UNIVERSIDADE FEDERAL DE OURO PRETO

tite

Rodolfo Labiapari Mansur Guimarães
Universidade Federal de Ouro Preto

Orientador: Ricardo Augusto Rabelo Oliveira

Dissertação submetida ao Instituto de Ciências Exatas e Biológicas da Universidade Federal de Ouro Preto para obtenção do título de Mestre em Ciência da Computação

Ouro Preto, Maio de 2017

tite

Rodolfo Labiapari Mansur Guimarães
Universidade Federal de Ouro Preto

Orientador: Ricardo Augusto Rabelo Oliveira

Uma Abordagem Híbrida para Resolver o Problema da Escala de Motoristas de Ônibus Urbano

Resumo

A alocação da tripulação (motoristas e cobradores) é uma etapa muito importante no planejamento operacional do Sistema de Transporte Público visto que custo operacional representado pelas escalas de trabalho compõe uma parcela significativa nos custos totais de uma empresa de transporte público. A redução dos custos das escalas de trabalho afetam não só as empresas operadoras, mas também os usuários deste serviço, pois com esta redução há a possibilidade de um maior investimento na qualidade do transporte público e a redução dos preços dos bilhetes. Estes custos, estão estritamente relacionados as normas operacionais impostas pelas empresas e legislações trabalhistas que se refletem na definição das jornadas de trabalho dos motoristas e cobradores.

Esse trabalho tem a finalidade de propor um novo método computacional capaz de auxiliar o processo da programação da tripulação em empresas de transporte público de ônibus urbano. Os métodos apresentados nesta pesquisa são baseados no uso de um modelo de programação linear inteira, ainda inédito na literatura, se diferindo dos demais pelo fato de que cada jornada é gerada diretamente a partir das tarefas a serem alocadas. Uma metaheurística *Late Acceptance Hill Climbing* (LAHC) também foi utilizada com o objetivo de resolver problemas de maiores dimensões. Um método híbrido, utilizando o método exato e a metaheurística LAHC, é proposto com o objetivo de obter um refinamento das soluções obtidas pela metaheurística, de modo a reduzir os custos das jornadas geradas. Para avaliar as abordagens apresentadas foram utilizadas instâncias geradas a partir de dados reais de uma empresa do setor de transporte público da cidade de Belo Horizonte/MG. Os modelos computacionais propostos apresentaram

resultados satisfatórios, sendo que os custos finais foram reduzidos para a maioria dos testes realizados. Por outro lado, há a necessidade de novos estudos sobre os métodos apresentados, a fim de que os mesmos se tornem mais eficientes.

Palavras-chave: Matheurística, Problema de Programação de Tripulações, Metaheurística.

A Hybrid Approach to Solve the Problem of Scale for Urban Bus Drivers

Abstract

The allocation of crew (drivers and collectors) is a quite important stage of operational planning of Public Transit System once the operational cost represented by the work schedules consist in a significant portion of total costs in a public transit company. Cost reduction of work schedules affects not only the operating company but also the users of this service since there is chance of higher investments in transit quality and reduction of ticket prices because of this cost-cutting. These costs are strictly related to the operational rules established by companies and work laws which reflects themselves in the transit drivers and collectors work schedule definition.

The goal of this work is to propose a new computational method able to assist the crew planning process in urban bus public transit companies. The methods presented in this work are based on use of an integer linear programming method even unpublished in literature, being different from others due the fact that each schedule is created directly from tasks to be allocated. A metaheuristic *Late Acceptance Hill Climbing* (LAHC) was also utilized with the purpose of solving bigger problems. A hybrid method using the exact method and the metaheuristic LAHC is proposed with the goal of refining solutions gotten through metaheuristic, reducing created schedule costs. To evaluate the presented approaches, problems generated from real data from a public transit company from Belo Horizonte/MG city were used. The proposed computational methods presented satisfactory results and final costs were reduced for most tests performed. On the other hand, other researches about the presented methods are necessary in order that they become more efficient.

Keywords: Matheuristic, Crew Scheduling Problem, Metaheuristic.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Nomenclatura

VLNS *Very Large-Scale Neighborhood Search*

Capítulo 1

Introdução

Segundo Sass, há trinta anos, quando se pensava em sistemas embarcados, falava-se em máquinas computacionais personalizadas construídas a partir de circuitos eletrônicos discretos e integrados. Isso significa que o foco era totalmente em hardware e em design de hardware. O software, por sua vez, era somente uma parte do sistema que integrava tudo. Somente produtos fabricados em grande escala justificavam o custo de construir hardwares personalizados. Com o passar do tempo a tecnologia foi reduzindo o volume dos componentes sem ter o trade-off [1] em vazão e custo e assim era melhor iniciar um projeto com componentes baratos, provendo uma plataforma computacional semi-personalizada. Entretanto, a solução final exigia mais esforço em software para gerar uma aplicação para um comportamento específico, e com isso, o desenvolvimento de software passou a ser mais custoso que o desenvolvimento em hardware se tornando principal fator no custo de desenvolvimento de um produto.

Dispositivos como Plataformas FPGA quebraram tal situação. Isso pelo fato de que Plataformas FPGAs permitem ao designer de sistemas embutidos ter uma "lousa branca" e que possa implementar hardwares computacionais personalizados tão facilmente como o desenvolvimento de um software. Infelizmente, enquanto configurar um FPGA é uma tarefa fácil, criar um design de hardware inicial não é. Seria inviável implementar toda vez cada hardware que fosse necessário em projetos. Por isso, utilizam-se hardwares de cores[2] já existentes e desenvolvimento de novas cores para seu reuso. Basta saber como quais cores serão e como são utilizados, suas performances e como desenvolve novos cores com intuito de deixá-los reutilizáveis. A partir de então, designers de Plataformas

[1] Conflito de escolha. A resolução de problema acarreta noutro.
[2]

FPGA devem começar a entender mais sobre desenvolvimento de software como sistemas operacionais e dispositivos controladores do que simples aplicações, para trazer o código escrito para a aplicação de fato. Utilizando hardwares comerciais, muito desses problemas de plataformas poderiam ser escondidos em sistemas embarcados porque grande parte dos componentes possuem ferramentas e sistemas de baixo nível de software para manuseio e dessa forma, ambos hardware e software agora são programáveis.

