tais resultados são comparados com uma implementação completamente paralela que chamada de “ideal”, por possuir 64 colunas CLBs e, portanto, não necessitar de reconfiguração dinâmica para sua execução (ver Figura 5-34).

Na Figura 5-34 é apresentado o tempo de resposta que o sistema é capaz de fornecer para cada aplicação requisitada. Esse tempo é calculado contando a partir do momento em que o usuário requisita uma instância de uma determinada aplicação até o momento em que a mesma está pronta para ser utilizada. Por exemplo, o tempo de resposta será de 300ms se esse for o tempo entre o clique no botão para baixar a janela do carro e o momento em que a mesma começar efetivamente a ser baixada. De acordo com as especificações que foram obtidas através de experimentos do sistema no silício e nos passada, o tempo de resposta tolerável é de até 100ms. Acima disso o atraso torna-se perceptível ao usuário e não desejado

no automóvel. Na Figura 5-34 esse limite situa-se na linha do número 2, já o gráfico foi gerado em escala logarítmica. Acima desse limite as respostas podem ser consideradas atrasadas.



Área Disponível (Colunas)	3	2	1	0.5	0.33	0.17
8	200734	145155	106164	5628,11	479,98	4,965
16	53320,2	40572,3	10097,7	32,94	40,78	4,32
24	29859,6	10783,6	121,82	9,415	4,295	3,655
32	5799,29	246,02	10,43	3,81	3,57	3,01
40	172,245	20,025	4,43	3,005	2,785	2,365
48	4,9	2,115	2,24	2,16	1,94	1,72
64 (ideal)	0,73	0,875	1,06	1,06	1,06	1,06

Requisições por segundo

Figura 5-34: tempo de resposta do sistema em milissegundos para diferentes áreas reconfiguráveis disponíveis.

Os resultados apresentados mostram que o sistema atual, com 32 colunas CLBs, responde bem quando as requisições são de até uma requisição por segundo. Requisições mais freqüentes do que isso, geraram atrasos nas respostas. Aumentando na simulação a área do chip para 40 colunas CLBs, suportando, então, até 5 aplicações simultâneas, o sistema responde a tempo até a 2 requisições por segundo. Acima disso, com 48 colunas CLBs, o sistema reponde no tempo limite para todas as freqüências de requisições simuladas.

A partir dos experimentos realizados pode-se concluir que o tempo de resposta obtido para o sistema é satisfatório para situações corriqueiras, onde não há excesso de requisições. Também mostra que o aumento de área em mais 8 colunas CLBs traria um desempenho maior, mesmo quando as requisições chegam a duas por segundo, número consideravelmente alto. Para uma maior garantia da qualidade da resposta, o sistema poderia utilizar 48 colunas CLBs, que respondeu em tempo hábil a todas requisições geradas na simulação.

A Figura 5-35 apresenta um resumo dos resultados exibindo a porcentagem de requisições que são respondidas atrasadamente para diferentes freqüências de requisições. Como dito anteriormente, o sistema com 32 colunas CLBs apresenta atrasos em muitos casos simulados. Cerca de 40% das requisições são respondidas atrasadas quando duas requisições por segundo são efetuadas. Já com 40 colunas esse número é melhor, menos de 10% das respostas são atrasadas a duas requisições por segundo, e cerca 30% a três requisições por segundo.

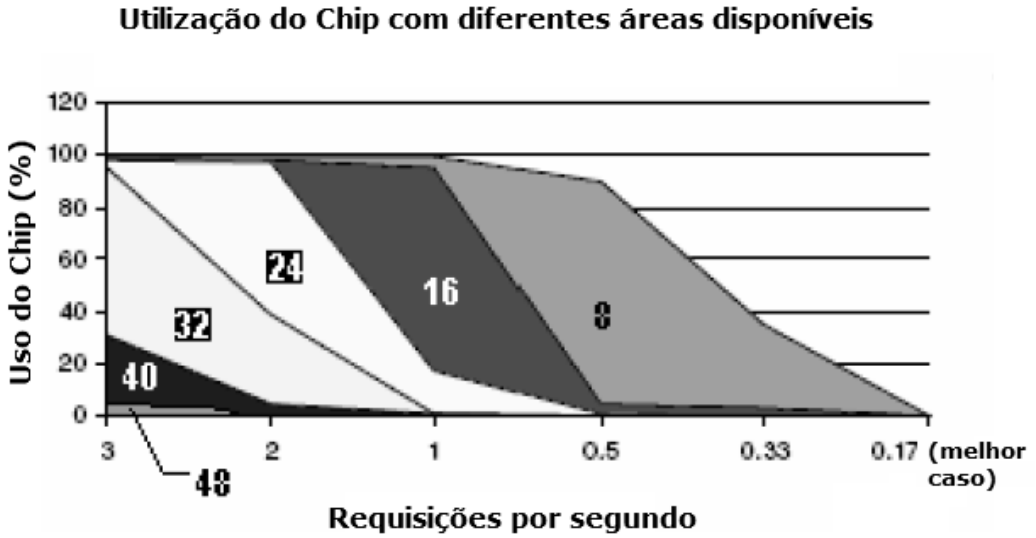


Figura 5-35: percentagem de aplicações atrasadas em diferentes configurações.

Este experimento mostra que com a simulação desenvolvida, vários cenários podem ser criados e simulados rapidamente, fazendo com que o sistema, mesmo já estando em funcionamento em hardware, possa ter suas decisões de projeto reavaliadas. O exemplo simulado mostrou que o sistema responderia melhor se 8 colunas CLBs a mais fossem disponibilizadas na

FPGA.

5.2.3. Simulação da prevenção de acidentes ("pre-crash situation")

Um cenário mais particular simulado é a chamada situação de pré-acidente, ou "*pre-crash*". Aqui o sistema deve realizar uma seqüência de ações objetivando diminuir os efeitos de um acidente eminente. Sistemas inteligentes de detecção de acidentes automotivos são uma tendência e capazes de prever e detectar acidentes de forma precisa, assim como, tomar atitudes preventivas para melhorar a segurança dos passageiros [86]. A questão é quanto tempo um sistema inteligente e preciso como esse necessita para tomar uma decisão em tempo hábil?

Este cenário não foi desenvolvido em hardware, apenas na simulação. O objetivo é aumentar o conhecimento sobre as limitações do sistema através da exposição do mesmo a situações críticas.

Neste cenário, a cada 2 segundos uma requisição é passada para o sistema, simulando um uso normal do sistema. Durante a situação de acidente as quatro janelas devem ser fechadas e as duas poltronas devem ser posicionadas verticalmente. Esses seis comandos são requisitados ao mesmo tempo. A Tabela 5-3 apresenta o tempo de resposta do sistema em milissegundos durante uma situação de pré-acidente típica para cada comando individualmente. O tempo de resposta total em milissegundos para a configuração dos seis comandos também é apresentado. Inicialmente é considerado que nenhuma aplicação está configurada no sistema, logo a área livre é total, porém o tempo necessário para configurar todas as aplicações é maior, já que não há aplicações já configuradas.

Tabela 5-3: tempo de resposta em milissegundos durante a situação crítica de prevenção de acidentes.

Comando requisitado	Número de colunas CLB					
	8	**16**	**24**	**32**	**40**	**48**
1º	26420	1805	5	5	5	5
2º	27421	5006	5	5	5	5
3º	2205	6007	2006	205	5	5
4º	32222	8008	3607	1806	5	5
5º	19217	8809	405	2007	405	5
6º	26420	1805	5	5	5	5
Tempo total (ms)	**133905**	**31440**	**6033**	**4033**	**430**	**30**

Os resultados encontrados mostram que para o sistema responder em

tempo hábil uma situação crítica como a de detecção de acidente, 32 colunas CLBs de área reconfigurável não são suficientes. O mais indicado seriam 40 colunas CLBs para um tempo de resposta maior (430ms em média), ou 48 colunas CLBs para maior segurança em todas situações encontradas, isso porque, com 48 colunas, o sistema comporta 6 instâncias. Exatamente o número de aplicações que o sistema deve carregar para se preparar para o acidente eminente.

No geral a simulação apresentou que a opção adota de 32 colunas CLBs é uma suficiente para as condições normais de funcionamento de um automóvel, mas não responde muito bem quando há muitas requisições por segundo, ou quando o sistema necessite efetuar respostas ágeis em situações críticas, como é o caso da prevenção de acidentes. Nesses casos 8 colunas CLBs a mais, totalizando 40 colunas CLBs seriam suficientes.

5.3. Estudo de caso 2: simulador de processadores parcialmente reconfiguráveis

Esta técnica objetiva desenvolver um simulador em SystemC parametrizável e reutilizável capaz de modelar e simular arquiteturas alvo de processadores dinamicamente e parcialmente reconfiguráveis [5]. Como por exemplo, processadores de granularidade grossa, como o XPP [9] da empresa PACT, que consiste de ULAs configuráveis que se comunicam através de uma rede de sincronização automática e orientada a pacotes.

Para este processador foi dado o nome de PReProS (*Partially Reconfigurable Processors Simulator*). Como pode ser visto na Figura 5-36, este simulador contém portas de dados de entrada, saída e de configuração. Possui unidades de área alocáveis para as aplicações que são conectadas às portas de dados. Todos esses parâmetros são dados pelo projetista e visam a melhor adequação ao processador alvo que deseja ser modelado.

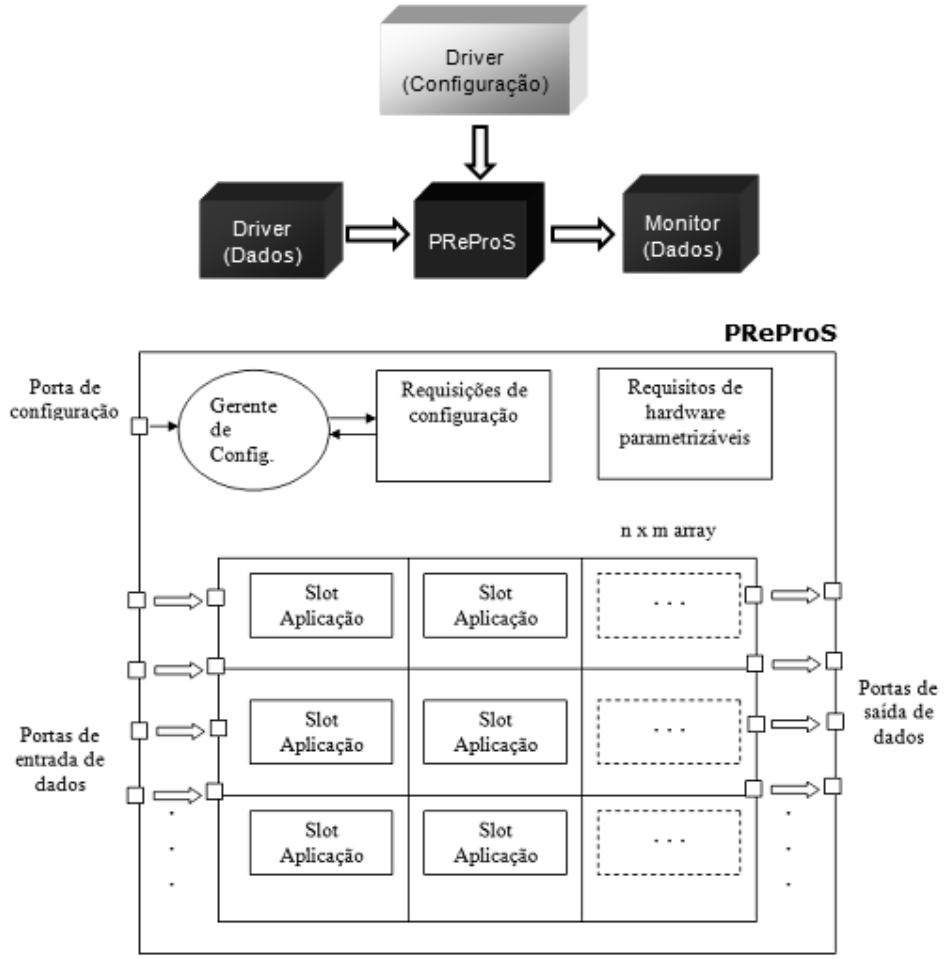


Figura 5-36: visão geral do PreProS

5.3.1. Arquitetura do Simulador

Internamente, o simulador é composto por vários *slots* que podem ser atribuídos às aplicações, como pode ser visto na Figura 5-37. Um barramento interno faz a conexão entre os *slots* e as portas de dados (de entrada e de saída). Um bloco responsável pela gerência das configurações interpreta bits recebidos pela porta de configuração e modifica a configuração atual do sistema, que pode ser o roteamento entre as aplicações nos *slots* e as portas, como também a localização de cada aplicação nos respectivos *slots*.

