DEPARTAMENTO DE COMPUTAÇÃO - UFSCAR

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

INFRAESTRUTURA DE COMPILAÇÃO PARA A IMPLEMENTAÇÃO DE ACELERADORES EM FPGA

PAULO HENRIQUE LOPES RETTORE

ORIENTADOR: MÁRCIO MERINO FERNANDES

São Carlos – SP

Novembro/2012

DEPARTAMENTO DE COMPUTAÇÃO - UFSCAR

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

INFRAESTRUTURA DE COMPILAÇÃO PARA A IMPLEMENTAÇÃO DE ACELERADORES EM FPGA

PAULO HENRIQUE LOPES RETTORE

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Processamento de Imagens e Sinais: Algoritmos e Arquitetura
Orientador: Márcio Merino Fernandes

São Carlos – SP

Novembro/2012

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

R439ic

Rettore, Paulo Henrique Lopes.
Infraestrutura de compilação para a implementação de aceleradores em FPGA / Paulo Henrique Lopes Rettore. -- São Carlos : UFSCar, 2012.
150 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2012.

1. Compiladores (Programas de computador). 2. Arquitetura de computador. 3. Alto desempenho. 4. FPGAs. 5. Cetus. I. Título.

CDD: 005.453 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"Infraestrutura de Compilação para a Implementação de Aceleradores em FPGA"

Paulo Henrique Lopes Rettore

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação

Membros da Banca:

Prof. Dr. Márcio Merino Fernandes
(Orientador - DC/UFSCar)

Prof. Dr. José Hiroki Saito
(PPG-CC/UFSCar)

Prof. Dr. Eduardo Marques
(ICMC/USP)

Prof. Dr. Valentin Obac Roda
(UFRN)

São Carlos
Novembro/2012

A meus familiares e amigos.

AGRADECIMENTOS

Inicialmente agradeço a Deus, pelas oportunidades. Agradeço o apoio recebido, nos momentos difíceis vividos ao longo desta trajetória, de meus familiares, amigos e namorada. Nessa caminhada o sentimento de entrega aos objetivos traçados sempre me acompanhou, e ao termino deste trabalho a sensação é de missão cumprida. Em especial agradeço ao Prof. e orientador Márcio M. Fernandes, que indubitavelmente desempenhou de maneira literal a orientação. Ao qual o agradecimento não se limita a pessoa acadêmica, pois em muito seus conselhos trasbordaram tais limites. Por fim, agradeço aos professores que tive a oportunidade de ser aluno e a instituição UFSCar.

A inspiração existe, mas tem que te encontrar trabalhando.

Pablo Picasso

RESUMO

O aumento no desempenho de processadores sequenciais tem sido limitado severamente por fatores físicos e tecnológicos nos últimos anos. Dessa forma, abordagens alternativas para a execução com alto desempenho ganharam maior importância nos últimos anos. Uma delas baseia-se na utilização de *hardware* customizado, implementado utilizando-se FPGAs. Entretanto, os métodos convencionais para programação desses dispositivos são notoriamente complexos, normalmente baseados em linguagens como VHDL e Verilog. Este trabalho apresenta o desenvolvimento de um *framework* de compilação para auxiliar a transformação de um *loop*, escrito em linguagem C, em sua versão para *hardware* customizado. A execução otimizada baseia-se na técnica de *loop pipelining*, a qual exige suporte avançado de compilação. Este é conseguido utilizando o compilador Cetus, que após uma série de modificações, pode ser utilizado como base para a geração semi-automática de aceleradores em *hardware* customizado. Como forma de guiar o desenvolvimento do compilador e validar suas funcionalidades básicas, dois casos de estudo foram considerados: um baseado na utilização de máquinas de estados finitos como método para a modelagem de *hardware* (EC-1), e outro baseado na linguagem de domínio específico LALP (EC-2). Em ambos os casos, o *framework* de compilação proposto mostrou-se útil como elemento facilitador ao desenvolvimento de *hardware* customizado de alto desempenho.

Palavras-chave: Compilador, Otimização, Paralelismo, *Loop*, HDL, HLS, FPGA, FSM, LALP, Cetus

ABSTRACT

In recent years, performance improvements in sequential microprocessors have been limited by physical and technological factors. For this reason, alternative approaches for high performance execution have gained importance. One of them is based in the use of reconfigurable hardware, implemented using FPGAs. However, conventional methods for programming those devices are notoriously complex, usually based on hardware description languages such as VHDL and Verilog. This work presents the development of a compilation framework to support the translation of a loop, described in C language, into its corresponding version for synthesis in reconfigurable hardware. The optimized execution is based on the loop pipelining technique, which requires advanced compiler support. That is achieved by using the Cetus compiler, enhanced by a number of modifications, and thus used as a basis for the semi-automatic generation of custom-hardware accelerators. In order to guide the compiler developments and validate its basic functionalities, two study cases were considered: one based on finite state machines as the method of choice for hardware modelling (EC-1), and another based on the LALP domain specific language. In both cases, the proposed compilation framework have shown to be a facilitator element for the development of high performance custom-hardware.

Keywords: *Compiler*, *Optimization*, *Parallelism*, *Loop*, HDL, HLS, FPGA, FSM, LALP, Cetus

LISTA DE FIGURAS

LISTA DE TABELAS

LISTAGENS

GLOSSÁRIO

ALP – *Aggressive Loop Pipelining*

ASIC – *Application Specific Integrated Circuit*

C2H – *C-to-Hardware Acceleration*

CDS – *Cadence Design Systems*

CFG – *Control Flow Graph*

CIRRF – *Compiler Intermediate Representation for Reconfigurable Fabrics*

CLB – *Configurable Logic Block*

DAG – *Directed Acyclic Graph*

DARPA – *Defense Advanced Research Projects Agency*

DDG – *Data Dependence Graph*

DFA – *Data Flow Analysis*

DFG – *Data Flow Graph*

DFI – *Data Flow Intensive*

DFS – *Depth-first search*

DG – *Directed Graph*

DMA – *Direct Memory Access*

DSP – *Digital Signal Processor*

DoD – *Department of Defense*

EEPROM – *Electrically-Erasable Programmable Read-Only Memory*

FPGA – *Field-Programmable Gate Array*

FSM – *Finite State Machine*

GCC – *GNU Compiler Collection*

GDA – *Gateway Design Automation*

GPL – *General Purpose Language*

GPP – *General Purpose Processor*

GP – *General Purpose*

HDL – *Hardware Description Language*

HLS – *High-Level Synthesis*

HPC – *High Performance Computing*

HTG – *Hierarchical Task Graph*

IEEE – *Institute of Electrical and Electronic Engineering*

II – *Initiation Interval*

IR – *Intermediate Representation*

LALP – *Language for Aggressive Loop Pipelining*

LE – *Logic Element*

LL – *Left-Left*

LM – *Logic Modules*

LR – *Left-Right*

LUT – *Look-Up Table*

OVI – *Open Verilog International*

PLD – *Programmable Logic Device*

RAM – *Random Access Memory*

ROCCC – *Riverside Optimizing Configurable Computing Compiler*

RTL – *Register Transfer Level*

SCC – *Strongly Connected Component*

SRAM – *Static Random Access Memory*

SUIF – *Stanford University Intermediate Format*

UFSCar – *Universidade Federal de São Carlos*

USP – *Universidade de São Paulo*

VHDL – *VHSIC Hardware Description Language*

VLSI – *Very Large Scale Integration*

SUMÁRIO

Capítulo 1

INTRODUÇÃO

Neste capítulo é apresentada uma breve introdução do trabalho desenvolvido. Para melhor compreensão, este está subdividido em contexto e trabalho proposto, que por sua vez contém os tópicos que sustentaram a pesquisa: premissas, problema, hipótese, objetivos gerais, objetivos específicos, motivações, justificativas e metodologia. Por fim é apresentada a organização do trabalho.

1.1 Contexto

Para a execução de algoritmos que demandem alta carga de processamento, tínhamos acesso a duas abordagens contrapostas em suas vantagens e limitações. Uma consistia em construir circuitos orientados à aplicação, garantindo alto desempenho e baixo consumo de energia. Contudo apresenta altos custos de projeto e implementação quando não produzidos em larga escala e após sua concepção o dispositivo se torna completamente inflexível. Para esta abordagem dá-se o nome ASIC (*Applications Specific Integrated Circuits*) (SMITH, 1997). Não menos importante, mas com objetivos diferentes da abordagem ASIC, surge o processador de propósito geral GPP (*General Purpose Processor*) (BLAKE; DRESLINSKI; MUDGE, 2009), que visa atender uma grande gama de funcionalidades, já que os algoritmos podem ser modificados a qualquer momento e facilmente incorporados ao código, o que permite uma grande flexibilidade na programação mesmo após sua fabricação. Porém esta abordagem apresenta desempenho insuficiente em determinadas situações e deve ser considerada em implementações que exigem altas cargas de processamento.

Observadas estas duas linhas para o desenvolvimento de aplicações computacionais, em *software* (implementado em computador de uso geral) e *hardware* (implementado em ASIC), é possível notar de forma clara suas vantagens e limitações. É portanto entre estes contextos que a computação reconfigurável, mais especificamente os dispositivos FPGAs (*Field Programmable Gate Arrays*) (COMPTON; HAUCK, 2002) se inserem. Tendo dessa forma o objetivo de preencher

o espaço que há entre *software* e *hardware*, possibilitando ganhos de desempenho mais expressivos que as soluções por *software*, enquanto mantém níveis de flexibilidades maiores que as soluções por *hardware* podem oferecer.

Apesar de os dispositivos reconfiguráveis apresentarem grandes vantagens em relação as abordagens ASIC e GPP, o aprimoramento das técnicas de otimização se faze necessário, tendo em vista a escassez de recursos e a necessidade de sua melhor utilização. Neste contexto um grande esforço é feito para que os compiladores consigam por meio de técnicas avançadas e ferramentas de HLS (*High-level synthesis*), otimizar da melhor maneira possível o código fonte, diminuindo a curva de aprendizado existente para se desenvolver aplicações utilizando linguagens de descrição de *hardware* como VHDL e Verilog.

Observadas as dificuldades para criar mecanismos facilitadores para programação nestes dispositivos, muito se tem feito na comunidade acadêmica e empresarial para a obtenção de técnicas de otimização e transformação, que por meio de ferramentas auxiliem a utilização destas linguagens de descrição de *hardware* para que, muito do que já dispomos, como documentações e sistemas desenvolvidos na linguagem de programação C não sejam desprezados.

Tanto na comunidade acadêmica quanto no âmbito comercial, ferramentas e técnicas são lançadas constantemente para possibilitar a exploração automática do espaço de projeto para diferentes requisitos e tecnologias. Uma das maneiras mais rápidas de modelar a funcionalidade de diferentes arquiteturas e estabelecer limites mínimos e máximos para a área, desempenho e potência, é proporcionado pela prototipação e exploração da arquitetura, que é o uso mais frequente das ferramentas de HLS. A Síntese de alto nível nada mais é do que a tradução de uma descrição de circuito em alto nível para uma descrição em um nível inferior. Segundo McFarland (MCFARLAND; PARKER; CAMPOSANO, 2002), HLS foi definida como a "tradução de uma especificação abstrata do comportamento de um sistema digital no nível algorítmico para a estrutura no nível RTL (*Register-Transfer Level*) que implementa aquele comportamento".

Podemos citar alguns dos projetos desenvolvidos com objetivo de realizar síntese de alto nível para FPGA: a ferramenta C2H (*C-to-Hardware Acceleration*) (NIOS, 2007) usada para geração de aceleradores para o processador Nios II, desenvolvida pela Altera; O compilador SPARK (GUPTA et al., 2004) desenvolvido na Universidade da Califórnia San Diego; A ferramenta C-to-Verilog (ROTEM, 2010) para compilação de programas em C para Verilog de forma online; O compilador ROCCC (*Riverside Optimizing Compiler for Configurable Computing*) (GUO; NAJJAR; BUYUKKURT, 2008; VILLARREAL et al., 2010) desenvolvido na Universidade da Califórnia Riverside; O *OpenMP extensions for FPGA Accelerators* (CABRERA et al., 2009), que é uma extensão da API 3.0 do OpenMP; Por fim, o sistema de compilação SystemC (GRÖTKER et al., 2002) que pode ser usado para a modelagem de *hardware* em diversos níveis de abstração.

Este trabalho de pesquisa utiliza como base um compilador que visa a paralelização como forma de otimização de código em alto nível. Conhecido como Cetus (LEE; JOHNSON; EIGEN-

MANN, 2004), foi criado por estudantes de graduação da Universidade de Purdue (*Purdue University*) nos EUA, e tem como objetivo a transformação automática de código fonte C serial (arquitetura convencional) para código C paralelizado e otimizado, possibilitando o aproveitamento dos recursos que uma arquitetura paralela pode oferecer.

A partir desta ferramenta, uma série de procedimentos foram criados com o intuito de modificar o compilador para atender aos objetivos deste trabalho, que visam criar facilitadores para a programação de *hardware* customizado. Dessa forma, dois estudos de caso moldaram a nova infraestrutura de compilação. Sendo o estudo de caso 1, a criação de mecanismos para facilitar a geração de Máquina de Estados Finitos - ou *Finite State Machines* (FSM), auxiliando na modelagem de *hardware* customizado. E o estudo de caso 2, a criação de facilitadores para a programação utilizando a linguagem e *framework* LALP (*Language for Aggressive Loop Pipelining*), que foi desenvolvida no projeto de doutorado da Universidade de São Paulo (USP) (MENOTTI et al., 2012), e tem como objetivo facilitar a descrição e geração de *hardware* de alto desempenho utilizando como base a técnica denominada ALP (*Aggressive Loop Pipelining*).

1.2 Trabalho Proposto

Para a definição dos objetivos do trabalho, algumas premissas e hipóteses foram consideradas, conforme apresentado a seguir.

1.2.1 Premissas

- Soluções computacionais baseadas em microprocessadores podem não ser a melhor solução em determinadas situações. Isso se deve a questões como desempenho, custo, consumo de energia e área física utilizada.
- Limites no aumento do *clock* de processadores levam a uma busca por soluções alternativas para o aumento de desempenho.
- *Hardware* customizado é uma das alternativas (total ou parcial) ao uso de microprocessadores.

1.2.2 Dificuldades

O projeto de *hardware* customizado requer conhecimentos específicos de técnicas e ferramentas, constituindo-se em uma barreira para desenvolvedores de *softwares*.

1.2.3 Hipótese

É possível utilizar uma infraestrutura de compilação voltada para microprocessadores, de modo a adaptá-la para o uso na implementação de *hardware* customizado?

1.2.4 Objetivos Gerais

Como objetivo geral do trabalho, é possível destacar a criação de uma infraestrutura de compilação, para a geração automática (ou semi-automática) de aceleradores em *hardware* customizado.

Para tal, um compilador com recursos avançados de análise de dependência de dados e otimizações visando a exploração do paralelismo foi utilizado. Mais especificamente, foi utilizado o compilador Cetus (LEE; JOHNSON; EIGENMANN, 2004), do tipo *source-to-source*, para a transformação de código em alto nível em uma estrutura de código ordenada respeitando o escalonamento em *modulo scheduling*, favorecendo a criação de aceleradores em *hardware* customizado.

Em suma, o objetivo deste trabalho é criar um sistema de auxílio a compilação (*framework*) para *hardware*, integrando ao compilador Cetus análises e transformações que viabilizam a exploração de *software pipelining* para a geração automática de aceleradores de *hardware* customizado, a partir de código implementado em uma linguagem de alto nível. Ou seja, o objetivo do trabalho é criar de um módulo de compilação capaz de tratar as análises e transformações em benefício dos *hardware* customizados, a partir de uma entrada código na linguagem C.

1.2.5 Objetivos Específicos

De maneira sucinta, este trabalho baseia-se nos seguintes objetivos específicos:

- Entendimento aprofundado da representação intermediária (IR) do compilador Cetus.
- Identificação e extensão das análises e transformações realizadas pelo Cetus.
- A partir do grafo de dependência de dados (DDG), efetuar as análises e/ou transformações visando o escalonamento das instruções utilizando a técnica de *software pipelining* para estruturas de repetição (*loop*).
- Estudo de Caso 1 (EC-1): FSM
  - O objetivo deste estudo de caso é testar a efetividade da infraestrutura de compilação como auxílio à geração de *hardware* customizado usando FSMs.

- Estudo de Caso 2 (EC-2): LALP
  - O objetivo deste estudo de caso é testar a efetividade da infraestrutura de compilação como auxílio à geração de *hardware* customizado usando LALP.
  - Criação de procedimentos a fim de resolver os pontos críticos da programação LALP.
  - Determinação das dependências de dados
  - Escalonamento das instruções considerando estas dependências
  - Geração de bloco de código LALP de forma automática a partir destas análises e transformações.
  - Realização de testes e comparações do código LALP gerado de forma automatizada com o código LALP escrito de forma manual.

1.2.6 Motivações e Justificativas

Para a validação deste trabalho foram analisados aspectos que justificassem a pesquisa realizada. Aspectos esses que consideraram os custos de projeto de *hardware* reconfigurável, o desempenho exigido e a realidade técnica. As justificativas que motivaram o desenvolvimento e conclusão deste trabalho são descritas a seguir:

- A grande demanda por sistemas de computação de alto desempenho tem-se tornado cada vez mais acentuada a medida que as aplicações que necessitam de alto processamento, se veem impedidas de obterem resultados significativos em tempo hábil, ou mesmo na impossibilidade de execução de aplicações utilizando a tecnologia atual. Isso é particularmente observado nos chamados sistemas embarcados.
- Para se alcançar um melhor desempenho na execução de aplicações complexas, é tido como alternativa a utilização de aceleradores de *hardware* ou *software* empregados em seções e funcionalidades críticas do programa. Tais soluções se constituem em plataformas híbridas de execução.
- A busca por uma solução adequada em uma plataforma híbrida, exige a exploração conjunta de uma série de alternativas de *hardware* e *software*, no sentido de balancear parâmetros como desempenho, consumo de energia, temperatura e custo.
- Existe uma grande necessidade para o desenvolvimento de novas técnicas que facilitem o projeto e implementação de aplicações para execução em plataformas híbridas, e a consequente utilização das mesmas de maneira eficiente e em larga escala, notadamente em sistemas embarcados.

- Com a utilização de técnicas que trabalhem em aproximar os problemas computacionais das linguagens mais próximas dos desenvolvedores de *software*, é possível viabilizar a formulação de melhores soluções para estes problemas.

1.2.7 Metodologia

Para a execução deste trabalho, alguns pontos importantes foram seguidos com o intuito de guiar o estudo de maneira evolutiva e construtiva. A seguir são apresentados estes pontos, tal como, o contexto geral do trabalho para o estudo de caso 1 na Figura 1.1 e o estudo de caso 2 na Figura 1.2:

- Aperfeiçoamento do conhecimento do compilador Cetus, em particular sua representação intermediaria com a análise de dependências de dados.
- Utilização do compilador Cetus como base para o *parser* de programação em C e análise de dependências de dados
- Extração do DDG em um formato que viabiliza manipulação.
- Criação de novos passos de análise e transformação de código, de modo a adaptar o compilador Cetus para o auxílio na geração semi-automática de *hardware* customizado.
- Estudos de Caso:
  - O objetivo dos estudos de caso é guiar o desenvolvimento dos novos passos de compilação, e avaliar as análises e transformações efetuadas no contexto dos objetivos do trabalho proposto.
  - Estudo de Caso 1: FSM
    - Aperfeiçoamento do conhecimento sobre máquinas de estados finitos, tal como suas características e funcionamento para a modelagem de *hardware* customizado.
  - Estudo de Caso 2: LALP
    - Aperfeiçoamento do conhecimento da linguagem LALP e *framework* existente na sua forma atual.
    - Criação de um gerador de linguagem LALP a partir do DDG gerado.
    - Geração de um novo código fonte na *syntax* LALP (ou próximo).

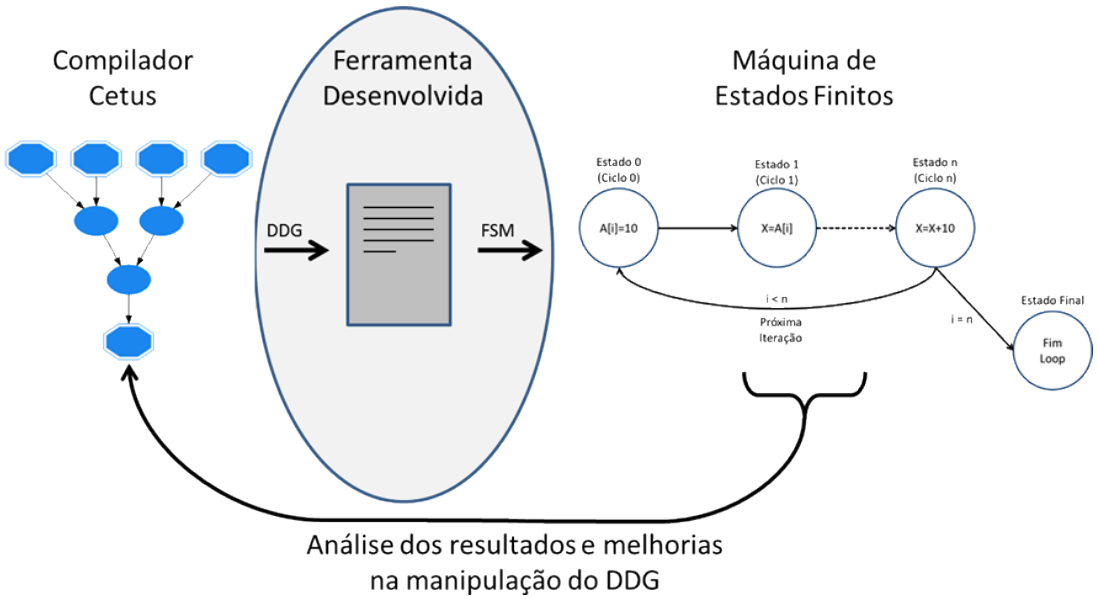

Figura 1.1: Contexto do trabalho para o EC-1 e guia para a metodologia utilizada.

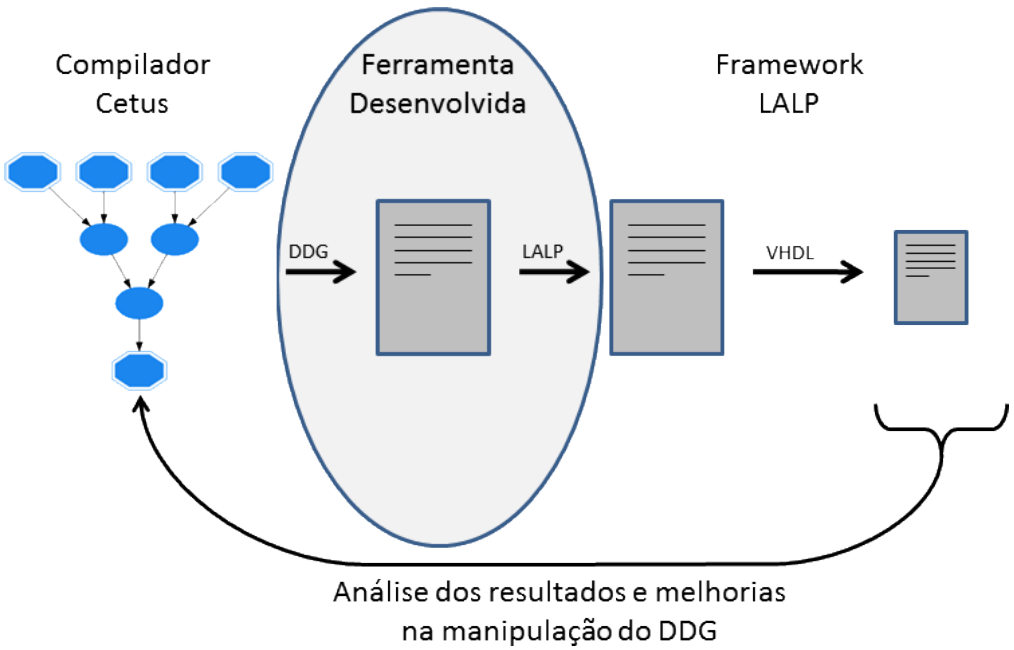

Figura 1.2: Contexto do trabalho para o EC-2 e guia para a metodologia utilizada.

1.3 Organização do Trabalho

Este trabalho encontra-se organizado da seguinte forma: No Capítulo 2 são abordados os principais temas que fundamentam a base teórica desta pesquisa. O Capítulo 3 apresenta as principais ferramentas disponíveis atualmente que tem alguma relação com o tema proposto por este trabalho. No Capítulo 4 são abordadas todas as modificações realizadas no compilador Cetus, tais como, melhorias nas análises, correções de erros, aprimoramento das técnicas e criação de procedimentos. Em continuidade, o Capítulo 5 apresentando os resultados e contribuições obtidas destas modificações. Por fim, o Capítulo 6 apresenta as conclusões deste trabalho, viabilidade do estudo, as principais dificuldades enfrentadas e os trabalhos futuros sugeridos.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os principais assuntos estudados, tendo em vista fundamentar a base teórica de todo o trabalho realizado.

Este capítulo dedica-se a fundamentar toda base teórica para a solidificação deste projeto de pesquisa que, por sua vez, visa facilitar a programação de *hardware* customizado. Para tal, a Seção 2.1 trata das plataformas de computação disponíveis e o destaque à computação reconfigurável do tipo FPGA, que possibilita a implementação de *hardware* de alto desempenho. Na Seção 2.2 é explorado o conceito da programação usando máquina de estados finitos (FSM), que por sua vez representa o estudo de caso 1 deste trabalho. Na Seção 2.3 são abordadas as linguagens de descrição de *hardware* VHDL e Verilog, que viabilizam vários níveis de abstração na implementação de sistemas digitais, além de universalizar a programação destes dispositivos. É também tratada nesta seção, o que é a síntese de alto nível das linguagens de programação utilizadas para os dispositivos digitais e o estudo de caso 2, o *framework* e linguagem LALP, que busca explorar o paralelismo de *loops* em computação reconfigurável permitindo a programação de aceleradores eficientes usando *loop pipelining* de forma agressiva, buscando melhor desempenho e aproveitamento dos recursos disponíveis. Na Seção 2.5 é descrita a estrutura básica de um compilador, algumas técnicas utilizadas para obtenção de melhores desempenhos e análises de dependências de dados para otimizações em soluções paralelizadas. Por fim, em continuidade a este tópico, é apresentado na Seção 2.6 o compilador Cetus, que a partir de técnicas consolidadas de *software pipelining* analisa as dependências de dados e otimiza o código com foco na exploração do paralelismo.

2.1 GPP/ASIC/Computação Reconfigurável

Os computadores convencionais foram desenvolvidos com o objetivo de atender uma grande gama de funcionalidades GP, sendo classificados como computadores de uso geral. Atendendo a este objetivo, os processadores de propósito geral comumente chamados de GPP (*General*

Purpose Processor) (BLAKE; DRESLINSKI; MUDGE, 2009), possuem arquitetura com instruções fixas. Portanto, as diversas aplicações desenvolvidas para este hardware tem como preocupação estas restrições físicas.

Contudo, com os avanços das técnicas de programação e as limitações do *hardware*, diversas ferramentas foram criadas para auxiliar o desenvolvimento de sistemas e possibilitar a extração de melhores desempenhos considerando os custos finais. Desta forma, esta abordagem apresenta uma vantagem significativa, já que o algoritmo pode ser modificado a qualquer momento e facilmente incorporado ao código, permitindo uma grande flexibilidade na programação, haja vista as linguagens orientadas a objeto (SKLIAROVA; FERRARI, 2003).

Embora pareça ser a melhor solução para nossos problemas computacionais, quando a situação exige uma carga de processamento que exceda as limitações de um computador de propósito geral, o desempenho desejado para realizar as funcionalidades necessárias deixa de ser alcançado. Dessa forma, uma nova abordagem para a solução do problema deve ser utilizada, ou seja, a criação de um sistema computacional de alto desempenho.

Uma das técnicas para se atingir níveis de desempenho mais altos é a utilização da computação paralela, que vislumbra a fragmentação de um grande problema em diversos problemas menores, possibilitando a execução de pequenos processos de forma paralela (PATTERSON; HENNESSY, 2009).

Apesar desta solução apresentar algumas vantagens, não é qualquer aplicação que pode ter seu desempenho melhorado utilizando a computação paralela. É necessário que o sistema possua uma estrutura compatível a este modelo de computação. Para muitas aplicações as técnicas de programação paralela não oferecem grandes alterações no *speedup*[1], a razão disso é parcialmente explicada pela lei de Amdhal[2]. Além disso estas aplicações sofrem grande sobrecarga de comunicação entre os vários núcleos (*overhead*), o que pode anular total ou parcialmente os benefícios do processamento paralelo (SKLIAROVA; FERRARI, 2003).

Outra solução para aplicações de alto desempenho, consiste em construir circuitos orientados à aplicação, ou seja, projetar e fabricar circuitos integrados projetados especificamente para executar uma determinada aplicação - ou *Applications Specific Integrated Circuits* (ASIC) (SMITH, 1997). Em função dessa especificidade, podem ser projetados *chips* menores, mais velozes e que consumam menos energia que um processador de uso geral. Isso é justificado com o controle fixo e unidades funcionais personalizadas e otimizadas para cada aplicação, podendo atingir ótimos resultados com a utilização reduzida dos recursos de *hardware*.

Contudo, é importante salientar, que há um preço muito alto a ser pago ao optar por desen-

[1] É a relação entre o tempo gasto para executar uma tarefa com um único processador e o tempo gasto utilizando N processadores

[2] O ganho de desempenho obtido melhorando uma determinada parte do sistema é limitado pela fração de tempo que essa parte é utilizada pelo sistema durante a sua operação (CARTER, 2002)

volver um projeto através de um ASIC. A princípio existe o custo do projeto e da implementação que é demasiadamente alto e só deve ser justificado em uma produção em larga escala, já que o custo do silício é relativamente alto para pequenas implementações. Esta situação se torna clara ao analisarmos o custo baixo dos processadores de propósito geral, que são projetados em larga escala amortizando o custo inicial do silício.

Outro ponto importante, é o tempo gasto para o desenvolvimento do projeto, que pela complexidade se torna lento enquanto que os processadores de uso geral continuam cada vez mais aumentando sua capacidade. Em virtude disso, os projetos baseados em ASIC se tornam obsoletos em pouco tempo, já que após sua concepção estes dispositivos se tornam completamente inflexíveis, impossibilitando qualquer tipo de alteração, seja para a melhoria de desempenho ou adaptação em uma nova aplicação.

Observadas estas duas abordagens de programação, em *software* (implementado em computador de uso geral) e *hardware* (implementado em ASIC), é possível notar de forma clara suas vantagens e limitações. A computação reconfigurável vem com o objetivo de preencher o espaço que há entre o *software* e *hardware*, possibilitando ganhos de desempenho mais expressivos que a solução por *software*, enquanto mantém um nível de flexibilidade maior que o *hardware* pode oferecer, como é ilustrado na Figura 2.1.

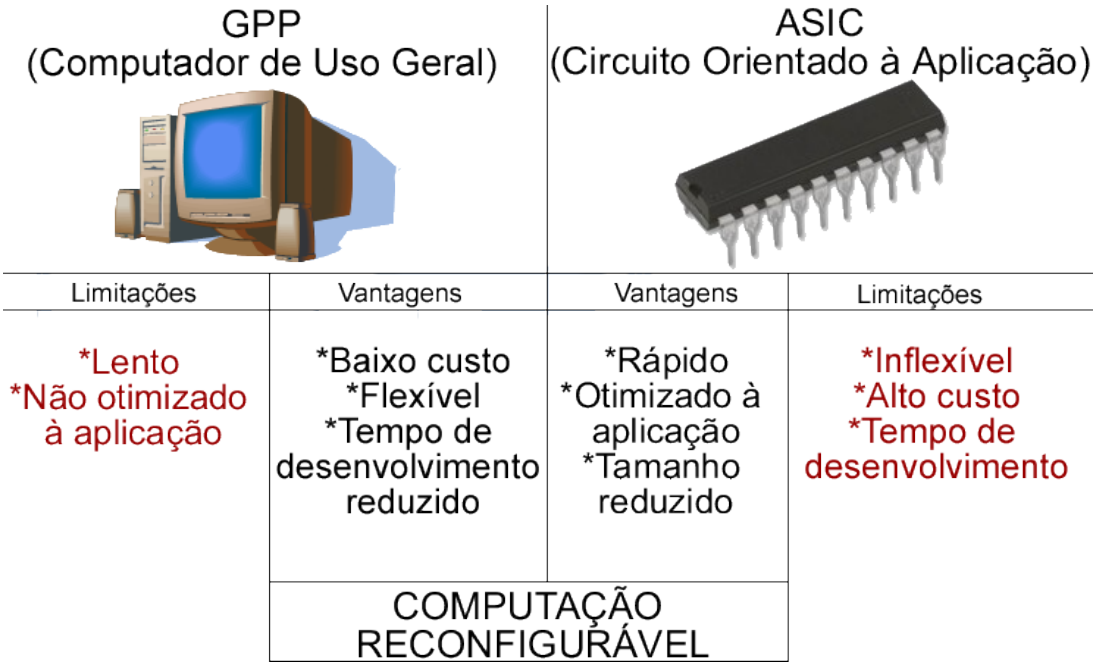

Figura 2.1: A computação reconfigurável integra as vantagens das abordagens GPP e ASIC

A tecnologia utilizada pela computação reconfigurável, é baseada em dispositivos lógicos reprogramáveis que fornecem desempenho e flexibilidade a nível de portas lógicas. Os dispositivos de *hardware* mais difundidos baseados nesta abordagem são os FPGAs (*Field Programmable Gate Arrays*) (HAUCK; DEHON, 2008; KUON; TESSIER; ROSE, 2008).