Princípios de Design de um Sistema Para dar um significado a alguns componentes de design, alguns conceitos devem ser definidos a priori. Para declarar um design bom ou ruim, precisa-se definir dois critérios, o critério externo e interno. Os critérios externos são características que um usuário observaria. Um exemplo é o mau funcionamento de um componente onde em vez de aumentar o volume, é reduzido ao tentar-se elevar, problema no qual pode ser percebido facilmente por um usuário. Critérios internos são características que são inerentes, ou seja, dependentes à estrutura ou organização do design, mas não necessariamente observável pelo usuário. Um exemplo é o usuário não saber que tipo de codificação é utiliza no seu dispositivo sonoro, mas algumas possam estar interessadas em sistemas com alta qualidade de design a ponto de impactar em seu conserto ou durabilidade. Algumas dessas características podem ser mensuráveis quantitativamente, mas muitas vezes de forma subjetiva. O primeiro conjunto de termos a ser definido está relacionado à performance do sistema. A corretude usualmente significa que o sistema está, matematicamente, atendendo a uma especificação formal. Isso pode ser um gasto grande de tempo, mas em alguns casos, algumas porções de sistemas devem ser formalmente verificadas, como por exemplo quando uma vida humana está em risco. Os dois outros termos estão relacionados com a corretude são, a segurança e a resiliência, também chamado de robustez. A segurança depende de onde é aplicado (hardware ou software). Um sistema confiável em hardware significa que ele funcionará corretamente mesmo com uma presença de falha física como por exemplo, corrompimento de memória por meio de radiação cósmica. Isso pode ser obtido por meio de redundância e procedimentos que utilizam a técnica on the fly que permite que o sistema se recomponha automaticamente mesmo com falha. Já em software significa que o sistema funcionará corretamente apesar da especificação formal estiver incompleta. Um exemplo é quando um disco enche, o sistema deve parar de escrever mesmo sem uma especificação formal sobre. Isso é importante pois a maioria dos sistemas são grandes demais para especificar formalmente cada comportamento. A resiliência (ou robustez), segue quase o mesmo caminho que a segurança. Enquanto segurança foca em detectar e corrigir todos os corrompimentos, a resiliência aceita o fato de que erros ocorrerão e que o design contornará o problema mesmo com degradações. Assim, segurança é agir sobre algo não

especificado e resiliência é agir sobre algo que nunca tenha acontecido. E por fim existe a confiabilidade onde pode-se pensar como um espectro, onde um lado tem-se proteção contra fenômenos naturais e em outro, ataques maliciosos. O sistema de proteção deve proteger de ambos os lados. Tais termos não levam uma quantidade associadas com eles. Mas, sendo conciso com cada um deles, o desenvolvimento de um sistema embarcado pode ser construtivo. Isso será descrito melhor na definição de módulos e interfaces.

Módulos e Interfaces Existem duas, filosofias de design para a construção de um sistema. É possível a especificação dos blocos básicos e conectando-os a fim gerar o sistema completo e tal abordagem é descrita como bottom-up. E também é possível descrever um sistema completo e em seguida ir definindo seus sub-blocos até chegar ao mais básico, sendo esta abordagem chamada de top-down. Para descrever com mais detalhes sobre ambas as abordagens, é preciso primeiro definir os conceitos de módulo e interface. Módulo é qualquer conjunto de operação autocontidas que tem um nome, uma interface e uma descrição funcional. Ou seja, módulo pode ser considerado como uma sub-rotina de software ou um componente em VHDL e pode ser representado como uma caixa, graficamente. Interface formal é um nome do módulo e uma enumeração de suas operações, incluindo as entradas caso existam, saídas e seu nome. É algo que é possível inspecionar mecanicamente. E a interface geral inclui a interface formal e qualquer protocolo adicional ou comunicação implícita. Enquanto a interface formal descreveria por exemplo as funções, entradas e saídas de um gerador de número pseudorrandômico, a interface geral descreveria como seria a interação das funções seed e random, mas sem formalismo. Um módulo pode ter também uma descrição funcional onde esta pode ser implícita ou informal. Quando a descrição é implícita, a descrição está presente de forma clara no nome da função como um fullAdder, onde não existe dúvida sobre seu funcionamento. Já a informal pode ser comentários descritivos mais claros e informais ao longo da escrita da função e a formal consiste na documentação em meios matemáticos e comentários diretos sobre o procedimento/comando. Outros termos importantes são implementação e instância. Uma implementação é a realização de uma funcionalidade pretendida de um módulo, incluindo módulos que tenham várias implementações, como é permitido criar várias arquiteturas em linguagem VHDL. Instância é o simples uso de uma implementação. Enquanto em instâncias são relações um-a-um entre implementação e instância, em hardware é comum o uso de copias, sendo cada cópia representa uma instância. Figura 3.2 e Figura 3.3 Tendo em mente todos estes conceitos, é possível descrever conceitos mais amplos no tema de design de sistema.

Abstração e Estado Abstração é definido no dicionário como operação intelectual em

que um objeto de reflexão é isolado de fatores que comumente lhe estão relacionados na realidade, ou seja, tirar fora, extrair, remover. Assim, uma abstração é a arte ou um artifício de abstrair ou retirar. Dessa forma, um módulo é uma abstração de alguma funcionalidade em um sistema. Um módulo se torna uma boa abstração se suas interfaces e descrições provêm uma fácil compreensão. Mas isso torna sua implementação mais complexa. Uma boa abstração capta todas características importantes e elimina tudo que não é importante para a ideia, ou seja, cria uma organização. Uma imagem rica em informação pode não ser tão relevante imediatamente. Se isso for uma abstração ruim de algo, nos forçará a pensar não sobre a singularidade do módulo, mas também como foi implementado, o que não é uma informação valiosa para uma abstração. Estado já são uma palavra bastante conhecida no âmbito de sistemas eletrônicos. Eles são explicitados num design de máquinas sequenciais e podem ser um ponto de memória de dispositivos eletrônicos como flip-flops. Já em sistemas embarcados, sua definição é um pouco mais abstrata já que um estado de módulo pode ser armazenado em vários lugares em várias formas distintas como flip-flops, static RAM, arquivo, ou até mesmo off-chip. Sendo assim, estado é uma condição de memória onde é possível segurar uma informação por um certo período de tempo.

Coesão e Acoplamento Dessa forma, para conceitos temos abstração e estado e para a mensuração teremos coesão e acoplamento. Coesão é uma métrica de abstração. Se os detalhes dentro de um módulo no ato da implementação são funcionalidades compreendidas facilmente, então o módulo possui coesão. O acoplamento é a forma de como os módulos estão relacionados uns com os outros. Uma dependência entre módulos é quando um comunica diretamente com outro. Quando um módulo A invoca um módulo B, então A depende de B. Mas B não necessariamente depende de A. Entretanto, dependência não é sempre explícita e com isso a o conceito estado entra em ação para auxiliar na formação de um módulo. Um exemplo disso é quando dois módulos parecem não ter nenhuma dependência entre si, mas que o sistema exige que ambos tenham terminado uma tarefa no mesmo tempo, criando assim uma dependência e consequentemente o sistema é acoplado no quesito tempo. Dependências não é algo ruim. São necessárias para que o procedimento possa funcionar corretamente com todos os outros módulos, mas precisa-se saber também o grau de acoplamento no sistema sendo essa em número e tipo. Dependência surgidas a partir de interfaces formais são as melhores formas de dependências. Uma forma de reduzir acoplamento em sistemas é por meio de encapsulamentos. Tais envolvem manipulações de estado e introdução à interface formal. A ideia é mover um estado dentro de um módulo e fazer com que isso seja exclusivo dele, também chamado de informação escondida. Isso permite uma maior liberdade de mudança in-