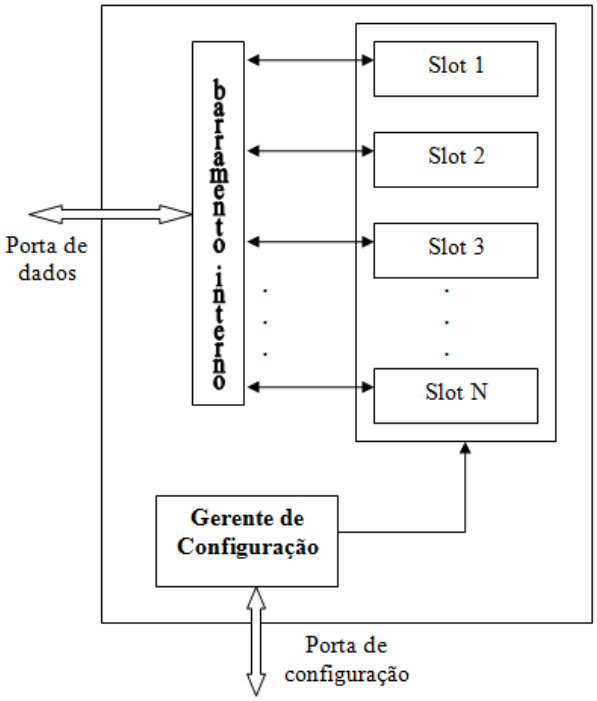


Figura 5-37: arquitetura do simulador

5.3.2. Simulando o XPP como exemplo

Como estudo de caso para o simulador, o processador XPP da PACT [9] foi escolhido para ser modelado e simulado. Como pode ser visto na Tabela 5-4, o PreProS pode receber, como parâmetro, a freqüência interna do processador, o número de ALUs, a quantidade de portas, o tamanho da seqüência de bits para uma unidade e para a configuração total do chip, e a largura de bits da porta de configuração. Com esses dados, o simulador desenvolverá comportamento semelhante ao processador XPP real, pelo menos no que diz respeito ao desempenho.

Tabela 5-4: parâmetros para o XPP

Freqüência	200MHz
ULAs (elementos de granularidades grossas)	8x8 = 64
Portas	4 portas de E/S de 2x16 bits
Tamanho do *bitstream* de configuração	16 Kbytes
***Bitstream* de configuração de cada unidade de área**	0,25Kbytes por ULA
Largura de bits de configuração	42

A Tabela 5-5 apresenta as aplicações que foram utilizadas como exemplo na simulação do processador XPP. Baseado nessas configurações, módulos são implementados emulando o comportamento de cada aplicação. O tempo de configuração, o número de configurações e a quantidade de área de chip necessária são parâmetros utilizados nas reconfigurações. A área de chip aqui é medida pela quantidade de ULAs necessárias para cada aplicação.

Já o desempenho, medido em operações por ciclo e a quantidade de portas utilizadas, são parâmetros utilizados para a emulação da execução das aplicações. Por exemplo, o filtro FIR tem desempenho de 128 operações por ciclo e utiliza 4 portas de dados, isso significa que o módulo que emula esta aplicação executa em laço constante uma função que lê e escreve dados em 4 diferentes portas de dados. As operações são emuladas e os valores são aleatórios. O que é simulado é a quantidade de dados recebida, processada e gerada, e não o significado dos dados. Isso também pode ser feito se os módulos que emulam as aplicações forem implementados na íntegra, mas não foi o caso deste estudo de caso.

Tabela 5-5: parâmetros das aplicações configuradas no XPP 8x8

Aplicação	Ciclos para configuração	Área usada	Área livre	Configurações	Operações por ciclo	Portas usadas
FIR	4000	64	0	1	128	4
IIR	4000	64	0	1	128	2
Viterbi multicanal	1340	22	42	2	43	1
Trans. Fourier	1250	20	44	3	40	1
Beamforming	1625	26	38	2	52	1

5.3.3. Resultados

Os resultados obtidos visam mostrar a corretude do simulador frente ao processador que se quer simular. No caso do XPP, a Tabela 5-6 mostra a quantidade de dados gerada e recebida pelas portas de dados, bem como a taxa de envio e recebimento, para cada aplicação. No caso do filtro FIR, por exemplo, 3200 mega bytes por segundo são enviados e recebidos pelas portas de dados (4 portas, no caso). Ao final da simulação, 6400 bytes foram recebidos pela aplicação, processados e enviados através das portas de saída.

Tabela 5-6: desempenho das portas de dados do processador XPP

Aplicação	Portas	Envio de dados (MB/seg)	Recep. de dados (MB/seg)	Dados enviados (bytes)	Dados recebidos (bytes)	Taxa de transmissão total (MB/seg)
FIR	4	3200	3200	6400	6400	6400
IIR	2	1600	1600	1600	1600	3200
Viterbi multicanal	1	800	800	400	400	1600
Trans. Fourier	1	800	800	400	400	1600
Beamforming	1	800	800	1600	1600	1600

Na Tabela 5-7 são apresentados os resultados referentes às portas de configuração. Para cada aplicação, todas as marcas de tempo para cada passo no processo de configuração é apresentado. Primeiramente a configuração é requisitada (*cfg_req*), depois a aplicação é configurada (*config*), inicia e termina. Isso é importante para se observar o tempo necessário para consumir cada uma dessas tarefas. A taxa de configuração também é apresentada, que é diretamente proporcional ao tamanho da seqüência de bits de configuração.

Tabela 5-7: tempo de configuração e desempenho

Aplicação	t (cfg_req)	t (config)	t (início)	t (fim)	Bits de config (bits)	Taxa de configução (MB/seg.)
FIR	10ns	1915ns	2020ns	4020ns	16000	8400
IIR	1925ns	3830ns	3930ns	4930ns	16000	8400
Viterbi multicanal	3840ns	4495ns	4595ns	5095ns	5500	8400
Trans. Fourier	4505ns	5105ns	5205ns	5705ns	5000	8300
Beamforming	5115ns	5890ns	5995ns	8000ns	6500	8400

Dependendo da quantidade de recursos utilizados por cada aplicação, é possível que mais de uma aplicação trabalhe simultaneamente no chip. Como é o caso apresentado na Figura 5-38. Aqui aplicações como o Filtro IIR e o Viterbi podem executar simultaneamente, já que a primeira necessita de 64 ULAs e a segunda de 22 ULAs, totalizando 86 ULAs. Como no exemplo foi configurado um XPP fictício com 144 ULAs, o paralelismo de aplicações torna-se possível. O que não seria verdade nesse caso, se o XPP tradicional de 64 ULAs fosse utilizado.

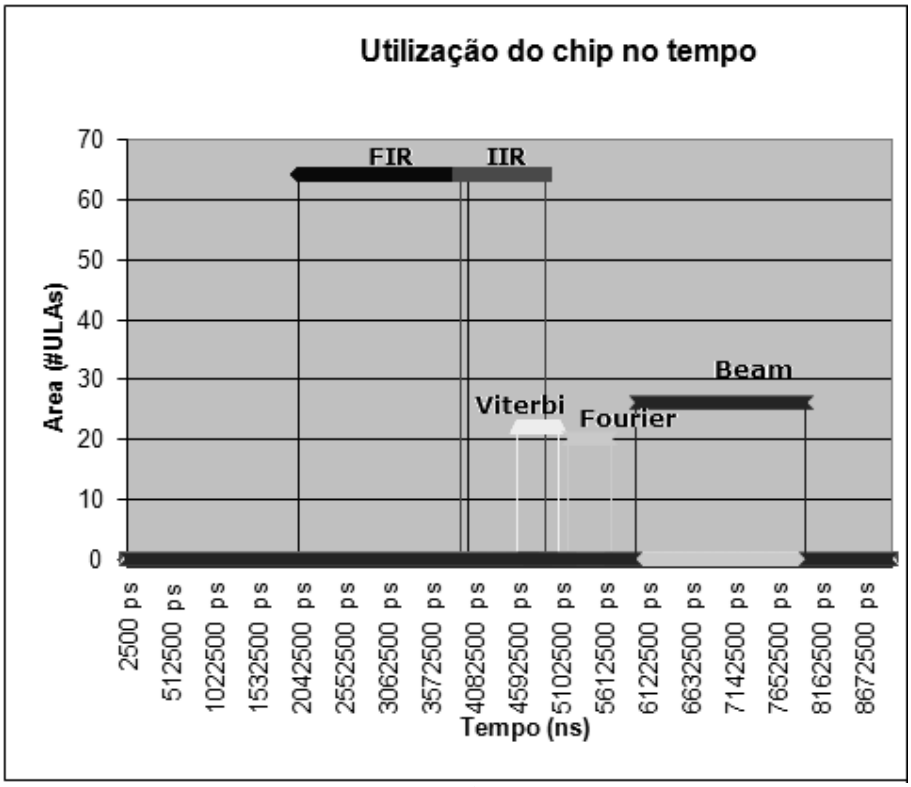


Figura 5-38: utilização de área por cada aplicação

Neste exemplo a utilização total de área de chip no tempo é apresentada na Figura 5-39. A parte escura do gráfico representa a área total do chip, enquanto que a clara, a área utilizada por cada aplicação durante a simulação. É importante ressaltar que esse gráfico é gerado com dados automaticamente coletados pela versão modificada do SystemC, que a partir das informações sobre as aplicações passadas anteriormente, é capaz de mostrar, a cada instante, o quanto de área de chip está sendo utilizada. Estes dados são essenciais no melhor entendimento das tecnologias e implementações utilizadas.

Esta aplicação mostra como um processador dinamicamente reconfigurável pode ser modelado utilizando o PReProS. O exemplo utilizado foi o XPP. Qualquer elemento dinamicamente reconfigurável pode ser utilizado, desde que haja dados sobre sua especificação e funcionamento, tanto no que se refere ao desempenho no tratamento de dados, quanto no que se refere ao processamento de *bitstreams* de configuração e ao próprio tempo de reconfiguração. Uma vez que o processador é modelado utilizando o PReProS, o modelo gerado pode ser integrado às outras plataformas a fim

de que seu comportamento frente a outros elementos de processamento (principalmente no projeto de *System-on-Chips* - *SoCs*).

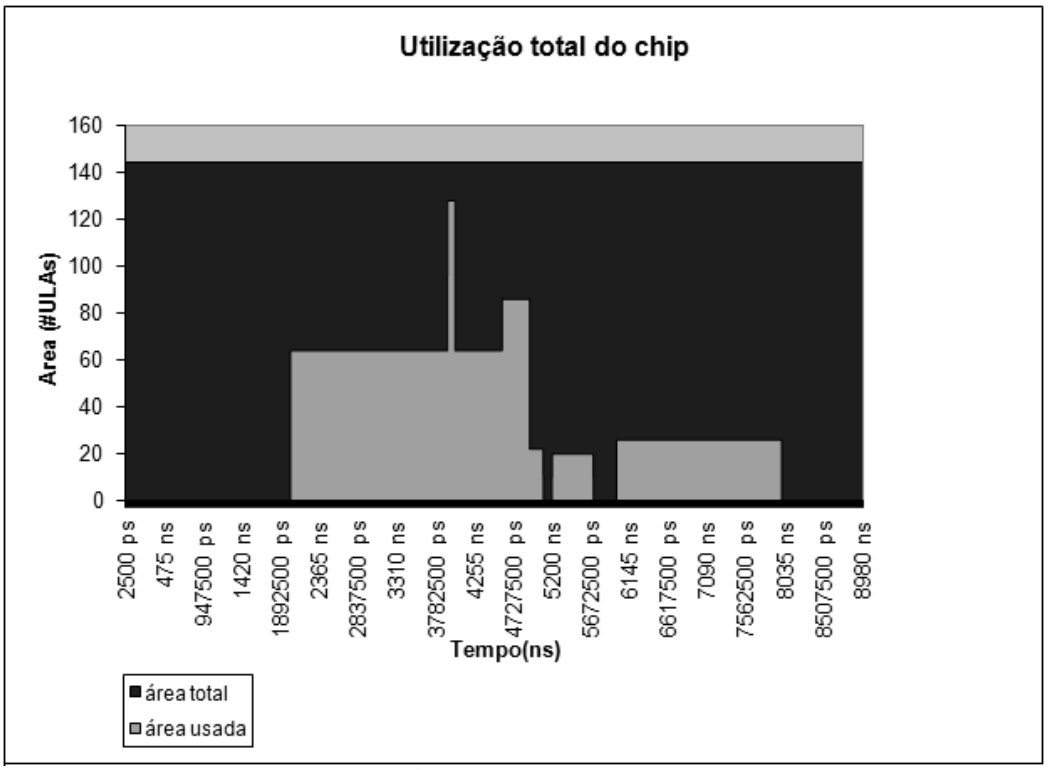


Figura 5-39: área total de chip utilizada no tempo

Capítulo 6

Análise dos Resultados e Considerações Finais

6.1. Introdução

A partir dos exemplos aqui apresentados, é possível classificar todas as aplicações desenvolvidas pelas técnicas, finalidade e nível de abstração utilizada.