2.1.1 FPGA

Dispositivo criado em meados dos anos 80 pela empresa Xilinx (CHAN; MOURAD, 1994; XILINX,) com o objetivo de possibilitar o molde do *hardware* de acordo com a aplicação. O FPGA (*Field Programmable Gate Array*) ou mesmo Matriz de Portas Programáveis em Campo, é um tipo de circuito integrado reconfigurável contendo uma matriz de blocos lógicos, uma rede de interconexão (*Switch Matrix*) e um conjunto de blocos de entrada e saída. A capacidade dos FPGAs serem reprogramáveis é resultado da configurabilidade destes três elementos de tal forma que qualquer função computacional pode ser implementada (BOBDA, 2007).

Os FPGAs atuais apresentam recursos que permitem a construção de sistemas extremamente complexos em um único chip, pois várias estruturas heterogêneas, tais como blocos de memória, passaram também a ser encapsuladas permitindo acelerar uma variedade de aplicações. Além disso, uma vez que os circuitos integrados realizam tarefas diferentes utilizando a mesma área, os dispositivos reconfiguráveis podem aproveitar melhor sua densidade (DEHON, 2002).

Embora o custo de engenharia e o tempo de desenvolvimento de um FPGA apresente vantagens quando comparado ao ASIC, projetos desenvolvidos com tecnologia VLSI, como processadores e memórias RAM utilizadas nos PCs, apresentam maior velocidade, densidade e complexidade. Contudo, tal tecnologia apresenta custos de produção extremamente elevados, tornando-se viável quando produzidos em larga escala (KUON; TESSIER; ROSE, 2008). A Figura 2.2 demonstra a relação entre PLDs[3],FPGAs,ASICs e os projetos VLSI (HAMBLEN; HALL; FURMAN, 2008).

Outra relação importante que deve ser feita diz respeito à flexibilidade e desempenho das soluções implementadas. Soluções baseadas no computador de Von Neumann (NEUMANN; ASPRAY; BURKS, 1987) possuem grande flexibilidade por serem capazes de computar qualquer tipo de tarefa. Por esta razão a terminologia GPP pode também ser utilizada para descrever a máquina de Von Neumann. Estes computadores não apresentam desempenho muito elevado, pois não foram desenvolvidos para processar dados de forma paralela, além de manterem suas instruções fixas independente da aplicação que será utilizada. Desta forma a flexibilidade é alcançada, pois a aplicação se adapta ao hardware. (HAMBLEN; HALL; FURMAN, 2008)

Os ASICs possuem grande desempenho, pois sua programação é otimizada para cada aplicação a ser desenvolvida, ou seja, cada instrução necessária à aplicação pode ser construída no próprio *chip*. Desta forma, o desempenho é obtido, pois o hardware é adaptado à aplicação. (HAMBLEN; HALL; FURMAN, 2008)

Os DSPs ou processadores de sinais digitais são utilizados em domínio específico e apre-

[3]Dispositivos Lógicos Programáveis Simples (PLDs), como Lógica de Matriz Programável (PAL) e Matrizes lógicas programáveis (PLAs), que têm sido utilizados por mais de trinta anos

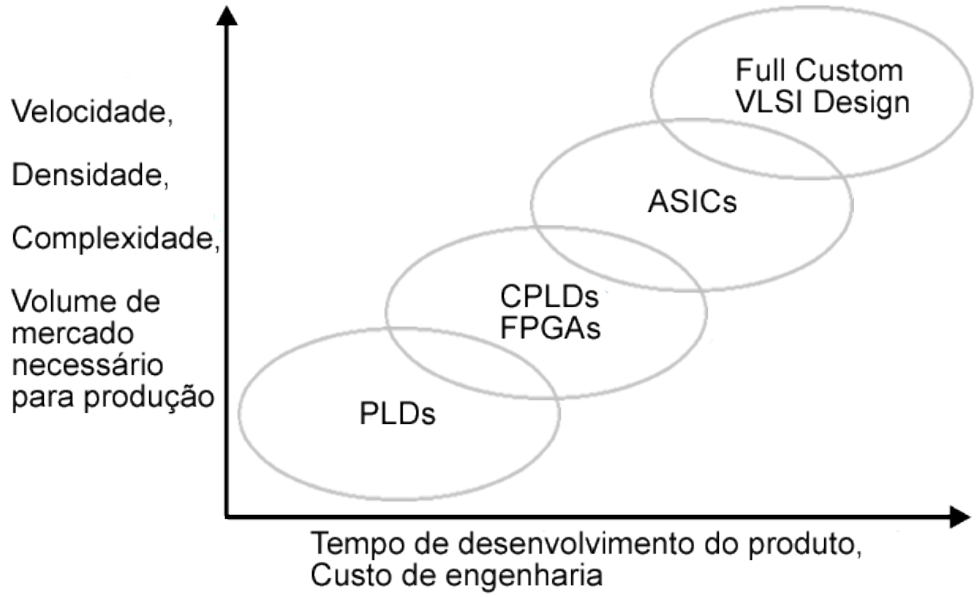

Figura 2.2: Relações de mercado de lógica digital.
Adaptado de (HAMBLEN; HALL; FURMAN, 2008)

sentam boa flexibilidade, porém são utilizados apenas por uma classe específica de aplicações que não exigem números elevados de operações por ciclo de *clock*. Já os dispositivos FPGAs alcançam alto grau de flexibilidade e performance, pois são programados através de ligações entre as portas formando multiplicadores, registos, somadores e assim por diante. Possibilitando sua utilização de forma generalizada, além de permitir elevado nível de performance (MEYER-BAESE, 2007). A Figura 2.3 demonstra a relação de flexibilidade e performance entres estes dispositivos.

Como citado, um FPGA tem em sua composição principal três tipos de recursos, sendo estes: blocos lógicos, blocos de entrada e saída e chaves de interconexão programáveis, além do roteamento entre eles. A Figura 2.4 demonstra a estrutura básica de um FPGA.

Os blocos lógicos são responsáveis pelas funções lógicas implementadas. Alguns destes possuem recursos sequenciais tais como *flip-flops* ou registradores, conforme a Figura 2.5. A nomenclatura dada a estes dispositivos pode variar de acordo com o fabricante. A Xilinx chama seu Bloco Lógico de CLB (*Configurable Logic Block*), a Actel usa o termo LM (*Logic Modules*), já a Altera utiliza o termo LE (*Logic Element*) e *Macrocell* para diferentes séries fabricadas.

As chaves de interconexão (*Switch Matrix*) permitem o roteamento entre os blocos lógicos

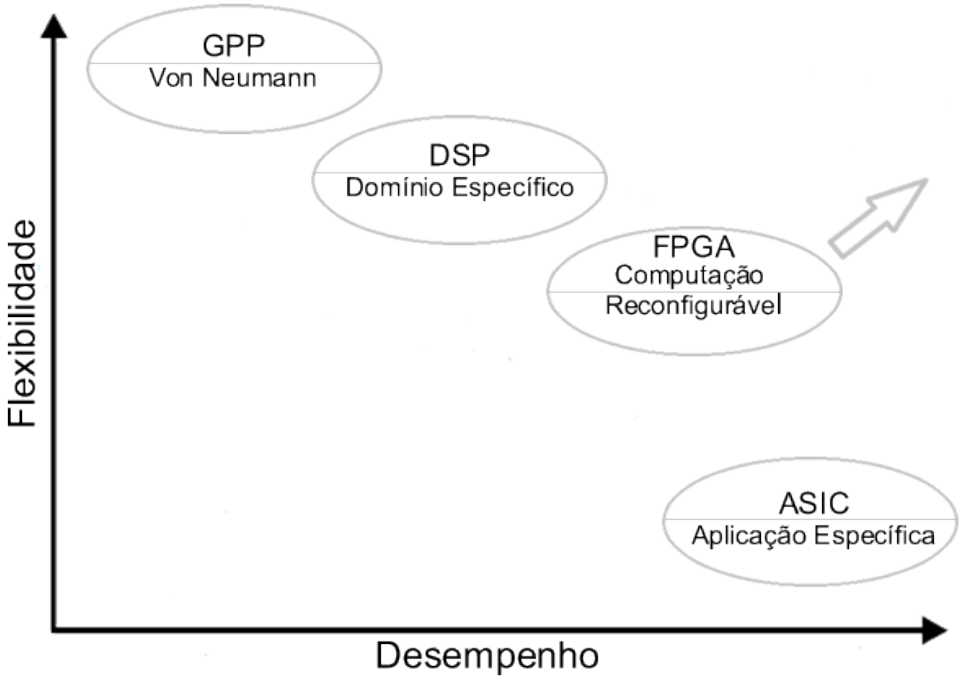

Figura 2.3: Relação entre flexibilidade e desempenho.
Adaptado de (BOBDA, 2007)

através das conexões globais[4] como ilha[5], linhas longas[6], celular[7] e horizontal[8] (DESCHAMPS et al., 2006). A Figura 2.6 apresenta a estrutura de uma *Switch Matrix*.

Os blocos de entrada e saída possuem *buffers tristate* para as saídas e um *buffer* para os sinais de entrada. (DESCHAMPS et al., 2006)

A técnica mais utilizada para a construção dos blocos lógicos programáveis é conhecida como *Look-Up Table* (LUT), que é uma estrutura lógica que implementa tabelas verdade de funções de k entradas. As entradas da função correspondem ao endereço enquanto a saída corresponde ao conteúdo da memória. Uma LUT implementa todas as 2^{2^k} funções lógicas de k entradas $k \geq 2$. Desta forma uma LUT de 2 entradas é capaz de implementar todas as 16 funções de 2 entradas. Estes blocos lógicos consistem em uma tabela verdade de k entradas, uma lógica de controle e uma parte sequencial que pode ser um *flip-flop* (LIMA, 1999). As LUTs utilizam funções com poucas entradas e saídas implementadas em células de armazenamento. As células de armazenamento dos LUTs podem ser voláteis, o que caracteriza a perda do conteúdo armazenado em caso de falta de energia, sendo necessário que o FPGA seja programado toda vez que for energizado. Na Figura 2.7 é possível observar o exemplo de um circuito de uma

[4]Rede de interconexão formada em linhas e colunas que se ligam através de *Switch Matrix*. Esta rede interliga os blocos lógicos e os blocos de E/S

[5]Os *clusters* lógicos são cercados por canais de roteamento verticais e horizontais segmentados

[6]São conexões que atravessam todo o circuito sem passar pelas *switch matrix*

[7]As conexões mais usadas são transformadas em locais entre *clusters* lógicos e somente um pequeno ou nenhum através de segmentos de conexões longas

[8]Usa principalmente canais de interconexão horizontais para roteamento de sinais entre dois *clusters* lógicos

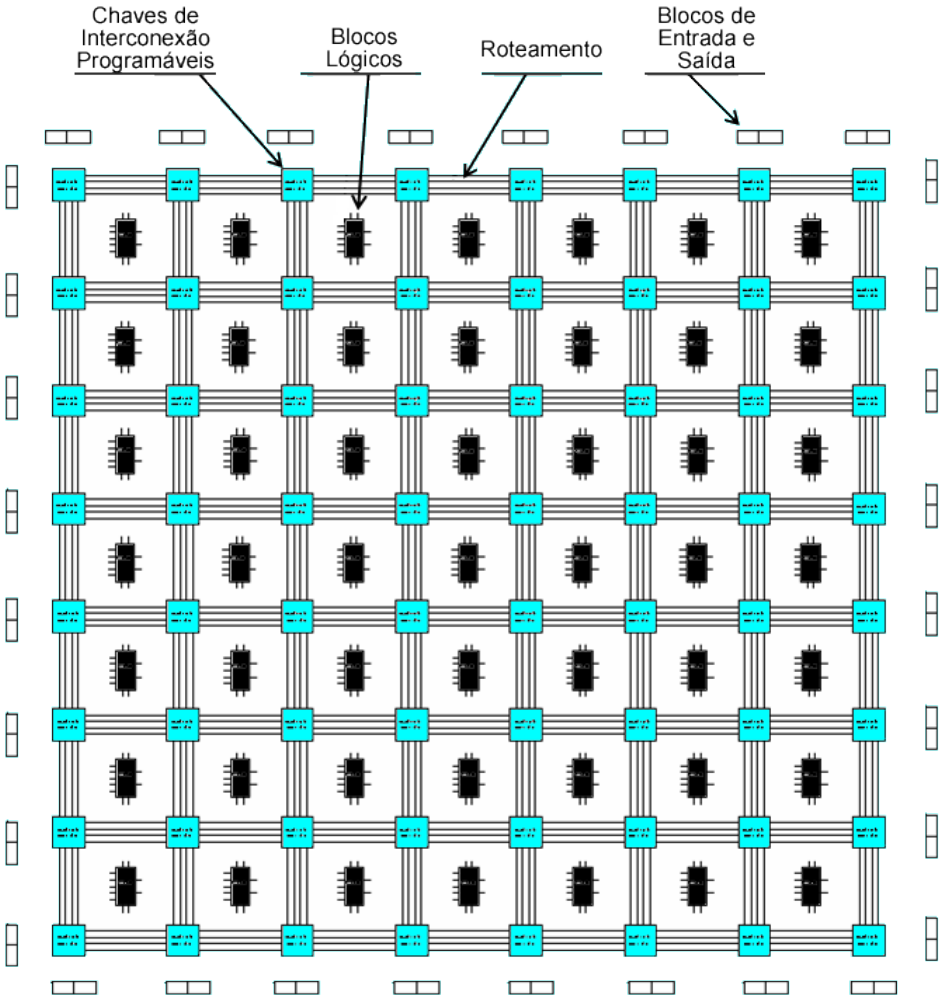

Figura 2.4: Estrutura básica de um FPGA.
Adaptado de (PELO; TERROSO, 1998)

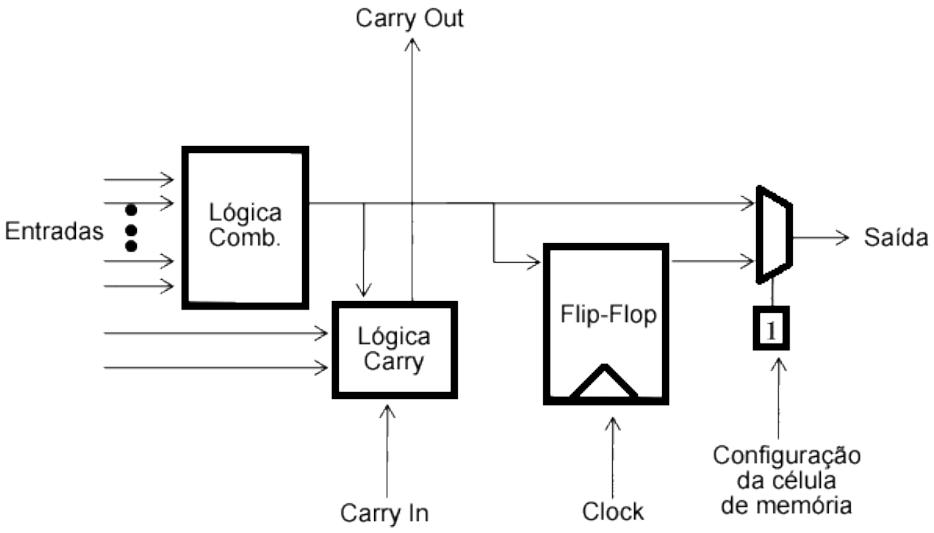

Figura 2.5: Bloco lógico programável.
Adaptado de (DESCHAMPS et al., 2006)

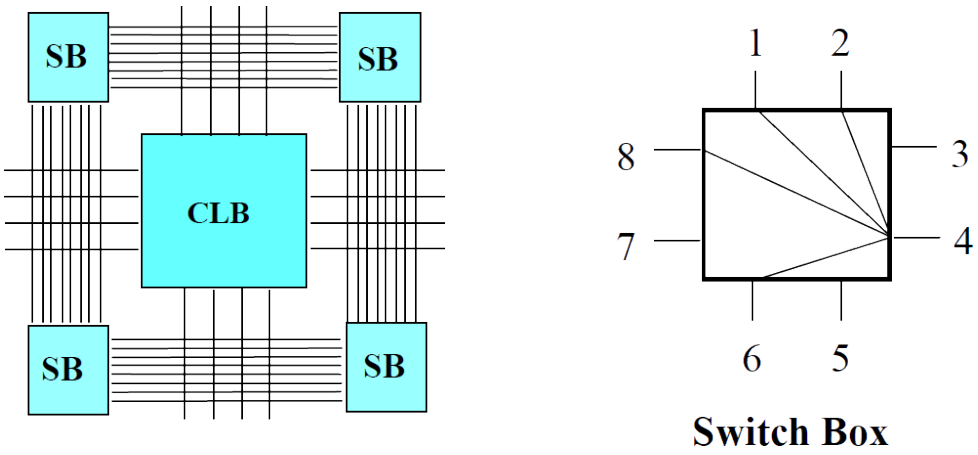

Figura 2.6: Representação de uma *Switch Box*.
Fonte: (PELO; TERROSO, 1998)

LUT e o seu uso para implementar uma função.

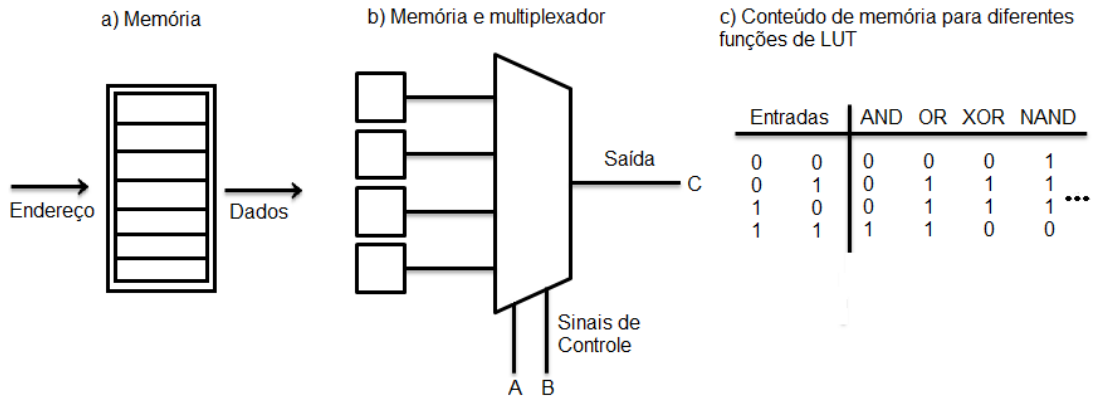

Figura 2.7: Implementação da lógica combinatória de uma LUT

A construção de sistemas embarcados tem se beneficiado com a evolução dos FPGAs, pois cada vez mais oferecem equilíbrio entre o desempenho e a flexibilidade (COMPTON; HAUCK, 2002). Outra área que se beneficia com a utilizações destes dispositivos é a computação de alto desempenho - ou *High Performance Computing* (HPC), já que proporcionam aumento considerável na capacidade computacional de forma a superar o obtido com os microprocessadores comuns, permitindo a criação de arquiteturas explicitamente paralelas e especializadas (HERBORDT et al., 2007; BAXTER et al., 2007). As aplicações que exemplificam a necessidade desta tecnologia são aquelas que tradicionalmente exigem alto poder de processamento como biologia molecular, dinâmica dos fluídos, aplicações financeiras, simulações diversas etc.

A programação em dispositivos FPGAs tradicionalmente respeita um fluxo de desenvolvimento como apresentado na Figura 2.8. Inicialmente o circuito é desenvolvido, seja por diagramas esquemáticos e/ou linguagens de descrição de *hardware*. No processo de síntese é verificada a consistência do que foi programado, podendo em seu término ser realizada simulações comportamentais do sistema, onde serão verificadas se as funções foram implementadas

corretamente. Em seguida é realizada a implementação do projeto, ou seja, os elementos são posicionados e as ligações entre eles são roteadas, mapeando o circuito desenvolvido no dispositivo alvo. Nesta fase do processo, as simulações se tornam mais reais, já que são consideradas as propriedades físicas do dispositivo. Logo é possível analisar o tempo de propagação do sinal elétrico no meio e realizar simulações para avaliar o circuito projetado. Por fim, é gerado um arquivo de configuração para o dispositivo, chamado de *bitstream*, que pode ser enviado ao FPGA utilizando um cabo ou mesmo um dispositivo de armazenamento não volátil, o que viabiliza embarcar o aparato de configuração no mesmo sistema.

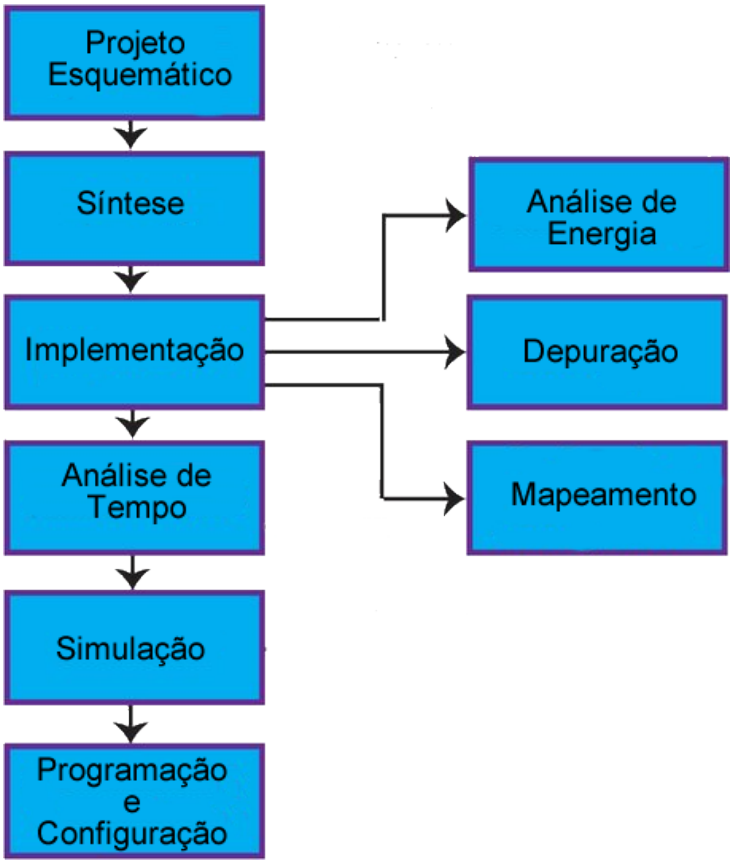

Figura 2.8: Fluxo de desenvolvimento para um FPGA.
Adaptado de (QUARTUS, 2007)

Hoje existem diversas ferramentas para auxiliar no desenvolvimento de projetos em FPGA, seja em cada processo citado ou mesmo de forma global. Contudo, os fabricantes destes dispositivos fornecem ferramentas que permitem realizar todos os processos necessários para a concepção do projeto, desde sua especificação, ao envio do arquivo de configuração ao FPGA.

Os principais fabricantes que produzem FPGAs e componentes para sua construção além

das tecnologias de programação utilizadas como *Antifuse*[9], EEPROM[10] e SRAM[11], são mostrados na Tabela 2.1. Na sequência, a Tabela 2.2 apresenta as principais características dos dispositivos mais recentes dos fabricantes Xilinx e Altera.

Fabricantes	**Tecnologia de Programação**
Achronix Semiconductor Corp.	SRAM
Actel Corp.	*Antifuse*/EEPROM
Altera Corp.	SRAM/EEPROM
Cypress Semiconductor Corp.	SRAM
Integrated Circuit Technology (ICT) Corp.	EEPROM
Lattice Semiconductor Corp.	EEPROM
Quicklogic Corp.	*Antifuse*
SiliconBlue Technologies Corp.	SRAM
Xilinx Corp.	SRAM/EEPROM

Tabela 2.1: Principais fabricantes de FPGA e componentes.
Adaptado de (COSTA, 2007)

	Virtex 7	**Stratix V GS**
Tecnologia	28nm	28nm
Células lógicas (Kgates)	285K a 2,443K	563K a 1,100K
Memória Total (Mbits)	18M a 77M	32M a 34M
DSP	420 a 2,350	1,620 a 1,840
Transceiver (numero/Gbps)	72 / 13.1[12]	27 / 12.5

Tabela 2.2: Comparação entre os modelos de placas das fabricantes de FPGA Xilinx e Altera.
Adaptado de (ASSUMPÇÃO, 2010)

Com a evolução constante dos FPGAs, tem-se aprimorado a capacidade de introduzir em uma mesma pastilha um ou mais processadores, como é o caso de algumas famílias de FPGAs da Xilinx que possuem processadores *hardcore* PowerPC internamente, viabilizando uma nova abordagem que possibilite extrair os benefícios de ambas arquiteturas, permitindo que sistemas possam ser desenvolvidos de forma híbrida, com parte da aplicação em *software* e parte em *hardware*. A este tipo de abordagem da-se o nome de *codesign*.

Apesar da combinação híbrida de FPGA/processador ser ideal para o desenvolvimento de sistemas complexos, as otimizações destes sistemas a partir de linguagens de descrições de alto nível, ainda possuem muitas lacunas e precisam ser melhor exploradas.

[9] *Antifuse* ou Anti-fusível é um dispositivo de dois terminais que quando programado se funde, criando então uma conexão

[10] *Electrically-Erasable Programmable Read-Only Memory - podem ser programadas reprogramadas várias vezes eletricamente*

[11] *Static Random Access Memory* - são memórias estáticas de acesso aleatório

2.2 Modelagem de *Hardware* Customizado usando FSM

Em um circuito combinacional, a saída depende apenas de uma combinação das entradas, enquanto que em um circuito sequencial, a saída depende além das variáveis de entrada, também de seus estados anteriores. Ou seja, um sistema sequencial deve ter a capacidade de capturar a influência de todas as entradas passadas sobe as saídas atuais e futuras. Consequentemente, é necessário memorizar a sequência de entrada, para ser capaz de determinar a saída no tempo. Uma vez que o numero de estados do sistema é finito, os sistemas são chamados de Sistemas de Estados Finitos ou Máquinas de Estados Finitos. (ERCEGOVAC; LANG; MORENO, 2000)

Desse modo, uma Máquina de Estados Finitos - ou *Finite State Machines* (FSM) é definida como uma técnica de modelagem de sistemas, seja este um programa de computador ou um circuito lógico. A descrição do estado de um sistema sequencial ou a modelagem de uma FSM, usa três variáveis no tempo: a entrada, o estado e saída. Além disso há a função de transição de estado e a função de saída.

Uma FSM pode estar em apenas um estado por vez e este estado é chamado de estado atual. Cada estado armazena as informações sobre o seu passado, ou seja, as mudanças desde a entrada em um estado, no início do sistema, até o momento presente são refletidos no estado. Uma transição indica a mudança de estado e é descrita por uma condição que precisa ser satisfeita para que a transição ocorra.

Os argumentos e o valor das funções de transição de estado e saída são variáveis com um número finito de valores e podem ser representados por tabelas, expressões ou mapas. Outra descrição gráfica comumente usada para representar uma FSM são os diagramas de estados, que são grafos orientados usados para demonstrar as funções de transições e de saída em um sistema sequencial.

Cada nó significa a representação de um estado assim como cada arco uma transição. Um arco que vai do nó S_k para o só S_j e rotulado como x/z especifica que, para cada estado atual S_k e uma entrada x, o próximo estado é o S_j e a saída é z. Na Figura 2.9 é possível demostrar como esta representação é feita.

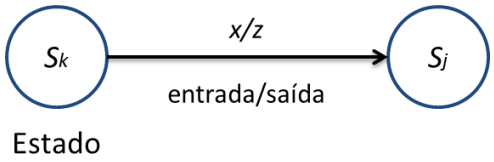

Figura 2.9: Representação em diagrama de estados.

Como forma de exemplificar a descrição de estados de uma FSM, a seguir são apresentados os parâmetros de entrada, saída, estado e estado inicial. Sendo a e b entradas pertencentes à $x(t)$, p e q saídas pertencentes à $z(t)$ e os estados S_0, S_1, S_2 pertencentes à $s(t)$, sendo S_0 o estado

inicial. Na Tabela 2.3 são demonstradas as funções de transição de estado e saída considerando os parâmetros listados.

Entrada:	x(t) $\in \{a, b\}$
Saída:	z(t) $\in \{p, q\}$
Estado:	s(t) $\in \{S_0, S_1, S_2\}$
Estado Inicial:	s(0) = S_0

s(t)	x(t)	
	a	b
S0	S1,p	S2,q
S1	S1,p	S0,p
S2	S1,p	S2,p
	s(t+1), z(t)	

Tabela 2.3: Funções de transição de estado e saída.

A partir da tabela de transição de estados e saída, é representado o respectivo diagrama de estados mostrado na Figura 2.10.

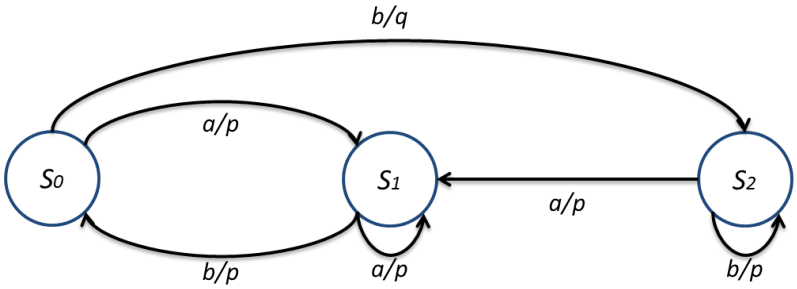

Figura 2.10: Representação em diagrama de estados referente a Tabela 2.3.

O tipo de função de saída de uma FSM determina um dos possíveis modelos utilizados. O modelo de máquina de Mealy é um sistema sequencial cuja saída no tempo t depende do estado e da entrada no tempo t, sendo incluso tanto o sinal de entrada como o de saída para cada vértice de transição. Já o modelo de máquina de Moore é um autômato de estado finito onde a saída no tempo t depende apenas do estado no tempo t, sendo dependente apenas do estado atual da máquina. (ERCEGOVAC; LANG; MORENO, 2000)

2.2.1 Pontos de Interesse do Trabalho

Como forma de enfatizar o interesse deste trabalho auxiliar a modelagem de *hardware* customizado usando máquinas de estados finitos são apresentados a seguir os pontos importantes relacionados a este EC:

- A abordagem padrão mais utilizada para projetos de *hardware* customizado é baseada no uso de FSM para modelagem de sistemas.
- Ferramentas de programação para FPGAs, como Quartus II, também utilizam FSM para modelagem do *hardware*.
- Nesse contexto, o trabalho proposto visa fornecer ao programador uma ferramenta que o auxilie na geração de máquinas de estado para a execução de trechos de código escritos em linguagem C. Em particular, a ferramenta visa auxiliar a geração de FSM para a execução otimizada utilizando a técnica de *loop pipelining*, processo esse altamente dependente de uma análise de dependência de dados sofisticada.

2.3 Programação em FPGA: Ferramentas/Linguagens

Nesta seção será tratada a programação em dispositivos FPGAs considerando duas abordagens, sendo na Subseção 2.3.1 um apanhado genérico ao modelo de programação, considerando o modelo esquemático e as linguagens de descrição de *hardware* VHDL e Verilog. Em seguida é apresentada na Subseção 2.3.2 a síntese de alto nível e ferramentas que fazem uso desta abordagem.

2.3.1 Abordagem Esquemática e HDL

A definição do comportamento de um dispositivo FPGA pode ser feita de duas formas, uma fornecendo um desenho esquemático e outra uma linguagem de descrição de *hardware*, sendo possível esta descrição também através de máquinas de estados finitos. A entrada esquemática é a utilização associada de portas lógicas formando um circuito contendo mecanismos de entrada e saída. A utilização desta forma de programação pode facilitar a visualização de um projeto quando o mesmo não se torna muito complexo, caso contrário o tempo gasto para que se defina o comportamento de um dispositivo FPGA se torna longo e moroso até sua concepção.

As linguagens de descrição de hardware (HDL) surgiram para criar alternativas para se especificar circuitos digitais de forma diferente a esquemática, utilizando ferramentas de projeto baseadas em textos ou linguagens que possibilitassem a descrição do sistema a ser desenvolvido. HDL descreve o que um sistema deve fazer e como fazer. Esta descrição é a modelagem de um sistema de *hardware* em um *software* simulador, podendo ser implementada em dispositivos programáveis como FPGAs possibilitando assim a utilização dos sistemas de forma mais abrangente, permitindo também a vantagem de alterar e reutilizar o código a qualquer momento.

Atualmente as linguagens de descrição de hardware VHDL (LIPSETT; MARSCHNER; SHAHDAD, 1986) e Verilog (THOMAS; MOORBY, 2002), são as mais utilizadas pelos desenvolvedores

de dispositivos FPGAs. O VHDL ou Linguagem de Descrição de *Hardware* com ênfase em Circuitos Integrados de Altíssima Velocidade, tem sua sintaxe baseada na linguagem Ada, que foi desenvolvida pelo Departamento de Defesa dos Estados Unidos (DoD) em 1970, com o objetivo de padronizar a programação dos *hardware* utilizados na época, muitos destes dispositivos desenvolvidos em ASIC. Originalmente o VHDL foi desenvolvido sob o comando da Agência de Pesquisa Avançada de Defesa dos Estados Unidos (DARPA) com o intuito de substituir os complexos diagramas esquemáticos dos ASICs, e em 1987 o VHDL foi padronizado pelo Instituto Elétrico e Eletrônico de Engenharia (IEEE) (BRANDÃO; ROSÁRIO, 2010).