terna no módulo já que a mudança de formatos de um estado não introduz um erro em outro e se o módulo possui uma boa abstração, então a informação escondida também permite que ele seja implementado de forma isolada. Assim, acoplamento é o resultado de dependências entre módulos. É preciso evitar acoplamentos quando desnecessário e para isso existem várias técnicas para manipulação do grau de acoplamento. É possível ver na Figura Xa que os sub-módulos B e C possuem dependência do módulo somatório A. Na Figura Xb a somatória A é duplicada dentro de cada sub-módulo eliminando cada uma das dependências existentes. Figura 3.4(a) Será proposta uma situação exibindo o porquê desta abordagem pode ter mais vantagens: suponha que A foi designado para calcular somente números não-sinalizados. Mas com o passar do tempo, C necessita de operar com número sinalizados. Dessa forma, na Figura xa o design deveria alterar tanto C quanto A. Alterando A, obrigatoriamente afetará B. Mesmo que B continue a trabalhar bem com a alteração de A, ouve uma cascata de alterações a serem feitas ao longo do tempo sobre algo que deveria ser pequeno, simples e isolado. Discutindo sobre desvantagens deste procedimento, talvez se pensa que isso pode aumentar no tamanho do design do projeto já que está duplicando um item, mas não. É possível que a funcionalidade de A seja simplesmente mesclada com o configurable logic block (CLB) já alocado para os sub-módulos B e C e consequentemente é possível que não haja nenhum ganho de conexão no CLB alocado. É claro que isso não é sempre verdade e é possível sim que tenha aumento de custo. Uma segunda desvantagem é, ao duplicar um módulo, agora se tem o mesmo componente em dois lugares. Se um erro for encontrado em um, deverá ser alterado noutro também.

Planejando para o Reuso Além de qualidade, outro princípio é o torná-lo reusável. Com o constante aumento de complexidade dos componentes, é imprescindível que construamos designs com intenção de serem reutilizáveis. Primeiramente é necessário criar e identificar designs reutilizáveis. Alta coesão e baixo acoplamento são indicativos de componentes reutilizáveis. Existem custos em escondidos por meio de relative cost of reuse (RCR) e relative cost of writing for reuse (RCWR) e eles serão dissertados a seguir. RCR são custos que fazem com que tenhamos que ler a documentação para entender como utilizar o módulo e o RCWR é o esforço extra que alguém deve exercer para projetar um módulo que outros possam utilizar (POULIN ET AL, 1993). Um exemplo é o trade-off de aprender a utilizar a função strcpy da biblioteca string.h contra o tempo de criar a nossa própria função de cópia. Em alguns casos, poderia ser fácil gerar nosso próprio a invés de aprender um componente potencialmente complexo. Isso provavelmente seria por causa do alto RCR do módulo. Uma forma de gerenciar o RCWR é tomar uma abordagem incremental onde é projetado um componente específico no design atual. Se ele

for necessário novamente, copie e generalize-o. Para vários designs, adicionar uma generalização torna o componente reusável. Em linguagem VHDL, isso pode ser conseguindo introduzindo generics no projeto. Um ponto importante de máquinas computacionais personalizadas são a vantagem de serem específicas. Simplesmente adicionando generalização sem deixar a opção de gerar versões específicas de aplicativos através de genéricos é improdutivo. Refatoração é a tarefa de procurar em um design existente e rearranjar os agrupamentos e hierarquia sem alterar sua funcionalidade, como é exibido muito bem na Figura 3.4 e é feito para fazer componentes reutilizáveis. É possível que haja situações onde a refatoração pode acidentalmente alterar a sua funcionalidade. Para isso existe o processo de teste regressivo. Ele é usado para prevenir tal situação. Geralmente é automatizado e ser uma simulação dirigida tal como os testsbenchs já comumente conhecidos ou mesmo uma série de sistemas que contornam o componente exercitando sua própria funcionalidade. São necessários vários sistemas pelo fato de quererem também testar todos os genéricos que estão no conjunto em tempo de compilação.

Grafo de Controle de Fluxo É possível ver que os detalhes abordados aqui são referenciados a um design de software. É possível representar um sistema, quanto em software ou em hardware, de várias formas. Uma delas é o desenvolvimento de rápidos protótipos e este é referenciado como um design de referencia de software. Por mais que o custo de sua criação é algo ruim, sua forma de especificação sistêmica traz várias vantagens. A mais notável é a generalização de uma especificação bastante completa, sendo qualquer dúvida sobre o comportamento do sistema, é possível olhar no design referencial. E sendo é um projeto que pode ser lido pelo computador, a sua especificação pode ser analisada por ferramentas computacionais. Assumindo que o design de referência de software já exista, será mostrado como este pode ser demonstrado matematicamente e auxiliar-nos em o que deve ser implementado em nível de hardware ou software. Para isso, utilizar-se-á o grafo de controle de fluxo (CFG). Ele é definido por $G = (V, E)$ onde V são vértices que representam os blocos básicos e as arestas indicam a todas as possibilidades de caminhos. Um bloco básico é uma sequência maximal de instruções sequenciais com single entry and single exit (SESE). Figura 3.5 O primeiro grupo A é um bloco não básico porque não é maximal, ou seja, a primeira instrução store word with update deveria estar incluída. Grupo B é um bloco básico. Já o grupo C não é pois tem duas entradas para o bloco sendo elas no store word e também pelo branching para L2. Figura 3.6 A Figura 3.6(a) exibe os blocos básicos em um código em alto nível. A Figura 2.6(b) exibe os o código gerado por um compilador para PowerPC onde os blocos básicos também são identificados e por fim o grafo de controle de fluxo. Sabe-se que, é só possível identificar o bloco básico em C desde que se sabe como compilador foi

utilizado para gerar código, o que não acontece com assembly já que é possível identificar os blocos já com as próprias instruções.

Design de Hardware Até agora foi discutido os tópicos de modo genérico. A partir de agora, a discussão terá foco em design em hardware, em especial design à Plataforma FPGA. Designers raramente queriam construir em sistemas embarcados a partir de simples rabiscos e para isso utilizavam arquiteturas já existentes removendo componentes desnecessários e então adicionando cores aos requisitos dos projetos. O modelo processador-memória, exibido na Figura 3.8, teve grande sucesso no início. Isso se tornou possível por causa da popularização do IBM Personal Computer em 80. Isso impulsionou desenvolvedores terceiros a criarem periféricos e máquinas compatíveis com outros fabricantes. E como a demanda aumentava com o passar do tempo, foi possível que os fabricantes pudessem tentar diferentes designs de arquiteturas de computadores. Figura 3.8 Computação embarcada não tinha grandes mudanças desde então, pois como é possível ver na Figura 3.8, esta arquitetura permitia o aprimoramento de componentes individuais do sistema. Por mais que o sistema de barramento seja insuficiente, ele serve bem para aplicações em geral e esse sistema é um bom ponto inicial para um design em Plataforma FPGA. A Plataforma FPGA tem adota este tipo de arquitetura pois provê um framework estável que pode ser construído sobre designs customizados. É possível construir complexos sistemas utilizando componentes existentes e cores, e com tempo reduzido em relação a sistemas embarcados tradicionais.