A Tabela 6-8 mostra um resumo das aplicações desenvolvidas. A aplicação de prova de conceito apresentada no Capítulo 4 foi desenvolvida para mostrar a viabilidade das rotinas desenvolvidas. Foi desenvolvida utilizando o nível RTL, modelando cada sinal e monitorando-os através de formas de onda geradas automaticamente pelo SystemC. Ele modela a reconfiguração dinâmica em granularidade fina, já que os módulos reconfigurados trabalham sinais em níveis de bits. Cada módulo recebe e processa apenas um bit por vez.

Tabela 6-8: resumo das aplicações desenvolvidas

Aplicações	Finalidade	Nível de Abstração	Granularidade
Prova de conceito	Prova de conceito	RTL	Fina
Automotivo	Avaliar hardware alvo	TLM	Grossa
PReProS	Gerar simuladores genéricos	TLM	Grossa

A aplicação automotiva foi desenvolvida para avaliar qual seria a configuração ideal para o hardware que atualmente funciona com essa

aplicação. Os resultados mostraram que o hardware atual é adequado para as situações corriqueiras, mas pode se tornar pequeno para situações mais críticas, como para a resposta rápida no caso da detecção de um acidente eminente. Ela foi desenvolvida utilizando nível TLM de abstração. Neste caso, o hardware final simulado é de granularidade grossa, e cada módulo simulado trabalha com processamento de dados no nível de bytes.

O simulador PReProS tem como propósito principal, emular processadores que deverão ser integrados em plataformas maiores. Ele foi desenvolvido pela necessidade que tivemos na Universidade de Karlsruhe de simular plataformas em TLM formada por processadores de diversos fornecedores, que nem sempre disponibilizam simuladores SystemC TLM de fácil integração, boa documentação e em tempo hábil. Muitas vezes esses simuladores não são o alvo principal no momento, mas apenas sua relação com os outros elementos do sistema. Neste caso, o PReProS é uma boa opção para uma rápida emulação de simuladores de terceiros. Apesar de se tratar de um simulador genérico, foi desenvolvido no exemplo para simular processadores de granularidade grossa, sendo o processador XPP utilizado como exemplo e parâmetro. Esta aplicação também foi desenvolvida no nível TLM de abstração.

6.2. Contribuições do trabalho

Este trabalho apresentou uma metodologia para modelagem e simulação de sistemas dinamicamente reconfiguráveis através da modificação do núcleo de simuladores orientados a eventos, capaz de auxiliar na avaliação da utilização da reconfiguração dinâmica em sistemas digitais de granularidades diversas.

Como estudo de caso, modificações foram realizadas no mecanismo de simulação do SystemC de forma a permitir a simulação de sistemas dinamicamente reconfiguráveis. A estratégia utilizada incorporou algumas características ao simulador, tais como:

- Facilidade – é muito simples simular a reconfiguração dinâmica, já que apenas três novas funções foram adicionadas e não houve alterações nas instruções já existentes do SystemC, facilitando também a

adaptação de sistemas tradicionais já implementados para o uso de reconfigurações dinâmicas.

- Generalidade – como as modificações foram realizadas na base do SystemC, modificando o núcleo do simulador, qualquer sistema que utilize SystemC pode usufruir das reconfigurações dinâmicas.
- Replicabilidade – a maneira como o SystemC foi modificado internamente mostra como é possível acrescentar outras funcionalidades para as mais diversas aplicações como, por exemplo, a medição do consumo de energia pelo simulador, a modificação do SystemC para simulação de sistemas baseados no comportamento da natureza (*Organic Computing*), ou quaisquer outros sistemas baseados na reconfiguração dinâmica.

Foram mencionadas na seção de objetivos do Capítulo 1, as metas que planejadas para este trabalho foram alcançadas. Pode-se dizer em relação a elas que:

- Foi desenvolvido e apresentado um mecanismo para a modelagem, simulação e o auxílio na análise de sistemas dinamicamente reconfiguráveis. Tal mecanismo foi aplicado utilizando SystemC, e exemplos práticos demonstraram a utilidade do mesmo em questões práticas e científicas e em granularidades e em níveis de abstração diversos.
- Uma prova do conceito foi apresentada, onde o conceito de reconfiguração dinâmica pôde ser realmente comprovado, reforçando o conceito defendido pela metodologia. Além disso, uma prova de generalidade foi elaborada, demonstrando que a metodologia desenvolvida é capaz de modelar e simular os principais casos de reconfiguração dinâmica definidos pela literatura.
- Dois estudos de caso mostraram a viabilidade prática da metodologia de modelagem e simulação em sistemas reais e científicos, incluindo um exemplo baseado num sistema já funcionando em hardware.

6.3. Considerações Finais

O aumento do poder computacional dentro de um mesmo chip relaciona-se com o aumento de temperaturas, necessitando mecanismos de resfriamento cada vez mais sofisticados e onerosos. O poder computacional é uma busca contínua, mas o alto consumo de energia tem tornado os computadores maiores e com maior custo para manter seus sistemas de resfriamento. Uma das principais opções para o aumento do desempenho dos computadores é através de outros paradigmas que não o emprego de um único elemento processador de arquitetura do tipo Von Neumann, que vem demandando demasiado consumo de energia para ser sustentado. O acréscimo de um elemento processador dinamicamente reconfigurável pode ser uma solução para determinadas aplicações.

Para sobrepor o problema de atraso causado por sucessivas configurações em sistemas dinamicamente reconfiguráveis, o uso de arquiteturas mistas é uma promessa. Sistemas de arquiteturas mistas, aqui considerados, são aqueles formados por um processador RISC trabalhando na mesma pastilha de um processador dinamicamente reconfigurável, recebendo originalmente a denominação de *Dynamically Reconfigurable Systems on Chips (DR-SoC)*. Classificamo-nas como mistas por possuírem tanto processadores dinamicamente reconfiguráveis, quanto processadores convencionais, geralmente RISC.

Em breve será possível que os equipamentos permitam implementação de sistemas dinamicamente reconfiguráveis de forma embarcada, ou seja, nativo neles. Nesse caso, este novo paradigma baseado na mutação de comportamentos será utilizado de forma eficiente. Tendo em vista a importância dessa evolução, pesquisas como esta, contribuem para a exploração do espaço de opções de implementação desse tipo de sistemas.

6.4. Trabalhos Futuros

Sempre que o simulador do SystemC for utilizado, o mecanismo de simulação de reconfiguração dinâmica pode ser aplicado, significando a possibilidade também de trabalhar com outros simuladores, como a co-simulação SystemC-Verilog, SystemC-NS (*Network Simulator*), muito utilizado para simulação de Redes em Chip (*Network-on-Chip - NoC*), entre

outras. A viabilidade dessa idéia merece um estudo mais aprofundado.

Uma das principais vantagens no uso de reconfiguração dinâmica é a redução no consumo de energia pelos sistemas de hardware. Sendo assim, estimar essa redução torna-se também relevante. Da mesma forma que parâmetros sobre área de chip e tempo de configuração dos módulos foram adicionados ao simulador, outros números podem ser acrescentados de forma a mensurar o consumo de energia pela atividade de cada módulo e a energia despendida no momento de configuração. Com isso, figuras sobre o consumo de energia podem ser analisadas e o uso da reconfiguração dinâmica pode ser justificado, ou não, nas fases mais iniciais de projeto.

Uma técnica para redução no consumo de energia por chips (não reconfiguráveis) é desligar a alimentação de módulos inativos momentaneamente, levando o consumo deles a zero. Com as técnicas apresentadas nesse trabalho, o processo de desligamento pode ser simulado em vários níveis de abstração. Investigar a aplicabilidade deste trabalho para o projeto de chips mais econômicos no que se refere à energia deve ser considerado.

Referências Bibliográficas

[1] DEHON, A.; WAVRZYNEK, J. Reconfigurable computing: What, why and implications for design automation. In: Proceedings of the 36th ACM/IEEE Conference on Design Automation. USA: ACM PRESS, 1999. p. 610 – 615.

[2] COMPTON, K.; HAUCK, S. Reconfigurable Computing: a survey of systems and software. ACM Computing Surveys, v. 34, n. 2, p. 171-210, June 2002.

[3] DENG, Y., HWANG, C., AND LIU, J. "An object-oriented cryptosystem based on two-level reconfigurable computing architecture". *Journal of Systems and Software*, vol. 79, ed. 4. Abril, 2006.

[4] OLDFIELD, J. V.; DORF, R. C. Field-Programmable Gate Arrays. USA: John Wiley & Sons Inc., 1995. ISBN 0-47155-665-1.

[5] LYSAGHT, P.; DUNLOP, J. "Dynamic Reconfiguration of Field Programmable Gate Arrays". In: MOORE, W.; LUK, W. (Ed.). More FPGAs: Proceedings of the 1993 International Workshop on Field-Programmable Logic and Applications. Oxford, England: Abingdom EE&CS Books, 1993. p. 82-94.

[6] GOVINDARAJAN, S., OUAISS, I., KAUL, M., SRINIVASAN, V., and VEMURI, R., "An Effective Design Approach for Dynamically Reconfigurable Architectures". in *Proc. IEEE Symposium on Field Programmable,* 1998.

[7] LI, Y., CALLAHAN, T., DARNELL, E., HARR, R., KURKURE, U., STOCKWOOD, J. "Hardware-software co-design of embedded reconfigurable architectures", in *Proceedings of the 37th ACM Conference on Design Automation*, Los Angeles, California, USA. June 2000

[8] BINGFENG, M., VERNALDE, S., DE MAN, H.,LAUWEREINS, R.., "Design and Optimization of Dynamically Reconfigurable Embedded Systems". In Proc. 1st Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA), CSREA Press, 2001, pp. 78-84.

[9] BECKER, J., VORBACH, M.,. "Architecture, Memory and Interface Technology Integration of an Industrial/Academic Configurable System-on-Chip (CSoC)", IEEE COMPUTER SOCIETY. ANNUAL Symposium ON VLSI, Tampa, Florida, February 20–21, 2003.

[10] HADLEY, J. D., HUTCHINGS, B. L.. "Design Methodologies for Partially Reconfigured Systems", In Peter Athanas and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 78-84, Los Alamitos, California, April 1995. IEEE Computer Society, IEEE Computer Society Press.

[11] MARKOVSIY, Y., CASPI E., HUANG R., YEH J., MICHAEL C., J. WAWRZYNCK. "Analysis of Quasi-Static Scheduling Techiniques in a Virtualizad Reconfigurable Machine". *Proceedings of ACM FPGAs 2002,* Monterey, California, USA. February 2002.

[12] KÖSTER, M., PORRMANN, M., RÜCKERT, U. “Placement-Oriented Modeling of Partially Reconfigurable Architectures”. In: *Proceedings of the 19th International Parallel and Distributed Processing Symposium - Reconfigurable Architectures Workshop (RAW 2005)*, 2005

[13] ZHANG, X.; NG, K. W. “A review of high-level synthesis for dynamically reconfigurable FPGAs”. *Microprocessors and Microsystems*, v. 24, n. 4, p. 199-211. August 2000.

[14] VASILKO, M., CABANIS, D., Improving Simulation Accuracy in Design Methodologies for Dynamically Reconfigurable Logic Systems , in Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'99), Napa, CA, USA, April 21-23, 1999.

[16] BRUNELLI, L., Abordagem para Redução de Complexidade de RNA usando Reconfiguração Dinâmica. Tese de Doutorado. Departamento de Engenharia Elétrica da Universidade Federal de Campina Grande. Fevereiro, 2005. Download: http://www.dee.ufcg.edu.br (acesso em 18/08/2005).

[17] BRUNELLI, L., MELCHER, E. U. K., BRITO, A. V., FREIRE, R. C. S., A Novel Approach to Reduce Interconnect Complexity in ANN Hardware Implementation, Proceedings of the International Joint Conference on Neural Networks (IJCNN’2005), Montreal, Canada, July, 2005.

[18] BROWN, S.; ROSE, J. FPGA and CPLD architectures: A tutorial. IEEE Design & Test of Computers, v. 13, n. 2, p. 42{57, 1996.

[19] KWIAT, K. A.; JR., W. H. D. Modeling a versatile FPGA for prototyping adaptive systems. In: Proceedings on Sixth IEEE International Workshop on Rapid System Prototyping. Chapel Hill, NC, USA: [s.n.], 1995. p. 174-180.