O Verilog HDL foi criado pela empresa *Gateway Design Automation* (GDA) em 1985 e em 1989 foi comprado pela companhia *Cadence Design Systems* (CDS) que tornou a linguagem de domínio público em maio de 1990 com a formação da *Open Verilog International* (OVI) também conhecido como projeto *Accellera*. Hoje o Verilog é padronizado pela IEEE como uma linguagem oficial de descrição de *hardware* (BRANDÃO; ROSÁRIO, 2010).

A estrutura de programação da linguagem VHDL mostrada na Figura 2.11, sugere como deve ser desenvolvido um sistema em toda sua concepção. Inicialmente é necessário a declaração das bibliotecas (*libraries*) que serão utilizadas pelo projeto. Em seguida a entidade (*entity*) define a interface do projeto, como as portas através dos pinos de entrada e saída e o tipo de sinal correspondente. Por fim é especificada a arquitetura (*architecture*) que define a lógica do circuito e que por sua vez pode ser decomposta em componente (*component*) e sinal (*signal*) que são declarações intermediárias opcionais, e a lógica (*logic*) que utiliza da combinação dos circuitos para gerar saídas esperadas. Na Figura 2.12 é apresentado de acordo com esta estrutura de programação, um exemplo da sintaxe utilizada pelo VHDL e o circuito resultante.

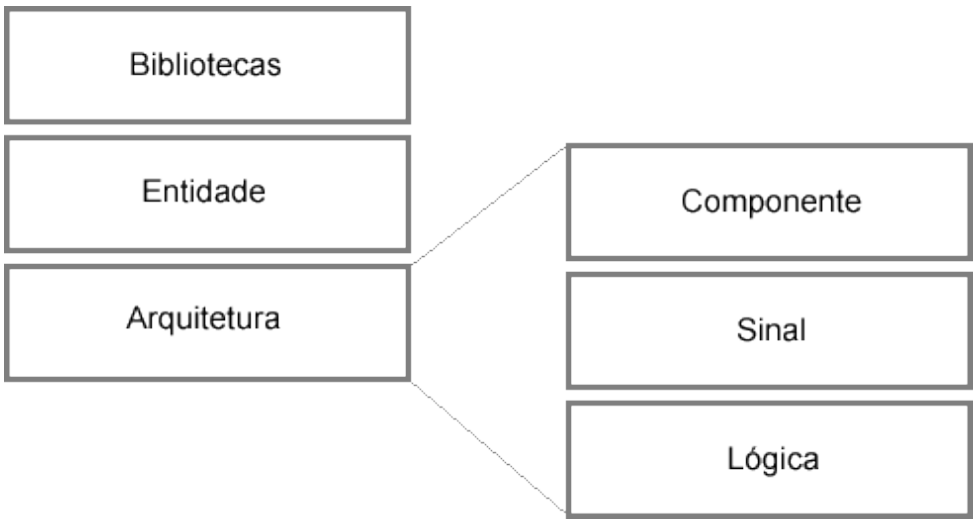

Figura 2.11: Estrutura de programação em VHDL

A linguagem VHDL apresenta algumas vantagens que a torna bastante utilizada entre os projetistas, como facilidade de atualização dos projetos permitindo diferentes formas de se implementar, verificação do comportamento do sistema digital através da simulação, redução do

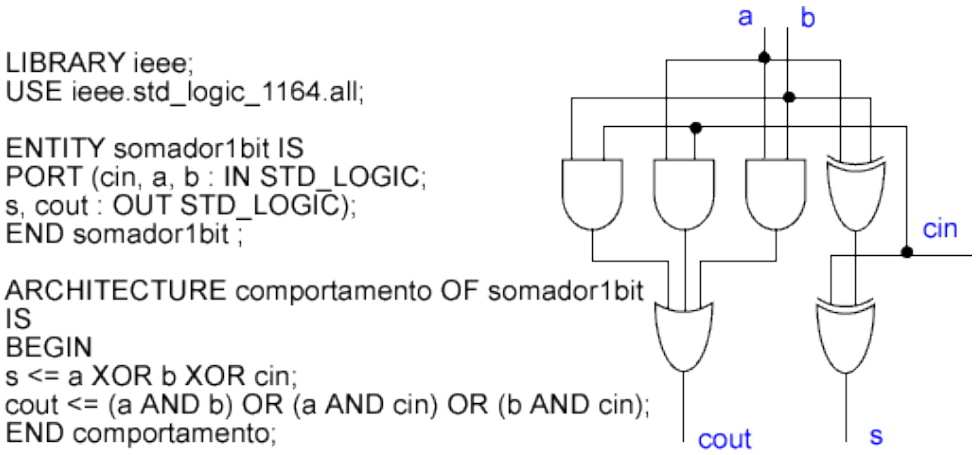

Figura 2.12: Exemplo de um somador completo em VHDL

tempo e custo do projeto e eliminação de erros de baixo nível. Porém existem grandes dificuldades para se otimizar o hardware gerado, além da necessidade de treinamento para se explorar os recursos que a linguagem oferece.

A linguagem Verilog, faz distinção entre o uso de maiúsculo e minúsculo, e os identificadores podem conter qualquer sequência de letras e dígitos, onde o primeiro caractere deve ser uma letra ou o símbolo "_". Outra característica da linguagem, é oferecer ao projetista os meios para se descrever um sistema digital de forma a obter vários níveis de abstração, possibilitando expressar ideias ou comportamentos, deixando os detalhes da programação para fases posteriores do projeto.

A estrutura de programação do Verilog é apresentada na Figura 2.13, e basicamente é a construção do módulo (*module*), ou seja, nesta declaração são informadas as entradas (*input*) e saídas (*output*) e a operação (*assign*) que se pretende realizar.

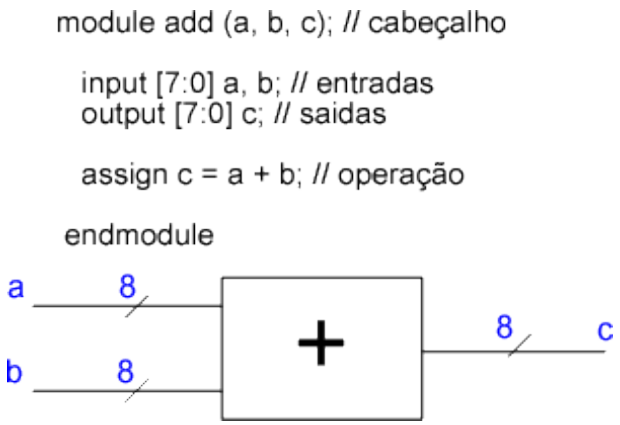

Figura 2.13: Estrutura de programação e exemplo de uma adição de dois números de 8 bits em Verilog

Não existem diferenças significativas quanto à capacidade de cada uma destas HDLs. A escolha de qual usar é muitas vezes baseada na preferência do projetistas, equipe de projeto ou

outras questões, tais como a disponibilidade de ferramentas e condições comerciais.

A grande vantagem de se utilizar as linguagens VHDL ou Verilog, é a possibilidade de serem utilizadas para simulação e síntese automática de circuitos descritos no nível da micro-arquitetura, por meio de ferramentas especializadas cada vez mais precisas em seus resultados. Contudo estas linguagens possuem uma curva de aprendizado muito acentuada, pois diferem bastante das linguagens de alto nível como C/C++, Java, entre outras, pelo fato de executarem toda sua lógica com base na temporização de um ou mais *clocks* de procedimento durante a execução do código, e não somente se preocupando com o fluxo do algoritmo descrito (BRANDÃO; ROSÁRIO, 2010).

Desta forma, é possível destacar dois principais problemas na programação de dispositivos FPGA que impedem sua utilização de forma mais abrangente. Um deles é a demanda de conhecimento específico não somente da arquitetura alvo, como de circuitos digitais e sua lógica combinatória. Outro problema que talvez possa ser derivado do primeiro é a dificuldade de programação destes dispositivos, isso considerando todo o processo de desenvolvimento que parte desde a especificação de um problema e proposta de solução computacional, até o processo de testes e manutenção.

Como alternativa à metodologia genérica de programação em dispositivos FPGAs, surgiram ferramentas que viabilizam a síntese destas linguagens de descrição de *hardware* por uma linguagem de mais alto nível, facilitando o uso por parte dos programadores de *software*. Deu-se o nome então a estas ferramentas de síntese de alto nível - ou *High-level synthesis* (HLS).

2.3.2 Abordagem à HLS

Com o crescente aumento da complexidade dos circuitos desenvolvidos em arquiteturas reconfiguráveis, a automação e otimização de níveis de abstração cada vez mais elevados se faz imprescindível. A Figura 2.14 mostra os diferentes níveis de abstração vistos em projetos de desenvolvimento de *hardware* customizado, e em destaque é apresentado o nível de algoritmo, que por sua vez é responsável pelos cálculos dos valores das saídas de acordo com as entradas, independente da forma de implementação, e o nível de transferência entre registradores (RTL) onde a estrutura do sistema é descrita como um circuito composto de unidades funcionais, elementos de memória e elementos de interconexão (MCFARLAND; PARKER; CAMPOSANO, 2002; COUSSY et al., 2009).

A Síntese de alto nível ou - *High-level synthesis* (HLS) nada mais é do que a tradução de uma descrição de circuito em alto nível para uma descrição em um nível inferior. Segundo McFarland, HLS foi definido como a "tradução de uma especificação abstrata do comportamento de um sistema digital no nível algorítmico para a estrutura no nível *register-transfer level* que implementa aquele comportamento" (MCFARLAND; PARKER; CAMPOSANO, 2002).

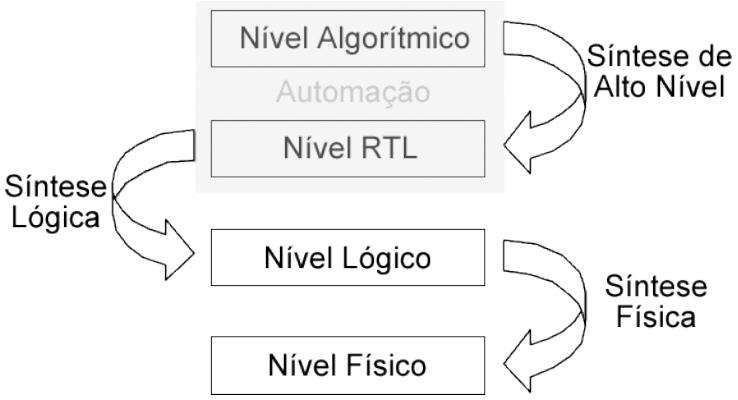

Figura 2.14: Níveis de abstração do projeto em computação reconfigurável

Segundo Martin e Smith (MARTIN; SMITH, 2009), HLS pode ser dividido em 3 gerações. Cronologicamente, a primeira geração iniciou em 1980 e parte da década de 90, a segunda começou em meados da década de 90 e do ano 2000, a terceira perdura até os dias de hoje. Por fim a quarta geração possivelmente dará sequência com a evolução da terceira.

Do ponto de vista técnico e do modelo de utilização, a primeira geração foi a era da pesquisa e de seus subprodutos específicos do domínio de *datapath*. A segunda geração foi o primeiro ano da comercialização do EDA (*Electronic Design Automation*), ferramentas de síntese comportamental dirigidas por linguagens de descrição de hardware, contudo foi um fracasso comercial na época. A terceira geração é caracterizada como a síntese de alto nível baseada na linguagem de programação C, que por sua vez é principalmente orientada a aplicações *datapath*. Por fim, o que se espera da quarta geração, é poder utilizar ferramentas multi-domínio baseadas em HLS que poderão proporcionar resultados mais satisfatórios tanto no projeto de *datapath* como em estruturas de controle (MARTIN; SMITH, 2009).

Geralmente o processo de síntese decompõe-se em vários subproblemas, tais como seleção de módulos (*module selection*), alocação (*allocation*), escalonamento (*scheduling*) e ligação (*binding*), como é apresentado na Figura 2.15. A seleção de módulos é responsável pela compilação da especificação. A alocação reserva os recursos de *hardware* (*functional units,storage components, buses*). O escalonamento determina os ciclos de *clock* das operações. A ligação atribui as operações para unidades funcionais, vincula as variáveis aos elementos de armazenamento e atribui as transferências aos barramentos. Por fim é gerada a arquitetura no nível RTL (COUSSY et al., 2009).

Como forma de exemplificar e mencionar alguns trabalhos que se relacionam e/ou assemelham em parte com o que se propôs este projeto de pesquisa, são apresentadas no Capítulo 3 algumas das principais ferramentas da literatura que realizam HLS.

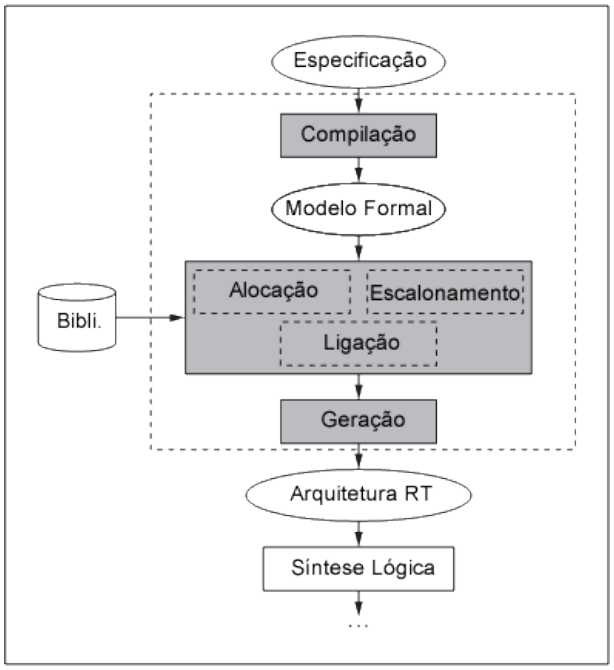

Figura 2.15: Etapas de projeto em HLS.
Adaptado de (COUSSY et al., 2009)

2.4 LALP

Muito se tem feito para o desenvolvimento de abordagens envolvendo síntese de alto nível visando diminuir os esforços no processo de programação dos dispositivos FPGA. Contudo, em muitos casos é evidente a deficiência dos compiladores para obter sistemas otimizados. Considerando estes aspectos foi desenvolvido por Menotti (MENOTTI et al., 2012; MENOTTI, 2010; MENOTTI et al., , 2009a, 2009b; MENOTTI; MARQUES; CARDOSO, 2007) uma nova linguagem que oferece a possibilidade de interferir no escalonamento das operações em termos de ciclos de *clock* sem a necessidade de se usar linguagens de baixo nível como VHDL e Verilog. O Objetivo desta linguagem é oferecer uma alternativa a mais no desenvolvimento de seções criticas de código (*loops*), em situações onde as ferramentas de alto nível se mostram incapazes de utilizar bem os recursos disponíveis ou mesmo atingir o desempenho desejado.

LALP (*Language for Aggressive Loop Pipelining*) é uma linguagem que facilita a descrição e geração de *hardware* de alto desempenho utilizando como base a técnica denominada ALP (*Aggressive Loop Pipelining*) (MENOTTI et al., 2012). Para isso foram utilizados contadores para cada repetição do código, esses contadores possuem sinais de controle que habilitam a execução das operações no correto ciclo de *clock*, como na execução preditiva presente em algumas arquiteturas. LALP foi desenvolvido utilizando uma sintaxe de operações lógicas e aritméticas de forma semelhante da linguagem de programação C e Java, mas com construtores simplificados para as instruções condicionais e de repetição. Já que o esforço usado para converter um programa escrito em C ou Java para LALP não deve ser maior do que o esforço usado para

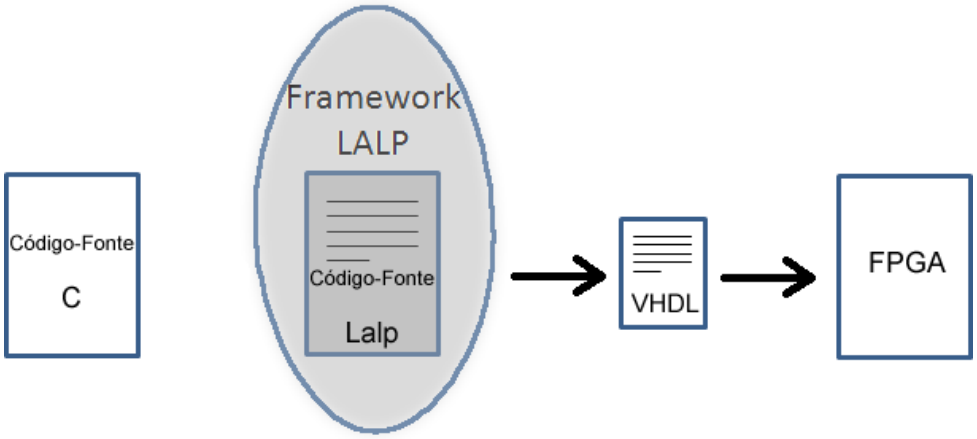

Figura 2.16: Contexto do *framework* e linguagem LALP

modicar o hardware gerado através de uma ferramenta capaz de interpretar diretamente estas linguagens.

É apresentado na Figura 2.16 o contexto geral da programação em dispositivos reconfiguráveis e mais especificadamente do *framework* e linguagem LALP. Nota-se que a ideia da linguagem LALP é aproximar-se da linguagem C, viabilizando uma maior utilização por parte dos programadores de linguagens de mais alto nível que as HDLs. Sendo assim, após a inserção da linguagem LALP ao *framework* base, este código é processado e transformado em uma linguagem de descrição de *hardware* VHDL. A partir desta linguagem gerada, é possível a sintetização por ferramentas fornecidas pelos fabricantes de FPGAs para gerar o *hardware* customizado e otimizado.

A forma geral de descrição de um programa escrito em LALP é apresentado na Listagem 2.1. Sendo inicialmente declaradas as constantes e tipos de dados definidos pelo usuário, logo a interface da entidade e atribuição dos pinos de entrada e saída e por fim são declaradas as variáveis em um bloco mais interno e as instruções do programa são listadas.

Listagem 2.1: Forma geral de um programa descrito em LALP. Fonte: (MENOTTI, 2010)

```
1 declaração de constantes
2 declaração de tipos
3 entidade (pinos de entrada/saída){
4     {
5         declaração de variáveis
6     }
7     instruções
8 }
```

Para demonstrar as características da linguagem LALP é apresentado na Listagem 2.2 um trecho de código simples escrito em C que calcula a soma do produto de dois vetores, e em

seguida na Listagem 2.3 é mostrado esta mesma soma descrita em LALP, sendo nas linhas 1 e 2 declaradas as constantes usadas para o número de bits de cada valor e para o número de iterações, respectivamente. Na linha 4 é definido um tipo de dado usado para 32 bits de ponto fixo com sinal e na linha 5 um tipo de um único bit para sinais de controle. A linha 7 inicia com um nome que será usado na criação da entidade em VHDL, seguido de sinais de entrada e saída desta entidade. O bloco que vai da linha 8 até a linha 12 contém as declarações das variáveis escalares e arranjos.

Listagem 2.2: Exemplo em C da soma do produto de dois vetores. Fonte: (MENOTTI, 2010)

```
1 #define N2048
2 int dotprod (){
3     int x[N],y[N];
4     int i,sum=0;
5     for(i=0;i<N;i++)
6         sum+=x[i]*y[i];
7     return sum;
8 }
```

Listagem 2.3: Exemplo em LALP da soma do produto de dois vetores. Fonte: (MENOTTI, 2010)

```
1  const DATA_WIDTH=32;
2  const N=2048;
3
4  typedef fixed(DATA_WIDTH,1) int;
5  typedef fixed(1,0) bit;
6
7  dotprod (out int sum,out bit done ,in bit init){
8      {
9          int x[N],y[N];
10         int acc;
11         fixed(16,0) i;
12     }
13     counter(i=0; i<N; i++@1);
14     i.clk_en=init;
15     x.address=i;
16     y.address=i;
17     acc+=x.data_out*y.data_out when i.step@1;
18     sum=acc;
19     done=i.done@2;
20 }
```

Na linha 13 é que as instruções propriamente ditas são iniciadas com o contador, sendo que a diretiva @1 indica que o componente irá gerar um novo valor para *i* a cada ciclo de *clock*. Na linha 14 o sinal externo de inicialização *init* é usado para habilitar a contagem. As linhas

seguintes indicam que o endereçamento dos vetores será determinado pela variável i. Estas instruções podem ser facilmente substituídas por uma macro nas formas $x[i]$ e $y[i]$. A linha 17 descreve as operações principais do código que devem aguardar um ciclo após o início da contagem para obtenção dos valores da memória. Por fim, a linha 18 indica que o sinal *sum* irá externar a soma dos valores e o sinal *done* do contador também será apresentado como um pino de saída da entidade, indicando o término das operações dois ciclos após o contador terminar. O compilador possui ainda uma diretiva, por meio de linha de comando, capaz de gerar saídas para todas as portas dos componentes, o que pode ser de grande utilidade para depuração do hardware gerado.

O escalonamento do *hardware* gerado a partir da Listagem 2.3 é apresentado na Figura 2.17, e sugere que cada interação necessita de 3 ciclos de *clock* para ser completada, mas como existem dependências entre elas, uma nova iteração é iniciada a cada ciclo.

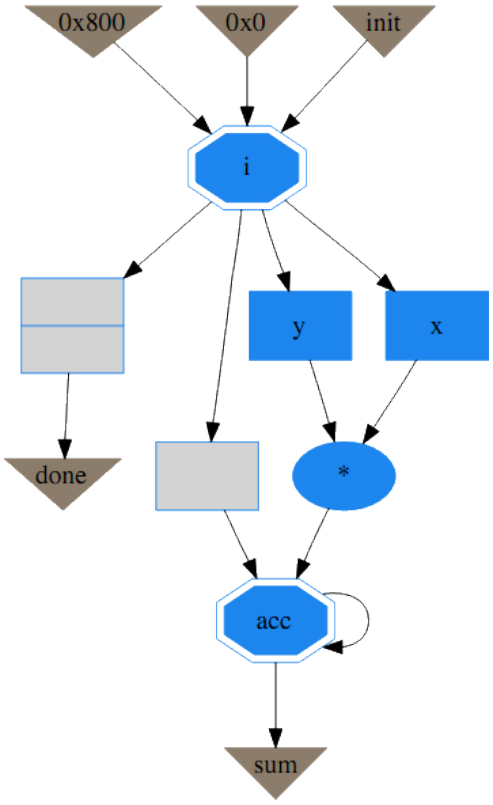

Figura 2.17: *Hardware* gerado a partir da Listagem 2.3.
Adaptado de (MENOTTI, 2010)

LALP assume em suas análises que as instruções sempre serão executadas de forma paralela salvo quando existem dependências, é por este motivo que se destaca das tradicionais linguagens de programação. O símbolo @ além de especificar o numero de ciclos de *clock*, ele tem a função de especificar se uma atribuição/operação será registrada e em quantos ciclos isso ocorrerá. Na Figura 2.18 são apresentados exemplos de como utilizar o operador @.

Dentre os diversos resultados obtidos com o uso do LALP, é exposto na Figura 2.19 o

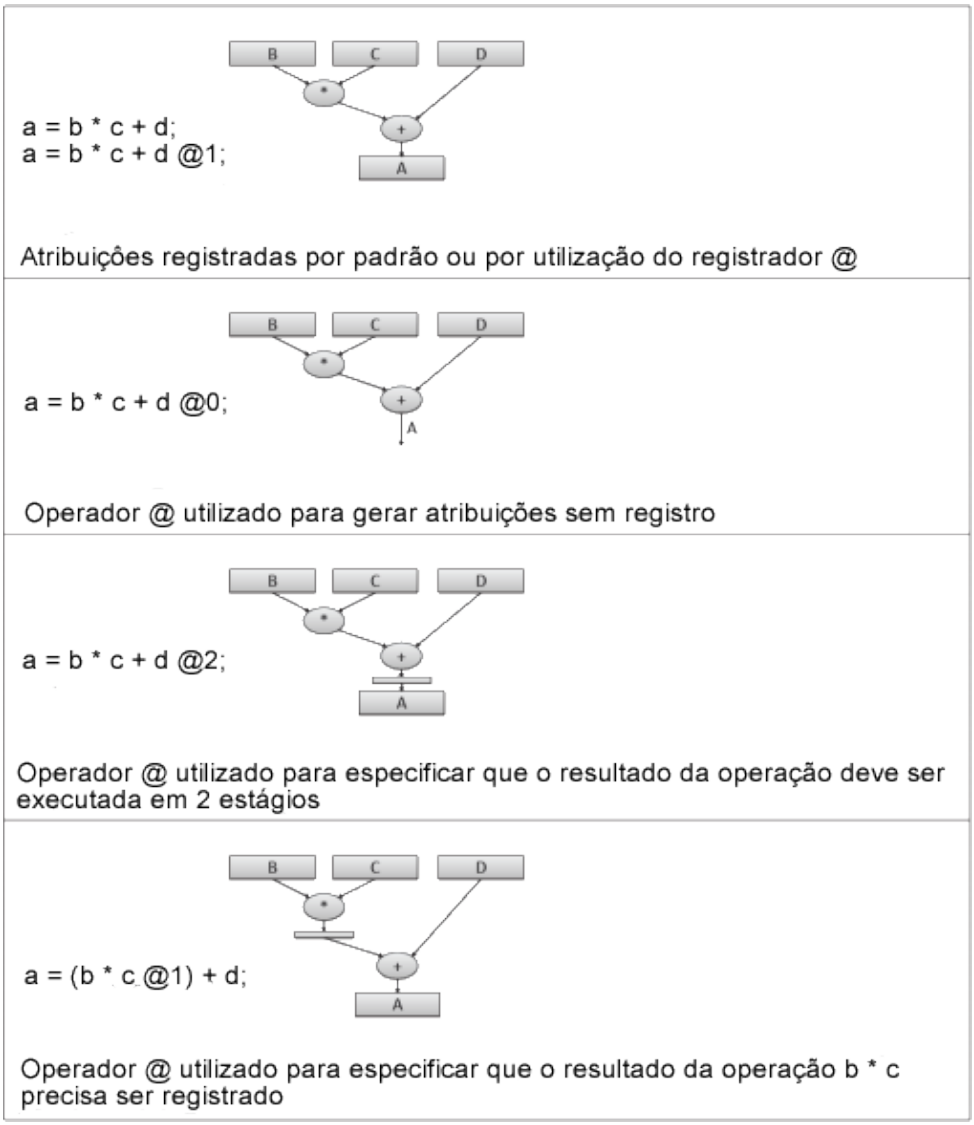

Figura 2.18: Exemplos do uso do operador @ na linguagem LALP.
Fonte: (MENOTTI, 2010)

resultado experimental do tempo de execução normalizado, comparando as arquiteturas obtidas com LALP às obtidas com as ferramentas ROCCC e C-to-Verilog (C2Verilog). O *speedup* relativo ao C-to-Verilog é em média 5,9 vezes maior e ao ROCCC 1,2 vezes maior. Embora as arquiteturas obtidas com LALP não ofereçam em todos os casos a maior frequência de operação, os ganhos em função do *throughput* permitem que se obtenha um tempo de execução menor (MENOTTI, 2010).

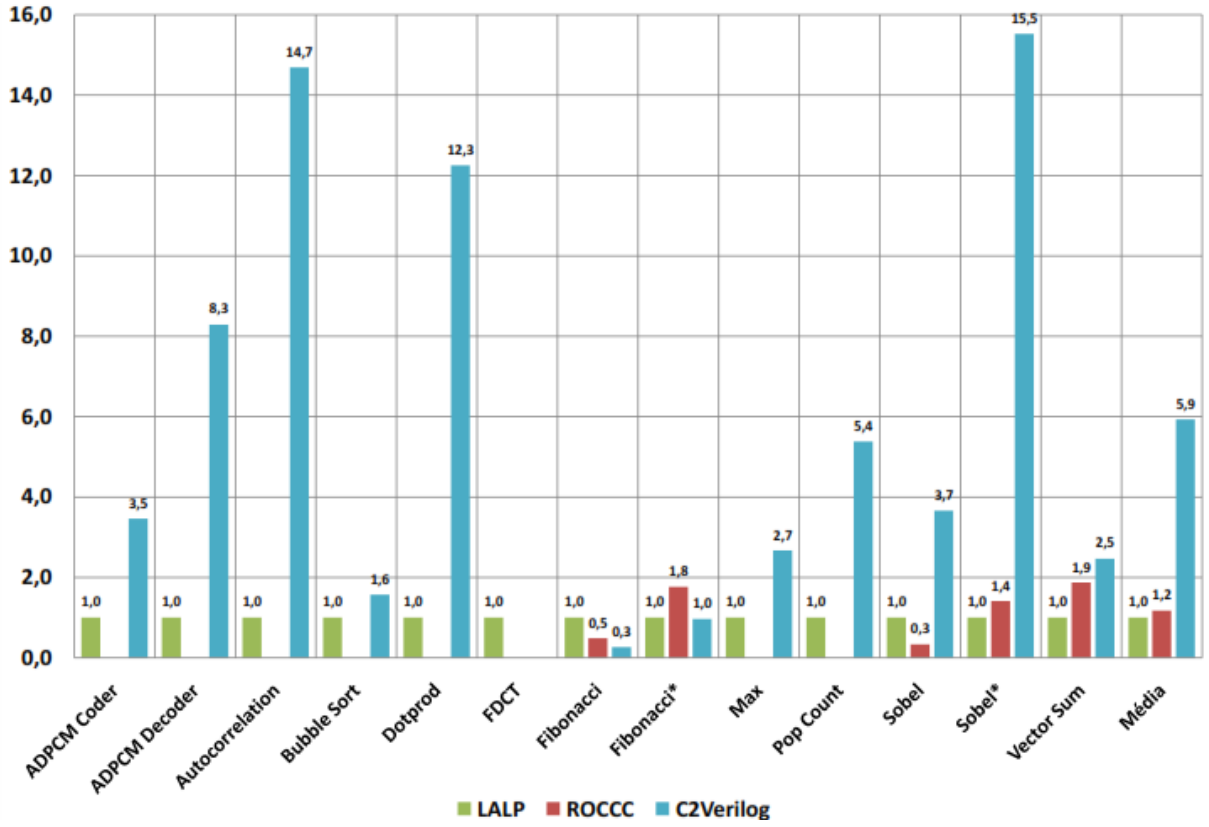

Figura 2.19: Tempo de execução normalizado comparando as ferramentas C-to-Verilog / ROCCC / LALP.

Fonte: (MENOTTI, 2010)

Vale ressaltar que a proposta do LALP é permitir uma programação de nível mais alto que VHDL/Verilog. Sendo esta programação a via para facilitar a geração de *loop pipelining* por parte do *framework*. Porém, o custo para programação em LALP é relativamente acentuado considerando que o conhecimento da linguagem se faz necessário para alcançar uma correta e precisa descrição de *hardware*.

Desta forma, é possível dizer que a principal dificuldade da programação em LALP é a necessidade do programador em incluir os @, que servem para guiar o *framework* na determinação das dependências de dados. Essa é a principal barreira para o uso do LALP e que constituiu a motivação para o EC-2 do presente trabalho. Ou seja, sabendo que os resultados obtidos a partir dos experimentos realizados pelo autor do *framework* LALP (MENOTTI et al., 2012) evidenciam uma maior eficiência da ferramenta em relação as demais na geração de *loop pipelining*. Este

trabalho propôs realizar um estudo para alavancar o uso do LALP, auxiliando o programador a determinar as dependências de dados que representavam a principal barreira do seu uso de forma mais abrangente.

2.4.1 Características do *Framework*

Para a determinação dos procedimentos a serem realizados para o EC-2 por este trabalho, se fez necessária a investigação do contexto onde serão aplicados, ou seja, as características da linguagem e *framework* LALP. Dessa forma, os procedimentos criados se fundamentaram em características da infraestrutura LALP que por sua vez é orientada pelas características dos dispositivos FPGAs.

As principais características da infraestrutura LALP e que conduziram a aplicação das modificações das análises e transformações do compilador Cetus no *framework*, são pontuadas nos itens a seguir:

- Recursos "ilimitados" de *hardware*, pois a princípio o número de registradores e unidades funcionais não são limitadores da arquitetura FPGA.
- LALP tem o foco em *loop pipelining*.
- Necessidade de escalonamento especial de instruções de acesso à memória. Sendo necessário aguardar dois ciclos de *clock* após seu endereçamento para que uma determinada instrução esteja disponível na memória e possa ser utilizada.
- Geração de única instrução para as estruturas condicionais *if-then-else*.
- Escalonamento respeitando as dependências de dados.

2.4.2 Pontos de Interesse do *framework* LALP para este Trabalho

Como forma de enfatizar o interesse deste trabalho em escolher o *framework* LALP como estudo de caso e fazendo um retrospecto ao Capítulo 1, são apresentados a seguir os pontos importantes que levaram a isso:

- Fato: O *Hardware* gerado a partir de LALP é de interesse da sociedade acadêmica, pelo bom desempenho e baixo custo.
- Problema: A programação em LALP apresenta barreiras que impedem seu uso mais abrangente por parte dos programadores destes dispositivos (FPGAs).