Componentes de uma Plataforma FPGA Com a ideia de modularidade, coesão, casamento de componentes e design, queremos iniciar uma construção de um sistema base que pode ser reutilizado como ponto de partida para um projeto de sistema embarcado, sendo este o modelo processador-memória. O processador oferece controle e um ambiente de design familiar ao designer. Mesmo que use pouca ou nenhuma relação com processadores, é usado para uma rápida prototipação. Pode existir dois tipos de processadores, os processadores hardware e software core, sendo o processador hard já é bem conhecido. Algumas Plataformas FPGA provê recursos reconfiguráveis suficientes para que um processador soft possa ser implementado em blocos lógicos. Eles fornecem flexibilidade e são naturalmente configuráveis. Inicialmente, mesmo processadores básicos, precisam de alguns recursos básicos e tal forma se chama stand-alone. Antes da implementação de fato, é preciso verificar se o FPGA já possui um processador hardware, se possui recursos suficientes para a implementação de um, qual o papel do processador e o software será utilizado nele e isso será discutido ao decorrer deste. A memória pode ser organizada e hierarquizada de várias formas diferentes como Von Neumann ou Harvard,

ou mesmo com níveis de caches diferentes. Tal como o processador, deve-se perguntar que tipos de memórias estão disponíveis, quanto há disponível e outros parâmetros de implementação. Em um sistema, ela pode ser considerada como um componente, um core, ou uma parte básica e sua localização determina a interface e acessibilidade. Um controlador de memória é requerido para controlar as transações de memórias. O uso eficiente é o item mais crítico pois ela pode gerar lags com o processador.

Barramento As interfaces de cada componente, como o processador e memória, se conectam via barramento padronizado. Um core que pode requisitar acesso é considerado barramento master, o que demonstra que nem as cores necessitam de ser master, o que tornam barramento slave. Um barramento no FPGA é implementado em lógica configurável, o que torna um soft core. Detalhes importantes são quais cores necessitam de comunicar diretamente, alguns comunicam mais frequentemente ou necessitam de uma alta vazão. Geralmente, o barramento com maior vazão é o que fornece a conexão entre processador e controlador de memória e geralmente é o primeiro barramento a ser adicionado ao projeto. Um segundo pode ser adicionado para separar o design em diferentes domínios, geralmente em low e high-speed ou com largura de banda dedicada à comunicação. Esses são conhecidos como barramento de periféricos. Utilizando barramentos simples, para realizar uma operação na memória, deve-se realizar uma requisição e o múltiplos barramentos permitem a comunicação paralela. Quando necessita de uma comunicação de um core com um periférico, utiliza-se de um bridge. É um core especial que reside em ambos os barramentos e propaga requisições de um barramento para outro. Trata-se de uma interface que funciona como um barramento mestre em um barramento e um barramento slave em outro e assim o slave responde à requisição que deve ser passada a outros os periféricos. Geralmente somente uma bridge simples é requerida, onde todos os periféricos são conectados nela, tendo um sistema de bridge para gerenciamento das solicitações e respostas.

Periféricos Quando se menciona periféricos, geralmente é referido à hardwares cores como impressoras, LCDs, GPSs e outros. Periféricos pode-se de dizer que são todos os componentes que estão em torno à unidade de processamento central. Todos eles possuem algum tipo de interface de comunicação, sendo ela uma PCI Bridge, Ethernet, USB, UART, I^2C, SPI e muitos outros.

O Sistema Base Para início, será montado um sistema conceitual simples constituído de um processador, dois tipos de memória e uma comunicação UART. Como resultado, este conceito será base para vários outros tipos de designs. Figura 3.9 O benefício de utilizar dois sistemas de barramento é a facilidade de modificação ao adicionar e

substituir componentes podendo assim termos o seguinte design de sistema exibido na Figura .3.10, de acordo com nosso propósito. Figura 3.10 Em uma perspectiva de hardware, a localização de dados é facilmente identificável. Memória off-chip é um modo separado e seu acesso é nada mais que seus endereços. O endereçamento é importante para qualquer core possa comunicar com o processador. A Figura 3.11 mostra o mapa de endereçamento dos dois barramentos. Cara core possui um intervalo na memória é considerado um slave e o processador não possui um espaço de memória no mapa. Em sistemas com dois barramentos, o bridge atua como um intermediário entre as requisições e o sistema de barramento periférico.

Montando Sistemas Personalizados Para iniciar, deve-se responder à pergunta de porque construir um sistema personalizado. Acreditava-se que desenvolvimento de hardware é um processo difícil porque existem mão de obra para softwares profissionais do que engenheiros de hardware. E a resposta para a pergunta 'porque desenvolver um hardware' é para obter performance além de eficiência e previsibilidade. Sabendo-se que um FPGA não terá o mesmo ganho que um processador projetado já em silício, deve-se analisar como o hardware FPGA supera um processador. Existem duas razões porque alguns designs de FPGA possuem vantagens em performance. A primeira é sobre o modelo de execução. O modelo de computação sequencial von Neumann possui uma barreira em sua performance por causa do fato de expressar suas tarefas como um conjunto sequencial de operandos. Em hardware, o paralelismo inerente da tarefa pode ser expressado diretamente. Para compensar sua operação serial inerente, processadores modernos possuem uma porção significante de recursos de hardware para extrair em nível de instruções o paralelismo como estratégia de aumentar a vazão. Para algumas aplicações, largura de banda de memória limita a performance e parte da banda é consumida pelas instruções sendo buscadas, da memória e assim, instruções fazem parte implícita do design. A segunda razão é que FPGA pode ter uma especialização. Em processadores de propósito data-path e tamanho de operações são organizados por meio de requerimentos gerais. Isso significa que, para multiplicar um inteiro por uma constante c é necessário de um multiplicador completo no sistema. Se essa informação já é conhecida, uma implementação baseada em FPGA pode ser criada com funções personalizadas. Supondo que uma implementação em hardware de uma tarefa A leva o mesmo tempo a ser executada no processador e ambas são de fácil implementação e questiona-se se ainda é viável a implementação desta. A resposta é sim quando a solução em hardware é mais eficiente. Eficiência é definida como facilidade de realizar uma tarefa fixa com uma quantidade variável de recurso sendo recurso a área de prototipação, número discreto de chips, ou o custo da solução. Um hardware personalizado e um pro-

cessador possuem são mais eficientes que dois processadores. É possível também o caso de um processador não ser totalmente utilizado de sua capacidade. Mesmo utilizando a propriedade de multitarefa em novas funcionalidades, existem razões para escolher a implementação. Alguns casos fazem sentido mover uma tarefa para o hardware se a torna mais previsível ou se torna o escalonamento no processador mais facilitado. Em casos onde restrição de tempo é importante, como sistemas de tempo real, a meta é satisfazer as restrições e assim previsibilidade é mais importante que performance. Entretanto, a única desvantagem de desenvolver em hardware, como já mencionado é o esforço requerido. O número de engenheiros de hardware é relativamente baixo comparado com profissionais que desenvolvem em software. Desenvolvimento em hardware não mais difícil que em software, mas sim, necessita de atenção no projeto. Em resumo, uma Plataforma FPGA oferece vantagens de performance, eficiência e previsibilidade sobre soluções somente em software. Como um projetista de FPGA, parte de sua tarefa inclui determinar quando um simples controlador é apropriado.