[20] LUK, W.; SHIRAZI, N.; CHEUNG, P. Y. K. Modelling and optimising run-time reconfigurable system. In: ARNOLD, J.; POCEK, K. L. (Ed.). Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines. Napa, CA, USA: IEEE Computer Society Press, 1996. p. 197-176.

[21] LYSAGHT, P.; STOCKWOOD, J. A simulation tool for dynamically reconfigurable field programmable gate arrays. IEEE Transactions on Very Large Scale Integration System, v. 4, n. 3, p. 381-390, September 1996.

[22] ROBINSON, D.; LYSAGHT, P. Methods of exploiting simulation technology for simulating the timing of dynamically reconfigurable logic. IEE Proceedings Computers an Digital Techniques, v. 147, n. 3, p. 175-180, May 2000.

[23] ROBERTSON, I.; IRVINE, J. A design flow for partially reconfigurable hardware. ACM Transactions on Embedded Computing Systems, v. 3, n. 2, p. 257-283, May 2004.

[24] FERRANDI, F., SANTAMBROGIO, M., SCIUTO, D., A Design Methodology for Dynamic Reconfiguration: The Caronte Architecture. *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS’2005)*, 2005, Denver, CA, USA

[25] GROTKER, T., LIAO, S., MARTIN, G., SWAN, S. System Design with SystemC. Kluwer Academic Publishers, 2002.

[26] PELKONEN A., MASSELOS K., CUPÁK M. System-Level Modeling of Dynamically Reconfigurable Hardware with SystemC. In International Parallel and Distributed Processing Symposium (IPDPS'03), Nice, France, April 22, 2003.

[27] CUMMINGS, CLIFFORD E. (Sunburst Design, Inc.), Synthesizable Finite State Machine Design Techniques Using the New SystemVerilog 3.0 Enhancements, SNUG, 2003, San Jose, CA.

[28] SYSTEMVERILOG, www.systemverilog.org (acesso em 20/03/2004).

[29] BELLOWS , P., HUTCHINGS, B., JHDL - An HDL for Reconfigurable Systems. In IEEE Symposium on Field-Programmable Custom Computing Machines, April 1998.

[30] CHU , M., WEAVER , N., SULIMMA , K., DEHON, A., WAWRZYNEK , J., Object Oriented Circuit-Generators in Java. In IEEE Symposium on FPGAs for Custom Computing Machines, April 1998.

[31] LEVI, D., GUCCIONE, S. Run-Time Parameterizable Cores. In ACM International Symposium on Field Programmable Gate Arrays, February 1999.

[32] BOOCH, G. Object-Oriented Analysis and Design with Applications. Addison-Wesley Pub Co; 2nd edition, 1993.

[33] QU, Y., TIENSYRJA, K., MASSELOS, K., “System-Level Modeling of Dynamically Reconfigurable Co-Processors”, International Conference on Field Programmable Logic and Applications, Antwerp, Belgium, August-September 2004.

[34] SCHALLENBERG, A., OPPENHEIMER, F., NEBEL, W., “OSSS+R: Modelling and Simulating Self-Reconfigurable Systems”. International Conference on Field Programmable Logic and Applications, pags 177–182, Aug. 2006

[35] LI, Y., CALLAHAN,T., DARNELL, E., HARR, R. E., KURKURE, U., STOCKWOOD , J., “Hardware-Software Co-Design of Embedded Reconfigurable Architectures”. In Proc. 37th ACM/IEEE Design Automation Conference DAC, Los Angeles, CA. 2000.

[36] KALAVADE, A., LEE, E. A., “A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem,” Proc. International Workshop on Hardware-software Co-design, 1994, pp. 42-48.

[37] DAVE , B., LAKSHMINARAYANA, G., JHA, N., “COSYN: hardware-software co-synthesis of embedded systems,” Proc. 34th Design Automation Conference, 1997.

[38] PRAKASH, S., PARKER, A., “SOS: synthesis of application specific heterogeneous multiprocessor systems,” Journal of Parallel and Distributed Computing, 1992, vol.16, pp.338-351.

[40] CUMMINGS, C. E., “Synthesizable Finite State Machine Design Techniques Using the New SystemVerilog 3.0 Enhancements”, SNUG’2003, San Jose, CA, 2003.

[41] LEVEUGLE , R., SAUCIER, G. "Optimized Synthesis of Concurrently Checked Controller," IEEE Trans. Computers, vol. 39, pp. 419-425, April 1990.

[42] ROBINSON , S. H., SHEN , J. P., "Direct Methods for Synthesis of Self-monitoring State Machines", International Symposium on Fault-Tolerant Computing, pp306-315, Jul. 1992.

[43] PAREKHJI , R. A., VENKATESH, G., SHERLEKAR, S. D., "Concurrent Error Detection using Monitoring Machines", IEEE Design & Test of Computers, vol. 12, pp.24-32, Fall 1995.

[44] HAREL, D., "Statecharts: A Visual Formalism for Complex Systems". Science of Computer Programming., vol. 8, pp 231-274, 1987.

[45] BUCHENRIEDER, K., PYTTEL, A., VEITH , C., "Mapping StateCharts Models onto an FPGA Based ASIP Architecture", Proceedings of the European Design Automation Conference with Euro-VHDL, September 1996.

[46] SELIC, B., GULLEKSON, G.., "Ward: Real-Time Object-Oriented Modeling", Wiley, 1994.

[47] KÖSTER, M., TEICH., J., "(Self-) Reconfigurable Finite State Machines: Theory and Implementation", Proc. Design, Automation and Test in Europe (DATE), 2002.

[48] BONDALAPATI, K., CHOI, S., VIKTOR, K., PRASANNA, SIDHU, R., "Computation Models for Reconfigurable Machines", International Symposium on Field-Programmable Gate Arrays, February 1997.

[49] GAMMA, E., HELM R., JOHNSON, R., VLISSIDES, J. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional; 1st ed., 1995.

[50] HAMMERSTROM, D. The connectivity requirements of simple association, or how many connections do you need? In: ANDERSON, D. Z. (Ed.). Neural Information Processing System. New York, USA: American Institute of Physics, 1998. p. 338–347.

[51] BEIU, V. How to build VLSI - efficient neural chips. In: ALPAYDIN, E. (Ed.). Proceedings of the International ICSC Symp. on Engineering of Intelligent Systems EIS'98. Tenerife, Spain: ICSC Academic Press, 1998. v. 2, p. 66–75.

[52] CRAVEN, M. P.; CURTIS, K. M.; HAYES-GILL, B. R. Consideration of multiplexing in neural network hardware. IEE Proceedings Circuit Devices System, v. 141, n. 3, p. 237–240, Jun 1994.

[53] BEIU, V. A survey of perceptron circuit complexity results. In: Proceedings of the International Joint Conference on Neural Networks 2003. USA: [s.n.], 2003. v. 2, p. 989–994.

[54] BACKUS, J. Can programming be liberated from the von Neumann style? A funcional style and its algebra of programs. Communications of the ACM, v. 21, n. 8, p. 613-641, August 1977. ACM Turing Award Lecture.

[55] MAJER, M., TEICH, J., AHMADINIA, A., BOBDA, C. "The Erlangen Slot

Machine: A Dynamically Reconfigurable FPGA-based Computer". *Journal of VLSI Signal Processing and Systems*, vol. 47, ed. 1, pp. 15-31. Abril, 2007.

[56] HARTENSTEIN, R. Coarse Grain Reconfigurable Architectures (invited embedded tutorial). *6th Asia and South Pacific Design Automation Conference 2001 (ASP-DAC 2001).* Pacifico Yokohama, Yokohama, Japan, February, 2001.

[57] ANDY, G., YE, ROSE, J., LEWIS, D., "Architecture of Datapath-Oriented Coarse-Grain Logic and Routing for FPGAs," Proceedings of the IEEE Custom Integrated Circuits Conference 2003, San Jose, CA, September 2003, pp. 61-64.

[58] HARTENSTEIN, R. The Microprocessor is no more General Purpose (invited paper), *Proc. ISIS'97, Austin*, Texas, USA, Oct. 8-10, 1997.

[59] KRESS , R. et al., A Datapath Synthesis System for the Reconfigurable Datapath Architecture; ASP-DAC'95, Chiba, Japan, Aug. 29 - Sept. 1, 1995

[60] BITTNER, R. A. et al. Colt: An Experiment in Wormhole Run-time Reconfiguration; SPIE Photonics East `96, Boston, MA, USA, Nov. 1996.

[61] MIRSKY , E., DEHON, A., MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources; Proc. IEEE FCCM'96, Napa, CA, USA, April 17-19, 1996

[62] CHEREPACHA, D., LEWIS , D., A Datapath Oriented Architecture for FPGAs; Proc. FPGA'94, Monterey, CA, USA, February 1994.

[63] HAUSER, J., WAWRZYNEK, J., Garp: A MIPS Processor with a Reconfigurable Coprocessor; Proc. IEEE FCCM'97, Napa, April 16-18, 1997.

[64] WAINGOLD, E. et al., Baring it all to Software: RAW Machines; IEEE Computer, September 1997, pp. 86-93.

[65] BECKER, J., et al.: Architecture and Application of a Dynamically Reconfigurable Hardware Array for Future Mobile Communication Systems; Proc. FCCM'00, Napa, CA, USA, April 17-19, 2000.

[66] DEHON, A., et. al. Design Patterns for Reconfigurable Computing. *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April, 2004.

[67] BECKER, J., HARTENSTEIN, R., Configware and morphware going mainstream. *Journal of Systems Architecture*. 49, 4-6, 127-142, September, 2003.

[68] HYLANDS, C., LEE, E. A., LIU, J., LIU, X., NEUENDORFFER, S., XIONG, Y., ZHENG, H., Heterogeneous Concurrent Modeling and Design in Java,(Volume 1:Introduction to Ptolemy II). Technical Memorandum UCB/ERL M03/27, University of California, Berkeley, CA USA 94720, July 16, 2003.

[69] LUK, W., GUO, S., Visualising reconfigurable libraries for FPGAs. In Asilomar Conference on Signals, Systems, and Computers, 1998.

[70] LYSAGHT, P. et al. Artificial Neural Network implementation on a fine-grained FPGA. In: HARTENSTEIN, R. W.; SERVI, M. Z.. Field-Programmable Logic. Prague, Czech Republic: Spring Verlag, 1994. p. 421-431.

[71] MOHAMED, A., A Tool Converting Finite State Machine to VHDL, In: IEEE CCECE/CCGEI, Niagara Falls, May, 2004.

[72] VCG GRAPH VISUALIZATION, http://www.cs.unisb.de, Universitat Saarlandes, Germany, 1996.

[73] DORAIRAJ, N., SHIFLET, E., GOOSMAN, M., "PlanAhead Software as a Platform for Partial Reconfiguration". *Xcell Journal*. Xilinx, Inc. Dezembro de 2005.

[74] PRUTEANU, C., "Kiss to Verilog FSM Converter", http://codrin.freeshell.org, 2000. Acesso em 15 de setembro de 2005.

[75] FSM GENERATOR, http://fsmgenerator.sourceforge.net. Acesso em Acesso em 15 de setembro de 2005.

[76] GUANGMING LU; SINGH, H.; MING-HAU LEE; BAGHERZADEH, N.; KURDAHI, F.J.; FILHO, E.M.C.; CASTRO-ALVES, V. "The MorphoSys dynamically reconfigurable system-on-chip". Proceedings of the First NASA/DoD Workshop on Evolvable Hardware, 1999. Págs: 152 – 160.

[77] BECKER, J. PIONTECK, T. GLESNER, M., "DReAM: A Dynamically Reconfigurable Architecture for Future Mobile Communication Applications". Journal Lecture Notes in Computer Science. Springer-Verlag, ISSN: 0302-9743. Págs. 312-321.1999.

[78] VAIDYANATHAN, R., TRAHAN, J. L., "Dynamic Reconfiguration: Architectures and Algorithms". Editora: Springer. ISBN: 0306481898, 2004.

[79] ROSENSTIEL, W. (Editor), LYSAGHT, P., "New Algorithms, Architectures and Applications for Reconfigurable Computing". Editora: Springer. ISBN: 1402031270, 2005.

[80] ATMEL CORPORATION, Datasheet: "AT94K FPSLIC" disponível em http://www.atmel.com/products/FPSLIC, acessado em maio de 2007.

[81] SMIT, G.J.M., SCHUELER, E., BECKER, J., Quevremont, J., Brugger, W. "Overview of the 4S project". *International Symposium on System-on-Chip* (SoC' 2005), Tampere, Finland, 2005.