- Hipótese: É possível automatizar a geração de código LALP a partir de código escrito na linguagem de programação C?
- Objetivo: Criar métodos e ferramentas para responder positivamente a pergunta da hipótese.
- Metodologia
  - Buscar na literatura, conhecimento das melhores e difundidas técnicas de otimização utilizadas pelos compiladores.
  - Utilizar o compilador Cetus como base para o *parser* de programação em C e análise de dependências de dados.
  - Gerar novo código fonte na sintaxe da linguagem LALP (ou próximo).

2.5 Técnicas de Compilação e Otimização

O compilador consiste basicamente em um sistema de *software* capaz de transformar um programa fonte escrito em uma linguagem de programação de alto nível (próximas das linguagens naturais) em um programa alvo equivalente escrito em código de máquina ou de baixo nível (linguagens entendidas pelos computadores), como é apresentado na Figura 2.20.

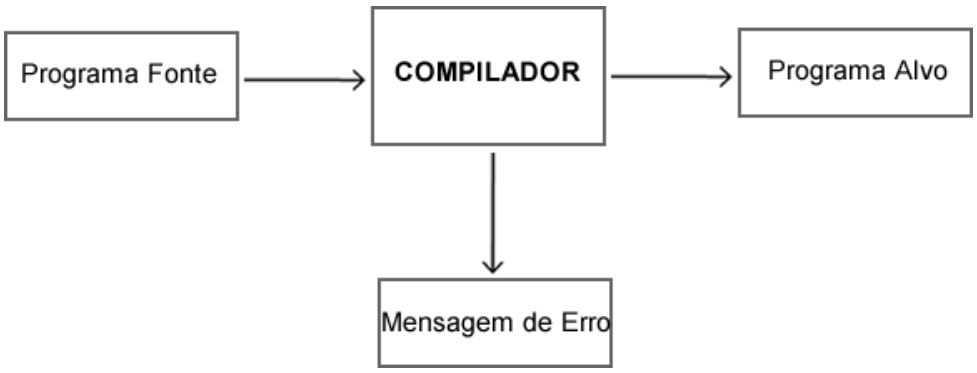

Figura 2.20: Processo executado pelo compilador.
Adaptado de (AHO, 2007)

A tradução de um código fonte de uma linguagem para outra ou mesmo de um código objeto de uma arquitetura para outra, também é um processo realizado pelo compilador. Para estes processos de transformações o compilador desempenha uma série de fases de análises e sínteses, de maneira que o processo de compilação pode ser decomposto nos seguintes passos (AHO, 2007):

1. **Análise Léxica:** É o processo de análise das linhas de caracteres, tal como, o código fonte e a reprodução de símbolos em pequenas sequências chamadas de símbolos léxicos

(*lexical tokens* ou *tokens*), facilitando sua manipulação através de um leitor de saída (*parser*). Ao analisarmos uma palavra, podemos definir se existe ou não algum caractere que não faz parte do alfabeto em questão, esta é a verificação desempenhada pelo analisador léxico do compilador.

2. **Análise Sintática:** Também conhecida como *parsing*, esta análise é responsável por processar os *tokens* e determinar a estrutura gramatical segundo uma gramática formal, gerando uma representação intermediária sequencial ou em árvore. A grande maioria do analisadores sintáticos implementados nos compiladores, aceitam alguma linguagem livre de contexto. Estes analisadores podem ser LL[13] e LR[14].

3. **Análise Semântica:** O analisador semântico preocupa-se com o significado ou semântica da representação intermediária que ele processa. Para isso, são criados métodos pelos quais as estruturas construídas pelo analisador sintático possam ser avaliadas ou executadas (WATSON, 1989).

4. **Geração de Código:** Transforma a árvore sintática em uma representação intermediária de um código de máquina equivalente, podendo estar na forma de um módulo re-alocável ou diretamente em um programa executável (AHO, 2007).

Fazem parte também destes componentes básicos de um compilador, a inclusão de uma tabela de símbolos e rotinas de acesso e uma interface para o sistema operacional. Interface esta, utilizada para estabelecer a conexão do usuário com o sistema, tal como, leitura e gravação de arquivos além de facilitar a portabilidade do compilador entre os sistemas operacionais. A Figura 2.21 mostra todo o fluxo básico enumerado anteriormente (AHO, 2007).

Uma nomenclatura bastante utilizada nos estudos dos compiladores é *frontend* e *backend*. Ambas denotam as fases descritas acima, sendo que nas fases iniciais do processo onde uma sequência de caracteres é recebida e logo gerada sua representação intermediária, da-se o nome de compilador *frontend*. Nas fases posteriores que recebem a representação intermediária e geram o código equivalente para a arquitetura alvo, dá-se o nome de compilador *backend*. Dispostos de forma modular, estas estruturas possibilitam a substituição do compilador *frontend* ou *backend* dependendo da linguagem de programação que se pretende utilizar ou mesmo da arquitetura alvo que este sistema será executado (AHO, 2007).

Apesar de toda complexidade, de maneira geral os compiladores apresentam as mesmas fases de análises. Contudo existem compiladores que acrescentam fases de otimização, pois

[13] É chamado de analisador sintático descendente (*top-down*), pois tenta deduzir as produções da gramática a partir do nó raiz. Lê-se a entrada de texto da esquerda para a direita, e produz uma derivação mais à esquerda (MARCUS, 1980)

[14] É chamado de analisador sintático de ascendente (*bottom-up*) pois ele tenta deduzir as produções da gramática a partir dos nós folha da árvore. Lê-se a entrada de texto da esquerda para a direita e produz uma derivação mais à direita (MARCUS, 1980; MENEZES, 2002)

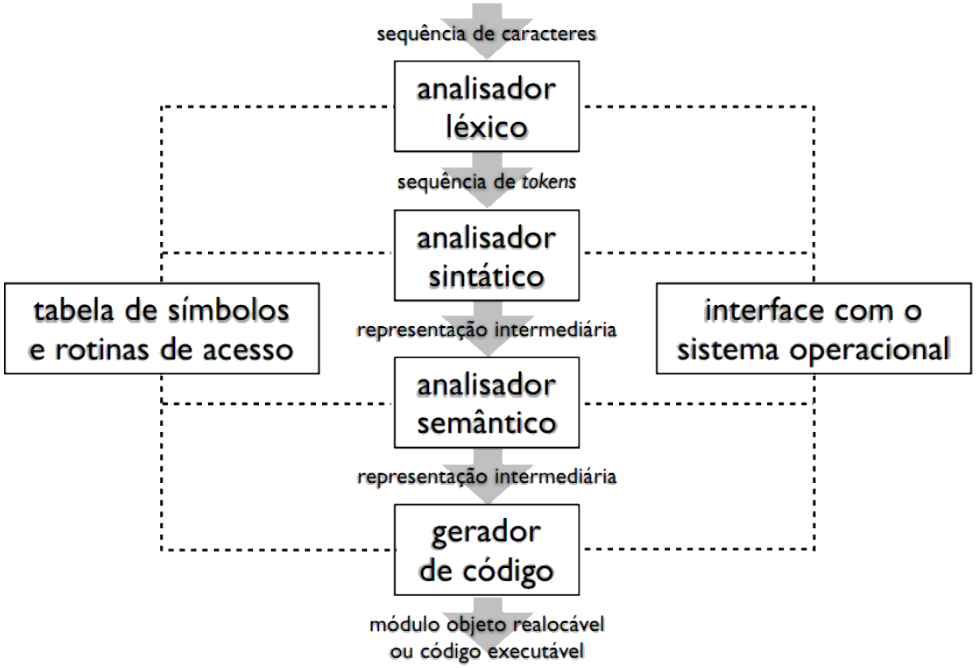

Figura 2.21: Estrutura básica de um compilador.
Adaptado de (WATSON, 1989)

os resultados obtidos pelos compiladores simples como citado anteriormente deixam muito a desejar, já que ao gerar códigos na sequência de cada análise as expressões são criadas uma após a outra sem considerar as instruções vizinhas, resultando em um programa sem nenhuma otimização. Por este motivo os compiladores otimizantes são mais bem equipados para tratar estas situações, melhorando os códigos na maioria dos casos.

2.5.1 Otimizações de Compilação

Para se alcançar a melhoria de desempenho desejada em um determinado sistema, várias técnicas de otimização podem ser utilizadas (MUCHNICK, 1997). Porém é importante salientar que é impossível criar um compilador otimizante que a partir de uma entrada P seja criado um programa P' equivalente, que é o melhor possível segundo os critérios considerados. Esta situação é explicada ao se estudar o "problema da parada"[15] em LFA (Linguagens Formais e Autômatos), pois um programa não pode obter informação suficiente sobre todas as possíveis formas de execução de outro programa (procedimento, máquina de Turing, etc.). Desta forma, o que se pode criar são programas que melhoram outros programas, tornando o programa P' na maioria das vezes, melhor do que o programa P original segundo o critério abordado (RANGEL, 2000).

As técnicas de otimização utilizadas pelos compiladores, são baseadas em um conjunto

[15]Determinar se um programa qualquer para ou não para. Dado um programa P e sua entrada de dados D, pode-se dizer ou não, se P para quando executado com a entrada de dados D

de heurísticas para detectar as sequências de código ineficiente e substituí-las por outras que removam estas situações. Apesar de existirem diversas técnicas de otimização de código o problema em questão não é apenas em como e quando utiliza-las, e sim se os resultados obtidos pelo programa otimizado não serão diferentes do programa original.

As otimizações podem ser classificadas nas seguintes categorias segundo Muchnick (MUCHNICK, 1997):

A: São otimizações geralmente aplicadas na representação intermediária de alto nível ou mesmo diretamente no código fonte, como sequência das instruções, repetições, formas de acesso aos arranjos etc.

B,C: São otimizações geralmente aplicadas à representação intermediária de nível médio ou baixo.

D: São otimizações geralmente aplicadas em código de baixo nível, sendo necessária informações do dispositivo de hardware.

E: São otimizações realizadas em tempo de ligação (*linking*) do código objeto.

Baseado na classificação segundo Muchnick, podemos exemplificar algumas propriedades das otimizações realizadas em A, ao considerarmos um trecho de programa em que aparece o comando $x = a + b$. Para otimizar este trecho de programa iremos retirar o comando respeitando a não alteração do funcionamento do programa, para isso algumas propriedades serão mostradas nas Listagens 2.4, 2.5 e 2.6 (RANGEL, 2000):

1. Quando o comando nunca é executado. Por exemplo, estando-o em seguida a um if cuja condição nunca é satisfeita:

Listagem 2.4: Comando nunca executado

```
1 if(0)
2 x=a+b;
```

2. Quando o comando é inútil. Por exemplo, x recebendo exatamente o mesmo valor atribuído anteriormente:

Listagem 2.5: Comando inútil

```
1 x=a+b;
2 x=a+b;
```

3. Quando o comando é inútil por nenhum comando executado posteriormente o utilizar. Por exemplo, o valor da variável *x* nunca será usado, pois após a saída da função a variável *x* não existirá mais.

Listagem 2.6: Comando inútil por não ser utilizado posteriormente

```
1 int f(int z) {
2 int x;
3 ...
4 x=a+b;
5 }
```

Apesar dos problemas destes exemplos serem simples e qualquer programador não muito experiente nota-los, é possível observar que não é prático otimizar um determinado programa tentando eliminar sucessivamente todos os comandos de uma única vez, sem que haja a preocupação com suas dependências em relação aos demais comandos do programa.

Para que se possa aplicar as transformações necessárias e posteriormente gerar o código objeto para a arquitetura alvo, serão apresentados alguns formatos importantes de representações intermediarias do código fonte de entrada ou simplesmente compiladores *frontend*, o qual tem o propósito de prover uma estrutura de dados simples que possibilite a aplicação destas transformações.

- **Código de três endereços:** Tem por objetivo criar uma representação que forneça uma visão simplificada do programa, ou seja, são inseridas quando necessário variáveis temporárias convertendo as operações em dois operandos e um resultado. Na Listagem 2.7 é apresentado um trecho de código com operações algébricas e logo a tradução em código de três endereços é mostrada na Listagem 2.8.

Listagem 2.7: Código com operações algébricas

```
1 a=x*y+z-w
```

Listagem 2.8: Tradução em código de três endereços

```
1 t1=x*y
2 t2=z-w
3 a=t1+t2
```

O mapeamento de estruturas diretamente para unidades funcionais da arquitetura alvo de dois operandos e um resultado, também pode ser realizado utilizando a representação de

código de três endereços. Isso faz com que o processo de decomposição em primitivas suportadas pela arquitetura esteja praticamente realizado, pois parte deste processo está na transformação do código para o formato desta arquitetura.

- **Eliminação de sub-expressões comuns:** Quando uma mesma expressão ocorre mais de uma vez em um mesmo trecho de programa, ou seja, quando as variáveis em uma dada expressão não tem seus valores modificados entre as duas ocorrências, é possível então realizar apenas uma vez os cálculos. A seguir é apresentado na Listagem 2.9 um trecho de código com operações algébricas que por sua vez possibilita a otimização por eliminação de sub-expressões comuns de duas formas. Sendo possível guardando o valor da expressão $a+b$ em uma variável temporária $t1$, como mostra a Listagem 2.10 ou mesmo em x se ainda estiver disponível, como mostra a Listagem 2.11.

Listagem 2.9: Trecho de código com operações algébricas similares

```
1 x=a+b;
2 ...
3 y=a+b;
```

Listagem 2.10: Eliminação de sub-expressões (1)

```
1 t1=a+b;
2 x=t1;
3 ...
4 y=t1;
```

Listagem 2.11: Eliminação de sub-expressões (2)

```
1 x=a+b;
2 ...
3 y=x ;
```

Este tipo de eliminação pode se deparar com situações um pouco mais complexas das consideradas anteriormente. Por exemplo, se uma das variáveis é a referência de um *array* $a[i]$, qualquer modificação de valor do mesmo *array* $a[j]$ entre as duas expressões, não permite garantir que o valor de $a[i]$ não se alterou. Isto pois não é possível garantir que $i \neq j$. A mesma situação acontece com atribuição de valores a ponteiros e atribuição de valores a chamadas de funções, podendo ter reações imprevisíveis (RANGEL, 2000).

- **Eliminação de código morto:** É o código que não pode ser alcançado durante a execução de um programa, ou seja, é o código que não é executado no programa. Por exemplo, ao definir uma instrução ao término de uma função como *return*, Listagem 2.12. Ao ocorre

após uma condição if nunca satisfeita, Listagem 2.13. Ou mesmo após um comando de desvio *goto*, que não possibilita o retorno para execução do código, Listagem 2.14.

Listagem 2.12: Eliminação de instrução ao término de uma função

```
1 int funcao(int x){
2 return x++; /* Código morto */
3 }
```

Listagem 2.13: Eliminação após uma condição nunca satisfeita

```
1 #define X 0
2 ...
3 if(X) {
4 ... /* Código morto */
5 }
```

Listagem 2.14: Eliminação após um comando de desvio

```
1 goto x;
2 i=5; /* Código morto */
3 ...
4 x: ...
```

A incidência de código morto nem sempre é ocasionada pelo programador, existem outras situações que geram este tipo de código, como é o caso da otimização de um programa que por sua vez sofre algumas transformações e em um estágio intermediário ocasiona a existência de código não alcançável (RANGEL, 2000).

- **Renomeação de variáveis temporárias:** As variáveis temporárias criadas durante a geração de código intermediário posteriormente podem não ser necessárias, desta forma são eliminadas dando outros nomes para as variáveis que guardarão os valores temporários. Diversas variáveis desnecessárias podem ser criadas na eliminação de sub-expressões, na Listagem 2.15 a seguir é apresentado o código fonte, na Listagem 2.16 o código intermediário, na Listagem 2.17 a eliminação das duas ultimas cópias de $a+b$ e na Listagem 2.18 o resultado de todo o processo (RANGEL, 2000).

Listagem 2.15: Código fonte

```
1 x=a+b;
2 y=(a+b)*c;
3 z=d+(a+b);
```

Listagem 2.16: Código intermediário

```
1 t1=a+b;
2 x=t1;
3 t2=a+b;
4 t3=t2*c;
5 y=t3;
6 t4=a+b;
7 t5=d+t4;
8 z=t5;
```

Listagem 2.17: Eliminação das ultimas cópias de $a+b$

```
1 t1=a+b;
2 x=t1;
3 t2=t1;
4 t3=t2*c;
5 y=t3;
6 t4=t1;
7 t5=d+t4;
8 z=t5;
```

Listagem 2.18: Resultado da renomeação de variáveis temporárias

```
1 x=a+b;
2 y=x*c;
3 z=d+x;
```

- **Transformações algébricas:** São transformações baseadas em propriedades algébricas, como a comutação, associação, identidade entre outros. Porém estas transformações de forma automática e aleatória podem gerar problemas nas convergências ou arredondamentos. Desta forma estes processos devem ser realizados apenas a com autorização explícita do usuário, como o uso de parênteses que indicada explicitamente a precedência das operações, evitando assim possíveis inconsistências. A soma comutativa nos possibilita transformar $x = a + b * c$, como mostrado na Listagem 2.19 em $x = b * c + a$, como mostrado na Listagem 2.20. Esta transformação pode reduzir o código dispensando a criação de uma variável temporária $t1$ e as instruções que a manipulam (RANGEL, 2000).

Listagem 2.19: Código algébrico $x = a + b * c$

```
1 Load b
2 Mult c
3 Store t1
4 Load a
5 Add t1
6 Store x
```

Listagem 2.20: Transformação algébrica comutativa $x = b * c + a$

```
1 Load b
2 Mult c
3 Add a
4 Store x
```

Outra transformação algébrica semelhante pode ser baseada na associação, possibilitando a troca de $x = (a+b)+(c+d)$, Listagem 2.21 por $x = (((a+b)+c)+d)$, Listagem 2.22. Esta transformação pode reduzir o código dispensando o uso das variáveis temporárias $t1$ e $t2$ e as instruções que as manipulam (RANGEL, 2000).

Listagem 2.21: Código algébrico $x = (a+b)+(c+d)$

```
1 Load a
2 Add b
3 Store t1
4 Load c
5 Add d
6 Store t2
7 Load t1
8 Add t2
9 Store x
```

Listagem 2.22: Transformação algébrica associativa

```
1 Load a
2 Add b
3 Add c
4 Add d
5 Store x
```

- **Dobramento de constantes:** É a avaliação em tempo de compilação das expressões ou sub-expressões compostas de valores constantes. Caso a arquitetura alvo não seja

a mesma do processo de compilação, esta avaliação é realizada uma vez logo no início da execução do programa quando estiver na arquitetura que foi projetado. Na Listagem 2.23 é apresentado um trecho de código que realiza o calculo repetidamente do valor de $N-2$, sendo que este valor poderia ser pré-calculado e substituído por 98 (RANGEL, 2000).

Listagem 2.23: Exemplo de onde pode ser utilizado o dobramento de constante

```
1 #define N 100
2 ...
3 while (i<N−2) {
4     ...
5 }
```

- **Redução de força:** É a substituição de expressões que possuem um custo elevado de processamento, por expressões com custos menores. Por exemplo, ao utilizarmos a função $pow(x,2)$ do C++, o código deverá calcular o logaritmo natural e a exponencial e^{2*lnx}. Para reduzirmos este esforço, poderíamos chegar ao mesmo resultado multiplicando $x*x$ (RANGEL, 2000).

2.5.2 Suporte para Otimizações Avançadas

As principais técnicas de otimizações possíveis de serem realizadas durante o processo de compilação como citado anteriormente, são as que atuam em alocação de registradores e escalonamento de instruções, e não mais importante mas que desprendem um maior esforço no aprimoramento das pesquisas, são as otimizações que atuam em estruturas de repetições (*loops*), pois geralmente é o trecho de código que consome maior processamento e por sua vez é onde se pode ter o melhor desempenho utilizando técnicas que melhor se aplicam a situação em análise (MUCHNICK, 1997).

Até o momento foi possível perceber que o fluxo de controle de um programa é essencial para a realização de otimizações no código, mas para o entendimento das técnicas utilizadas para extrair o máximo de desempenho de uma estrutura de repetição, é necessário introduzir conceitos básicos como fluxo de controle, fluxo de dados e suas dependências.

Para a representação do fluxo de controle, é utilizada a criação de blocos básicos[16] de instruções. A identificação destes blocos em um programa é realizada localizando instruções que podem alterar o fluxo de execução, como instruções de desvio (*goto*), chamadas de procedimentos (*call*), retorno (*return*) e rótulos (*labels*).

A representação de todas as possibilidades de fluxo de controle entre os blocos básicos

[16]Trecho de código executado sempre sequencialmente

de um programa é dada por uma estrutura de dados organizada em forma de grafo direcionado, chamada de grafo de fluxo de controle CFG. Os nós representam blocos básicos e os arcos representam precedência ou transferência de controle. Na Figura 2.22 é mostrada uma representação básica dos CFGs de algumas estruturas típicas de computação. Em seguida uma representação estrutural é exemplificada, sendo que na Figura 2.23 é mostrado um programa com apenas um caminho, na Figura 2.24 um programa com dois caminhos e por fim a Figura 2.25 apresenta a estrutura de um *loop*.

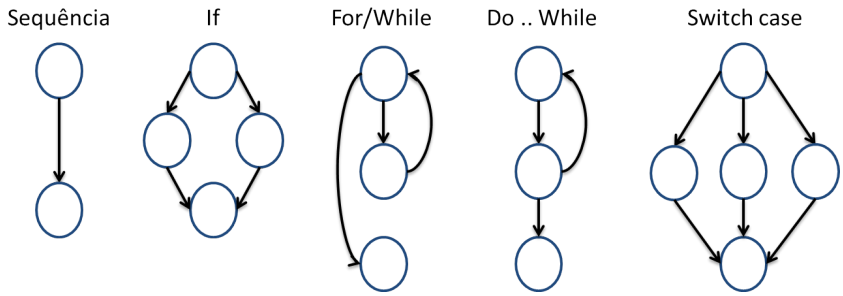

Figura 2.22: Representação funcional básica dos CFGs

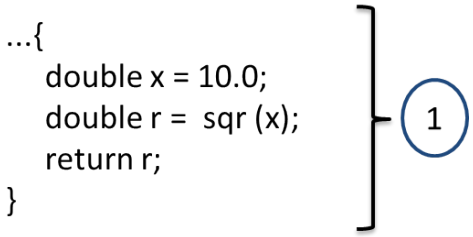

Figura 2.23: Representação estrutural de um CFG com um caminho

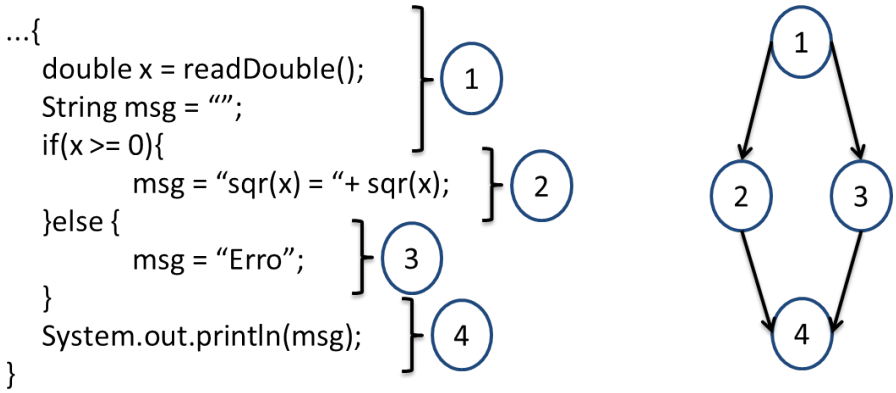

Figura 2.24: Representação estrutural de um CFG com dois caminhos

O fluxo dos dados de um programa e suas dependências, são imprescindíveis para a realização de otimizações. O cálculo destas dependências de dados é realizada através da análise do fluxo dos dados DFA. O resultado da DFA é o grafo de fluxo de dados DFG direcionado e acíclico. Sendo cada nó a representação de uma entrada primária, uma operação ou uma saída e

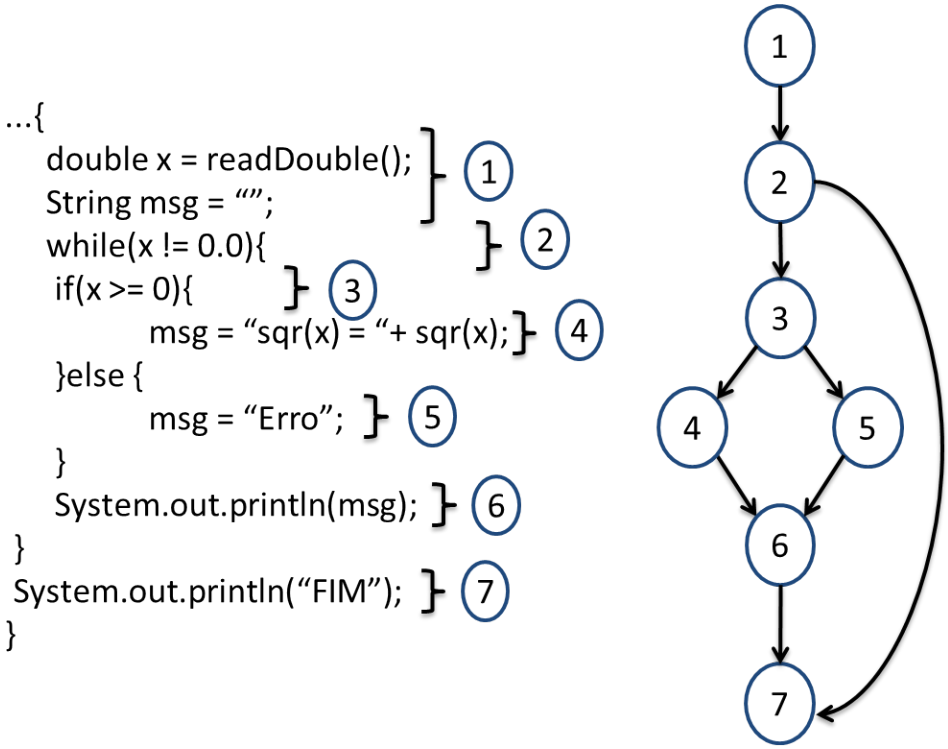

Figura 2.25: Representação estrutural de um CFG de um *loop*

cada aresta uma representação de que o valor definido pela operação(i) é usado pela operação(j). Essas dependências podem ser classificadas pelas seguintes situações (PATTERSON; HENNESSY, 2009):

- **Dependência de fluxo (*read-after-write*):** Quando a operação(j) lê o resultado escrito pela operação(i).
- **Antidependência (*write-after-read*):** Quando a operação(j) escreve em uma variável após ela ter sido lida pela operação(i).
- **Dependência de saída (*write-after-write*):** Quando a operação(j) escreve a mesma variável escrita pela operação(i).
- **Dependência de entrada (*read-after-read*):** Quando a operação(j) lê a mesma variável lida pela operação(i).

Na Listagem 2.24 é apresentado um bloco básico de instruções, e em seguida a representação deste bloco em um DFG é mostrado na Figura 2.26.

Listagem 2.24: Bloco básico de instruções

```
1 t1=a+b
2 t2=t1*c
3 t3=d*e
4 x=t2-t3
```

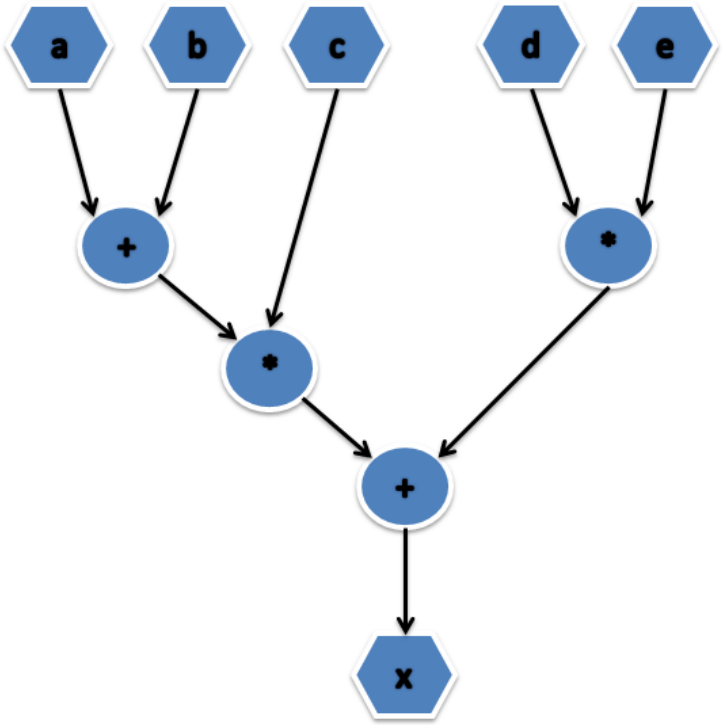

Figura 2.26: Representação de um bloco básico em DFG

As principais técnicas de otimização utilizadas em estruturas de repetição são conhecidas como *loop pipelining* e *loop unrolling* (HENNESSY; PATTERSON; GOLDBERG, 2003).

2.5.2.1 *Loop Pipelining*

A técnica de *loop pipelining* também conhecida como *software pipelining* (ALLAN et al., 1995; GOOSSENS; VANDEWALLE; MAN, 2006), é uma técnica que tem como objetivo paralelizar instruções de iterações diferentes se preocupando com as dependências de dados e a utilização de recursos. O bloco básico de instruções mostrado na Listagem 2.25, apresenta um *loop* que compreende 5 instruções que serão executadas em 10 iterações. Sendo $i++$ a instrução 1, a leitura na memória do valor $B[i]$ a instrução 2, a leitura na memória do valor $C[i]$ a instrução 3, a soma dos registradores das instruções 2 e 3 é armazenada na instrução 4 e o resultado desta soma é atribuída à posição de memória $A[i]$ que corresponde a instrução 5.

Listagem 2.25: Exemplo de *loop*

```
1 for(i=0;i<10;i++)
2       A[i]=B[i]+C[i];
```

Assumindo que todas as instruções conseguem ser executadas em um ciclo de *clock*, o modo de execução sequencial exigirá um total de 5 ciclos por iteração. Por outro lado se forem desprezados os conflitos de recursos, é possível executar uma iteração por ciclo utilizando a técnica de *loop pipelining*, como é apresentado na Figura 2.27.

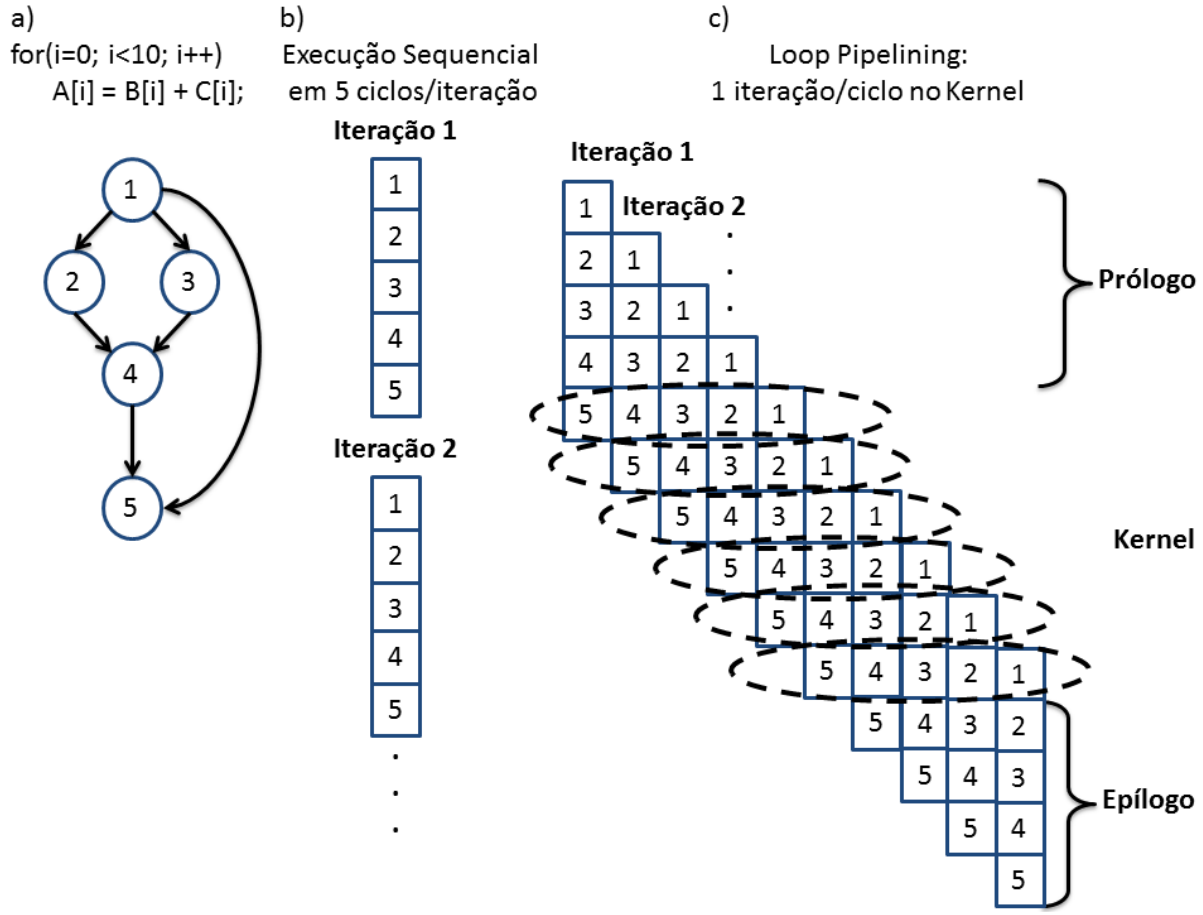

Figura 2.27: Exemplo de iteração sequencial e iteração em *loop pipelining*

2.5.2.2 *Loop Unrolling*

A transformação baseada no desenrolamento de repetições (*loop unrolling*), visa substituir o corpo de uma repetição por réplicas deste mesmo corpo, realizando as alterações necessárias para não afetar o correto funcionamento do *loop*. Considerando o bloco básico representado na Listagem 2.25, este será desenrolado em fator de 2 como mostra a Listagem 2.26, isso que dizer que, em uma única passagem pela estrutura de repetição 2 iterações serão realizadas

simultaneamente. Na Figura 2.28 é apresentada a execução de 2 iterações por ciclo de *clock* no *kernel*.