Composição de design Existem três passos para fazer um design de um core modular customizado. O primeiro passo é identificar as entradas e saídas, onde em muitos casos estas são baseadas em suas funcionalidades. O segundo é identificar os operandos e compor o data-path, geralmente uma coleção de computações de multi-estágios (isto é, pipeline). Cada componente é designado a uma funcionalidade particular. Os operandos exatos podem não ser claros no início da fase de design, mas determinando a funcionalidade em baixo nível necessárias permite a construção de um data-path. Um data-path representa o fluxo de dados por meio do componente. Uma vez definido, é possível construir o pipeline, no qual contribui com a performance e eficiência do design. Capturando os estágios do pipeline pode ser árduo inicialmente, mas iniciando um design com o conceito de suporta às operações pipeline faz com que o projeto seja mais gerenciável. O terceiro paço é o desenvolver o circuito de controle que sequencia as operações, frequentemente a máquina de estados finitos. Frequentemente referenciamos um hardware em termos de paralelismo, onde cada operação é independente e podem ser executadas ao mesmo tempo. Com a máquina de estados finitos é possível gerenciar a computação executando as operações paralelas e em seguida as sequencias e dependentes. Existem duas abordagens de design, bottom-up e top-down. Em muitos casos, utiliza-se a abordagem bottom-up no desenvolvimento em FPGA. Em método estrutural HDL, cada componente é construído por seus subcomponentes e assim, antes do top-level, todos os subcomponentes devem ser construídos e testados. Nessa abordagem, cada subcomponente pode ser tratado como uma black-box onde as entradas e saídas são conhecidas. Na abordagem top-down, quando se quer desenvolver um core

personalizado, o designer deve começar com as interfaces de entrada e saída, criando assim o início da black-box. Uma vez definida, é possível decompor sistematicamente em subcomponentes. Esse processo repete até os blocos de baixo nível onde podem ser simples o suficiente. Esta abordagem não é associada a nenhum HDL estrutural ou comportamental, mas é possível utilizada com o último. Em ambas as abordagens, internamente podem ser construídas de formas distintas, mas no final todas possuem a mesma funcionalidade. Considere o seguinte componente somador de quatro números da Figura 3.12. Na implementação temporal da Figura 3.13 existe um multiplexador que realizará a soma de acordo com o controlador da máquina de estados finitos. A máquina de estados finitos terá quatro estados ao todo e percorrerá de forma sequencial. Utilizando somente uma ALU e um registrador, temos o circuito que utiliza menor quantidade de recursos possível, mas não temos a solução com menor tempo. Para aumentar o speed-up, devemos considerar a abordagem paralela e adicionar mais recursos. Um sistema com três ALUs poderá realizar as operações `temp1 = a + b`, `temp2 = c + d` e `temp1 + temp2`. O trade-off, da latência e recursos fica a cargo do designer. Figura 3.12 Figura 3.13 Quando existem operações sequencias, implica que existe um controle onde a segunda toma lugar em seguida da primeira após sua computação e isso degrada no caso de composição espacial o que diminui a estrita ordenação de operações. Dessa forma, quando um hardware especifica duas operações, elas serão executadas simultaneamente ao menos que o designer tenha especificado previamente uma ordem. A Figura 3.14 mostra a implementação espacial do somador. Nesse caso, as operações de adição são pipeline tais que os resultados são alimentados para frente para o próximo somador. E execução solta em FPGA pode gerar vantagens e desvantagens. A concorrência é o que fornece ao sistema performance, e controle do tempo é o que fornece previsibilidade. Expressar simplesmente relações de tempo entre operações é um desafio. 3.14

Design de Software No passado, software era simples. Tinham uma simples tarefa, e executava um papel relativamente menor comparado com o design de software da época. Se duas tarefas eram solicitadas, eram frequentemente deixadas independentes, lógico e fisicamente. Com microcontroladores tornando mais veloz, sistemas embarcados adicionaram sistemas de software para gerenciar controle de múltiplas tarefas. Isso permite quem um microcontrolador simples possa fazer a multiplexação de tarefas separadas por tempo. Em sistemas embutidos atuais, seus processadores possuem unidade de gerenciamento de memória, suporte de memória virtual e com tecnologia suficiente para suportar sistemas operacionais. Tais características são tidas como vantagens pois resulta em uma explosão de novas características e assim permitindo a incorporação e adaptação de aplicações de softwares grandes que originalmente eram escritos para

computadores de propósito geral e máquinas servidoras.

Opções de Sistema de Software Um desenvolvimento de um sistema embarcado possui vasto escolhas quando vem para sistema de software. Sistema de Software referimos a qualquer software que assista à aplicação, geralmente adicionando uma interface em software para acesso ao hardware. Isso segue desde simples bibliotecas de rotinas até sistemas operacionais totalmente desenvolvidos que virtualizam o hardware para processos individuais. Em muitos os casos, não é necessário um sistema de software. Nesses casos, os arquivos iniciais executados antes do `main` em C, por exemplo, são modificados. Sem um sistema operacional, essas rotinas são responsáveis por definir o estado inicial do processador e periféricos. Mesmo que o processador tenha uma unidade de controle de memória, simples casos que a aplicação executa em modo real e privilegiado, nenhuma proteção de memória é utilizada. Isso se chama programa standalone C, pois executa sem nenhum sistema de software adicional e com isso, tem uma abordagem de sistema simples. Para FPGA, isso é frequentemente o primeiro passo quando se testa um novo hardware core, pois, um programa em C tem total acesso ao hardware. Geralmente, essa solução produz um executável suficientemente pequeno que o sistema de software inteiro pode ajustar dentro de um bloco de memória RAM no FPGA. A desvantagem é a necessidade de desenvolvimento. Não á proteção contra erros de software. Talvez o grande retrocesso hoje é que é difícil ter vantagem de software existente que assume que admite uma biblioteca C completa e um workstation. As vezes a adição de funcionalidades de um sistema de software como o suporte a multithreads pode ser útil, diferente da indesejável sobrecarga ao adicionar recursos completos de um sistema operacional. Vários produtos e soluções Open-Souce são feitos para sistemas embutidos. Um passo acima de standalone é uma simples biblioteca de threads. Sistemas operacionais provem um número de serviços para uma aplicação, mas isso tem um custo. Os sistemas operacionais de sistemas embarcados são diferentes dos utilizados nos desktops, significando que o desenvolvedor tem que aprender novas interfaces, convenções e o que está ou não disponível. Analisando de forma espectral, em um lado tem-se um sistema operacional com recursos completos, utilizados por workstations. A maioria dos sistemas de software descritos podem ser executados sem um subsistema de armazenamento secundário, ou seja, um sistema de arquivo. Entretanto, sistemas completos necessitam, no mínimo, sistema de arquivo raiz. Até recentemente, era inviável pensar em um sistema operacional completo em um sistema embarcado por causa de recursos requeridos. Entretanto, com novos dispositivos como a Plataforma FPGA isso se torna mais comum. Quanto mais serviços é colocado mais peso é adicionado ao desenvolvedor em saber o que é provido num sistema de software e saber usá-lo. Com sistemas completos, os programadores

já estão familiarizados com estes. Como sistemas operacionais são comuns, existem várias aplicações disponíveis. Com os sistemas embarcados se tornando mais obliquo e conectado à internet, eles necessitam suportar mais interfaces e mais protocolos de comunicação, o que o sistema operacional completo já fornece.