[82] HUEBNER, M., BECKER, T., BECKER, J., "Real-Time LUT Based Network Topologies for Dynamic and Partial FPGA Self-Reconfiguration", 17th Brazilian Symposium on Integrated Circuit Design (SBCCI 04), Brasil

[83] BRITO, A. V., KUEHNLE, M., HUEBNER, M., BECKER, J., MELCHER, E. U. K. "Modelling and Simulation of Dynamic and Partially Reconfigurable Systems using SystemC". In: *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures, (ISVLSI'2007)*, 2007, Porto Alegre. IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures. Piscataway, New Jersey: IEEE 2007 v.1. p.200 – 203.

[84] BRITO, A. V., KUEHNLE, M., HUEBNER, M., BECKER, J., MELCHER, E. U. K. "A General Purpose Partially Reconfigurable Processor Simulator (PReProS)" In: *15th Reconfigurable Architecture Workshop (RAW'2007)*, 2007, Long Beach. 21st International Parallel & Distributed Processing Symposium. Piscataway, New Jersey: IEEE, 2007.

[85] BRITO, A. V., ROSAS, W., MELCHER, E. U. K. “An open-source tool for simulation of partially reconfigurable systems using SystemC”. In: *IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2006)*, 2006, Karlsruhe, Germany.

[86] CHAN, Ching-Yao, “Trends in Crash Detection and Occupant Restraint Technology”. In: Proceedings of IEEE. Vol. 95, No. 2, Fevereiro de 2007.

[87] XILINX, “Two Flows for Partial Reconfiguration: Module Based and Difference Based”. *Xilinx Application Note: Virtex, Virtex-II and Virtex-II Pro. Families. XAPP290 (v1.2)*. Setembro, 2004.

[88] RECONF 2, “Specification of the Design Environment”, www.reconf.org, acesso em 28 de outubro de 2007. Janeiro de 2003.

[89] PLEIS, M. A., OGAMI, K. Y. "Dynamic reconfiguration interrupt system and method". *Cypress Semiconductor Corporation*, San Jose, CA, US. 2007.

[90] ROBINSON, S., “Simulation: The Practice of Model Developmentand Use”. ISBN: 978-0470847725. John Wiley & Sons, 2004.

[91] LEANDRO, K., “Metodologia de Projeto de Sistemas Dinamicamente Reconfiguráveis”. Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Sistemas Eletrônicos. São Paulo, 2007.

[92] Rocha, A. K., LIRA, P., JU, Y. Y., BARROS, E., MELCHER, E. U. K., ARAUJO, G. “Silicon Validated, IP Cores Designed by The Brasil-IP Network”. IP/SOC Conference, Grenoble, França, 2006.

Apêndice A

PROJETO E PESQUISA I: ESTUDO DE RECONFIGURAÇÃO DINÂMICA EM MODELOS COMPORTAMENTAIS COM VISTAS A MODELAGEM DE FIGURAS DE EXECUÇÃO

Período: **2004.1**
Aluno: **Alisson Vasconcelos de Brito**
Orientador do Projeto: **Elmar Uwe Kurt Melcher**

1. Introduction

A vast number of computational applications are susceptible to enjoy performance benefits [A4] using reconfigurable computing [A1, A2, A3], but the absence of specific design tools and techniques has retarded research in this area.

One of the greatest problems with dynamically reconfigurable architectures is the deficiency of computational platforms which help the engineering during the development process. Many of the existing tools that are based on RTL or behavioral level analyze the hardware behavior of each configuration seperately without considering the reconfiguration logic [A5, A6]. The few existing tools, which analyze the system as a set of all its elements including the transitions form one configuration to another are netlist-based and specific to determined architectures [A7, A8].

Enabling a high level simulation of dynamically reconfigurable architectures implies the use of some hardware simulation language which supports their elements reconfiguration during simulation-time.

Some important dynamic reconfiguration concepts are clearly implemented in object oriented languages. These languages allow [A9]:

- Abstract data models (class) creation, which can be instantiated dynamically during execution-time;
- Reuse of classes and project's modules (heritage);
- Behavior changes during execution-time (polymorphism).

SystemC [A10, A17], as a hardware description language which is based on the object oriented language C++ can be used for this purpose. Another option is the SystemVerilog [A11, A12] which supports the modeling and verification of hardware based on transactions. SystemVerilog is also object-oriented and has support of modeling and

verification at the “transaction” level of abstraction, enables to “call” C/C++/SystemC functions, and vice versa. It enable co-simulation with SystemC blocks.

The problem of these two languages is that they have no support for the instance creation during execution-time this limits the possibility to use them to simulate dynamically reconfigurable architectures at a higher level than the structural (netlist) level.

Other project is the JHDL that supports dynamic reconfiguration [A5]. JHDL is a hardware description language based on Java, which allows the user to design the structure and layout of a circuit, debug the circuit in simulation, *netlist* and interface for bit-stream synthesis.

The Ptolemy Project [A13] studies modeling, simulation and design of concurrent systems, real-time and embedded systems, based on concurrent components assembly. It is implemented with Java and allows the instantiation and the destruction of elements (actors) during execution time, a technique denominated mutation [A6]. With Ptolemy II is possible to generate hardware models through JHDL [A5], which transforms a Ptolemy model into a HDL. Other tools based on Java for hardware project can be found in literature [A5, A15, A16].

The concept of Ptolemy mutation is not extensively explored by literature, neither used to design hardware architectures. The Ptolemy Project’s manuals do not demonstrate clearly how to implement the Mutations, neither what are this technique’s capabilities.

One of these new applications for the reconfigurable computing [A14] is the Execution Patterns which defend a different manner of data flow, where the data is not moved through buses to the processing units, the hardware is reconfigured making the processing units to reach the data on the way. Using this technique the area used for interconnections is reduced. What is still unknown is the reconfigurations cost for these computations, and what applications can really take advantage from Execution Patterns. In Figure 1 and 2 is possible to see an example circuit executing a sum and a multiplication. In Figure 1 is used the traditional computation model. The Execution Pattern example is presented in Figure 2 where the execution is divided in two moments. In the first moment only the adder circuits are configured and the results are stored in two registers. In the second moment the adder is disconfigured and a multiplier is configured, it receive the stored data from the two result registers, process the multiplication and store the result in the same register that, in the first moment, stored data to the adder.

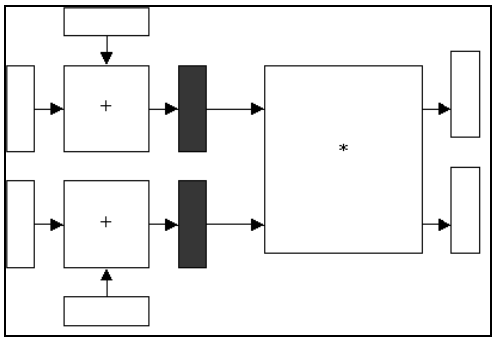


Figure 1: Circuit execution without dynamic reconfiguration

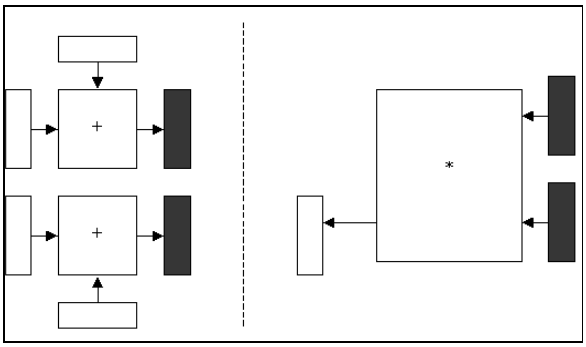


Figure 2: The same circuit using Execution Patterns (dynamic reconfiguration)

The goal of this research project is to explore the true potential of the Ptolemy mutations in the modeling of dynamically reconfigurable hardware. Ptolemy mutations might then be used for the evaluation of Execution Patterns.

2. An Example Application

A model of a dynamically reconfigurable computing was implemented using the concept of Execution Patterns and the Ptolemy tool. The simulated model is presented in Figures 3, 4 and 5. This model computes the expression (a+b)*(c+d). In Figure 3 the conventional model without reconfiguration is presented, where one clock is connected to four register elements, which send the stored data to two adders, when a clock pulse is received. There are synchronous digital system clock signal connected to all registers. The addition result is passed to a multiplier as its input. Finally the multiplier will send the result to be stored in another register.

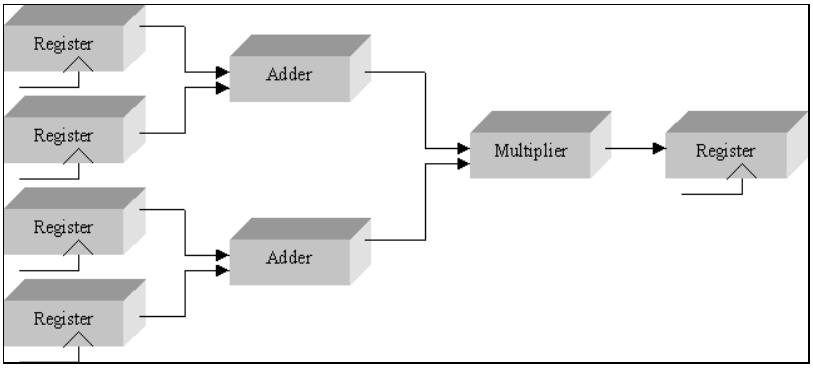


Figure 3: computational model without dynamic reconfiguration

In Figure 4 one can see the same computation but using dynamic reconfiguration. In this model the computation is divided into two steps. During the first step only the addition is computed. The results are passed to registers five and six to be used during the second step.

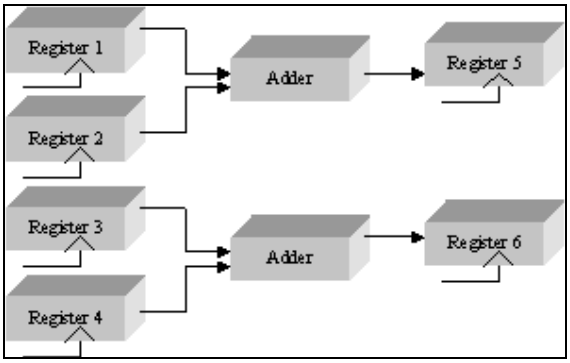


Figure 4: Computation with dynamic reconfiguration - first step

During the second step the chip is reconfigured and a multiplier is configured to receive the data from the resultant register (see Figure 5). The clock is plugged in the register and the computation resumes. The multiplier will receive the data stored in registers five and six and send the multiplication result to register seven. As the computation presented in Figure 3, the final result will be stored in a register.

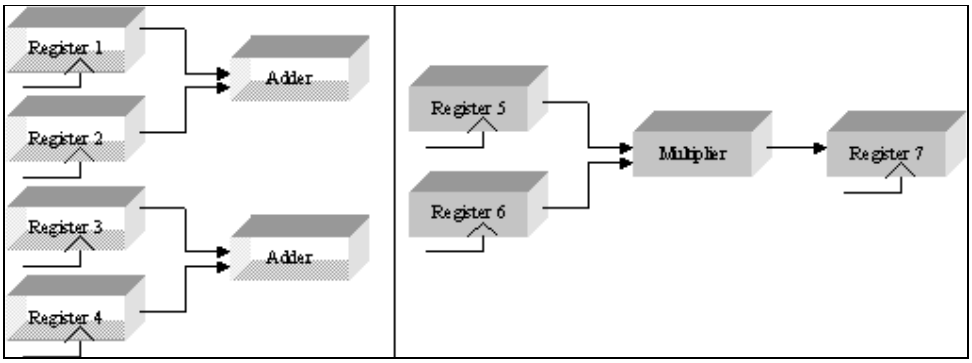


Figure 5: Computation with dynamic reconfiguration – second moment

2. Methodology

The simulation of Execution Pattern scenarios must be done using some dynamic reconfiguration tool. The solution proposed is to implement the simulations with the Ptolemy tool. Executing these simulations with Ptolemy may help to answer some still opened questions about Execution Patterns:

- What is the reconfiguration costs to the Execution Patterns?
- What is the chip area gain applying consecutive reconfigurations?

Trying to answer these questions introduce two problems. First one, each application can be reconfigured at different times, by different manners. The strategy used to implement the dynamic reconfiguration will change the final results. A simulation tool should be general

and not measure the *Execution Patterns* costs only for a group of applications, neither generate different results for the same application.

Second one, the answers for both questions depend on the reconfiguration hardware used. Each hardware uses its own reconfiguration logic and structure, this will generate different performance results for equivalent applications.