Listagem 2.26: Exemplo de *loop unrolling*.

```
1 for(i=0;i<5;i++){
2       A[i]=B[i]+C[i];
3       A[i+5]=B[i+5]+C[i+5];
4 }
```

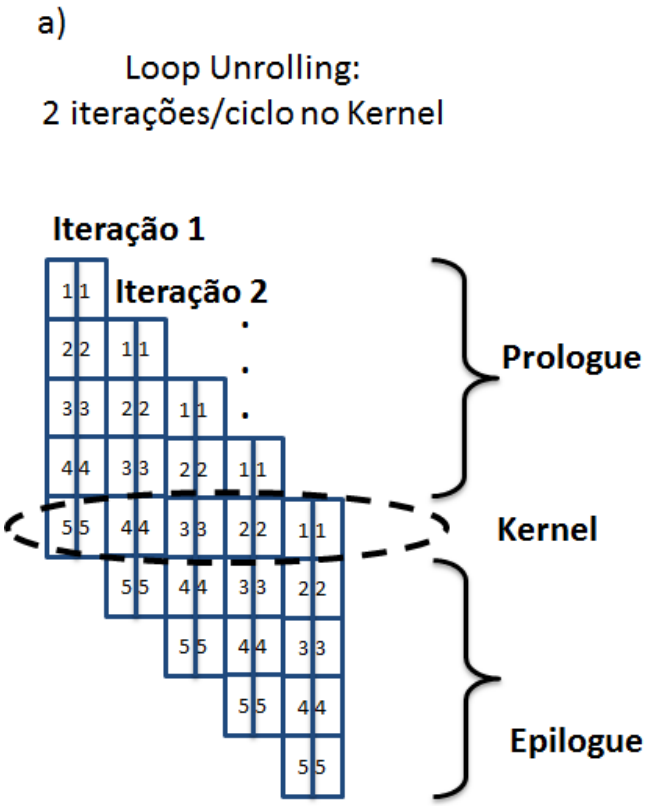

Figura 2.28: Exemplo de código utilizando a técnica de *loop unrolling*.

Para a exploração do *loop pipelining* os algoritmos geralmente seguem duas abordagens, sendo: o escalonamento módulo, proposto por Rau (RAU, 1994) e a identificação de *kernel*, proposto por Aiken e Nicolau (AIKEN; NICOLAU, 1988). O escalonamento módulo, procura um *kernel* escalonável com base nas restrições de recursos e de dependências, e buscam melhorar-lo reduzindo o intervalo de iniciação[17]. A identificação de *kernel*, desenrola completamente a repetição e busca por padrões de execução para construir um novo *kernel*.

As técnicas de *software pipelining* podem ser utilizadas visando a melhoria de desempenho de uma aplicação, contudo existem custos que devem ser ponderados. Ao serem executadas em um processador com recursos especiais utilizando um compilador, obtêm-se menos eficiência e

[17] É a quantidade de ciclos de *clock* entre a primeira instrução de iterações do *kernel*

mais facilidade de uso para o programador. Ao serem executadas em um *hardware* customizado utilizando uma HDL, obtêm-se mais eficiência, mas com um grau de complexidade elevado para o programador. Já com a execução de *software pipelining* em um *hardware* customizado, utilizando uma linguagem de alto nível e recursos avançados de compilação, é possível obter eficiência e facilidade de uso pelo programador.

É portanto a motivação da utilização destas técnicas em arquiteturas reconfiguráveis, pois possuem uma maior disponibilidade de recursos de *hardware* e flexibilidade que um microprocessador pode oferecer. Dessa forma, as arquiteturas reconfiguráveis apresentam maior potencial para a exploração de *software pipelining*.

2.6 CETUS

O Projeto Cetus (LEE; JOHNSON; EIGENMANN, 2004; DAVE et al., 2009) foi iniciado por estudantes de graduação da Universidade de Purdue (*Purdue University*) nos EUA a quase 8 anos, e conquistou o apoio da NSF (*National Science Foundation*). Cetus é uma infraestrutura de compilador desenvolvida em Java para a transformação de código fonte para código fonte (*source-to-source*) na linguagem de programação C. Esta infraestrutura tem como objetivo a transformação de código fonte C em um novo código fonte C paralelizado, ou seja, o compilador Cetus objetiva a geração automática de código C paralelizado e otimizado a partir de um código fonte C serial (arquitetura convencional). Dessa forma, fazendo uso de um conjunto de análises e transformações que visam a paralelização e otimização automática do código fonte.

Cetus pode se assemelhar a três infraestruturas de compilador que tem objetivos próximos ao projeto, estes compiladores são o GCC, Polaris e SUIF. A seguir são descritos os motivos da não utilização destas infraestruturas como base para o Cetus e as características mais importantes que podem ser acopladas ao projeto (LEE; JOHNSON; EIGENMANN, 2004).

- **GCC:** É umas das infraestruturas de compilador mais robusta disponível para a comunidade de pesquisa, pois é capaz de gerar código altamente otimizado para uma variedade de arquiteturas, sua distribuição é aberta (*open-source*) e continuamente atualizada, além de possuir farta documentação para estudo. Contudo a curva de aprendizado é muito íngreme para a realização de modificações em sua estrutura, sendo desprendido grande investimento de tempo para implementar transformações mesmo triviais.

  O GCC não foi projetado para as transformações *source-to-source*, pois a maioria de suas fases operam sobre o nível inferior da representação RTL. A compilação é feita a partir de um arquivo de origem por vez, para depois ser realizada a análise de forma separada dos procedimentos. A partir da versão 3.0 o GCC incluiu uma representação da arvore de sintaxe, que por sua vez tem sido utilizada pelo projeto de classes para realizar uma série

de fases do compilador Cetus.

Alguns problemas foram enfrentados pelos integrantes do projeto Cetus, pois o GCC não fornece uma API de fácil entendimento, ela consiste basicamente de macros. Faltam operações em agrupamentos lógicos como, classes e *namespaces*, como seria de se esperar de um compilador desenvolvido em uma linguagem orientada a objetos. Estas dificuldades foram consideradas as principais responsáveis para os integrantes do projeto Cetus optarem por não trabalhar em cima de modificações nos processos do GCC e sim criar um novo projeto de compilador gastando menos tempo e obtendo mais eficiência.

- **Polaris:** É um compilador desenvolvido em C++ que opera no programa Fortran 77. O Polaris foi escrito antes da *Standard Template Library* (STL C++) se tornar disponível à comunidade, por isso suas implementações são inclusas no próprio programa. O Polaris usa um dialeto pré-ISO do C++, hoje incomum para os programadores, além de apresentar muitos avisos e até alguns erros que limitam sua portabilidade para outras plataformas.

  De maneira geral, o Polaris é o representante dos compiladores projetados para uma determinada linguagem, servindo bem ao seu propósito mediante dificuldades em seu entendimento. O projeto de compilador Cetus não deve ser pensado como o "Polaris para C", pois é projetado para evitar esses problemas. No entanto, ainda existem vários recursos do compilador Polaris que poderiam ser adotados ao Cetus. A representação intermediária (IR) do Polaris pode ser impressa no formato de código, se assemelhando ao programa de origem. Esta propriedade faz com que seja fácil para o usuário rever e compreender as etapas envolvidas nas transformações geradas pelo Polaris. Além disso, a API do Polaris consiste em uma IR por estado após cada chamada, assim os erros mais comuns podem ser evitados.

- **SUIF:** É a junção dos compiladores *National Compiler Infrastructure* (NCI) e o Zephyr. O SUIF foi desenvolvido em C++ como o Polaris, e sua versão mais atualizada suporta análises de programas em C. Foram lançadas duas distribuições do compilador, sendo o SUIF-1 um compilador de paralelização e o SUIF-2 que realiza análises de performance inter procedural (HALL et al., 1996).

  O Cetus e o SUIF são classificados como compiladores extensíveis *source-to-source*, porém duas razões principais eliminaram a utilização do SUIF-1 para a implementação da infraestrutura do Cetus, são elas: O projeto SUIF-1 não está mais ativo e a última atualização aconteceu em 2001. Embora entende-se que o SUIF-1 suporte múltiplas linguagens, não foi encontrada documentação a não ser para a linguagem C e uma versão antiga para o Java. Foram iniciados os trabalhos em *frontends* para Fortran e C++, mas não estão disponíveis na versão atual. Desta forma o SUIF suporta essencialmente uma única linguagem.

Por fim, visto todos os aspectos citados, os autores do projeto Cetus optaram por utilizar o Java como linguagem para a implementação do compilador. Pois, a linguagem de programação oferece diversos recursos que favorecem à boa engenharia de software, como suporte a uma boa depuração, a uma alta portabilidade, ao *garbage collection* (GC)[18], além de possuir um sistema de documentação automático. Dessa forma, viabilizando a construção de toda infraestrutura de compilador do projeto Cetus.

O compilador Cetus é um conjunto de classes com objetivo de extrair a IR para possibilitar as transformações a fins de otimização. Atualmente inclui um *parser* do C escrito em ANTLR[19]. Dessa forma, para o completo funcionamento do compilador é necessário o pacote de classes inclusas no projeto Cetus (escritas em Java) e o ANTLR v2. Para trabalhar com a linguagem de programação C++ é necessário um analisador externo que possa produzir árvores e analisar a saída formatada como arquivo graphviz[20], pois até o momento o compilador não suporta esta linguagem de programação.

2.6.1 Análises e Transformações

A partir da inserção de um código fonte em sua forma serial no compilador Cetus, diversas análises e transformações são realizadas a fim de modificar e adequar este código para sua execução de forma paralela, obtendo assim desempenho relativamente melhor. Isto é possível por ter uma rica análise de dependências de dados e geração de grafos que viabilizam estes processos. Faz-se necessário ressaltar que estas modificações realizadas pelo compilador, visam a obtenção de melhores resultados para a mesma plataforma do código de entrada[21].

Como o objetivo deste trabalho é implementar aceleradores para auxiliar a programação em uma plataforma diferente para o qual se prestam os objetivos do Cetus. Este trabalho se ateve a identificar e extrair somente as informações necessárias para serem manipuladas a posteriori, com o intuito de facilitar a programação em FPGA.

A seguir é apresentada uma sequência de passos executados pelo Cetus, visando a otimização e transformação do código fonte. Sendo "passo" a nomenclatura convencional adotada por compiladores para designar uma determinada etapa de análise ou transformação, e portanto também é adotada pelo Cetus e por este trabalho.

A sequência apresentada respeita a ordem de execução do compilador a partir da inserção de parâmetros de configuração de entrada, que por sua vez, foram escolhidos após diversas

[18] É uma forma de gerenciamento automático de memória, caso especial de gestão de recursos, em que o recurso limitado a ser gerenciado é a memória

[19] Ferramenta que fornece uma estrutura para construir reconhecedores, compiladores e tradutores de descrições gramaticais das linguagens Java, C++, *Python* etc.

[20] Ferramenta de visualização gráfica das informações estruturadas como diagramas de grafos abstratos

[21] O compilador Cetus realiza otimizações de códigos para as plataforma de uma CPU e GPU.

combinações com o intuito de extrair o máximo de informações que poderiam ser utilizadas nos passos seguintes do trabalho. A Figura 2.29 exemplifica os passos efetuados pelo Cetus em uma tarefa típica de compilação visando paralelizar um programa em C. Para que seja possível visualizar o exato momento em que cada passo é iniciado e finalizado pelo compilador, cada procedimento é distinguido através de suas cores.

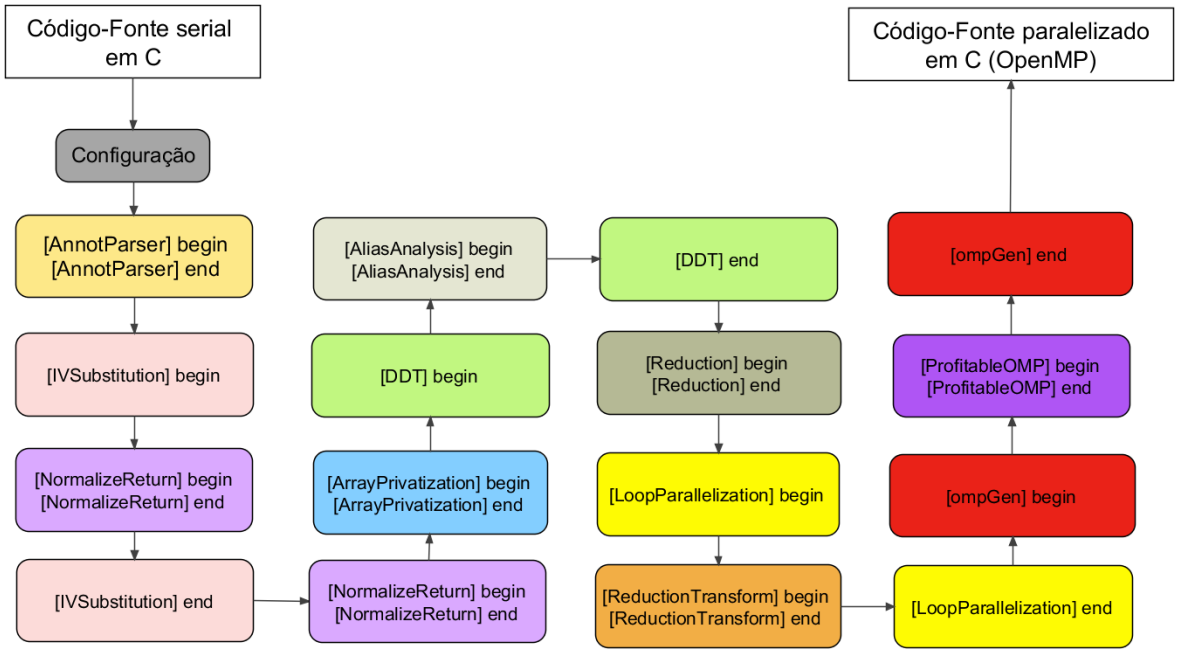

Figura 2.29: Passos efetuados pelo Cetus a partir de uma configuração.

A partir destes passos realizados pelo compilador, é possível acompanhar cada análise e transformação realizada no código fonte. A seguir é apresentada uma breve explicação do que é realizado em cada passo do Cetus, que por sua vez, este projeto em fez uso:

- *AnnotParse*: Este passo é responsável por analisar e converter as anotações externas presentes no código fonte em C, para anotações internas ao Cetus.
- *IVSubstitution*: Responsável por identificar e substituir as variáveis de indução. Para isso dois procedimentos são seguidos, sendo o primeiro de análise, que detecta as potenciais variáveis de indução e o segundo de transformação, que realiza a substituição e remoção do *statement* dependendo do tipo de declaração de cada um.
- *NormalizeReturn*: O passo de normalização é entendido como o processo de simplificação dos *statements*, ou seja, a simplificação das expressões matemáticas e operações simbólicas.
- *ArrayPrivatization*: Este passo tenta encontrar as variáveis de privatização (escalares e vetoriais) que são escritas antes do corpo do *loop*.

- *DDT*: Este passo é responsável por realizar os testes de dependência de dados.
- *AliasAnalysis*: São usadas para obter os alias das variáveis que fazem referência aos mesmos locais de memória. As informações dos alias podem ser de fluxo e/ou sensíveis ao contexto ou nenhum. O Cetus implementa estes dois níveis diferentes.
- *Reduction*: Realiza a análise de redução de variáveis para detectar e anotar declarações como $x = x + i$ em *loops*. Assim uma anotação é adicionada imediatamente antes do *loop* que contêm variáveis de redução.
- *LoopParallelization*: Todo o programa é varrido levando em consideração as análises de dependência de dados, a fim de localizar e anotar os *loops* que são executados em paralelo.
- *ReductionTransform*: Este passo é responsável por auxiliar a produção de código que é aceito pelo *backend* OpenMP do compilador. A transformação assume que o passo *loop-parallelization* não permite o paralelismo aninhado.
- *ompGen*: Responsável por procurar anotações que forneçam informações suficientes para adicionar os *pragmas* do OpenMP e em seguida os insere no código fonte.
- *ProfitableOMP*: Realiza o pós-processamento do programa em paralelo com o OpenMP. Sendo que, dois métodos um com modelo de desempenho simples e outro com a orientação de perfil são fornecidos e ambos os métodos fazem uso do if previsto pelas especificações do OpenMP.

O resultado de todas as análises e transformações com o objetivo de paralelizar o código fonte, é a geração de um novo fonte otimizado e paralelizado utilizando os *pragmas* do OpenMP. Além do suporte a transformações C serial para C paralelo (OpenMP)[22], o projeto Cetus e a comunidade acadêmica tem dado continuidade às pesquisas para aprimorar as transformações de OpenMP para MPI que é um padrão para comunicação de dados na computação paralela, como trafegar informações entre os vários processadores ou nodos de um *cluster* e de OpenMP para CUDA que corresponde a uma arquitetura *software* de propósito geral desenvolvida pela NVIDIA, que utiliza as unidades de processamentos gráficos (GPUs) para processamento paralelo de determinadas aplicações.

Alguns dos passos anteriormente descritos e apresentados na Figura 2.29 que abrangem os processos de análises do compilador, podem ser melhor entendidos nos seguidos itens:

- **Manipulação Simbólica**[23]: Trata a manipulação de equações matemáticas e expressões em sua forma simbólica, viabilizando a simplificação de expressões ao integrar símbolos e substituir uma expressão por outra como mostra a Listagem 2.27.

[22] É uma API para a programação utilizando multiprocessadores

[23] Referenciada também como: *symbolic manipulation, symbolic processing, symbolic mathematics,* ou *symbolic algebra*

Listagem 2.27: Cetus - Manipulação Simbólica. Fonte: (DAVE et al., 2009)

```
1+2*a+4-a --> 5+a (dobramento)
a*(b+c) --> a*b+a*c (distribuição)
(a*2)/(8*c) --> a/(4*c) (divisão)
(1-a)<(b+2) --> (1+a+b)>0 (normalização)
a && 0 && b --> 0 (avaliação de curto-circuito)
```

- **Análise de Seção de Matriz**: Descreve o conjunto de elementos de uma matriz acessados por instruções de programa. Um aspecto importante nessas análises é formulação de métodos para descrever estas seções, pois fatores como precisão, eficiência e praticidade influenciam a escolha de um determinado método. Um exemplo do resultado da análise da seção de Matriz expressa como uma anotação *pragma* é mostrado na Listagem 2.28.

Listagem 2.28: Cetus - Análise de Seção de Matriz. Fonte: (DAVE et al., 2009)

```
c = 2;
N = 100;
#pragma cetus USE(A[0:100][0:100])
DEF(B[1:99][1:99])
for(i=1; i<N; i++){
    for(j=1; j<N; j++){
        B[c*i-i][j] = (A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1])/4;
    }
}
```

- **Análise de Dependência de Dados**: Esta análise proporciona técnicas de desambiguação de memória, ou seja, procura identificar as referências de dados que acessam o mesmo local da memória durante a execução do programa e que caracteriza as dependências entre essas referências. O Cetus implementa um conjunto de analisadores de dependência de dados, sendo uma interface de coleta de informações embutida na IR com o objetivo de recolher informações sobre a matriz de acesso e *loops* relacionados. Esta análise é o ponto chave do Cetus para apoiar a geração de *loop pipelining* neste projeto.

- **Faixa de Análise**: Esta técnica calcula os intervalos das variáveis inteiras, ou seja, o valor destas variáveis em cada ponto do programa e retorna um mapa de cada declaração a um conjunto de intervalos de valores válidos antes de cada instrução.

Em continuidade aos passos anteriormente descritos e apresentados na Figura 2.29, mas que abrangem os processos de transformações do compilador, podem ser melhor entendidos nos seguidos itens:

- **Privatização**: A identificação de variáveis privadas em um *loop* é uma importante tarefa para a execução da paralelização automática. Uma variável privada atua como uma variável temporária em um *loop*, estas variáveis não precisam ser expostas para outros segmentos em tempo de execução, assim o analisador de dependência de dados pode seguramente assumir essas variáveis como não dependentes.

- **Reconhecimento de Variável de Redução**: São operações bastante utilizadas nas aplicações computacionais. EX: $vr = vr + expr$. O Cetus analisa as variáveis e verifica se satisfazem os critérios para a redução, sendo: o ciclo possui uma ou varias expressões de atribuição no formato $vr = vr + expr$, onde vr é uma variável escalar ou um acesso de *array*, e $expr$ não aparece em nenhum outro lugar do *loop*.

- **Substituição de Variável de Indução**: Da mesma forma que na Redução, a declaração de indução tem o formato $vi = vi + expr$, e deve ser substituído por outro que não induza a dependência de dados. Se o lado direito da formula anterior pode ser expressa como uma expressão de forma fechada que não contém vi, a dependência na declaração anterior será removida.

Como forma de exemplificar uma ação típica de compilação do Cetus, é apresentado na Listagem 2.29 um código fonte contendo uma estrutura de repetição (*loop*) executado de forma serial e logo na Listagem 2.30 é apresentado o código gerado e anotado após análises e transformações (como descritos nos passos da Figura 2.29) realizadas pelo compilador Cetus.

Listagem 2.29: *Loop* executado de forma serial. Fonte: (DAVE et al., 2009)

```
1  int foo(void)
2  {
3      int i;
4      double t,s,a[100];
5      for(i=0;i<50;++i)
6      {
7          t=a[i];
8          a[i+50]=t+(a[i]+a[i+50])/2.0;
9          s=s+2*a[i];
10     }
11     return 0;
12 }
```

Listagem 2.30: Cetus - *Loop* executado de forma paralela utilizando OpenMP. Fonte: (DAVE et al., 2009)

```
1  int foo(void)
2  {
3      int i;
4      double t,s,a[100];
5      #pragma cetus private(i,t)
6      #pragma cetus reduction(+:s)
7      #pragma cetus parallel
8      #pragma omp parallel for reduction(+:s)
9      private(i,t)
10     for (i=0;i<50;++ i)
11     {
12         t=a[i];
13         a[(i+50)]=(t+((a[i]+a[(i+50)])/2.0));
14         s=(s+(2*a[i]));
15     }
16     return 0;
17 }
```

O projeto Cetus passou de um simples projeto para traduzir código fonte, em um sistema robusto apoiado pela *National Science Foundation* como uma infraestrutura de compilação para a comunidade. Visto ser uma ferramenta de alta qualidade, fácil de usar e pronta para o uso através do portal da Universidade de Purdue[24]. Mediante suas particularidades que se destacam se comparado as demais ferramentas de paralelização automática de código, o Cetus tem grandes chances de tornar-se uma infraestrutura de pesquisa amplamente utilizada e aplicada ao nível de fonte de otimizações e transformações para ambos os núcleos e programas paralelos de grande escala.

Como forma de demonstrar o interesse pela comunidade acadêmica e suas recentes descobertas ao utilizarem o Cetus como ferramenta de apoio, são apresentados a seguir alguns projetos que merecem destaque. No artigo (ZHOU et al., 2004) foi proposto com o uso do compilador Cetus o monitoramento e a verificação de *bugs* nas alocações de memória. Já o artigo (DAVE; EIGENMANN, 2010) apresenta uma "sintonia" automática que visa uma melhor paralelização do código fonte utilizando as diretivas de análises do Cetus, possibilitando a posteriori uma sugestão de nova configuração que obtenha melhor desempenho do que as definições padrões do Cetus. Na proposta dos autores do artigo (TINEO et al., 2006), uma nova técnica busca auxiliar nas análises de estruturas como ponteiros. E não por fim, mas sim como término dos exemplos da viabilidade de uso do compilador Cetus, são apresentados no artigo (CASTILLO et al., 2006) alguns experimentos iniciais que o utilizam para obter informações detalhadas so-

[24] cetus.ecn.purdue.edu

bre como se organiza a área de memória dinâmica (*heap*), informação essa, de fundamental importância para encontrar o paralelismo nos acessos ao *heap*.

2.6.2 Pontos de Interesse do Compilador Cetus para este Trabalho

Os pontos que despertaram interesse e motivaram a utilização do compilador Cetus por este trabalho, podem ser pontuados a seguir:

- Recursos avançados para análise de dependência de dados
  - Determinação de partes do programa que podem ser executadas independentemente de outras.
  - Realização de otimizações para uma execução eficiente em paralelo.
  - Testes eficientes de dependências de dados.
  - Criação de um grafo de dependência de dados (DDG).
- Projeto ativo e em constante aprimoramento
- Compilador fonte para fonte
  - Importante pois o interesse deste trabalho está em transformar código C em um novo código C anotado e sincronizado.
- Um dos objetivos principais do Cetus é gerar código anotado para compilação OpenMP, e muitas das análises e transformações realizadas são úteis para o contexto do trabalho.

Os aspectos abordados nos item anteriores reforçaram a escolha do compilador Cetus por este trabalho. Contudo vale mencionar que outras infraestruturas foram analisadas e classificadas em pontos importantes que determinaram a conclusão do trabalho. Foram considerados:

1. Foco na paralelização e otimização de código.
2. Ferramenta *source-to-source*.
3. Tempo de aprendizado.
4. Documentação.
5. Projeto ativo e atualizado.
6. Suporte.

As ferramentas GCC, Rose, Coins e Cetus foram analisadas e comparadas segundo os itens anteriores. Sendo o GCC umas das infraestruturas de compilador mais robusta disponível, é capaz de gerar código altamente otimizado para uma variedade de arquiteturas e sua distribuição é *open-source*. Rose (QUINLAN et al., 2011) é uma infraestrutura de compilação de código aberto, desenvolvida no Laboratório Nacional Lawrence Livermore (LLNL) que visa a geração de analisadores e tradutores de fonte para fonte para múltiplas linguagens, que inclui C, C++ e Fortran, além de oferecer suporte a OpenMP, UPC e alguns arquivos binários. Rose destina-se a permitir o desenvolvimento de customizado e otimizados compiladores, contendo seus próprios analisadores. Por fim, o compilador Coins (SASSA et al., 2003) que visa a paralelização e otimização de fonte para fonte utilizando as linguagens C ou Fortran para arquiteturas como Intel x86, Sparc, Arm, Mips e PowerPC. Facilitando a construção de compiladores customizado, combinando ou modificando os componentes ou mesmo adicionando novos componentes. O desempenho obtido na execução dos códigos gerados a partir do compilador Coins se aproximam aos dos códigos gerados pelo GCC. Coins foi desenvolvido em Java e seu código é aberto está disponível para uso acadêmico e industrial.

A Figura 2.30 apresenta o comparativo entre as ferramentas GCC, Rose, Coins e Cetus segundo pontos determinados anteriormente. Para tal, foram utilizados três níveis de classificação, sendo *Bom* para o cumprimento do ponto em análise, *Regular* para o cumprimento parcial e *Ruim* para o não cumprimento. Esta forma de classificação, levou em conta a dificuldade em estimar quantitativamente cada ferramenta perante os pontos analisados.

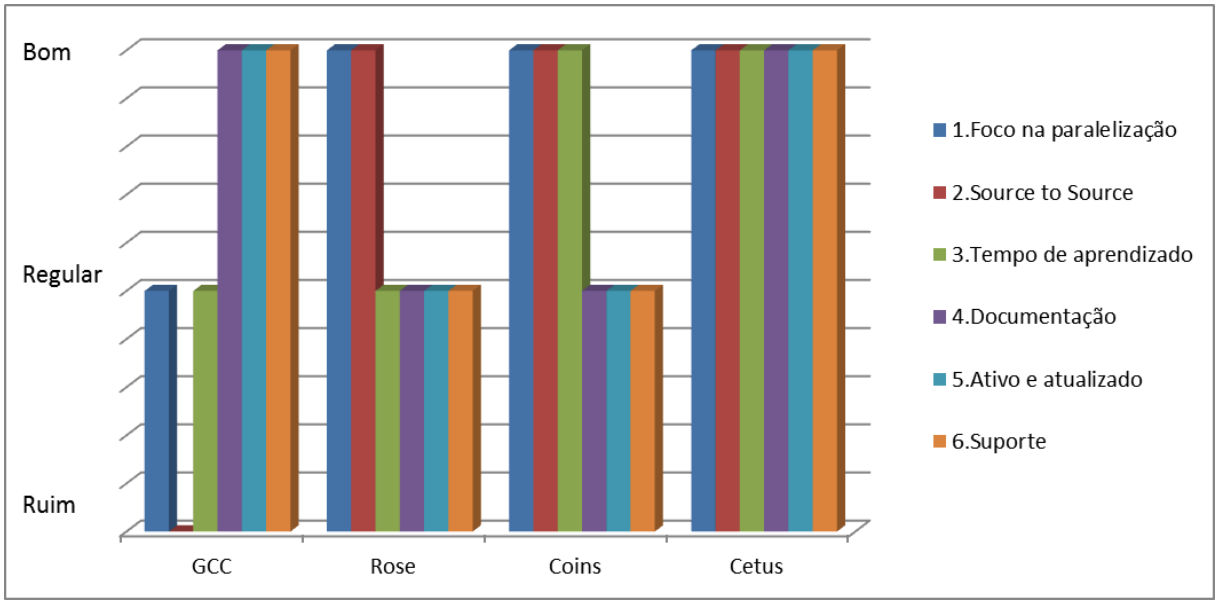

Figura 2.30: Comparativo que sugeriu a utilização do compilador Cetus.

Segundo análises contínuas realizadas pelos desenvolvedores do projeto Cetus, considerando as características de paralelização de código realizadas pelo compilador, os resultados mostraram qualidade considerável dos processos de paralelização e facilidade de utilização ao

ser comparada com ferramentas de mesmo propósito como: o compilador ICC Intel, o compilador Coins, as infraestruturas SUIF (suif.stanford.edu), Open64 (www.open64.net), Rose (rosecompiler.org), GNU C, Pluto (pluto-compiler.sourceforge.net) e compilador C do Grupo Portland (PGI) (DAVE et al., 2009).

Capítulo 3

TRABALHOS RELACIONADOS

Neste capítulo são apresentadas algumas das principais ferramentas recentemente desenvolvidas, voltadas para síntese de alto nível em FPGAs.

Diversas técnicas estão sendo desenvolvidas para aproximar os problemas computacionais, das linguagens mais próximas de nossa realidade. Pois partindo desta situação, a formulação de melhores ou novas soluções se tornam mais frequentes e aprimoradas, já que a linguagem utilizada faz parte do contexto do problema. Porém, muito deve ser feito para que essas técnicas ou mesmo o conjunto delas na forma de aplicativos automáticos, consigam tratar o máximo de problemas ao se preocuparem com desempenho e otimização de recursos.

A baixa aceitação no mercado para o uso de ferramentas automáticas de geração de *hardware* pode ser explicada pelo desejo dos projetistas de controlar como um todo os projetos, ou mesmo devido a ausência de uma linguagem universal que viabilize este controle (COUSSY; MORAWIEC, 2008). Contudo a otimização dos circuitos em se tratando da potência dissipada, tem se tornado de fundamental importância e de difícil realização em todos os níveis de abstração quando não se utiliza ferramentas automáticas (CONG; ROSENSTIEL, 2009).

Tanto na comunidade acadêmica quanto no âmbito comercial, ferramentas e técnicas são lançadas constantemente para possibilitar a exploração automática do espaço de projeto para diferentes requisitos e tecnologias. Uma das maneiras mais rápidas de modelar a funcionalidade de diferentes arquiteturas e estabelecer limites mínimos e máximos para a área, desempenho e potência, é proporcionada pela prototipação e exploração da arquitetura, que é o mais utilizado das ferramentas de síntese de alto nível.

Alguns dos principais projetos recentemente desenvolvidos que realizam a síntese de alto nível para FPGA, são descritos ao decorrer deste capítulo na seguinte sequência: Na seção 3.1 é apresentada a ferramenta C2H usada para geração de aceleradores para o processador Nios II, desenvolvida pela Altera. Na seção 3.2 é apresentado o compilador SPARK, desenvolvido na Universidade da Califórnia em San Diego. Na seção 3.3 é apresentada a ferramenta C-

to-Verilog para compilação de programas em C para Verilog de forma online. A seção 3.4 apresenta o compilador ROCCC, desenvolvido na Universidade da Califórnia Riverside. Na seção 3.5 é abordada a infraestrutura de *hardware/software* Impulse CoDeveloper. Na seção 3.6 é apresentada a biblioteca SystemC, que corresponde a um conjunto de classes em C++ para modelagem em nível de sistema como as linguagens VHDL e Verilog. Por fim, a seção 3.7 apresenta uma extensão de OpenMP para aceleradores FPGA.