Monitores e Bootloaders Nos primeiros microprocessadores baseado em sistemas embarcados, simples 8-bit migraram de computadores de hobbies e games para outros produtos comercias que agora chamamos de sistemas embarcados. Fabricantes desses microprocessadores geralmente desenvolvem kits que incluem board fabricadas que aumentam as capacidades dos chips e uma Board Support Package (BSP) que inclui compiladores, software power-on-self-test (POST), bibliotecas de software de Basic Input/Output System (BIOS) e um debugger. POST é executado antes de qualquer outro software para verificações de que nada foi esgotado desde a última vez que foi ligado. Usando sub-rotinas da BIOS, o tamanho da aplicação continua pequeno. Monitor é um simples software de tipo primitivo ao debug. Debuggers modernos executam em um processo separado, tem acesso à tabela de símbolos do compilador e fornece uma rica e flexível interface. Ao contrário, um monitor é um driver de interrupção e suporta algumas funcionalidades básicas. Monitores tinham uma funcionalidade que não existem nos debuggers atuais, no qual suporta a transferência de memória sobre canal de comunicação serial transmitindo ASCII usado para interação. Como os caracteres ASCII possuem 7 bits e executáveis utilizam tudo como 8-bits de um byte, blocos de memória foram codificados para a transmissão. Enquanto é desenvolvido a aplicação, o designer poderia iniciar o monitor e copiar a aplicação para a RAM, ajudando o curto teste/debug dos softwares. Isso é importante pois, até o GNU debugger (ou gdb) possui subsistemas capazes de comportar como monitores. Sistemas modernos moveram um passo a diante. Uma moderna substituição de um monitor pode ser a interface joint test action group (JTAG). Tal controlador gerencia qualquer endereço físico, incluindo a memória principal, provendo uma alternativa a abordagem dita dos monitores. Nesse caso, o debugger comunica com uma interface até o controlador JTAG. Da mesma forma as funcionalidades de POST/BIOS têm transformado em software BIOS de desktops no qual inicia logo em seguida da energização do computador. Para alguns computadores, é critico que o BIOS coloque o computador e seus periféricos em um estado conhecido. Linux por exemplo não assume nada e permite que cada hardware inicialize sozinho. Parcialmente concorrente com o desenvolvimento de PC, workstations surgem com uma abordagem levemente diferente. Elas utilizam um pequeno software chamado bootloader ou também chamado de PROM. Era simplesmente um programa que lia o primeiro setor de um disco, no qual continham outros programas mais avançados, e carregava-os

para a memória principal, apontando o PC para o primeiro endereço desse setor. Esse então processava o carregamento do sistema operacional por completo. Essa técnica vem surgindo em PCs com boa aceitação. BIOS continua sendo o primeiro a ser executado, então o bootstrap é iniciado carregando o sistema operacional. Alguns conhecidos hoje são GRUB, U-Boot e o RedBoot. Para sistemas embarcados, a abordagem BIOS/monitor ainda domina os pequenos microcontroladores e o legado sistêmico enquanto o bootloader está ganhando espaço em sistemas operacionais completos.

Particionamento A decomposição de um design de referencial de software pode gerar dois componentes: uma porção a ser realizada em hardware e outra executada em software, num processador e isso é chamado de problema de particionamento. Para sistemas baseados em Plataformas FPGA, particionamento é um subproblema de um problema mais geral localizado no codesign de hardware e software. Para início, será considerado uma aplicação como um conjunto de instruções organizadas como uma coleção de grafos de fluxo de controle especificando a ordem de execução. Sendo assim, a ideia do particionamento é o grupo de específicos conjuntos de instruções em uma aplicação e então mapear esses grupos tanto em hardware e software. Os grupos designados ao software são executados sequencialmente enquanto os mapeados em hardware são implementados por uma combinação customizada ou por circuitos sequenciais. Se uma aplicação é totalmente funcional em design referencial de software, o resultado do particionamento é conhecido como decomposição. Alguns fatores podem ajudar nas decisões de particionamento tal como expectativa de ganho de performance, os recursos utilizados em hardware, como são usados e talvez o mais importante quanto de sobrecarga de comunicação a decomposição impõe ou dificuldade de implementar um conjunto específico em hardware. Recurso por definição são instruções de um cluster conectado de aplicações de design referencial de software adequado para uma implementação de hardware. Adequado será utilizado para definir 'o projetista do sistema antecipa que uma implementação de hardware se mostrará vantajosa'. Para obter uma boa partição, geralmente tem que examinar grupos que podem ser maiores ou menores que sub-rotinas definidas pelo programador. Recurso pode variar de um pequeno conjunto de instruções para um kernel de loops aninhados até um modulo de software completo consistente de múltiplas sub-rotinas. Como o tamanho dos recursos afetam na performance, a decisão de implementação em hardware depende da sua melhoria no sistema por inteiro e os recursos utilizados relativos a outros recursos candidatos. Se determinado a ser valido a pena, então os recursos de implementação em hardware aumentam a arquitetura de hardware. Profile, também conhecido como recorte, é uma técnica para coletar informações em tempo de execução de uma aplicação. O Software referencial é executado com uma

entrada representativa e o tempo gasto em várias partes da aplicação é mensurado. Um exemplo é exibido na Figura 4.1 onde exibe a codificação de uma imagem. Figura 4.1. Aqui será assumido a técnica simplista que recortará uma aplicação por interrupção periódica e amostrar o *program counter*. Um histograma pode ser utilizado para contar quando um programa é interrompido em um endereço particular e a partir desse, uma fração aproximada do tempo total de execução gasto em várias partes da aplicação pode ser computado. A análise de performance é definida pela Lei de Amdahl que aplica a lei de retornos decrescentes à utilidade de uma única arquitetura, ou seja,

$$\text{Speedup}_{overall} = \frac{1}{(1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}}}$$

articulando os limites de um melhoramento geral para uma aplicação que um simples realce pode fazer em tempo de execução. Generalizando para incluir todos os potenciais realces e métricas de performance arbitrária, pode-se caracterizar os limites como coleções de realce. Se forem em recursos de hardware e pode-se antecipar que requer ganho potencial de recursos para cara tal, então pode-se desenvolver um modelo que irá orientar ao processo de particionamento. Tal lei não é adequada pois: • Visa um simples realce e não nos auxilia a selecionar um subconjunto de uma coleção de recursos potenciais; • Foca somente no tempo de execução; • Não visa recursos requeridos; • Não visa custo de comunicações. Perfis baseados em exemplos requerem representações de entrada e é uma aproximação de tempo gasto. Sem implementar todo o potencial de recursos em hardware, pode ser difícil antecipar os recursos requeridos. Pode-se com determinação analítica calcular quantos ciclos de clock um recurso de hardware irá tomar, podendo assim calcular métricas como speedup. Entretanto, é impossível saber quando um processo arbitrário será completado. Ao invés de prover uma solução automatizada, a solução analítica simplesmente provê um framework para guiar um designer de sistema a criar uma solução criativa.