The solution for the first problem is to implement a tool able to simulate dynamic reconfiguration and *Execution Patterns* for all kind of applications. It is possible only if there to be a unique strategy to map each conventional application into its respective solution with *Execution Patterns*. A generator must be defined to receive as input the conventional application and returns a reconfiguration model, which uses the Figures de Execution facilities as output. This will be presented as a future work.

Solving the second problem is simpler because it is possible to generate performance results based on the reconfigurations quantity and complexity. This complexity can be measured by the kind of elements and the quantity that is reconfigured in each instant. If for a specific application is possible to know how many reconfiguration are necessary, how many elements of each kind, and how many connections between elements are stabilized at each reconfiguration time, it is possible to estimate a relative performance and a relative chip area necessary for this system be executed using *Execution Patterns*.

3. Implementation

Ptolemy´s mutation is changes to the model during execution. All model mutations should be requested using *requestChange()* method. While invoking those changes, the method *invalidateSchedule()* is expected to be called, notifying the director that the topology it used to calculate the priorities of the actors is no longer valid. This will result in the priorities being recalculated for the next execution time [A18].

The steps followed to implement Mutation with Ptolemy can be divided into three parts. In the first part the initial application model (presented in **Figure 4**) is configured. In the source-code below, the Mutation class is the complete model that will be executed by the Ptolemy. All models to be executable by the Ptolemy must extend the *TypedCompositeActor* class and must have a constructor as presented below. The initial model configuration must be implemented in the class constructor because the Ptolemy will need this implementation to create the model.

In the class constructor six register elements, two adders and one clock are instantiated. The connections between the elements are made using the *connect()* method, when the elements ports are passed as parameters.

```
//The model class
public class Mutation extends TypedCompositeActor {

//Simulation manager and director are declared.
public Manager manager;
```

```
public DEDirector director;

//Mutation class constructor. The initial configuration is
implemented.
public Mutation(Workspace workspace) throws IllegalActionException,
NameDuplicationException {
      super(workspace);

      //Elements are instantiated.
      data1 = new Register(this,"Register1");
      data2 = new Register(this,"Register2");
      data3 = new Register(this,"Register3");
      data4 = new Register(this,"Register4");
      data5 = new Register(this,"Register5");
      data6 = new Register(this,"Register6");
      acc = new Adder(this,"Adder");
      acc2 = new Adder(this,"Adder2");
      clock = new Clock(this,"Clock");

      //Connections for the first Adder
      connect(clock.output,data1.trigger);
      connect(clock.output,data2.trigger);
      connect(data1.output,acc.input);
      connect(data2.output,acc.input);

      //Connections for the second Adder
      connect(clock.output,data3.trigger);
      connect(data3.output,acc2.input);
      connect(clock.output,data4.trigger);
      connect(data4.output,acc2.input);

      //Connections for the results
      connect(acc.output,data5.input);
      connect(acc2.output,data6.input);
}
```

The second part implements the mutation actions. At this point the elements, which will not be necessary any more, must be disconnected from the others and the new elements must be instantiated and receive new connections.

In the source-code below it is possible to observe the Mutations' inner class *ConfigMultiplication* being implemented. This class requests the Ptolemy to a model Mutation. It must extend the *ChangeRequest* class and implements the _execute() method. When the simulation director handles a specific *ChangeRequest* it calls the respective _execute() method. In the example below the _execute() method configures the model to use the multiplier instead of the adders (as seen in **Figure 5**). It creates two new elements (the multiplier and another register), disconnects all unnecessary elements' ports and establishes new connections. All disconnected ports must call the *createReceivers*() method to avoid eventual conflicts. This method sends a message to simulation's director showing it which elements will be disconnected. Finishing, the simulation director (declared in the Mutation class) must call the *invalidateSchedule*() method to recreate the elements' execution schedule. The execution schedule dictates the simulation execution sequence. When any

element's port changes the simulation's director must re-organize the execution schedule to avoid any try at executing not connected elements.

```
//Create the inner class the request the new model configuration.
private class ConfigMultiplication extends ChangeRequest{

      TypedCompositeActor owner;
      String name;

      //Constructor
      ConfigMultiplication(TypedCompositeActor owner,String name){
        super(owner,name);
        this.owner = owner;
        this.name = name;
      }
      //This method execute the model mutation.
public void _execute() throws IllegalActionException,
NameDuplicationException {

      //New elements are created
      multi = new Multiplier(this,"Multiplier");
      data7 = new Register(this,"Register7");
      //Disconnection
acc.output.unlinkAll();
acc.input.unlinkAll();
acc2.output.unlinkAll();
acc2.input.unlinkAll();
      data1.trigger.unlinkAll();
      data1.input.unlinkAll();
      data2.trigger.unlinkAll();
      data2.input.unlinkAll();
      data3.trigger.unlinkAll();
      data3.input.unlinkAll();
      data4.trigger.unlinkAll();
      data4.input.unlinkAll();
      data5.trigger.unlinkAll();
      data5.input.unlinkAll();
      data6.trigger.unlinkAll();
      data6.input.unlinkAll();
      data7.trigger.unlinkAll();
      data7.input.unlinkAll();

      //New connections are established.
      owner.connect(clock.output,data7.trigger);
      owner.connect(clock.output,data5.trigger);
      owner.connect(clock.output,data6.trigger);
      owner.connect(data5.output,multi.input);
      owner.connect(data6.output,multi.input);
      owner.connect(multi.output,data7.input);
      owner.connect(data7.output,display.input);

      // Any pre-existing input port whose connections
      // are modified needs to have this method called.
      data5.input.createReceivers();
      multi.input.createReceivers();
      data7.input.createReceivers();
      data7.trigger.createReceivers();
      data6.input.createReceivers();
```

```
      data5.trigger.createReceivers();
      data6.trigger.createReceivers();

      //Recreate the elements' execution schedule
      director.invalidateSchedule();
  }
}
```

The last part is to decide when the Mutation should be executed, the Mutation condition. In the presented application the condition is the simulation-time be less than one (the Ptolemy simulation-time starts with zero). As can be observed in the source code below, this condition was implemented in the Mutation's method *postfire*(). Ptolemy calls this method after every execution cycle, when all elements execute their actions (called Fire by Ptolemy). The Mutation condition should be anyone and be implemented at any desired moment.

When the Mutation condition is valid, a *ConfigMultplication* object is instantiated, which requests the simulation director to model mutation. The *changeRequest*() method is called passing the *ConfigMultiplication* as parameter. When this occurs the director will queue the request and as soon as possible it will retrieve it from the queue and execute its *_execute()* method.

```
public boolean postfire() throws IllegalActionException {
super.postfire();

//Request mutation after 1 simulation cycle.
if(director.getCurrentTime() < 1){
      ConfigMultiplication changeMulti = new
ConfigMultiplication(this,"ExecMutation");
      //Mutation request is queued
      requestChange(changeMulti);
}
return true;
}
```

4. Results

In code below is possible to analyze the execution time and the mutation execution. At simulation time t=0.0, the register are initialized with two (as test). At time t=1.0 the registers data are passed to the adders which execute the sum and store the result in the registers five and six.

In real hardware the reconfiguration would be done during t=1.0 clock signal, after the sum execution and before the t=2.0 clock signal to come to the registers five and six. The multiplication hardware would be programmed and prepared before time t=2.0, so it could execute with the registers five and six as inputs.

In simulation it is different, at time t=2.0 the registers five and six receive a clock signal (before the reconfiguration), but do not send data to any device because it has no connections. During this clock period (t=2.0) the simulation is paused, the reconfiguration is

done and the simulation is resumed with the multiplication ready on the next clock signal. It is not necessary to worry about synchronization because the simulation is paused during model reconfiguration. The reconfiguration time delay can be modeled and analyzed using a Ptolemy timed delay element.

When the simulation postfire() method is called, the time is not less than one, and the Mutation is requested. At time t=2.0, the register five and six still stores the same values which are passed to the multiplier, which execute the multiplication and store the result in the Register7.

The Ptolemy does not support the actor destruction during mutation executions. So it is only possible to know which elements are configured at any simulation time listing all elements which has neither input nor output ports with active connections. As the objects were not destroyed, just disconnected, Java's garbage collector will not make memory available, so they can still be accessed.

```
t=0.0 -> Register1 = 2
t=0.0 -> Register2 = 2
t=0.0 -> Register3 = 2
t=0.0 -> Register4 = 2
t=1.0 -> Register5 = 4
t=1.0 -> Register6 = 4

t=1.0 -> *** Reconfiguration to Multiplication ***
t=2.0 -> Register5 = 4
t=2.0 -> Register6 = 4
t=3.0 -> Register7 = 16
```

The **Table 1** below shows the number of elements and connections necessaries when the application uses *Execution Patterns* and when it does not. It is possible to note that in the first execution of the *Execution Patterns* (first step), the numbers are better but similar to the number without *Execution Patterns*. The performance gain is better in the second moment when just the multiplier is configured.

Number of Elements	Without Execution Patterns	First moment with Execution Patterns	Second moment with Execution Patterns
Register	7	6	3
Adder	2	2	0
Multiplier	1	0	1
Connection	9	6	4
Total	19	13	8

Table 8: Allocated elements compared with and without Figures de Execution.

4. Final Considerations

During this research a dynamically re-configurable model, which uses the Execution Patterns, was simulated using the Ptolemy tools.

The connections gain is visible is these simulation when the Execution Patterns is used. The Execution Patterns potential are still undefined and other applications will be

studied to this potential be achieved. A higher connection gain is expected for Execution Patterns when high-connected applications are simulated.

Here a different simulation way was presented, where the elements are really removed and created during simulation time. In the usual strategies gates are used to simulate the change of elements during reconfigurations. The reconfigurations are just a change of the execution path but this reconfiguration logic could not be directly synthesized. The simulation using Ptolemy Mutation enables a future creation of a strategy to synthesize dynamically reconfigured hardware directly.

As future works the development of a dynamically reconfiguration logic generator which uses a unique strategy to map each conventional application into its respective solution with Execution Patterns. This generator could also be part of a complete Execution Patterns simulation tool where the conventional applications are converted to their respective Execution Patterns solution and are simulated, enabling performance reports creation contrasting each solution. Studies are also being developed to use others application as SystemC integrated with Ptolemy, and to use the JHDL for dynamically reconfiguration hardware projects. In [A17] is presented a methodology for modeling of dynamically re-configurable blocks at the system-level using System-C is presented, in future works this methodology can be used to our purpose.

References:

[A1] DEHON, A. Comparing Computing Machines. *Proceedings of SPIE: Configurable Computing Technology and Applications vol. 3526,* November 2-3, 1998.

[A2] W.H. MANGIONE-SMITH, B. HUTCHINGS, D. ANDREWS, A. DEHON et. al. Seeking solutions in Configurable Computing. *Computer*, 30(12):38-43, December 1997.

[A3] A.DEHON, J. WAWRZYNCK. Reconfigurable Computing: What, Why, and Implications for Design Automation. *Proceedings of the 1999 Design Automation Conference*, DAC'99, June 21-25, 1999 (http://www.cs.berkeley.edu/~amd /papers.html).

[A4] MARKOVSIY, Y., CASPI E., HUANG R., YEH J., MICHAEL C., J. WAWRZYNCK. Analysis of Quasi-Static Scheduling Techiniques in a Virtualizad Reconfigurable Machine. *Proceedings of FPGAs 2002,* ACM, Monterey, California, USA. February 2002.

[A5] P. BELLOWS AND B. HUTCHINGS. JHDL - An HDL for Reconfigurable Systems. In *IEEE Symposium on Field-Programmable Custom Computing Machine*s, April 1998.

[A6] W. LUK AND S. GUO. Visualising reconfigurable libraries for FPGAs. In *Asilomar Conference on Signals, Systems, and Computer*s, 1998.

[A7] G. BREBNER. CHASTE: a Hardware/Software Co-design Testbed for the Xilinx XC6200. In *Reconfigurable Architectures Workshop, RAW'97*, April 1997.

[A8] P. MACKINLAY, P. CHEUNG, W. LUK, AND R. SANDIFORD. Riley-2: A Flexible Plat-form for Codesign and Dynamic Reconfigurable Computing Research. In *7th International Workshop on Field-Programmable Logic and Application*s, September 1997.

[A9] BOOCH, GRADY. Object-Oriented Analysis and Design with Applications. Addison-Wesley Pub Co; 2nd edition, 1993.

[A10] GROTKER, T., LIAO, S., MARTIN, G., SWAN, S., System Design with SystemC. Kluwer Academic Publishers, 2002.

[A11] CUMMINGS, CLIFFORD E., Synthesizable Finite State Machine Design Techniques Using the New SystemVerilog 3.0 Enhancements*, SNUG*, 2003, San Jose, CA.

[A12] SystemVerilog, www.systemverilog.org (acesso em 20/03/2004).