Faz-se necessário ressaltar, que a priori não foram realizados estudos comparativos destas tecnologias, já que o período de tempo a fim de alcançar resultados relevantes para este projeto de pesquisa não pode se prolongar o bastante. As comparações foram realizadas no nível funcional, e não em termos do desempenho das aplicações geradas. Contudo esta tarefa será de fundamental importância já que estas análises apresentarão os pontos fortes e fracos do sistema de compilação desenvolvido. A ferramenta LALP se mostrou mais eficiente que as demais ferramentas na aceleração de *loops* citadas (MENOTTI et al., 2009b), o que motivou sua utilização como EC por este trabalho, com o intuito de facilitar o seu uso.

3.1 C2H

C2H (C-to-*Hardware Acceleration*) é um compilador desenvolvido pela Altera (NIOS, 2007) com o objetivo de acelerar os algoritmos criados utilizando a linguagem de programação C para o processador Nios II. Tal ferramenta não é responsável pela construção de hardware a partir de uma descrição em C, mas sim pela melhoria de desempenho dos programas executados no processador utilizando de aceleradores construídos em hardware.

A integração do acelerador no processador Nios II é feita por meio de um barramento dedicado para dados e instruções, que viabiliza a realização de operações de acesso direto a memória (DMA)[1] acessando uma ou mais memórias de dados, a fim de compartilhar dados com o processador como mostra a Figura 3.1

A ferramenta C2H realiza os seguintes procedimentos para a concepção do seu objetivo:

- Utiliza o compilador GCC como pré-processador para o código.
- Cria um grafo de dependência de dados (DDG).
- Realiza algumas otimizações (não especificadas).
- Determina a melhor forma de execução das operações
- Gera um arquivo com uma linguagem de descrição de hardware (HDL) para o módulo acelerador.

[1] *Direct Memory Access*

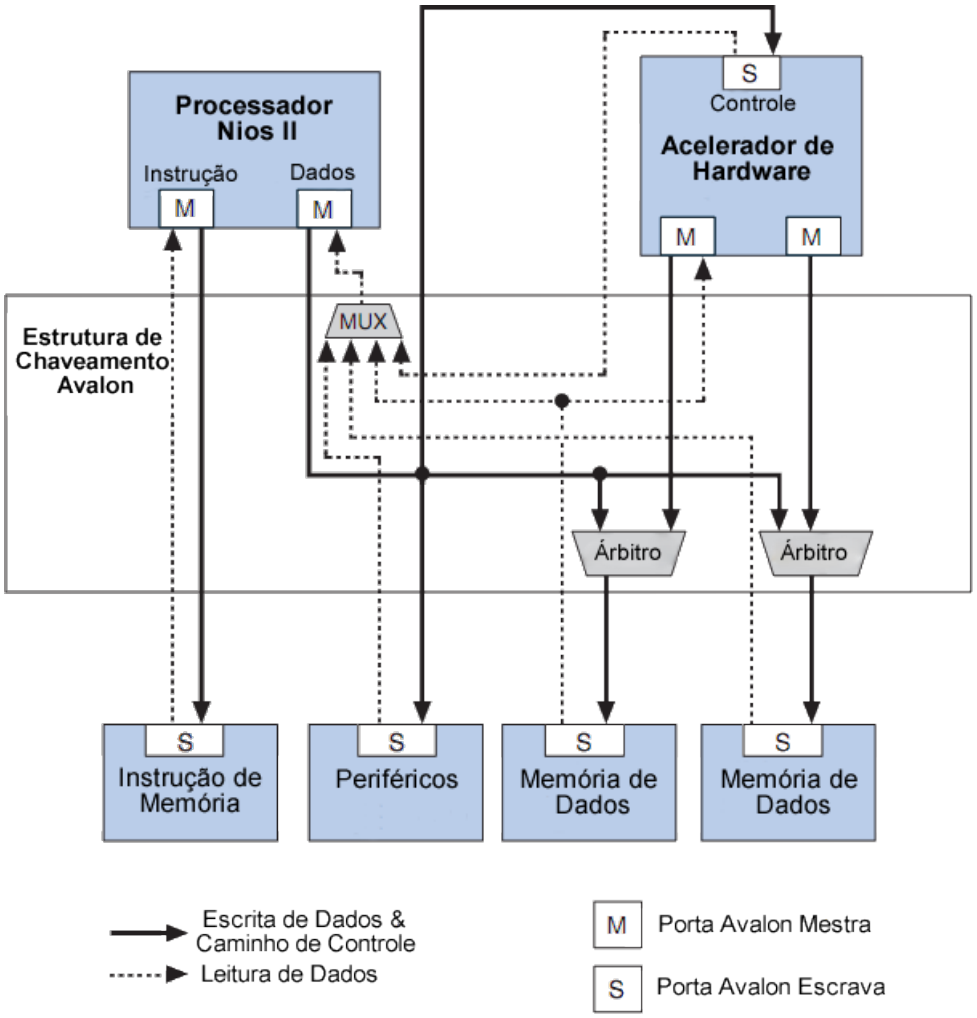

Figura 3.1: Exemplo de uma topologia de sistema com um único acelerador de *hardware*.
Adaptado de (NIOS, 2007)

- Gera um *wrapper*[2] para a função selecionada que será invocada no lugar da função original, se tornando transparente os detalhes da execução da função para o desenvolvedor.

Para a realização dos procedimento citados, o desenvolvimento utilizando o compilador C2H para acelerar uma função deve envolver os seguinte passos:

1. Desenvolver e depurar a aplicação em C para o processador Nios II.
2. Obter o perfil do código (*profiling*) para determinar as áreas que podem ser beneficiadas com a ferramenta.
3. Isolar o código desejado em funções separadas e se possível também em arquivos separados.
4. Especificar as funções que deseja acelerar na ferramenta.
5. Reconstruir o projeto.
6. Obter novamente o perfil do código.
7. Se os resultados não atenderem os requisitos de desempenho, modicar o código C e a arquitetura do sistema.
8. Retornar ao passo 5.

3.2 SPARK

O *framework* desenvolvido no projeto SPARK (GUPTA et al., 2004) apresenta uma metodologia para a síntese de alto nível de sistemas digitais baseada na paralelização. Desta forma, um código escrito em linguagem C (com algumas restrições) é inserido no sistema, para que daí seja gerado um código em VHDL para ser sintetizado.

Devido as dificuldades de se sintetizar algumas estruturas da linguagem C diretamente em hardware, a ferramenta possui restrições no código de entrada, tais como, o suporte ao uso de ponteiros, a recursão de funções, os saltos irregulares (*goto*) e por não estarem implementadas, restrições como o uso de estruturas (*struct*), *loops* que utilizam os comandos *break* e *continue* e o suporte a arranjos multidimensionais, ainda existem no *framework*.

O fluxo de compilação do SPARK é apresentado na Figura 3.2 e descreve em sua primeira fase a transformação do código fonte de entrada em uma representação intermediária em formato de grafos hierárquicos de tarefas (HTG)[3], grafos de fluxo de controle (CFGs) e grafos de

[2] Função cujo principal propósito é chamar uma outra função.
[3] *Hierarchical Task Graph*

fluxo de dados (DFGs). Em seguida são aplicadas as principais técnicas citadas na Seção 2.5 segundo literatura (AHO, 2007; MUCHNICK, 1997).

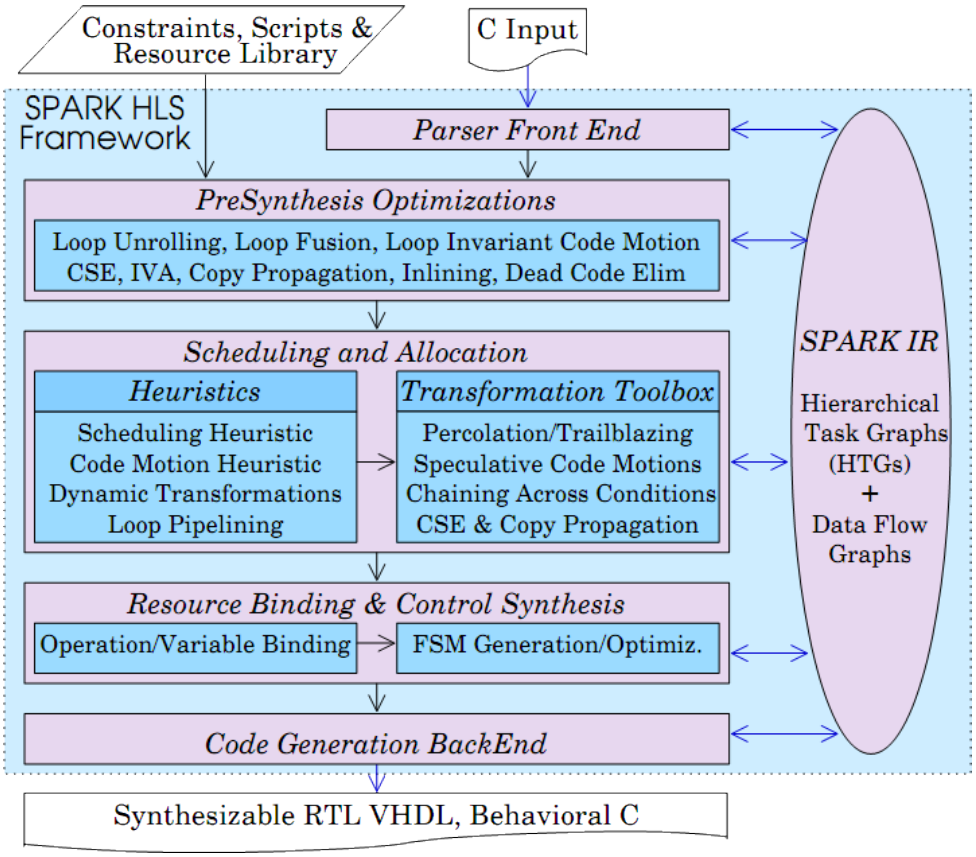

Figura 3.2: Fluxo de compilação do *framework* desenvolvido no projeto SPARK.
Fonte: (GUPTA et al., 2004)

Em continuidade, a próxima fase é responsável pelo escalonamento, que visa aumentar o desempenho e tentar extrair o paralelismo inerente das aplicações realizando para tal movimentações no código fonte. Porém, estas transformações aumentam significativamente a complexidade das conexões do hardware a ser gerado. A fim de diminuir esta complexidade as operações que possuem as mesmas entradas ou saídas são agrupadas em uma única unidade funcional da mesma forma que as variáveis são agrupadas em um registrador quando representam entradas ou saídas de uma mesma unidade funcional. Por fim, uma máquina de estados finitos (FSM)[4] é criada com o intuito de controlar cada estágio do projeto, e ão gerados também neste estágio os arquivos VHDL das unidades funcionais para serem sintetizados pela ferramenta adequada.

3.3 C-to-Verilog

A ferramenta C-to-Verilog (ROTEM, 2010) é o resultado do estudo acadêmico na área de síntese de alto nível da Universidade de Haifa (*Haifa University*) em Israel. O compilador

[4] *Finite State Machine*

usado no projeto é uma versão modificada do sistema de síntese *SystemRacer*. O código fonte do compilador está disponível para fins de pesquisa e tem sido usado por diversos grupos para o desenvolvimento de compiladores otimizados.

O compilador tem como objetivo a síntese de alto nível para a geração de código fonte em Verilog de forma online, a partir de um fonte de entrada escrita na linguagem C. Este código pode ser sintetizado em um FPGA ou ASIC. O compilador utiliza o escalonamento módulo para a geração do pipeline, o que proporciona uma boa alternativa para comparações de desempenho.

Algumas limitações são apresentadas pelo compilador para a geração de código verilog a partir de uma entrada em C, tais como, o uso de funções recursivas, estruturas, ponteiros para funções e chamadas às funções de biblioteca como *printf* e *malloc*. Múltiplas funções é algo possível de ser realizada, desde que sejam declaradas com os modificadores *staticinline*. A Listagem 3.1 apresenta um exemplo de entrada válida para o compilador C-to-Verilog.

Listagem 3.1: Exemplo de código C para o compilador C-to-Verilog

```
1  #define ITEMS (1024)
2  //retorna o numero de 1's em uma palavra
3  static inline unsigned int popCnt(unsigned int input){
4      unsigned int sum=0;
5      for (int i=0;i<32;i++){
6          sum+=(input)&1;
7          input=input/2;
8      }
9      return sum;
10 }
11 //este programa irá colocar o popCnt
12 //de cada palavra do B[] em A[]
13 void my_main(unsigned int* A, unsigned int *B) {
14     for (int i=0;i<ITEMS;i++) A[i]=popCnt(B[i]);
15 }
```

3.4 ROCCC

ROCCC (*Riverside Optimizing Compiler for Configurable Computing*) (GUO; NAJJAR; BUYUKKURT, 2008; VILLARREAL et al., 2010) é um projeto desenvolvido pela *University of California Riverside* que tem como objetivos maximizar o paralelismo dentro das limitações do dispositivo de *hardware* reconfigurável, otimizar a frequência de *clock* com técnicas eficientes de *pipelining* e minimizar a área de utilização destes dispositivos.

A ferramenta ROCCC é um compilador de código aberto baseado na plataforma SUIF (HALL et al., 1996) que utiliza aceleradores em hardware, ou seja, a partir de um subconjunto da

linguagem C ou FORTRAN é gerada a linguagem de descrição de hardware VHDL, utilizada em dispositivos FPGAs.

O compilador analisa o perfil do código de entrada para identificar os blocos executados com mais frequência. A partir disso as otimizações são realizadas nos *loops* e aplicadas a uma representação intermediária gerada pelo compilador, denominada CIRRF.

Como a ferramenta foi construída para otimizar aplicações com intenso fluxo de dados (DFI), desvios no fluxo de controle prejudicam seu desempenho. Para a otimização de determinadas operações, *buffers* e controladores são utilizados para a alimentação e coordenação das operações. Desta forma os dados são movidos de uma memória externa para um bloco de memória RAM interno ao FPGA e os circuitos gerados processam esses dados que são movidos para outro bloco de memória interno. Por fim, é importante salientar que algumas restrições são impostas no código de entrada do compilador, como a utilização de funções recursivas e o uso de ponteiros.

3.5 Impulse CoDeveloper

Impulse CoDeveloper (IMPULSE, 2010) é uma ferramenta de *design* de *hardware/software* que permite como entrada a linguagem de programação C a fim de desenvolver aplicações para plataformas mistas, como FPGAs combinados com outros processadores, como por exemplo o MicroBlaze, o PowerPC, o Nios, ou DSPs. A biblioteca Impulse C e ferramenta de compilação para apoiar modelos múltiplos de programação, permite que um programador de *software* faça uso de recursos de FPGA disponíveis para co-processamento de *hardware* sem a necessidade de utilizar linguagens de descrições *hardware*.

As ferramentas de programação CoDeveloper proporciona a geração automática e optimizada de interfaces de *hardware/software*. Essa capacidade torna possível a engenheiro criar aplicações completas, sem a necessidade de utilizar linguagens como VHDL ou Verilog. Em vez disso, as ferramentas CoDeveloper criam as necessárias descrições de baixo nível na forma de saídas em HDL e bibliotecas de *software* geradas automaticamente, que são importadas diretamente para as ferramentas dos fabricantes de FPGAs para a síntese de *hardware* e implementação.

O fluxo de programação do Impulse CoDeveloper é apresentado na Figura3.3 e pode ser pontuado de maneira mais agrangente da seguinte forma:

1. Inserção ou importação de projeto em C usando as ferramentas Impulse C e Design CoDeveloper Assistant.

2. Compilação e depuração utilizando ferramentas como CodeWarrior, GCC, Visual Studio

ou outras para desenvolvimento em C.

3. Utilização do CoDeveloper e Impulse C para exploração do paralelismo e redução dos gargalos de performance, acelerando a aplicação.

4. Seleção do FPGAs alvo e núcleos de microprocessadores.

5. Geração automaticamente de HDL incluindo interfaces.

6. Sintetização do *hardware* gerado utilizando ferramentas padrões de programação em FPGA.

7. Compilar elementos de software usando o padrão cross-compilador.

8. Verificação do *hardware* usando ModelSim ou qualquer outro simulador HDL.

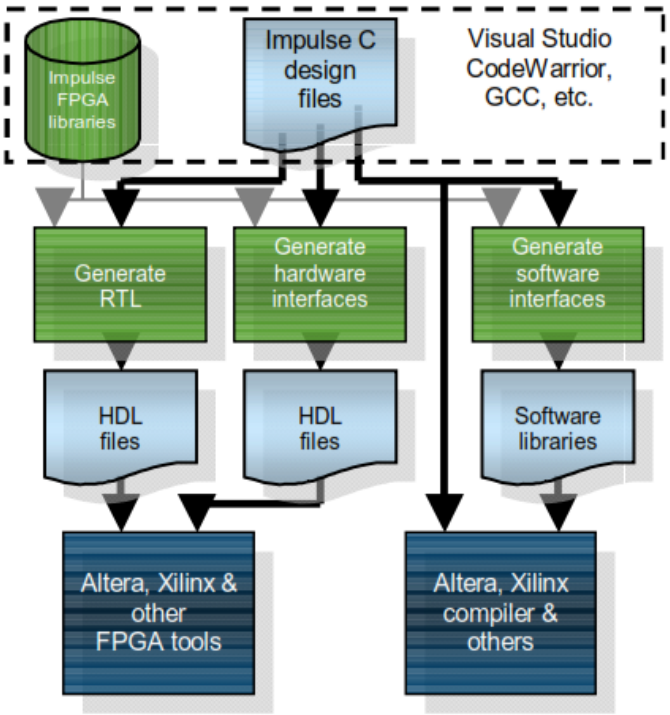

Figura 3.3: Fluxo de programação ImpulseC/RTL/FPGA.
Fonte: (IMPULSE, 2010)

3.6 System C

SystemC (GRÖTKER et al., 2002) é uma linguagem padrão da indústria para nível de sistema eletrônico (ESL - *Electronic System-Level*) *design* e modelagem, permitindo a criação de descrições abstratas de arquitetura de *hardware/software* sistemas digitais, também conhecido como protótipos virtuais.

A partir de um conjunto de classes e macros do C++, é fornecido um *kernel* de simulação orientada a eventos. Essas facilidades permitem simular processos simultâneos descritos usando a sintaxe do C++. Em certos aspectos, o SystemC pode ser comparado às linguagens de descrição de *hardware* VHDL e Verilog, mas é melhor descrita como uma linguagem de modelagem em nível de transação (TLM - *Transaction Level Modeling*).

Empresas líderes na propriedade intelectual (IP), automação de *design* eletrônico (EDA), semicondutores, sistemas eletrônicos e indústrias de *software* embarcados, usam atualmente o SystemC para a exploração de arquitetura. Como entrega de blocos de *hardware* de alto desempenho em vários níveis de abstração e desenvolvimento de plataformas virtuais para *hardware/-software*. A comunidade SystemC tem recebido grande apoio destas empresas com o intuito de promover o padrão.

3.7 OpenMP extensions for FPGA Accelerators

O crescente interesse no uso de arquiteturas híbridas, ou seja, que incluem um processador principal e outros dispositivos auxiliares como GPUs ou FPGAs, motivou o desenvolvimento de uma extensão da API OpenMP 3.0 (CABRERA et al., 2009). Neste trabalho, os autores propõem recursos que facilitam a especificação de trechos de processamento a serem executados em um FPGA, permitindo que o ambiente de execução gerencie a transferência de dados entre esses elementos.

Capítulo 4

COMPILADOR CETUS MODIFICADO

Este capítulo dedica-se a expor de forma gradual todo o processo para a realização das modificações ao compilador Cetus.

Os motivos pelos quais este trabalho fez uso do compilador Cetus, são expostos na Subseção 2.6.2 do Capítulo 2. Desse modo, a partir da avaliação e validação da ferramenta em acordo com os objetivos propostos, consolidam-se todas as modificações apresentadas neste Capítulo.

4.1 Preparação do Ambiente de Trabalho

Esta seção expõe algumas das necessidades levantadas no início deste trabalho de pesquisa, refletindo diretamente na boa organização, manutenibilidade, eficiência e usabilidade dos procedimentos posteriormente desenvolvidos. Sendo que esta preparação refere-se à escolha de ferramentas de *software* para apoiar as modificações a serem feitas no Cetus.

4.1.1 Biblioteca para Manipulação de Grafos

Ao decorrer dos estudos do compilador Cetus, foram observadas diversas classes que o auxiliam na criação e manipulação dos grafos internos a partir do código fonte de entrada. Dessa forma, a boa continuidade deste projeto dependia necessariamente da interpretação destas classes, o que apresentou relativa complexidade de entendimento e induziu a busca por ferramentas que realizassem a manipulação de grafos, para que apenas as informações das análises e transformações do Cetus fossem extraídas e utilizadas pelo projeto.

Esta decisão não apenas facilitou as análises das informações que são necessárias para este trabalho, mas como também simplificou a interpretação do grafo gerado, viabilizando possíveis alterações sem que o funcionamento normal do compilador Cetus fosse afetado.

A escolha de uma ferramenta que possibilitasse a realização destes processos, foi imprescindível para dar sequência às análises necessárias por este trabalho. Dessa modo, algumas ferramentas que auxiliam na criação e manipulação de grafos foram selecionadas com base nos requisitos citados a seguir:

- Biblioteca em Java, haja visto ser a linguagem utilizada no desenvolvimento do compilador Cetus.
- Código fonte aberto.
- Farta documentação.
- Robusta em suas análises e algoritmos para manipulação do grafo.
- Utilização simples e facilitada.
- Possibilitar a criação e manipulação de grafos:
  - Direcionados e Não Direcionados.
  - Com peso e rótulos nas arestas.
  - Personalização dos vértices e arestas.
  - Com algoritmos que facilitem a manipulação das informações do grafo criado.

A partir dos requisitos levantados, as ferramentas selecionadas foram analisadas e classificadas a fim de que fosse escolhida a que melhor atendesse as necessidades do projeto. Na Figura 4.1 são apresentadas as bibliotecas selecionadas e suas análises baseadas nos requisitos levantados. Para tal, foram utilizados três níveis de classificação, sendo *Sim/Bom* para o cumprimento do ponto em análise, *Regular* para o cumprimento parcial e *Não/Info. não disponível* para o não cumprimento ou a falta de informação explicita nas documentações existentes.

Após análise das ferramentas, a que melhor atendeu aos objetivos do projeto foi a biblioteca *JgraphT* (NAVEH et al., 2008). Pois, além de conter todos os recursos necessários para dar sequência ao projeto, esta biblioteca apresentou uma curva de aprendizado aceitável considerando o tempo disponível para dar sequência ao trabalho.

A biblioteca JgraphT viabilizou o desenvolvimento de classes vértices e arestas customizadas para a criação do grafo. Estas novas classes foram criadas com o objetivo de receber as informações do *loop* extraídas do compilador Cetus, tais como: instruções do corpo do *loop*, seus predecessores e sucessores com base nas análises de dependências de dados e fluxo de controle, os tipos de dependências de dados e suas respectivas distâncias, estruturas condicionais, vetores de uso ou definição.

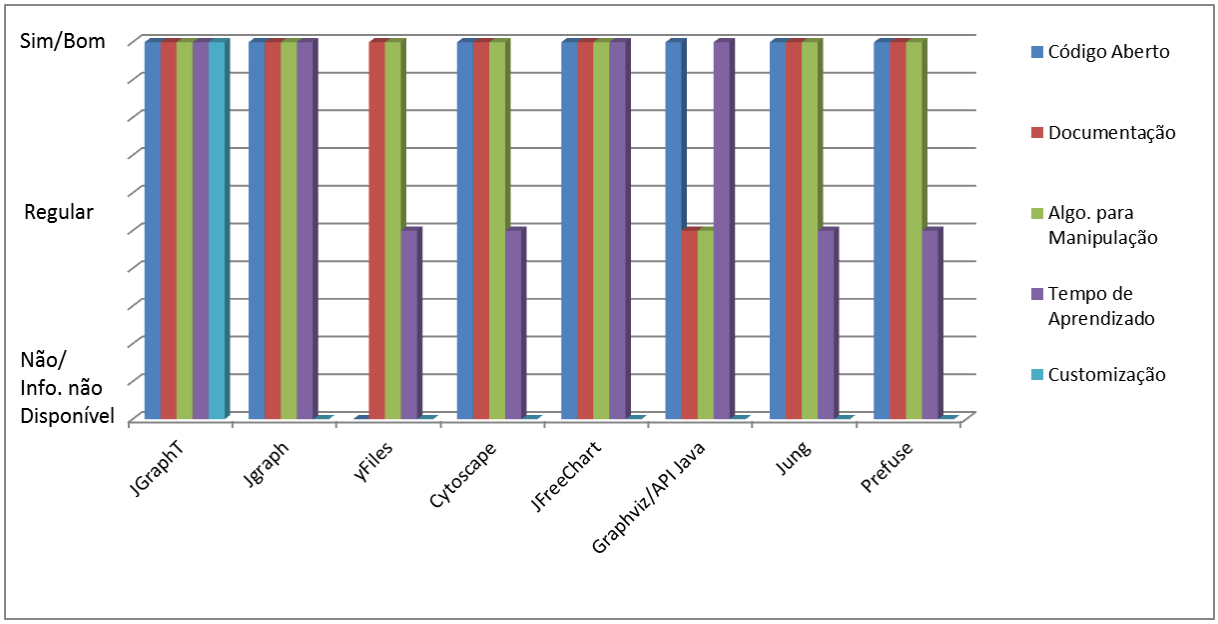

Figura 4.1: Comparativo que sugeriu a utilização da biblioteca JgraphT

A partir da exploração das informações processadas pelo Cetus, uma cartela de parâmetros foi criada para compor o novo grafo. Na Figura 4.2 é apresentado o mapa dos parâmetros contidos nos novos vértices e arestas criados com auxilio da biblioteca JgraphT. Em seguida na Tabela 4.1 é feita uma breve explicação do conteúdo do vértice e da mesma forma para o conteúdo do arco do grafo, a Tabela 4.2.

Com a utilização destas classes customizadas, foi possível manipular as informações importantes para a continuidade do trabalho. Permitindo a preservação do código original do Cetus e o encapsulamento de todos os procedimentos criados neste trabalho.

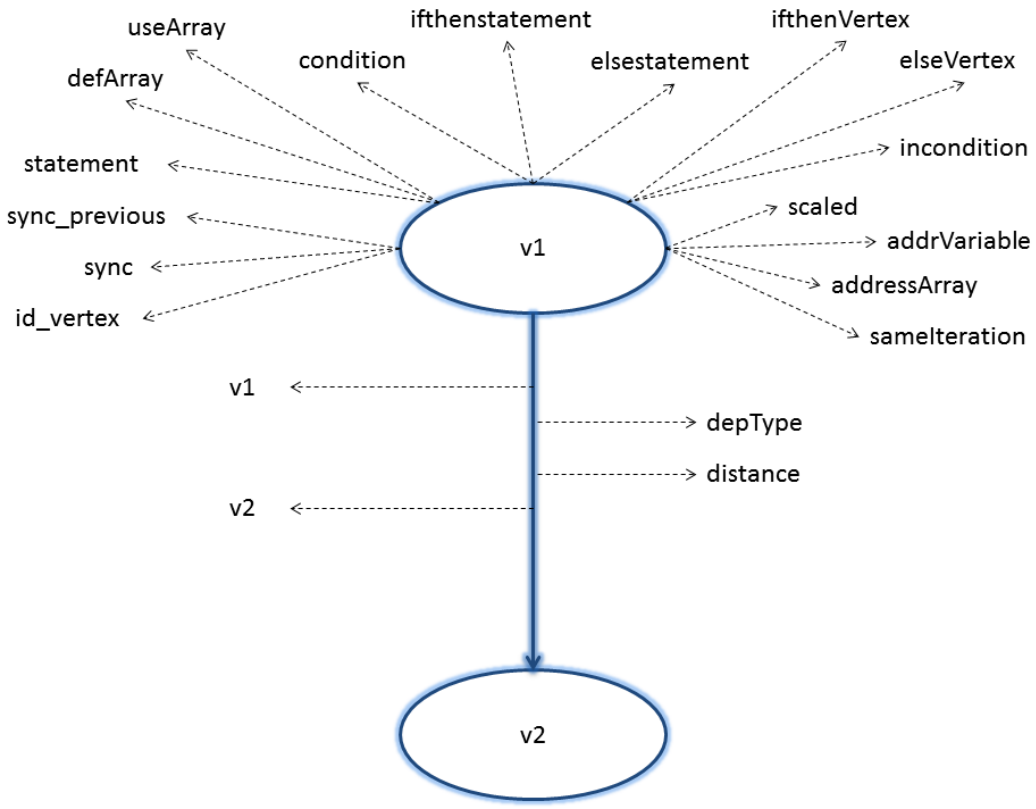

Figura 4.2: Mapa das informações do novo grafo

Parâmetro	**Função**
id_vertex	identificação do vértice
sync	sincronismo do vértice
sync_previous	variável auxiliar que armazena o último sincronismo atribuído ao vértice
statement	instrução do vértice
defArray	definição de variável de acesso à memória do *statement* do vértice
useArray	uso de variável de acesso à memória do *statement* do vértice
condition	instrução condicional (*IF*) do vértice
ifthenstatement	instrução interna (*then*) da condicional (*IF*) do vértice
elsestatement	instrução interna (else) da condicional (IF) do vértice
ifthenVertex	vértice interno (if then) da condicional (IF) do vértice
elseVertex	vértice interno (else) da condicional (IF) do vértice
incondition	determina se o vértice está ou não dentro da condicional(IF)
scaled	determina se o vértice foi ou não escalonado
addrVariable	determina o endereçamento do vetor
addressArray	determina se o vértice possui ou não endereçamento a vetor
sameIteration	determina se o vértice está na mesma iteração que o contador do *loop*

Tabela 4.1: Parâmetros do vértice do grafo.

Parâmetro	Função
v1	vértice origem
v2	vértice destino
depType	tipo de dependência de dados 1 - Verdadeira , 2 - Anti , 3 - Saída , 4 - Entrada , 5 - Inválida
distance	distância entra duas instruções dependentes

Tabela 4.2: Parâmetros da aresta do grafo.

4.1.2 Visualização dos Grafos

Após a extração das informações do compilador Cetus e a criação do novo grafo a partir da biblioteca JgraphT, surgiu a necessidade de externalizar este grafo para melhor interpretação das informações dispostas. Portanto a visualização dos grafos em formato amigável e preciso foi muito importante no processo de desenvolvimento da infraestrutura de compilação, além de agir como elemento auxiliar para o usuário programador do sistema.

Dessa forma, uma classe foi criada com o intuito de exportar o grafo gerado em qualquer momento das análises e transformações realizadas. A classe é responsável por exportar este grafo para a linguagem Dot[1], que foi escolhida por apresentar boa qualidade visual na produção de grafos utilizando a ferramenta *Graphviz* (ELLSON et al., 2002).

Esta exportação leva em consideração o identificador do vértice (ID), o trecho de código (*statement*), o tipo de dependência (verdadeira, anti-dependência, saída), a distância entre os vértices e o sincronismo (*Sync*). Sendo impresso um grafo direcionado com os valores das distâncias plotados logo acima de cada aresta correspondente, o identificador do vértice e trecho de código são impressos dentro de cada nó, o sincronismo é precedido da diretiva @ e é impresso logo acima do nó correspondente. Por fim, as arestas podem ser sólidas nas cores preto e vermelho ou tracejadas nas cores marrom, verde e azul, sendo considerado para as arestas sólidas e na cor preto as dependência verdadeiras entre os vértices, na cor vermelho são representadas as arestas não validadas ou incoerentes após verificações internas. As arestas tracejadas na cor marrom, representam as anti dependências, na cor verde as dependências de saída e na cor azul as dependências de entrada existentes entre os vértices.

A função para a exportação do grafo em formato Dot, foi incluída aos procedimentos posteriormente criados. Sendo estabelecido um parâmetro de entrada (0 ou 1) que determina o tipo de visualização que se deseja ter. Dessa forma, ao optar pelo parâmetro 0 (*default*) o grafo será plotado apenas com as dependências verdadeiras, como mostra a Figura 4.3. Já ao utilizar como parâmetro o valor 1, o grafo será plotado contendo todas as dependência existentes entre os vértices, como mostra a Figura 4.4. Em ambos os grafos apresentados, foi utilizado o mesmo

[1] *Script* utilizado para descrever grafos desenvolvida pela empresa *AT & T Labs* para uso em sua ferramenta *Graphviz*

trecho de código fonte como mostrado na Listagem 4.1.

Listagem 4.1: Exemplo de um trecho de código fonte para geração do grafo em formato Dot

```
1 ...
2   c[i]= a[i];
3   a[i]= 20 * i;
4   b[i]= a[i] * k[i];
5   k[i]= c[i] + b[i];
6 ...
```

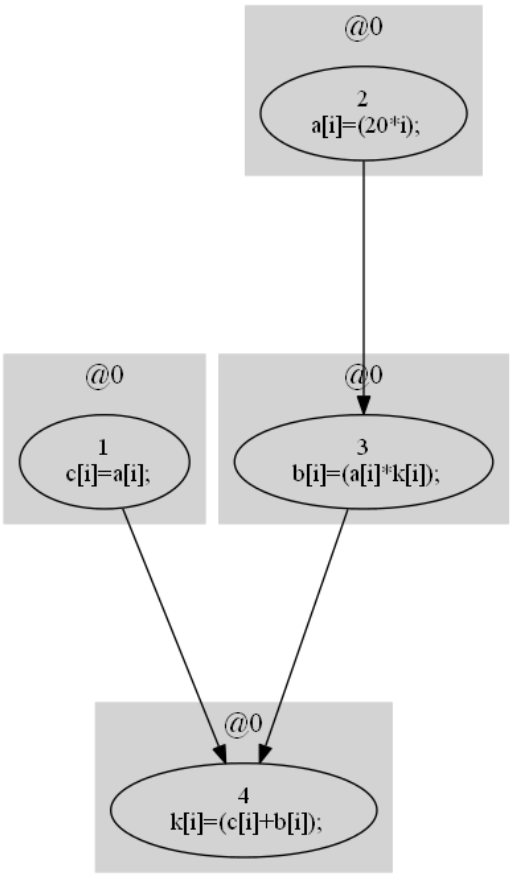

Figura 4.3: Exemplo de grafo exportado para Dot e plotado no Graphviz com apenas as dependências verdadeiras.