Solução analítica para particionamento Para definir o problema, será descrito fórmulas matemáticas do problema de agrupamento de instruções em recursos e então os seus mapeamentos em hardware ou software e ao final, converter os recursos em implementações em FPGA. Atualmente, a forma mais comum de transcrever é descrever manualmente o core com um HDL utilizando design referencial de software como especificação. Muitos problemas práticos impactam na performance atual do sistema. Nem todos os problemas podem ser incorporados a um modelo analítico e por isso podemos caminhar no sentido de soluções matemáticas que irão produzir uma resposta aproximada ao problema de

particionamento. Muitas das entradas do nosso modelo são estimadas ou aproximações no qual futuramente degrada a fidelidade de resultados. Deve-se importar com isso pois resolvendo o problema de particionamento 'no papel', tem-se um particionamento que é próximo ao ótimo. Daqui, cabe ao designer ser habilidoso em usar os guias e projetar uma solução mais refinada. Ao final, é mais eficiente usar uma combinação de técnicas ad hoc e matemáticas para encontrar uma solução ótima do que simplesmente confiar numa intuição de engenheiro. Para continuar, deve-se definir alguns conceitos básicos. Sabe-se já de antemão o modelamento de uma sub-rotina de um design referencial de software utilizando o grafo de controle de fluxo (CFG) e além desse, será descrito uma nova notação para incluir o grafo de chamada (CG), no qual consiste num conjunto de CFGs, um por sub-rotinas

$$\mathcal{C} = C_0, C_1, \ldots C_{n-1}$$

onde $C_i = (B_i, F_i)$ é o CFG de uma sub-rotina i. Sendo assim, a CG da aplicação é escrita por

$$\mathcal{A} \subseteq (\mathcal{C}, \mathcal{L})$$

onde $\mathcal{L} \in \mathcal{C} \times \mathcal{C}$. Duas sub-rotinas são relacionadas $(C_i, C_j) pertence \mathcal{L}$ if podem ser determinadas no tempo de compilação que a sub-rotina i tem potencial de invocar a sub-rotina j. É assumido que os blocos básicos de cada sub-rotina são disjuntos, ou seja, cada bloco básico em uma aplicação pertence a exatamente um CFG. Além do mais, é assumido que um nó raiz para o CG é implícito, ou seja, uma sub-rotina é designada a iniciar a execução. Nem todos os executáveis podem ser expressados nesse modelo. O manuseio de sinais e interrupções não são representadas e assim, não é possível determinar todos vértices, F_i em uma dada sub-rotina C_i de um CFG antes da execução. Finalmente, o paradigma de orientação à objeto depende do tempo de execução para conectar os métodos virtuais invocados. Por design, esse paradigma nos previne de saber todos os vértices antes da execução. Para agora, será considerado que o modelo é suficiente para ser expressado em design referencial de software. Um equívoco comum é que uma definição formal de particionamento só aplica a separação de aplicação componentes de hardware e software. Além do mais, para fazer o problema mais tratável é comum agrupar operandos de primeiro recurso, ou seja, uma partição com um grande número de subconjuntos, e então mapeia esses recursos em ambos hardware e software. Assumindo que esses recursos são razoavelmente bem clustered, então a decomposição de uma aplicação em componentes de hardware e software pode ser dirigido por comparações de ganho de performance de um recurso contra outro. Primeiramente, definiremos uma partição formalmente. Uma partição $\mathcal{S} = S_0, S_1, \ldots$ de um conjunto universal U é um

conjunto de subconjuntos de U sendo que

$$\bigcup_{S \in \mathcal{S}} S = U$$

$$\forall S, S' \in \mathcal{S} | \mathcal{S} \cap \mathcal{S}' = \emptyset$$

e

$$\forall S \in \mathcal{S} \cdot S \neq \emptyset$$

A equação 4.1 diz que cada elemento de U é um membro de pelo menos um subconjunto $S \in \mathcal{S}$. A equação 4.2 e 4.3 diz que os subconjuntos $S \in \mathcal{S}$ são emparelhados disjuntos e não vazio. Em outras palavras, cada elemento do nosso universo U termina exatamente em um dos subconjuntos de $\mathcal{S}$ e nenhum dos subconjuntos são vazios. Por exemplo, considere as vogais da língua inglesa onde $U = a, e, i, o, u, y$. Uma partição $\mathcal{X}_a$ de U é

$$\mathcal{X}_a = a, e, i, o, u, y$$

e outro exemplo é

$$\mathcal{X}_b = a, e, i, o, u, y$$

A figura 4.2 ilustra o $\mathcal{X}_a$ graficalmente. Figura 4.2 Pode-se também aplicar em uma aplicaç]ao $\mathcal{A}$. Se assumirmos que nosso universo é o conjunto de todos os blocos básicos de cada sub-rotina,

$$U = \cup_{C \in \mathcal{C}} V(C)$$

então U é as partições de sub-rotinas e chamaremos de partição natural de aplicação onde

$$\mathcal{S} = \left\{ \underbrace{\{b_0, b_1, \ldots, b_i\}}_{\text{sub-rotine } C_0}, \underbrace{\{b_i, b_{i+1}, \ldots, \}}_{\text{sub-rotine } C_1}, \ldots \underbrace{\{b_j, b_{j+1}, \ldots, \}}_{\text{sub-rotine } C_{n-1}} \right\}$$

Nossa tarefa será reorganizar a partição de blocos básicos e então mapear cada subconjunto de ambos os hardwares e software. Dessa forma, estamos livres para criar e remover subconjuntos não vazios, e mover blocos básicos ao redor até termos uma nova partição e assim termos um novo resultado $\mathcal{A}' = (\mathcal{C}', \mathcal{L}')$, inferido a partir da reorganização da partição $\mathcal{X}'$. O segundo passo é mapear cada subconjunto de $\mathcal{X}$ para ambos hardware e software como é exibido abaixo

$$\mathcal{X}' = \left\{ \underbrace{\{b_j, b_{j+1}, \ldots, \}\, \{b_k, b_{k+1}, \ldots, \} \ldots}_{\text{software}} \underbrace{\{b_0, b_1, \ldots, b_i\}\, \{b_i, b_{i+1}, \ldots, \} \ldots}_{\text{hardware}} \right\}$$

Ganho de performance esperado Para explicar como performance pode ser utilizado para guiar o particionamento, será utilizado uma métrica simples chamada taxa de execução. É parcialmente motivada pelo fato de que o ganho de performance é relativamente fácil de ser mensurado e por causa de que de todas as métricas comumente utilizadas, speedup é frequentemente a mais importante. Diferente do mundo software onde tense análise de ordem de complexidade, hardware não possui um guia geral para comparação. Ganho de performance para aplicações pode residir em uma acumulação de pequenos ganhos que deveria ser perdido numa aplicação direta na teoria de complexidade. Sendo assim, será usado a informação de profiling para coletar o tempo total de execução bem com uma fração do tempo gasto em casa sub-rotina. O produto é a aproximação entre o tempo necessário para executar uma porção de aplicação em software e usar isso como o tempo que se espera que tomará em futuras execuções. Será utilizado $s(i)$ para representar o tempo de execução esperado para uma invocação de uma sub-rotina i, ou seja, bloco básico. É uma aproximação para um número de razões, não é o mínimo que depende dos conjuntos de dados de entrada para muitas aplicações. Mesmo assim, existe também exemplos de erros que podem impactar a performance. Seguindo, precisa-se aproximar o tempo que é uma implementação equivalente em hardware que iria tomar. No caso dos blocos básicos, isso é frequentemente mais preciso. Sem fluxo de controle, não presente em blocos básicos, uma ferramenta de síntese pode dar bastante aproximação de acurácia de propagação de tempo. Ou se o recurso é pipelined, o número de estágios é mais precisamente conhecido. Se o recurso inclui fluxo de controle, mas não contém nenhum loop, o caminho mais longo pode ser usado como uma estimativa conservativa. Recursos com um número de variáveis de iterações através de um loop apresentam o maior obstáculo para encontrar um tempo de hardware aproximado. Nesse caso, implementação e profiling com recurso em hardware pode ser a única solução. Independente, assumimos que uma aproximação apropriada $h(i)$ para o existente tempo de execução em hardware. Finalmente, a interface entre software e hardware requer tempo e este custo precisa ser contabilizado também. Pode-se aproximar este custo pela aproximação do montante total do estado que necessita ser transferido ou o custo de configuração e latência. Em ambos os caso, são representados por $m(i)$ para recursos $i \in \mathcal{H}$. Taxa de execução é a velocidade na qual um sistema computacional completa