[A13] C. HYLANDS, E. A. LEE, J. LIU, X. LIU, S. NEUENDORFFER, Y. XIONG, H. ZHENG (eds.), Heterogeneous Concurrent Modeling and Design in Java,(Volume 1:Introduction to Ptolemy II). Technical Memorandum UCB/ERL M03/27, University of California, Berkeley, CA USA 94720, July 16, 2003.

[A14] LUIZ BRUNELLI, Neurocomputação e Execution Patterns. http://www.dee.ufcg.edu.br/~lubru/qlf_lb.pdf. (acesso em 20/03/2004).

[A15] M. CHU, N. WEAVER, K. SULIMMA, A. DEHON, AND J. WAWRZYNEK. Object Oriented Circuit-Generators in Java. In *IEEE Symposium on FPGAs for Custom Computing Machine*s, April 1998.

[A16] D. LEVI AND S. GUCCIONE. Run-Time Parameterizable Cores. In *ACM International Symposium on Field Programmable Gate Arrays*, February 1999.

[A17] PELKONEN A., MASSELOS K., CUPÁK M. System-Level Modeling of Dynamically Reconfigurable Hardware with SystemC. In *International Parallel and Distributed Processing Symposium (IPDPS'03)*, Nice, France, April 22, 2003.

[A18] DEECS (Department of Electrical Engineering and Computer Sciences), "Ptolemy II, Heterogeneous Concurrent Modeling and Design in Java. Volume 3: Ptolemy Domains", julho de 2003. *University of California at Berkeley*. Disponível em http://ptolemy.eecs.berkeley.edu. Acesso em: maio de 2004.

Apêndice B

PROJETO E PESQUISA III: MODELAGEM E SIMULAÇÃO DE RECONFIGURAÇÃO DINÂMICA EM MODELOS COMPORTAMENTAIS COM VISTAS AS FIGURAS DE EXECUÇÃO

Período: **2005.1**
Aluno: **Alisson Vasconcelos de Brito**
Orientador do Projeto: **Elmar Uwe Kurt Melcher**

1. Introduction

One of the problems of Dynamically Reconfigurable (DR) systems is to know which and when the reconfigurable elements will make use of this functionality. DeHon [B1] lists many relevant works in Reconfigurable Systems research classifying them by purpose. According to this paper, problem can be classified as an "Area-Time Tradeoffs Pattern" and as a "Partial Reconfiguration Pattern". It is an "Area-Time Tradeoffs Pattern" because it aims to reduce chip area with the least delay overhead possible, and it is a "Partial Reconfiguration Pattern" because it uses partial reconfiguration to reduce this delay overhead. Other important contribution of this paper is in the design process. It proposes a new methodology in order to analyze and suggest the use of partial reconfiguration with minimum overhead. We believe that the better strategy to attack partial reconfiguration and design process overhead is using the combination of system-level modeling and object-oriented software methodology. Actually the open-source project which better combines these two points is the SystemC [B10] hardware description language.

There are results in both of the approaches which use hardware system-level specification. One project that can be classified as "Area-Time Tradeoffs Pattern" and make use of dynamically reconfigurable system-level analysis and modeling using SystemC is the Adriatic Project [B2], which proposes a system-level modeling technique to dynamically reconfigurable systems with co-processors which concentrate in reconfiguration candidates selection, reconfiguration overhead

modeling for fast space exploration and project reuse. In order to do candidate selection it receives a special tasks specification in C language and generates a Control Data Flow Graph (CDFG). After that, all necessaries resources are estimated for all tasks. A Control Scheduler is created to act during simulation which is connected to all modules to determine when which one must be activated. The system-level modeling and simulation are implemented with SystemC. This language allows hardware description in Register Transfer Level (RTL) and system-level. The implementation is made using C++ language and it is an open-source project. The approach proposed in this paper aims at a different methodology using the same system-level specification with SystemC to analyze the possible reconfiguration candidates and simulate them with minimum new code insertion. This approach has the advantage to selects the reconfiguration candidates based on the behavior during simulation, so the designer does not need to write an extra reconfiguration specification for his system.

In the approaches of the "Partial Reconfiguration Pattern" type a technique to solve temporal partitioning based on two phases is presented [B3], the properties of interconnected operations is estimated and the communications between temporal partitions are qualitatively evaluated The main goal is system partitioning with minimum communication cost using operation granularity estimations at system-level . The implementations of dynamically reconfigurable system using DR-FPGAs (Dynamically Reconfigurable-FPGAs) and proprietary tools from Xilinx are shown [B4].

Our approach is based on Execution Patterns [B5]. This work specifies a technique to group elements physically in a circuitry forming Execution Patterns. The idea is to insert, or remove, these Execution Patterns as soon as they are necessary, or unnecessary in run-time. This work has shown that reductions in interconnections and chip area can be achieved, but it is impracticable with available DR-FPGAs. The circuitry placement in such a fine granularity is another problem [B6] and a methodology to physically positioning and allocation of modules in a DR-FPGA is proposed. Our idea now is to perform the strategy of how the elements are chosen to form each Execution Pattern, analyzing how is their behavior during simulation, so we can decide which elements should be grouped with each other in a coarse granularity, decreasing the placement and timing tradeoffs. Our solution also analyze

the systems during project simulation phase, so the designers can know in advance if where to use the Partial Reconfiguration, enabling also the simulation and comparison of the results for a implementation in DR-FPGA with respect to implementation in traditional FPGA.

Another problem is to simulate partial reconfiguration at system level. The OSSS+R project [B7] uses the concepts of Object-Orientation to simulate Partial Reconfiguration implementing a SystemC extension. The simulation is based on adaptation of the SystemC language to accept the Object-Orientation concept of Polymorphism, enabling modules with the same interfaces to be exchanged at simulation-time.

This paper presents the Execution Pattern concepts in session 2. During session 3 is presented our methodology for system analysis based on system-level behavior. A practical example simulated and some results are presented in session 4.

2. Execution Patterns

Execution Patterns are based on spatial placement and partial reconfigurability. Spatial placement maps computation directly in hardware at a specific area using a device like a Dynamic Reconfigurable Field-Programmable Gate Array (DR-FPGA) [B11-B15].

Partial reconfigurability allows the reconfiguration of a part of a programmable logic device while other parts continue to operate. This decreases the devices resources necessary because some parts of a logic circuit may get loaded into the device only when they are actually needed.

In the approach presented in this paper, partial reconfigurability is used to instantiate each Execution Pattern (EP), while spatial placement is used to place each EP at a determined and geometric position in the DR-FPGA. An EP can receive and provide data at its border, where so-called data bars must be instantiated. Data bars can be implemented for example as latches. Data exchange between EPs occurs when data are latched to a data bar from an adjacent EP border and then passed on to another EP which is placed adjacent to this same fixed data bar. Note that in this approach the only elements that are moving during execution are the processing elements. The data bars remain in place.

Considering a simplified Neural Network implementation, only a set of

Multiply and Accumulates Units (MACs) are used. The following Equation 1 of six MACs and two sums illustrate how EPs can be used.

$$\begin{cases} z_1 = x_1 w_{11} + x_2 w_{21} \\ z_2 = x_1 w_{12} + x_2 w_{22} \\ z_3 = x_1 w_{13} + x_2 w_{23} \end{cases} \qquad (1)$$

A possible implementation, including place and route of the EPs is shown in Figures 1 and 2.

Initially three identical sets of data and EPs, called scenes, are configured, making up Act 00 (Figure 1). After execution of Act 00 the results *y1* to *y6* are obtained.

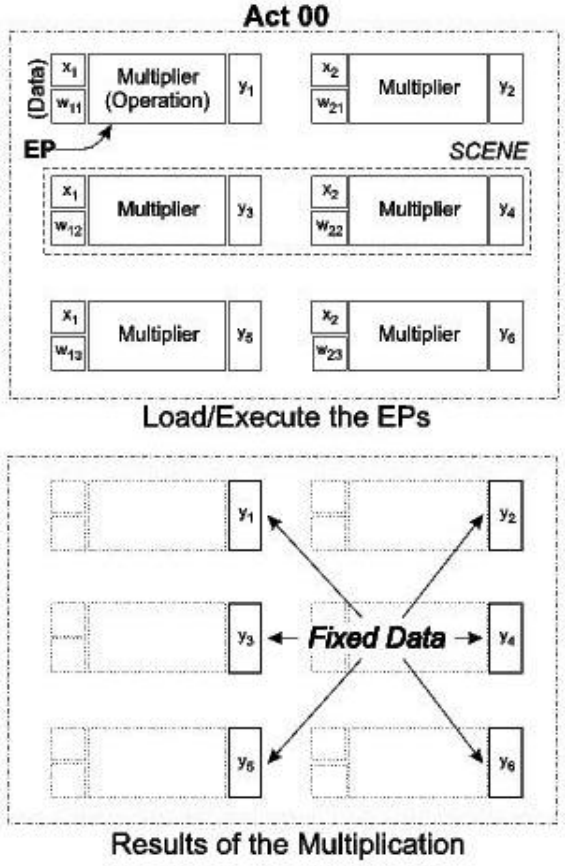


Figure 3: Results of the Multiplication

Following this execution, Act 01 (Figure 2) is configured, maintaining the position of the data bars that hold *y1* to *y6*. The sum EPs are placed adjacent to the two data bars that hold their respective argument values. After execution of Act 01, *z1*

to *z3* are obtained.

Processing was performed without data transport, scenes were executed in parallel and the resources of the DR-FPGA are used exclusively for the resources of the DR-FPGA are used exclusively for the execution of the processing step at hand, thus increasing its functional density. Instead of using reconfigurable resources of the DR-FPGA for data transport, fixed and dedicated resources for reconfiguration are used to place the processing units next to the data.

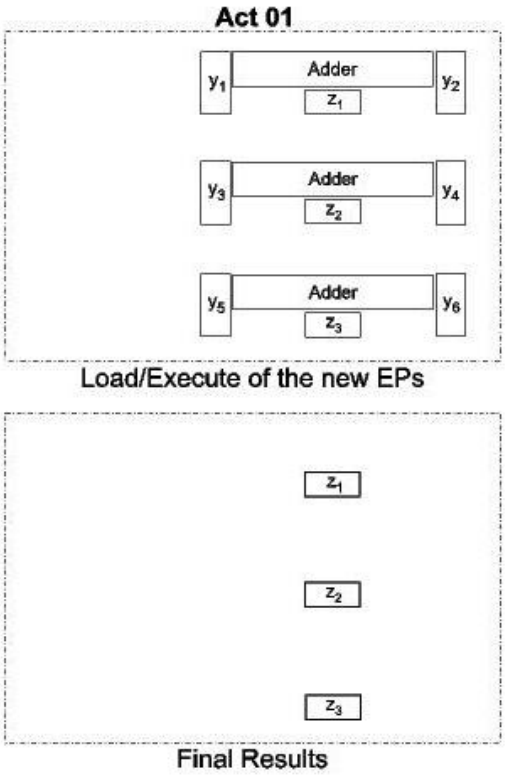


Figure 4: MACs in a DR-FPGA using EPs (accumulating phase)

3. System-Level Simulation Behavior

The main contribution of our methodology is in how the designer can know in advance the benefits of Partial Reconfiguration that its system can acquire. The transparency of this analysis is in implementation of an Observer software design pattern from Erick Gamma [B8]. In our solution the Observer is implemented in an extension of SystemC input port (*sc_in*). This port extension (*sc_dr_in*) is the same port as its base class, but it also contains an attribute called "Observer". Figure 3 presents a Class Diagram with the relationship among *sc_dr_in* and basic SystemC

classes.

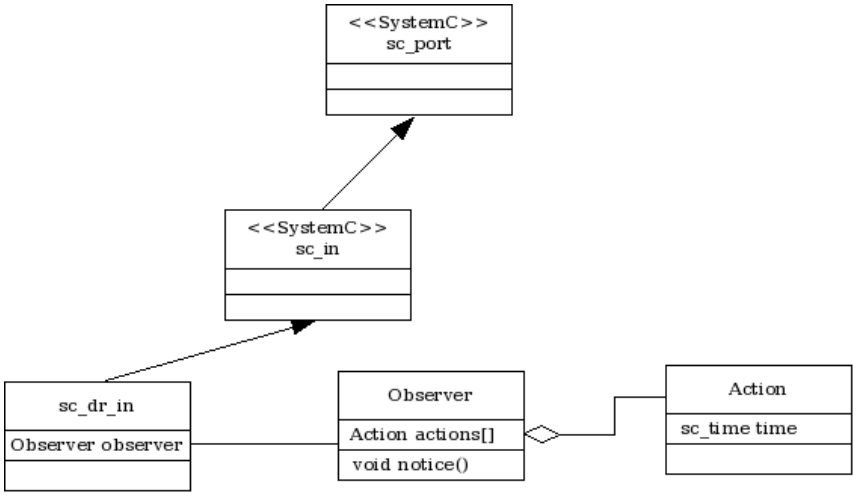


Figure 5: *sc_dr_in* relationship with SystemC classes.