É possível identificar nos grafos plotados o número que identifica cada vértice (ID), o trecho de código (*statement*), o valor do sincronismo (*Sync*) precedido da diretiva @ (a mesma utilizada pela linguagem LALP para determinação das dependências entre as instruções) e as distâncias representadas junto a cada aresta. A linguagem Dot pode ser interpretada por qualquer ferramenta gratuita que se propõe a tal, e logo exportada em formato de imagem. Vale salientar que estes @ referem-se ao sincronismos das instruções, o que neste momento ainda não é abordado. Este tema será explicado mais detalhadamente na Subseção 4.2.5, mais à frente.

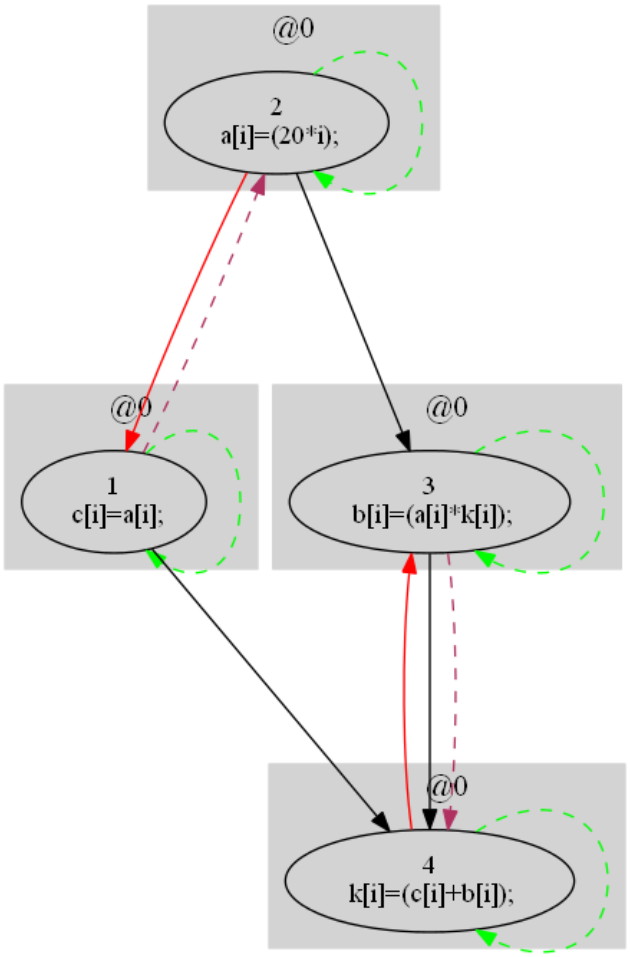

Figura 4.4: Exemplo de grafo exportado para Dot e plotado no Graphviz com todas as dependências existentes.

A visualização em forma de imagem foi de grande importância para a continuidade deste trabalho, pois a cada nova análise e transformação realizada no grafo, a visualização imediata em forma gráfica proporcionou o *feedback* de cada procedimento testado, validando cada passo desenvolvido.

4.2 Modificações Efetuadas

Esta seção apresenta as principais modificações no compilador Cetus realizadas por este trabalho, que visam facilitar a geração de *loop pipelining* em *hardware*, seja usando LALP, ou FSM (*Finite State Machine*). Em resumo à seção, é possível citar a extração do DDG (*Data Dependence Graph*) dos *loops* mais internos (*innermost*), a extensão do DDG para inclusão de dependências escalares, o escalonamento de instruções usando conceitos de *modulo scheduling* e menor II (*Initiation Interval*), a geração e escalonamento de instruções de acesso à memória respeitando as restrições do *framework* LALP e por fim, a geração de código fonte sincronizado próximo ao esperado para a linguagem LALP.

4.2.1 Acesso aos Passos de Análises do Compilador Cetus

A partir da inserção de um código fonte executado de forma serial no compilador Cetus (LEE; JOHNSON; EIGENMANN, 2004), diversas análises e transformações são realizadas a fim de modificar e adequar este código para sua execução de forma paralela, obtendo desempenho relativamente melhor. Faz-se necessário ressaltar que estas análises e transformações realizadas pelo compilador, visam a obtenção de melhores resultados para as plataformas de uma CPU ou GPU.

Como o objetivo deste trabalho é implementar aceleradores para auxiliar a programação em uma plataforma diferente para o qual se presta os objetivos do Cetus. Este trabalho se ateve a identificar e extrair somente as informações necessárias para serem manipuladas a posteriori, com o intuito de facilitar a programação em FPGA.

A identificação destas informações, foi obtida após um trabalho árduo de investigação do código fonte do compilador Cetus. A partir destas análises, foram localizados os passos que realizam a manipulação do trecho de código em repetição, haja visto ser o foco deste trabalho, as análises de dependências de dados e a identificação das instruções contidas nesta estrutura.

Ao fim desta tarefa, iniciou-se o processo de identificação das classes que integram estes passos e em seguida o reconhecimento de suas estruturas internas. O que possibilitou a extração das informações necessárias após a obtenção pelo compilador. A Figura 4.5 apresenta parte da representação intermediária do Cetus e junto sua hierarquia de classes. Nesta ordem, a representação intermediária e sintaxe estão descritos na cor azul e sua correspondência na forma

hierárquica de classes na cor vermelho.

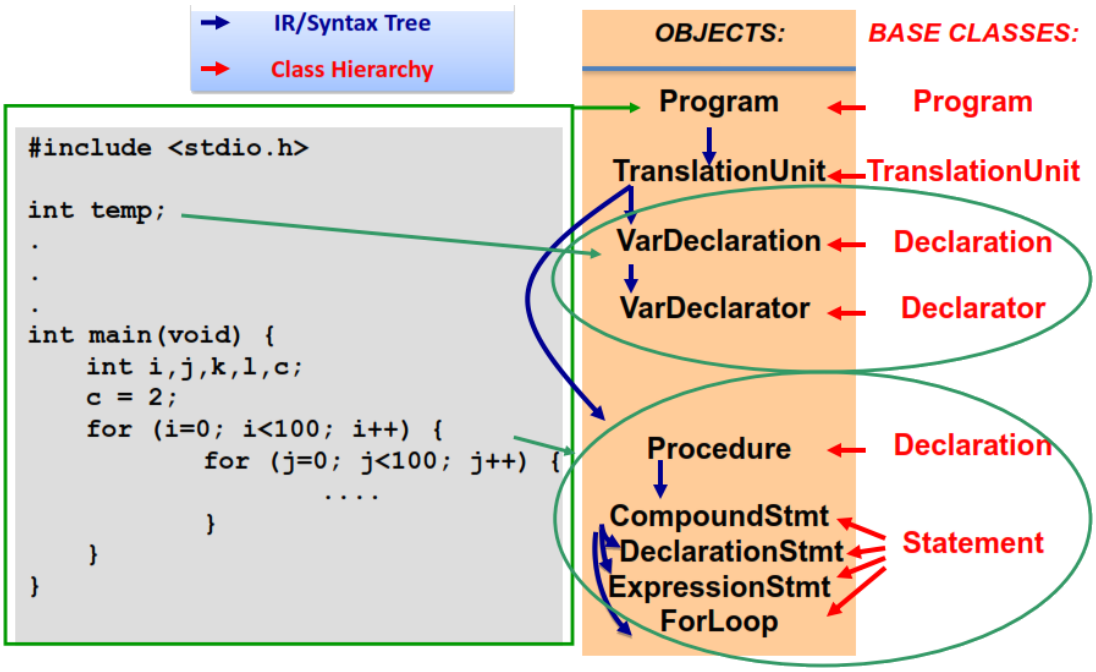

Figura 4.5: Parte da composição da IR e hierarquia de classes do Cetus.
Fonte: (LEE; JOHNSON; EIGENMANN, 2004)

A Figura 4.5 demonstra também, a conexão do código fonte com as estruturas de classes e IR do Cetus. É possível destacar três grandes estruturas a partir desta figura, sendo a primeira o programa fonte inserido pelo usuário (*Program*), a segunda as declarações das variáveis inseridas neste programa (*VarDeclaration* etc..) e por fim a identificação da estrutura de repetição e suas subestruturas (*Procedure*).

Dessa forma, a partir do entendimento interno do funcionamento do Cetus, deu-se início o processo de investigação do código fonte do compilador a fim de utilizar as corretas estruturas necessárias para cumprir os objetivos deste trabalho.

4.2.2 Criação do Passo FPGA no Compilador Cetus

Após a identificação das informações necessárias para dar sequência a este trabalho, uma rotina dentro do compilador Cetus foi criada. Rotina essa, criada a fim de customizar e reunir todas as informações relevantes por este trabalho, para que as análises e transformações fossem realizadas com o intuito de facilitar a programação em *hardware* customizado. Ou seja, um conjunto de classes foi criado visando a distinção dos procedimentos normais do Cetus, dos procedimentos criados para analisar, transformar e otimizar o código fonte para auxiliar na programação de *hardware* customizado.

Desse modo, foi desenvolvido um novo passo ao compilador Cetus, chamado *FPGAParser* que está contido dentro do pacote de classes *cetus.analysis* do compilador. A Listagem 4.2

apresenta a forma como é realizada a invocação do método *Driver* do Cetus. É possível notar que dois parâmetros são necessários para que o perfil de análise seja definido, sendo [*option*] responsável por definir a linha de comando que será utilizada, ou seja, o tipo de perfil de análise que se deseja processar no código fonte e o parâmetro [*files*], responsável por determinar o caminho do arquivo fonte de entrada. Em seguida na Listagem 4.3 é apresentada a forma de chamar o passo FPGA criado.

Listagem 4.2: Como invocar o método *Driver* do Cetus.

```
1 java cetus.exec.Driver [option] [files]
```

Listagem 4.3: Como invocar o passo FPGA no Cetus

```
1 java cetus.exec.Driver -fpga teste.c
2 ou
3 cetus -fpga teste.c
```

O passo FPGA invoca de forma implícita dois métodos do compilador Cetus, sendo o *parallelize-loops* contido no processo de análise e responsável por criar as devidas anotações para as tomadas de decisões de paralelização dos *loops*, mais internos ou não. E o segundo método o *ddt=2*, também contido no processo de análise do compilador e responsável pelos testes de dependências de dados.

A razão pela qual estes dois métodos foram utilizados, está na prévia investigação realizada ao percorrer todas as possíveis combinações de parâmetros do compilador, ou seja, foram realizados testes com todas as variações de parâmetros de configuração permitidos pelo Cetus. Dessa forma, a partir da depuração de um mesmo código fonte para todos os testes, foi identificada a ordem de execução que mais forneceu dados para a posterior manipulação. A seguir são apresentados nas Listagens 4.4 à 4.11 os perfis de configurações testados no compilador Cetus, sendo que cada perfil permite a combinação de dois parâmetros no campo [*option*] ao invocar o método *Driver*:

Listagem 4.4: Configuração 1

```
1 java cetus.exec.Driver -parallelize-loops -profile-loops=1 teste5.c
2 java cetus.exec.Driver -parallelize-loops -profile-loops=2 teste5.c
3 java cetus.exec.Driver -parallelize-loops -profile-loops=3 teste5.c
4 java cetus.exec.Driver -parallelize-loops -profile-loops=4 teste5.c
5 java cetus.exec.Driver -parallelize-loops -profile-loops=5 teste5.c
6 java cetus.exec.Driver -parallelize-loops -profile-loops=6 teste5.c
```

Listagem 4.5: Configuração 2

```
1 java cetus.exec.Driver -parallelize-loops teste5.c
```

Listagem 4.6: Configuração 3

```
1 java cetus.exec.Driver -profile-loops=2 teste5.c
```

Listagem 4.7: Configuração 4

```
1 java cetus.exec.Driver -parallelize-loops -verbosity=2 teste5.c
```

Listagem 4.8: Configuração 5

```
1 java cetus.exec.Driver -parallelize-loops -verbosity=3 teste5.c
2 java cetus.exec.Driver -parallelize-loops -verbosity=4 teste5.c
```

Listagem 4.9: Configuração 6

```
1 java cetus.exec.Driver -parallelize-loops -ddt=1 teste5.c
2 java cetus.exec.Driver -parallelize-loops -ddt=2 teste5.c
```

Listagem 4.10: Configuração 7

```
1 java cetus.exec.Driver -parallelize-loops -ddt=3 teste5.c
```

Listagem 4.11: Configuração 8

```
1 java cetus.exec.Driver -verbosity=2 -profile-loops=4 teste5.c
```

A partir dos testes realizados, foi possível identificar o perfil de configuração que permitisse a obtenção mais detalhada das informações de interesse deste trabalho. Dessa forma, a Figura 4.6 apresenta os perfis de configurações que foram avaliados considerando as seguintes classificações, sendo que todas as configurações foram aplicadas no mesmo código fonte apresentado na Listagem 4.12.

- *Bom*: Para o perfil que mais informações disponibiliza sobre as dependências de dados.
- *Regular*: Para o perfil que disponibiliza algumas informações sobre as dependências de dados, mas ainda assim de forma limitada.
- *Erro*: Para o perfil de configuração que não enquadra no modelo do código fonte de entrada, ou seja, as análises e transformações são direcionadas para códigos fonte diferentes

dos códigos utilizados, o que foge ao foco deste trabalho.

- Perfil não classificado: Corresponde a não obtenção de informações sobre as dependências de dados.

Listagem 4.12: Programa utilizado para a realização dos testes das análises e transformações do Cetus.

```
1  int main(){
2      int i;
3      int n=100;
4      int s=0;
5      double a[n], b[n], c[n];
6      for ( i=0; i<100; ++i ){
7          c[i]= a[i] + b[i];
8          a[i]= c[i−1] ∗ 2;
9          b[i]= c[i−2] ∗ 5;
10         s+= c[i];
11     }
12 }
```

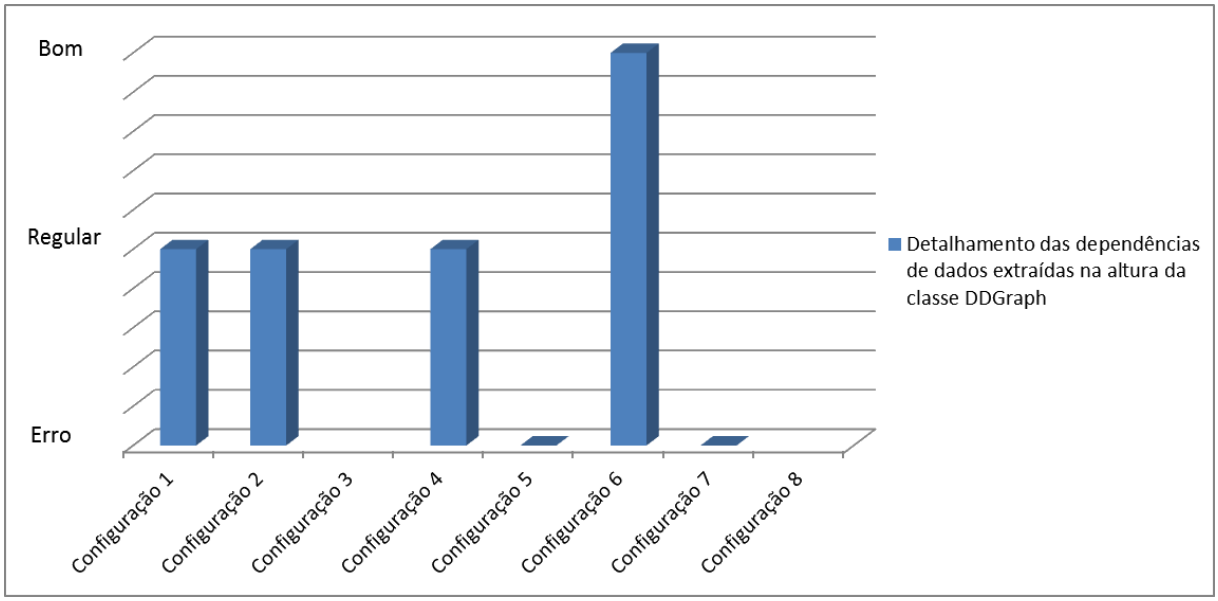

Figura 4.6: Comparativo entre os perfis de configuração avaliados na execução do Cetus.

É importante salientar que esta classificação determina a altura das análises realizadas pelo Cetus, ou seja, cada configuração inserida ao compilador determina o nível de análises que se deseja obter do código. Desse modo, o objetivo neste momento foi identificar o perfil que mais níveis abrange, retornando o máximo de informações a partir das análises realizadas.

4.2.2.1 Resumo das Funcionalidades Criadas no Passo FPGA

Após todas as análises e transformações realizadas a partir da invocação do passo FPGA, diversos métodos foram criados a fim de facilitar a obtenção das informações necessárias para que os devidos procedimentos fossem executados. Como forma de demonstrar o fluxo de análises e transformações feitas no código fonte pelo passo FPGA, a Figura 4.7 apresenta os principais procedimentos realizados visando a transformação do código fonte em C para referência ao uso de uma FSM (EC-1) e para um bloco de código em LALP (EC-2).

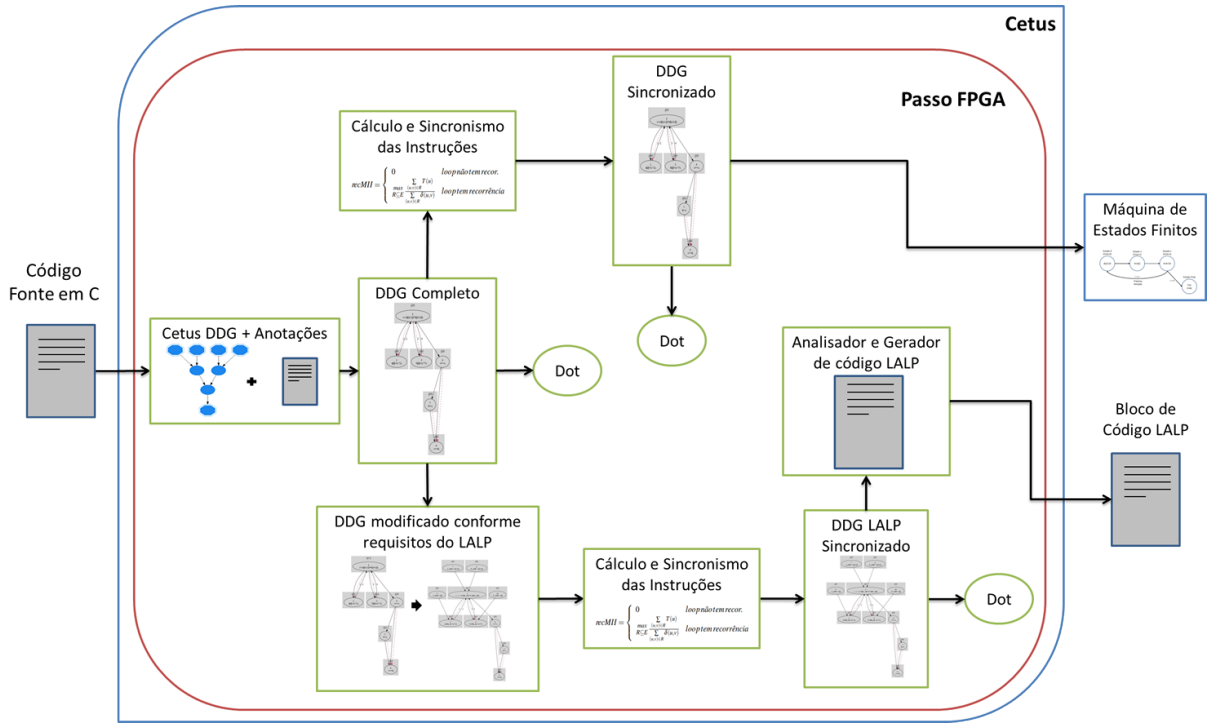

Figura 4.7: Visão geral do fluxo de transformações realizadas no código fonte, para gerar uma FSM e um bloco de código LALP

Na sequência a Tabela 4.3 apresenta de forma detalhada a sumarização dos métodos até o momento criados no passo FPGA, tal como uma breve explicação dos mesmos.

Um dos principais procedimentos realizados no passo FPGA e que caracteriza o objetivo deste trabalho, são os métodos de sincronização das instruções (*synchronizeInstructionsinC()* e *synchronizeInstructionsinLalp()*). Estes métodos são responsáveis por coletar todas as informações necessárias com o intuito de escalonar as instruções de uma estrutura de repetição, obtendo a ordem de execução de cada instrução considerando o menor custo. Este escalonamento é realizado de forma automática considerando o *merge* do DDG e CFG previamente extraídos e transformados, a tradução e modificação do grafo na sintaxe C para a sintaxe LALP para o EC-2 e por fim, o menor intervalo de iniciação (II) entre as instruções. Dessa forma, ao invocar o método de escalonamento das instruções seja para FSM ou LALP, todas suas dependências são executadas para que a nova estrutura seja montada usando os conceitos de *modulo scheduling*.

Resumo dos métodos do passo FPGA	
void	**rumFPGAPass()** - Executa os passos previamente determinados do Cetus, para a posterior análise e transformação do código C para FSM e LALP
void	**getProgram()** - Executa rotina para extrair o programa fonte completo
void	**getInfotoDDG()** - Executa a extração das informações dos arcos de dependência de dados analisados pelo Cetus
void	**getLoop()** - Identifica e extrai a estrutura de repetição do programa
void	**completeGraph()** - Executa duas rotinas internas e o *merge* para a obtenção do DDG completo em C: dfGraph() - Criar o grafo de fluxo de dados a partir das análises de fluxo de dados e fluxo de controle geradas pelo Cetus ddGraph() - Cria os vértices e arcos a partir das análises de dependências geradas pelo Cetus
void	**completeLalpGraph()** - Executa as mesmas rotinas que o método **completeGraph()**, mas para a obtenção do DDG completo em LALP
void	**dotExportCGraph(int dependenceType)** - Método que exporta o grafo em C para o formato Dot considerando o parâmetro: dependenceType = 0 (Default) - Imprime apenas as Dependências Verdadeiras dependenceType = 1 - Imprime todas as Dependências
void	**dotExportLalpGraph(int dependenceType)** - Executa as mesmas rotinas que o método **dotExportCGraph(int dependenceType)**, mas para a linguagem LALP
void	**extractStatementsinC()** - Método que extrai os *statements* e suas declarações do programa fonte em C
void	**exportLalpCode()** Método que exporta todo o grafo LALP analisado e transformado, para a versão texto seguindo a sintaxe da linguagem
void	**synchronizeInstructionsinC()** - Metodo que realiza o escalonamento das instruções em C, após sincronismo em *loop pipelining*
void	**synchronizeInstructionsinLalp()** - Método que realiza o escalonamento das instruções em LALP, após sincronismo em *loop pipelining* e requisitos impostos pela linguagem
void	**translateLalpGraph()** Método que realiza a tradução das instruções em C para as instruções em LALP

Tabela 4.3: Sumarização dos métodos do passo FPGA

Os detalhamentos destas e das demais funcionalidades descritas, serão realizados nas próximas seções.

4.2.3 Extensão do DDG para Inclusão de Novas Análises de Dependência de Dados

A criação de um DDG que suprisse as necessidades das arquiteturas de *hardware* customizado, não se limitou às informações extraídas de forma automática do compilador Cetus. Haja visto, os requisitos das arquiteturas alvo que o compilador deve atender, serem diferentes dos requisitos dos *hardware* customizado, tal como, os FPGAs. Dessa forma, várias análises e transformações no DDG extraído do compilador Cetus foram necessárias para a continuidade deste trabalho.

Ao decorrer da investigação das estruturas internas do compilador, foram identificadas as instâncias responsáveis pelas análises de testes das dependências de dados. A partir desta identificação, um conjunto de procedimentos foi realizado com o intuito de filtrar as informações para utilização no passo FPGA.

Contudo, foi observado que apenas estruturas de dados compostas (arranjos) eram analisadas pelo compilador, sendo desprezadas as estruturas simples. Ou seja, apenas o vetor de dependências era identificado, analisado e externalizado, sendo desconsiderada das análises as variáveis escalares. Característica essa que atende aos objetivos proposto pelo Cetus, mas é limitada para os objetivos deste trabalho.

Considerando as informações processadas pelo compilador, o vetor de dependência é capaz de identificar as instruções que possuem dependências entre si, identificar o tipo de dependência (*true, anti, in* e *out*) e calcular a distância entre elas. Na Listagem 4.13 é apresentado o código fonte utilizado para ilustrar a análise de dependências de dados.

Listagem 4.13: Código exemplo para ilustrar a análise de dependências.

```
1  int main()
2  {
3      int i;
4      int s=0;
5      int n = 100;
6      double a[n], b[n],c, k;
7      for (i=0; i<n; ++i)
8      {
9          c = a[i−2] + b[i−1];
10         a[i] = c ∗ 2;
11         b[i] = c ∗ 5;
12         s += c;
13         k = s;
14         s += k;
15     }
16 }
```

Com as informações extraídas do DDG do Cetus, um novo grafo foi criado contendo os vértices e seus respectivos arcos de dependências. A construção dos vértices considera os identificadores (*Vertex_Id*) e instruções (*statements*) extraídas. A partir das análises de dependência realizadas pelo compilador Cetus, foram identificados os vértices fonte e alvo (*source* e *target*), os tipos de dependências entre os vértices (*true, anti, in* e *out*) e suas respectivas distâncias, para que os arcos fossem criados.

Contudo é possível notar que as informações são incompletas, pois ao analisar o código da Listagem 4.13 apenas 3 das 6 instruções são processadas pelo compilador. O resultado desta análise pode ser mostrado no grafo da Figura 4.8. É importante ressaltar que para a criação deste novo grafo, são executados procedimentos para remover as arestas (ou arcos) duplicadas e incoerentes.

Observada esta característica das análises do compilador, o que representa uma limitação para a continuidade da proposta deste trabalho, foi necessária a retomada da investigação das estruturas internas do compilador, a fim de que pudessem ser extraídas também as variáveis escalares. Desse modo, foi identificada a estrutura que permitia acesso ao fluxo de controle do código fonte. A partir do acesso a esta informação, um novo grafo foi criado considerando todo o fluxo de controle do programa. Na Figura 4.9 é apresentado o CFG (*Control Flow Graph*) criado da Listagem 4.13. É importante ressaltar que para a criação do novo grafo, a exploração de cada elemento (variáveis e constantes) da instrução (vértice) se fez necessária

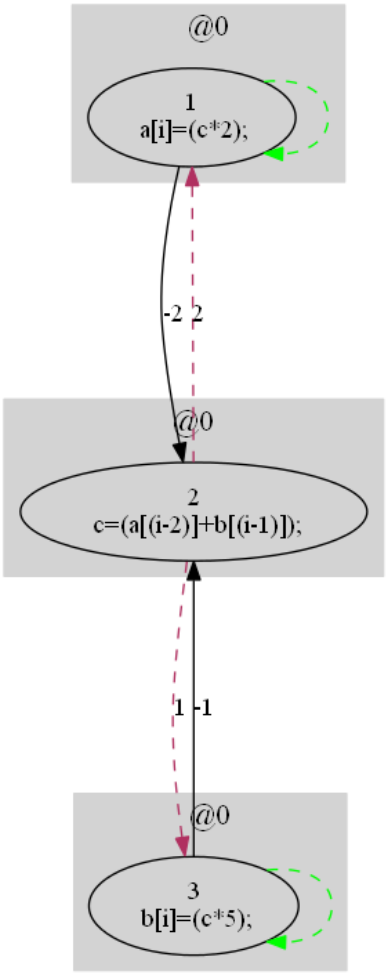

Figura 4.8: DDG criado a partir das análises automáticas de dependências de dados do Cetus.

para a determinação de cada dependências existente entre as instruções.

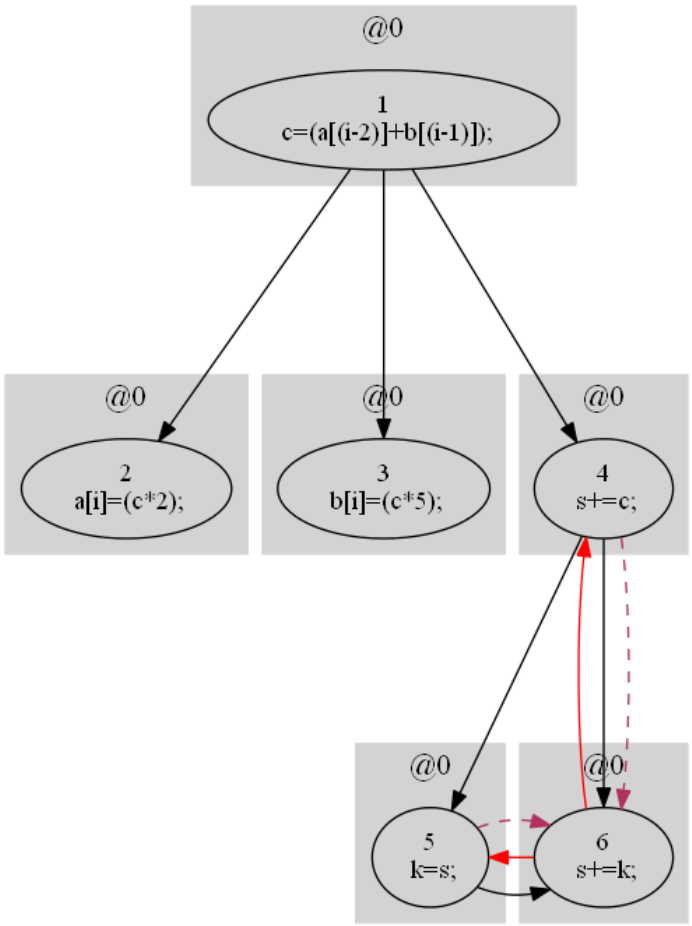

Figura 4.9: CFG criado a partir das análises do fluxo de controle do programa.

Considerando as informações isoladas dos dois grafos criados DDG e CFG, os objetivos deste trabalho não poderiam ser cumpridos. Todavia, serem informações do código que de nada serviriam separadamente. Portanto, a junção destes dois grafos possibilitou a exploração de todo o fluxo de controle do programa (código fonte) e dependências entre as instruções. Na Figura 4.10 é apresentada a junção (*merge*) do DDG mostrado na Figura 4.8 e do CFG mostrado na Figura 4.9.

Além do *merge* entre o DDG e o CFG, gerando um novo e completo grafo de dependência de dados contendo todas as dependências, tipos e distâncias, são executados também procedimentos para validar este grafo. Ao percorrer o DDG completo, são verificadas se as arestas respeitam o tipo de dependências existente entre as instruções. Caso exista alguma inconsistência, esta aresta é alterada para o tipo dependência inválida ou errada (*wrong dependence*) demonstrada na cor vermelho. Após a identificação das dependências erradas, o DDG completo é processado novamente para que todas as inconsistências sejam removidas, permitindo a continuidade dos objetivos deste trabalho. Vale ressaltar, que o DDG analisado refere-se apenas às instruções do corpo do *loop*.

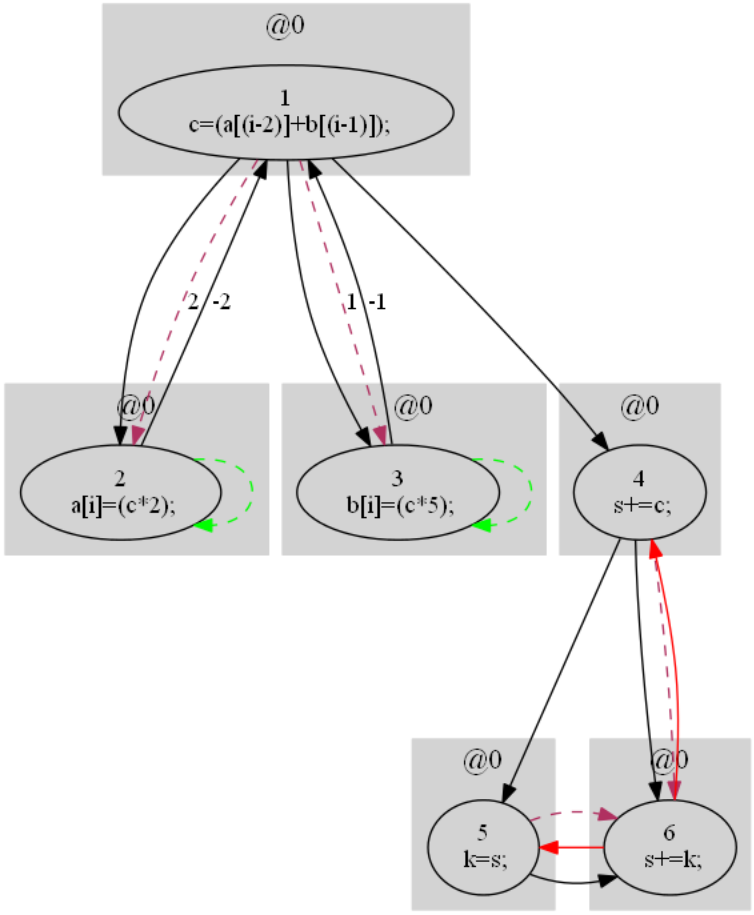

Figura 4.10: DDG completo criado a partir do *merge* entre DDG do Cetus e CFG.