uma aplicação, e em um sistema de plataforma FPGA olhamos para o hardware para melhorar sua taxa de execução. Esse ganho, no qual comparado com uma solução hardware e software contra uma solução puramente software, é tipicamente mensurada como speedup. Utilizamos $\mathcal{Y}$ para sua representação e isso nos permitira comprar recursos diferentes contra outros para determinar melhores particionamentos. Dessa forma, qualquer subconjunto de blocos básicos que não produzem um ganho de performance, podem ser excluídos geralmente de consideração. Em outras palavras, somente subconjuntos de blocos básicos para qual $\mathcal{Y} > 1.0$ são considerados recursos candidatos. Em geral, não mensuramos taxa de execução, mas ao invés disso, o tempo de execução, que no caso é inverso. Então quando considerando se um conjunto particular de blocos básicos deveriam ser mapeados ao hardware ou software, estamos interessados em seu ganho em speedup ou

$$\mathcal{Y} = \frac{\text{hardware speed}}{\text{software speed}} = \frac{\frac{1}{\text{hardware time}}}{\frac{1}{\text{software time}}} = \frac{\text{software time}}{\text{hardware time}}$$

Mais especificamente, estamos interessados no ganho de performance individual de cada recurso e assim, definindo $\mathcal{Y}(i), i \in \mathcal{C}$

$$\mathcal{Y}(i) = \frac{s(i)}{h(i) + m(i)}$$

onde $h(i)$ e $s(i)$ são o tempo de execução de uma implementação de um recurso i em hardware w software. A função $m(i)$ é o tempo que se leva para sincronização, estado preso, ou seja, o tempo que leva para guiar um dado entre o processador e o item reconfigurável. Assumindo por um momento que usaremos esse recurso separado em nosso design, deve-se questionar sobre o quão rápido é a aplicação. Isso é dependente em ambos o ganho de performance do recurso e o qual frequentemente é utilizado no design referencial de software. Pode-se ter uma fração do tempo gerado em um recurso particular $f(i)$ a partir de informações de profile. Então, o speedup da aplicação no geral será

$$\Gamma = \left[(1 - f(i)) + \frac{f(i)}{\mathcal{Y}(i)} \right]^{-1}$$

A inversão representa que estamos movendo entre tempo de execução e taxa de execução para manter o sentido de ganho de performance. A partir dessa equação, podemos observar que aumentando a velocidade do hardware de um único recurso tem-se menos e menos impacto na performance da aplicação a medida que sua frequência decresce. Para aumentar a performance sistêmica de uma aplicação no geral, também queremos olhar sobre uma outra dimensão: aumentando o sistema com múltiplos recursos que

aumentará a performance de componentes individualmente assim como aumentando a fração agregada de tempo gasto em hardware. Reconhecendo isso, queremos computar o speedup de múltiplos recursos em hardware, ou seja, quer-se avalizar o ganho sistêmico de um conjunto de recursos $\mathbb{D}$ onde cada membro do conjunto contribui à performance do sistema baseado na fração do tempo gasto em cada característica. Para estimar a performance desta partição, podemos adicionar recursos e rearranjar os termos para ter um ganho de performance esperado no geral

$$\Gamma(\mathbb{D} = \left[1 + \sum_{i \in \mathbb{D}} (\frac{f(i)}{\mathcal{Y}(i)} - f(i))\right]^{-1}$$

Considerações de Recursos Seguindo a equação exibida, se tentaria adicionar recursos na abordagem $\sum_i f_i$. Em outras palavras, seria implementar tudo em hardware para maximizar a performance, ignorando todos os custos de desenvolvimento e recursos limitados. Num FPGA, há um número finito de recursos disponíveis para implementação de circuitos em hardware. Tais recursos são limitados e a maioria das aplicações realísticas irão exceder esse limite disponível. Um meio de aproximação de recursos é contar o número de células logicas requeridas para cada recurso. Um chip que terá um valor escalar r_{FPGA} que representa o total de números de células lógicas disponíveis. Então $r(i)$ pode ser usado para representar a quantidade de células lógicas requeridas por cada recurso i. Uma simples relação

$$\sum_{i \in \mathbb{D}} r(i) < r_{FPGA}$$

restringe quão largo $\mathbb{D}$ pode crescer. Sabendo que dispositivos modernos são heterogêneos, uma típica plataforma FPGA tem múltiplos tipos de recursos além de células lógicas como memória, blocos DSP, etc. Sendo assim, um vetor seria uma melhor representação

$$\vec{r}_{FPGA} = \begin{pmatrix} r_0 \\ r_1 \\ \dots \\ r_{n-1} \end{pmatrix}$$

Também promovemos a função de requerimentos de recursos para um vetor onde $\vec{r}(i)$ representa o recurso requerido pelo recurso i. Assim, a nova equação de restrição de

recurso é similar

$$\sum_{i \in \mathbb{D}} \vec{r}(i) < \vec{r}_{FPGA}$$

onde $\mathbb{D}$ é o conjunto de recurso incluídos no design. Infelizmente, esse modelo não toma conta o fato de que alguns recursos alocados podem interferir em outros além de que a estimativa de performance é frequentemente baseada na suposição que recursos são próximos um do outro e recursos de rotas não são parte integral do modelo.

Abordagem Analítica Já descrito as ferramentas matemáticas necessárias para descrever o problema fundamental do particionamento, pode-se então, primeiramente descrever formalmente o problema em termos de variáveis e descrever um algoritmo para encontrar uma solução aproximada.

Declaração do problema A ideia básica é encontrar um particionamento para todos os blocos básicos de uma aplicação e então separá-los em hardware e software. Formalmente, procura-se por uma partição $\mathcal{P}$ de todos os blocos básicos U de uma aplicação

$$U = \cup_{C \in \mathcal{C}} V(C)$$

Um subconjunto, $\mathbb{C} \subseteq U$ onde C é um vértice de um grafo $A = (C, L)$ de um design referencial de software, é chamado conjunto de candidatos e contem todos os recursos arquiteturais potenciais, ou seja, o subconjunto de U que é esperado para melhorar a performance do sistema se implementado em um hardware reconfigurável. Devido ao limite de recursos, deve-se refinar para o subconjunto $\mathbb{D} \subseteq \mathbb{C}$ que maximiza nosso métrica de performance

$$\Gamma(\mathbb{D}) \text{é maximizado, e } \sum_{i \in} \vec{r}(i) < \vec{r}_{FPGA}$$

Algoritmicamente, uma abordagem desse problema seria encontrar todas as partições de U, sintetizando e profiling cada partição, e então, quantitativamente avalizar cada Γ, mas uma aplicação simples que utilizaria cerca de 10 mil blocos básicos, seria mais que 10^7, significando que o número de partições seria aproximadamente 2^{10^7} e por isso será discutido métodos heurísticos para tal.

Referências Bibliográficas