When every relevant input passes throw the input port, it notifies the Observer, which stores when it has occurred. The Figure 4 below shows how modules are linked in the example and the relationship among them. Each module has a *sc_dr_in*, which is connected to an Observer. An Observer is not a module, but a data structure that stores each module occurrence during simulation. It is not synthesizable and, as it is an attribute of *sc_dr_in*, after simulation and system analysis, the *sc_dr_in* can be replaced by a traditional *sc_in* and the Observer will also be consecutively removed from system.

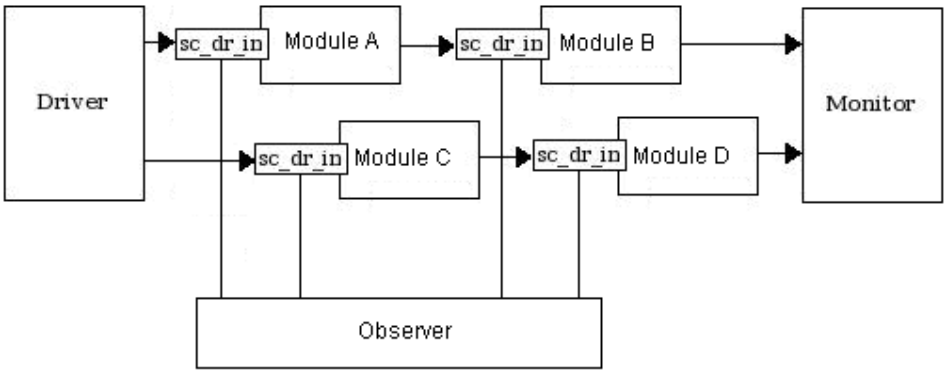


Figure 6: Example showing how *sc_dr_in* can be connected to modules.

4. Simulation and Results

A simulation using our methodology is presented in this session. The additional functionalities of our methodology are easy to implement and almost

transparent, differing from traditional SystemC programming only in declaration of *sc_dr_in* instead of traditional SystemC *sc_in*. Figure 5 below presents the use of *sc_dr_in* implemented in *ModuleA* (*moduleA.h)* from example of Figure 4.

```
//moduleA.h
#include "lib.h"

SC_MODULE(moduleA){

 sc_out<bool> out;
 sc_dr_in <bool> in;

 void prc_mod_a();

 SC_CTOR(moduleA){
   SC_METHOD(prc_mod_a);
   sensitive << in;
 }
};
```

Figure 7: Implementation of ModuleA from Figure 4 with a *sc_dr_in* as input port.

At the end of the simulation, it is possible to know when exactly each module port was activated. In the main routine it is necessary to specify how Observer data should be processed and printed. Figure 6 shows this routine.

```
#include <systemc.h>
#include <driver.h>
#include <monitor.h>
#include <moduleD.h>

int sc_main(int argc, char* argv[]){

 sc_signal<bool> input1("in1"),input2("in2"),input3("in3"),input4("in4");
 sc_signal<bool> ouput1("out1"),output2("out2"),out3("out3"), out4("out4");

 moduleA *modA = new moduleA("moduleA");
 modA->in(input1);
 modA->out(output1);

 moduleC *modC = new moduleC("moduleC");
 modC->in(input2);
 modC->out(output2);

 driver *_driver = new driver("signals");
 _driver->d_a(input1);
 _driver->d_b(input2);

 monitor *_monitor = new monitor("monitor");
 _monitor->m_a(output3);
 _monitor->m_b(output4);

 moduleB *modB = new moduleB("moduleB");
 modB->in(input1);
 modB->out(input3);

 moduleD *modD = new moduleD("moduleD");
 modD->in(output2);
 modD->out(output4);
 Analyzer analyzer;
 analyzer.add( modA->in.getObserver());
 analyzer.add( modB->in.getObserver());
 analyzer.add( modC->in.getObserver());
 analyzer.add( modD->in.getObserver());
 return(1);
}
```

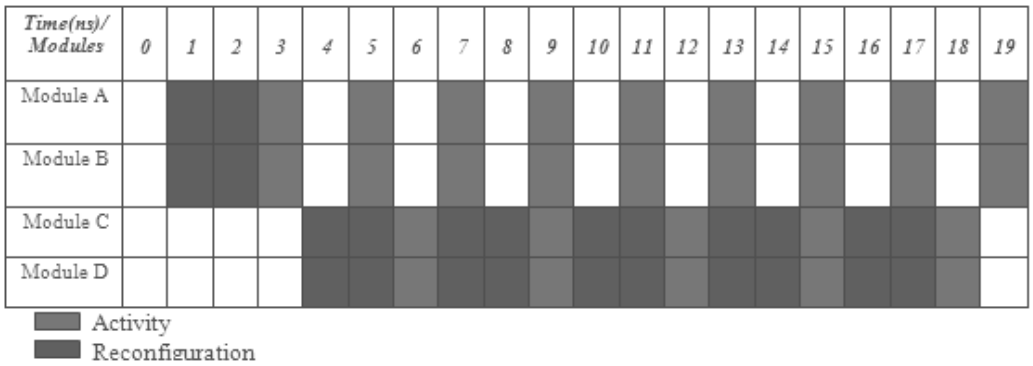
Figure 8: Simulation main routine of example from Figure 4.

This information is passed as a graph to a Neighborhood Analyzer, which will examine at each simulation instant in which ports were activated. Paper [B9] also uses Finite-State Machines, possible to be presented as a graph to model systems and their dynamic reconfiguration. The modules that are frequently activated together are considered neighbors modules, and elements that are rarely or never activated together are considered separated modules. The concept of distance here depends on what is considered as "activated together", so the Neighborhood Analyzer is parameterized enabling the designer to configure what is consider by his project as near. This parameterization can be used by designer to set reconfiguration performance constraints. If the dynamic reconfiguration implemented by the used chip is slower than another, it is necessary to decrease the distance concept to avoid the separation of modules when the chip is not able to configure and remove the

respective modules on time.

Table 1 below shows graphically the analysis made by Observers and the analysis result obtained by simulation of example (Figure 4). The portions on red represent the modules activities and the portion on blue represents when each module should be reconfigured.

Table 1: Modules Activities



The analyzer try to use dynamic reconfiguration as much as possible, so, if there is time, it will remove all modules that are not working, and reconfigure them again when necessary. In this example the reconfiguration constraint passed to Neighborhood Analyzer is 1ns, so each module needs 1ns to be configured in the chip.

This table shows that modules A and B were considered neighbors, should always be configured together and they cannot be remove from the chip because there is no time do reconfigure them on time. Modules C and D are also neighbors, needing to be configured together and there is enough time to remove them and reconfigure when necessary.

In this example, the Analyzer suggested the separation of the system in two different Execution Patterns: {*ModuleA*, *ModuleB*} and {*ModuleC*, *ModuleD*}. This grouping strategy is important to decrease the time lost occurred during successive reconfigurations. As module C and D will always be configured together, the system will lost time just once to reconfigure both of them, instead of losing time for each module on different instants.

All these information should be passed again to the simulator, so it could simulate how the system behavior with dynamic reconfiguration is. Figure 7 presents a time diagram of how the modules would work after the creation of the Execution Patterns. The absence of a simulation tool not allows the estimation of the benefits

that this and other systems would achieve using the Execution Patterns concept, neither how much their implementation would cost for a project.

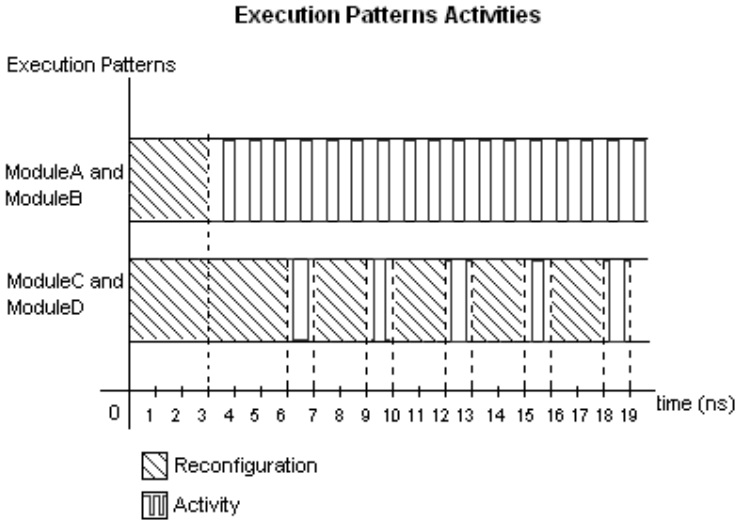


Figure 9: Execution Patterns activity.

5. Final Considerations

The absence of simulation tools which support dynamic reconfiguration does not allow evaluate the benefits that the techniques presented here can provide. The implementation of a simulation tool is the focus of further works.

6. References

[B1] DEHON, A., ADAMS, J., DELORIMIER, M., KAPRE, N., MATSUDA Y., NAEIMI, H., VANIER, M., WRIGHTON, M. Design Patterns for Reconfigurable Computing. *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April, 2004.

[B2] QU, Y., TIENSYRJA, K., MASSELOS, K., System-Level Modeling of Dynamically Reconfigurable Co-Processors, International Conference on Field Programmable Logic and Applications, Antwerp, Belgium, August-September 2004

[B3] ZHANG, XJ., NG, KW. A Temporal Partitioning Approach based on Reconfiguration Granularity Estimation for Dynamically Reconfigurable Systems. Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT'2003), Tokyo, Japan, December, 2003.

[B4] FERRANDI, F. MARCO D. SANTAMBROGIO, SCIUTO, D. A Design Methodology for Dynamic Reconfiguration: The Caronte Architecture. Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS'2005), 2005, Denver, CA, USA

[B5] BRUNELLI L., MELCHER, E. U. K., BRITO, A. V., FREIRE, R. C. S., A novel Approach to reduce Interconnect Complexity in ANN Hardware Implementation, Proceedings of the International Joint Conference on Neural Networks (IJCNN'2005), Montreal, Canada, July, 2005.

[B6] KÖSTER, M., PORRMANN, M., RÜCKERT, U. Placement-Oriented Modeling of Partially Reconfigurable Architectures. Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS 2005)- Reconfigurable Architectures Workshop, Denver, CA, USA, 2005.

[B7] SCHALLENBERG, A., OPPENHEIMER, F., NEBEL, W. Designing for dynamic and partially reconfigurable FPGAs with SystemC and OSSS, Forum on Specification and Design Languages (FDL ´04), Lille, France, Sept. 2004

[B8] GAMMA, E., HELM R., JOHNSON, R., VLISSIDES, J. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional; 1st ed., 1995.

[B9] MARKOVSIY, Y., CASPI E., HUANG R., YEH J., MICHAEL C., J. WAWRZYNCK. Analysis of Quasi-Static Scheduling Techiniques in a Virtualized Reconfigurable Machine. Proceedings of FPGAs 2002, ACM, Monterey, California, USA. February, 2002.

[B10] GROTKER, T., LIAO, S., MARTIN, G., SWAN, S. System Design with SystemC. Kluwer Academic Publishers, 2002.

[B11] DEHON, A.; WAVRZYNEK, J. Reconfigurable computing: What, why and implications for design automation. In: Proceedings of the 36th ACM/IEEE Conference on Design Automation. USA: ACM PRESS, 1999. p. 610 – 615.

[B12] COMPTON, K.; HAUCK, S. Reconfigurable Computing: a survey of systems and software. ACM Computing Surveys, v. 34, n. 2, p. 171-210, June 2002.

[B13] ATMEL. Application Note: Implementing Cache Logic with FPGAs. USA, Sep. 1999. Avalilable in http://www.atmel.com/dyn/resources/prod_documents/DOC0461.PDF. Accessed in November, 2003.

[B14] OLDFIELD, J. V.; DORF, R. C. Field-Programmable Gate Arrays. USA: John Wiley & Sons Inc., 1995. ISBN 0-47155-665-1.

[B15] LYSAGHT, P.; DUNLOP, J. Dynamic Reconfiguration of Field Programmble Gate Arrays. In: MOORE, W.; LUK, W. (Ed.). More FPGAs: Proceedings of the 1993 International Workshop on Field-Programmable Logic and Applications. Oxford, England: Abingdom EE&CS Books, 1993. p. 82-94. Available in

http://oak..eee.strath.ac.uk/publications.html , accessed in October, 2003.