4.2.4 Geração de Instrução *if-then-else*

Como citado na Subseção 2.4.1 do Capítulo 2, o *framework* LALP possui características que impuseram procedimentos específicos a este trabalho. Característica essa que transforma a estrutura condicional *if-then-else* em uma única instrução. Dessa forma, a partir da criação do CFG, anotações nos parâmetros dos vértices como mostrado na Figura 4.2 são realizadas com o objetivo de determinar a posteriori as instruções condicionais e suas respectivas instruções internas. A criação de instruções únicas para a condicional *if-then-else* também é utilizada para a geração de uma máquina de estados finitos, já que a estrutura condicional é problemática para o bom desempenho de *hardware* customizado.

Estas informações, direcionam a criação do DDG completo considerando a instrução condicional como sendo um único vértice no grafo. E as dependências com as instruções internas à estrutura condicional, passam a depender da instrução única criada. Na Listagem 4.14 é apresentado o código fonte utilizado para testar a criação do DDG completo considerando estruturas condicionais e a Figura 4.11 apresenta o DDG completo correspondente.

Listagem 4.14: Código utilizado para teste de estrutura condicional na geração do DDG.

```
1  int main()
2  {
3      int A[100];
4      int B[100];
5      int C[100];
6      int x1, x2, x3;
7      int i;
8      for (i=0; i<100; i++)
9      {
10         x1 = A[i];
11         x2 = B[i];
12         if(i<50)
13             x3= x1;
14         else
15             x3= x2;
16         C[i] = x3;
17     }
18     return 0;
19 }
```

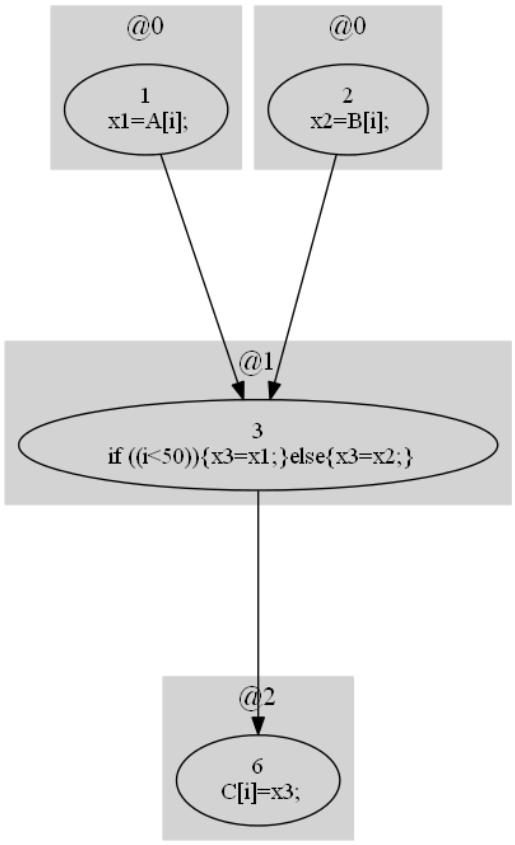

Figura 4.11: DDG completo considerando a estrutura *if-then-else*

4.2.5 Escalonamento das Instruções para *Loop Pipelining*

Com a criação do DDG completo a partir das análises e transformações da estrutura de repetição mais interna do código, foi dado início o processo de sincronismo das instruções. Este sincronismo é entendido como a ordenação dos vértices (instruções) do grafo com o objetivo de melhorar o aproveitamento do *pipelining* e o paralelismos entre as instruções. Este processo é utilizado tanto para o EC-1 como para o EC-2.

Para a realização do sincronismo das instruções, alguns procedimentos foram criados com o intuito de minimizar os esforços nas buscas pelo interior do grafo, o que poderia dispender alto custo de tempo nas análises. Dessa forma, em um primeiro momento são removidas do grafo as dependências que possam causar recursividade infinita na execução do algoritmo de busca. Logo, uma rotina é executada para determinar o melhor vértice do grafo para iniciar o procedimento de busca. Esta rotina considera os graus de entrada e saída dos vértices, a fim de selecionar o vértice mais ao topo do grafo sempre que possível, tornando o algoritmo de busca mais eficiente.

Em continuidade aos procedimentos que antecipam o escalonamento das instruções. São identificados dois tipos de grafos, os cíclicos e os acíclicos e para cada um deles um conjunto de procedimentos diferentes é utilizado. Desse modo, para cada tipo de grafo é identificado de forma automática a maneira mais eficiente para percorrer os vértices.

Com o DDG completo e processado para o sincronismo das instruções, é dado início o processo de escalonamento. O escalonamento das instruções deve respeitar incondicionalmente a não violação entre elas, pois esta é a característica que fundamenta o conceitos da técnica *loop pipelining*. Dessa forma, para alcançar o correto escalonamento, é necessário a obtenção de informações como as dependências de dados, as distâncias entre instruções e o intervalo de iniciação (II).

4.2.5.1 Cálculo do Intervalo de Iniciação

Como parte importante para o escalonamento das instruções, encontra-se o calculo do intervalo de iniciação. Este cálculo é responsável por determinar o salto entre as instruções de iterações diferentes. Desse modo, para calcular o intervalo de iniciação (II), foi utilizado o algoritmo conhecido como *Recurrence Minimum Initiation Interval* (RecMII) que considera os ciclos existentes no grafo para que o II seja encontrado. O RecMII pode ser demonstrado na expressão matemática a seguir (MOONEN; CATTHOOR, 1995), onde R é o ciclo do grafo de dependência, E o arco, T a latência e δ a distância entre os vértices.

$$recMII = \begin{cases} 0 & loop\,não\,tem\,recor. \\ \max\limits_{R \subseteq E} \dfrac{\sum\limits_{(u,v)\in R} T(u)}{\sum\limits_{(u,v)\in R} \delta(u,v)} & loop\,tem\,recorrência \end{cases} \tag{4.1}$$

A reprodução em forma computacional da expressão matemática citada, pode ser demonstrada no Algoritmo 1. Este algoritmo calcula para cada par de operações $(u,v) \in$ DDG o menor intervalo de tempo entre as operações de mesma iteração.

Algoritmo 1: Algoritmo para calcular o mínimo intervalo de iniciação.

```
Cálculo_RecMII (candidato)
II = candidato;
//Inicializa a matriz de distâncias com o delay
//mínimo entre os pares de dependência u e v
forall the operacao u ∈ DDG do
    forall the operacao v ∈ DDG do
        MinDist[u,v] = - ∞;
        forall the edge e(u,v) ∈ DDG do
            MinDist[u,v] = max(MinDist[u,v], (λe - II * δe));
//Considera todos os caminhos possíveis passando pelo nó w
forall the operacao w ∈ DDG do
    forall the operacao u ∈ DDG do
        forall the operacao v ∈ DDG do
            dist = MinDist[u,w] + MinDist[w,v];
            if dist > MinDist[u,v] then
                MinDist[u,w] = dist;
            if u == v and dist > 0 then
                return 0;
return 1;
```

As estruturas de dados do algoritmo consistem nas operações u e v, no *delay* ou latência λ, no candidato ao mínimo intervalo de iniciação II e na distância δ. Sendo que a principal estrutura do algoritmo é a matriz de distâncias $MinDist[u,v]$.

A entrada $[u,v]$ especifica o mínimo intervalo entre as operações de u e v. Se não houver caminho de u para v no DDG, o valor da entrada $[u,v]$ é definido como -∞. Se o valor de $MinDist[u,u]$ for positivo para qualquer u, isso significa que u deve ser programado depois dele mesmo, o que é impossível e indica que o II é pequeno e não satisfaz. Por outro lado, se todas as entradas diagonal forem negativas, indica uma falta em torno de todo o circuito, resultado de um II mais elevado do que o necessário. Dessa forma, o objetivo do algoritmo consiste

em encontrar o mínimo II para entradas positivas e pelo menos uma entrada igual a zero, na diagonal.

4.2.5.2 Escalonamento das Instruções

Ao fim de todas as análises e processamento das informações do DDG completo e acesso ao valor mínimo do II, é iniciado o procedimento de escalonamento das instruções. Este procedimento utiliza uma busca em profundidade (*Depth-first search*) no grafo, ou seja, a partir do vértice raiz é explorado cada um de seus ramos até que não seja mais identificados sucessores para este vértice (nó filho). Assim, a busca retrocede (*backtrack*) e começa no próximo vértice.

As informações necessárias para que o procedimento de escalonamento das instruções seja realizado, foram identificadas e tratadas ao longo de todo o processo evolutivo do trabalho. Desse modo, com o II é possível identificar o intervalo mínimo de iniciação que será utilizado e com o DDG completo, o acesso as dependências de dados entre as instruções e suas respectivas distâncias. A Listagem 4.15 apresenta o cálculo realizado para a obtenção do escalonamento das instruções do grafo percorrido.

Listagem 4.15: Cálculo para obtenção do escalonamento das instruções do DDG completo.

```
1 syncB = syncA + 1 − ( II ∗ |(distAB| )
```

Ao percorrer o DDG completo, cada vértice pertencente ao arco de análise ($edge(A,B)$) é processado e recebe o sincronismo devido. Sendo *syncB* o sincronismo do vértice *B*, *syncA* o sincronismo de seu predecessor $\in edge(A,B)$, II o intervalo mínimo de iniciação e *distAB* a distância entre as instruções *A* e *B*. Vale ressaltar que para cada novo sincronismo atribuído à instrução, é executado um processo de validação para que violações entre as dependências não ocorram.

Como citado no início da Subseção 4.2.5, são removidas do DDG completo as dependências que possam causar recursividade infinita na execução do algoritmo de busca. Ou seja, o DDG completo é submetido a um filtro que remove as dependências que possam causar problemas no escalonamento das instruções. Na Figura 4.12 é apresentado o DDG completo contendo todas as dependências (*true, anti, in, out* e *wrong*) em seguida na Figura 4.13 o DDG completo após o filtro executado para que o grafo seja analisado e submetido ao escalonamento. Ambos os grafos foram gerados a partir do código fonte da Listagem 4.13 apresentada na Subseção 4.2.3.

Em continuidade aos exemplos que demonstram a evolução do DDG partindo do *merge* entre DDG e CFG e filtro das dependências para o escalonamento das instruções. Na Figura 4.14 é apresentado o DDG escalonado segundo a técnica *loop pipelining* e considerando o II obtido de forma automática igual a 2.

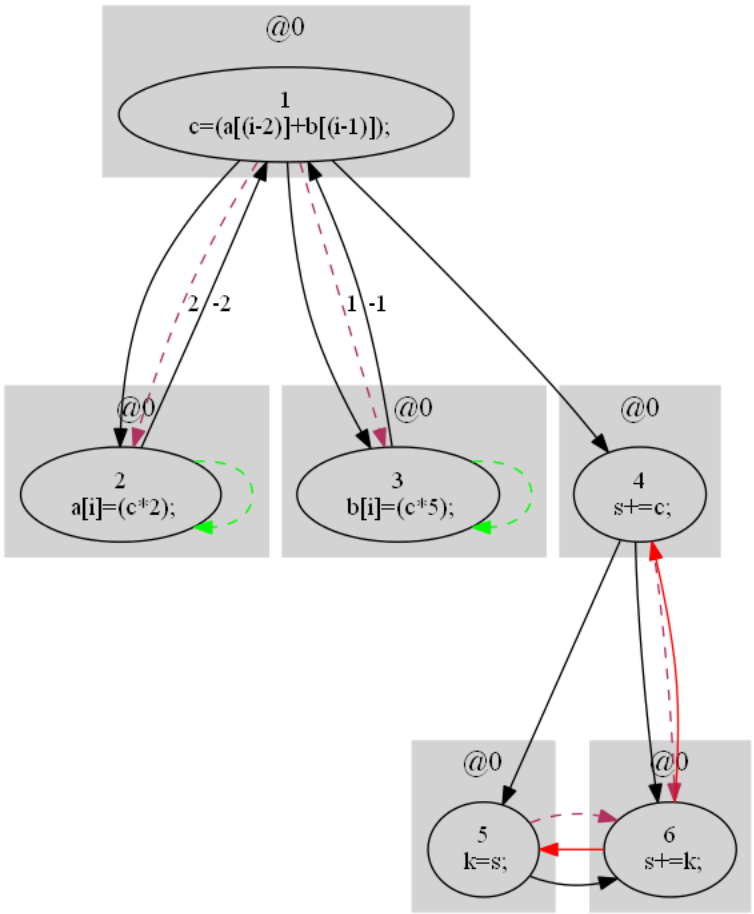

Figura 4.12: DDG completo criado a partir do *merge* entre DDG e CFG

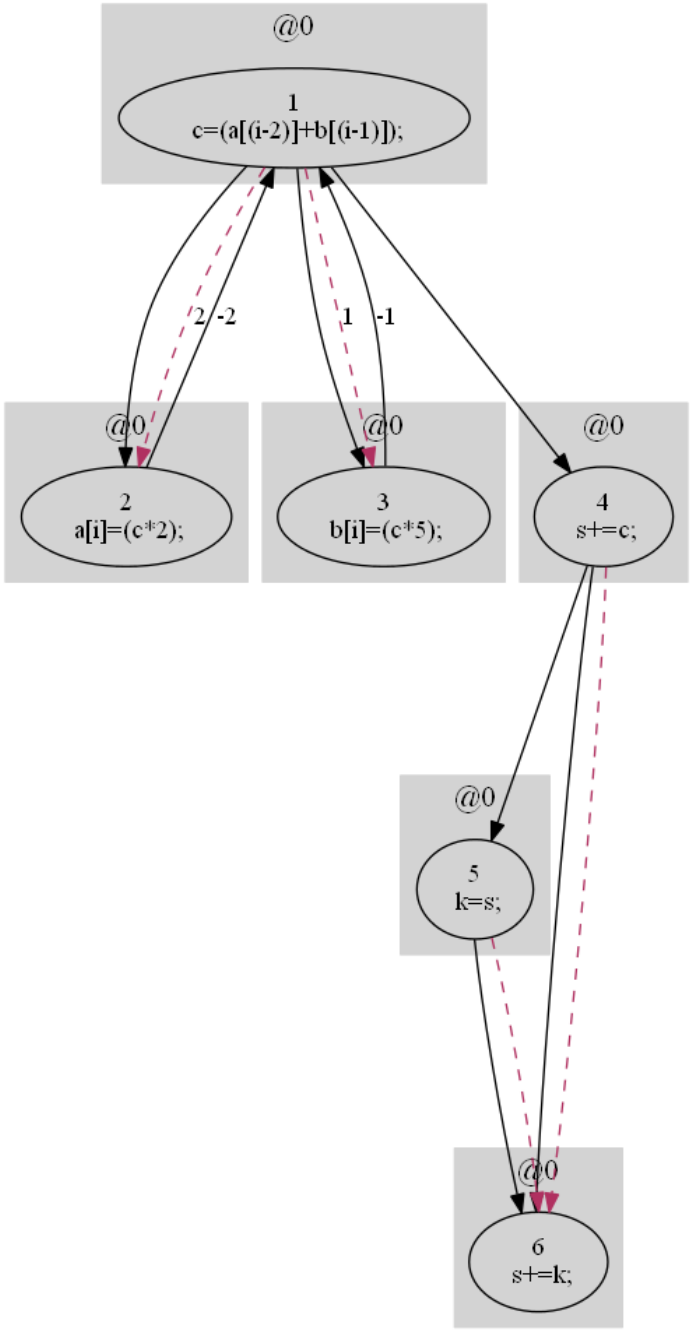

Figura 4.13: DDG completo submetido ao filtro de dependências.

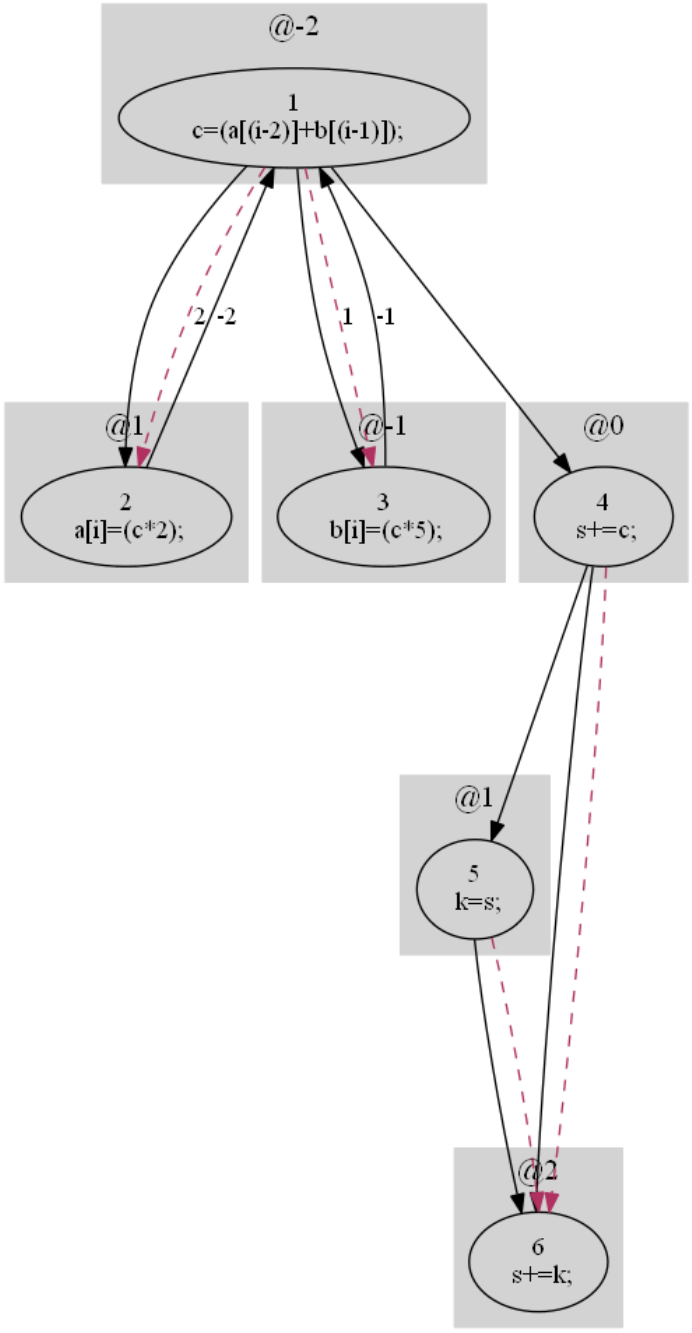

Figura 4.14: DDG escalonado.

Em suma, o resultado de todas as análises e transformações realizadas, é o escalonamento em *loop pipelining* das instruções internas da estrutura de repetição processada e a exportação do grafo em formato Dot, para sua posterior plotagem utilizando ferramentas como Graphviz. É apresentado na Listagem 4.16 o corpo de um *loop* escrito em C. E logo, na Listagem 4.17 este mesmo código é apresentado após ser processado e escalonado. Nota-se que a mudança realizada no código fonte processado, acrescenta a diretiva @ seguida de um valor numérico. Esta informação é responsável por dizer a ordem de execução das instruções, respeitando as dependências e possibilitando a otimização na execução do programa.

Listagem 4.16: Código C original do corpo de um *loop*.

```
1 ...
2 for ( i=0; i<n; ++i ){
3  a[i] = 10;
4  b[i] = a[i−2]∗2;
5  c[i] = b[i−1]+5;
6 }
7 ...
```

Listagem 4.17: Código C do corpo de um *loop* após o escalonamento.

```
1 ...
2 for ( i=0; i<n; ++i ){
3  a[i] = 10; @0
4  b[i] = a[i−2]∗2; @3
5  c[i] = b[i−1]+5; @2
6 }
7 ...
```

A Figura 4.15 apresenta o grafo plotado da estrutura de repetição processada. Estrutura essa que contem dependências de dados entre iterações diferentes, o que no grafo é demostrado nas distâncias plotadas em cada aresta correspondente. A coloração e estilo linear entre os arcos, informam o tipo de dependência existente, no caso o $arco(1,2)$ possui uma dependência verdadeira com distância igual a -2, o $arco(2,1)$ uma anti dependência com distância igual a 2, o $arco(2,3)$ uma dependência verdadeira com distância igual a -1 e o $arco(3,2)$ uma anti dependência com distância igual a 1. Por fim, a cada vértice é atribuído o valor do sincronismo calculado e representado com a diretiva @ mais o valor do sincronismo. Em seguida é demostrada na Figura 4.16 a mostra que todas as dependências são honradas do escalonamento destas instruções considerando o intervalo de iniciação (II) igual a 2.

Vale ressaltar que todos estes procedimentos são realizados de forma automática. Dessa

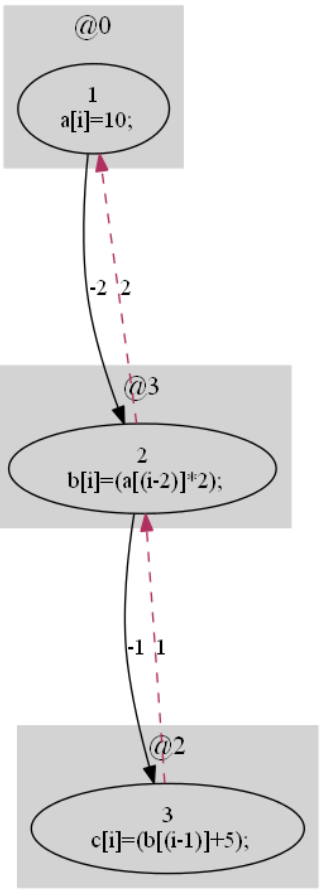

Figura 4.15: DDG escalonado.

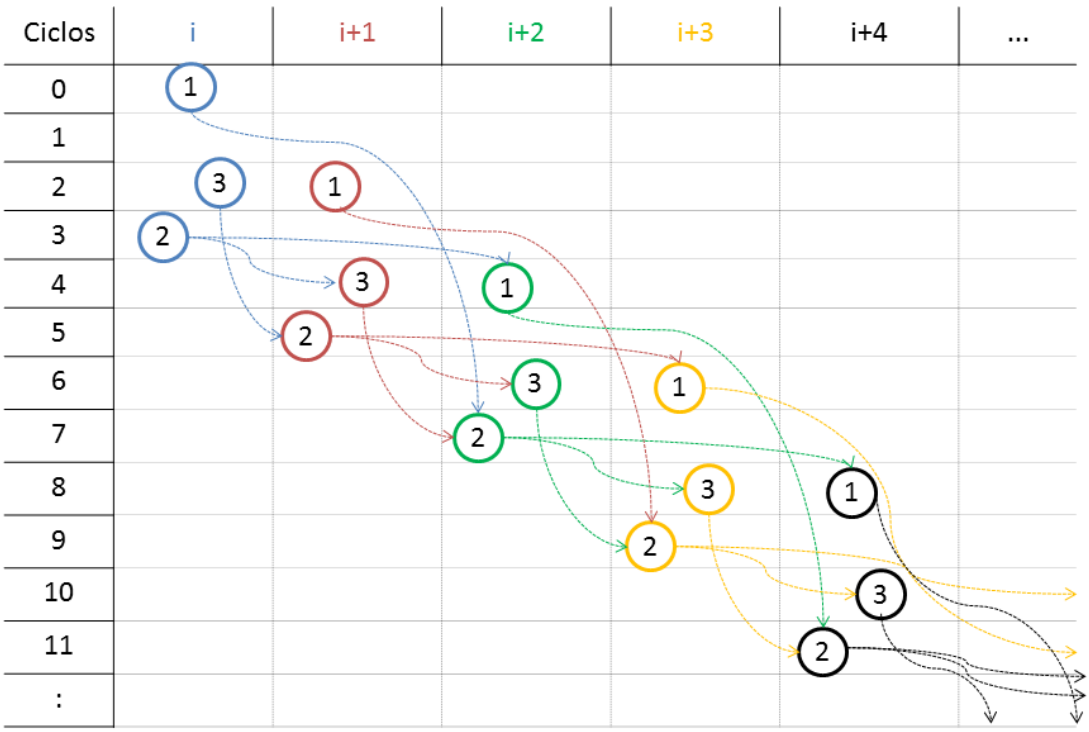

Figura 4.16: Teste de escalonamento em *loop pipelining* do DDG.

forma, o necessário é selecionar o código fonte de entrada (escrito em C), especificar o nome da função[2] para o qual se quer estes tipos de análises e os demais passos até a plotagem do grafo escalonado, é realizado de maneira automatizada pelo conjunto de procedimentos criados.

4.2.6 Geração de Código Fonte LALP

Ao término de todas as análises e transformações realizadas com o objetivo de gerar um novo e completo DDG. Deu-se início o EC-2 que corresponde a fase de geração de um novo bloco de código fonte na linguagem LALP. A seguir são apresentados os passos/modificações específicos para a geração de código LALP:

1. Tradução do DDG completo para a sintaxe LALP.
2. Modificação e reconstrução do DDG para que novas estruturas fossem criadas respeitando as necessidades da linguagem.
3. Escalonamento das instruções.
4. Escalonamento especial das instruções que fazem acesso à memória.
5. Normalização do sincronismo criado para as instruções.
6. Geração do bloco de código na sintaxe LALP.
   (a) Criação das declarações das variáveis.
   (b) Criação do inicializador do contador da iteração.
   (c) Criação do cabeçalho da estrutura de repetição.
   (d) Criação do contador da estrutura de repetição.

As modificações apresentadas serão detalhadas a seguir utilizando um exemplo a fim de demonstrar cada passo/modificação realizados. Desse modo, para a obtenção do bloco de código LALP o primeiro passo foi traduzir o DDG completo para a sintaxe LALP seguindo as anotações descritas em cada vértice, como apresentado na Figura 4.2 da Subseção 4.1.1. Sendo executada uma rotina transformando o grafo na sintaxe C para um novo grafo na sintaxe LALP.

A partir da inserção do código fonte, o trecho de código de interesse do trabalho (*loop*) é isolado de forma automática, para que as análises, visando o sincronismo das instruções, sejam realizadas. O código fonte completo é mostrado a Listagem 4.18, tal como o DDG do corpo do *loop* na Figura 4.17. Vale destacar que o código apresentado não realiza cálculos sofisticados ou mesmo representa algo conhecido na literatura, ele representa apenas uma série de atribuições

[2]Função que contenha o *loop*. Visto ser o foco deste trabalho.

a vetores. Pois o que se pretende não é testar o desempenho do algoritmo e sim o acessos à memória e as dependências entre as instruções.

Listagem 4.18: Código C inserido no Cetus modificado.

```
1  int main() {
2   int i;
3   int s = 0;
4   int n = 5;
5   int a[n],c[n],e[n];
6   for (i = 0; i < n; i++) {
7       c[i] = i + 1;
8       a[i] = c[i] + 2;
9       e[i] = a[i];
10      s += e[i];
11  }
12 }
```

Na sequência um conjunto de procedimentos é realizado visando a geração de um novo grafo traduzido para a sintaxe LALP. Este DDG na linguagem LALP é apresentado na Figura 4.18. Note que este novo grafo não foi apenas traduzido, e sim modificado e reconstruído para que novas estruturas fossem criadas respeitando as necessidades da linguagem. Razão pelo o qual o escalonamento não é realizado neste momento. Haja visto, o correto escalonamento das instruções, necessitar que todas as estruturas do grafo tenham sido criadas.

Desse modo, duas novas estruturas são criadas quando vetores são manipulados. Sendo, uma correspondendo a atribuição da posição do endereço de memória a uma variável auxiliar e a outra estrutura à conexão entre o vetor e esta variável auxiliar. Nesta ordem é possível identificar no grafos as variáveis auxiliar nas instruções 6,8,10,12,14,16 em seguida a conexão entre vetor e posição de memória atribuída às variáveis auxiliares é mostrada nas instruções 7,9,11,13,15,17. As instruções de conexão não são conectadas ao grafo, pois não necessitam ser sincronizadas e sim apenas declaradas no código fonte LALP.

Por característica da linguagem LALP as instruções que fazem acesso à memória necessitam ser endereçadas antes da execução de suas operações, sendo que este endereçamento deve respeitar a distância de dois ciclos de *clock* até que esta informação esteja disponível e possa ser utilizada pela instrução. A partir do DDG na sintaxe LALP, o grafo é submetido ao processo de escalonamento das instruções como citado ao longo da Subseção 4.2.5 e normalizado em situações que o sincronismo é menor que zero. Este grafo escalonado considerando as características citadas, é apresentado na Figura 4.19.

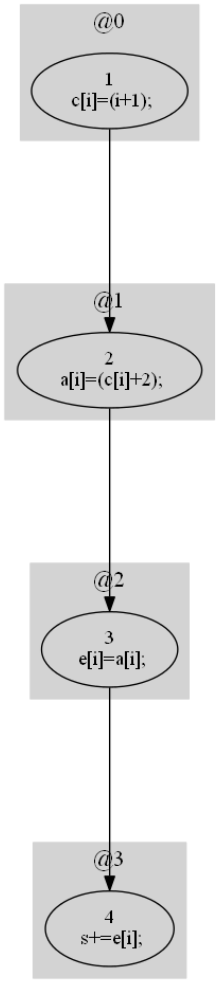

Figura 4.17: DDG referente a Listagem 4.18.

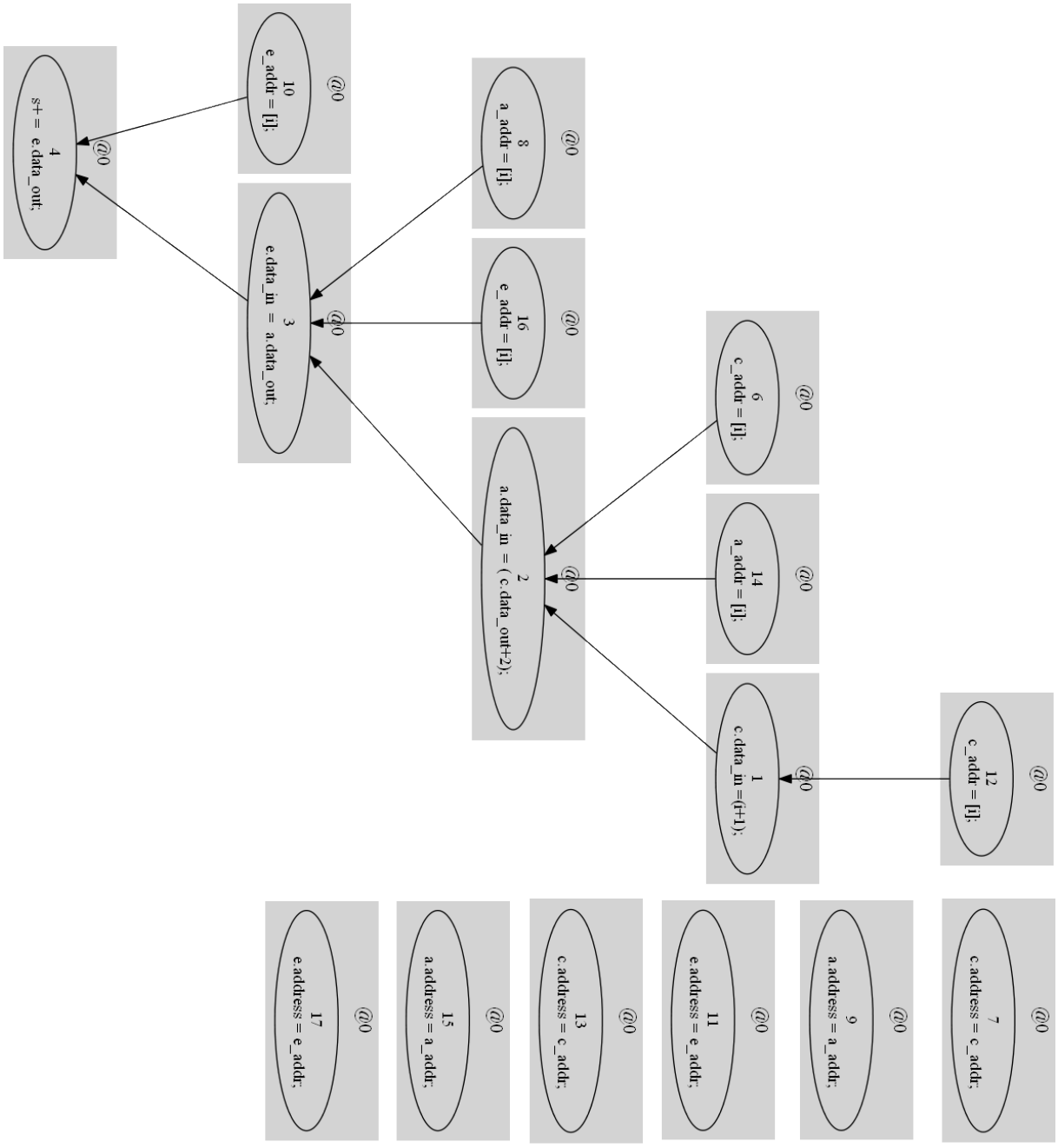

Figura 4.18: DDG reconstruído a partir do DDG em C Figura 4.17, para gerar um novo DDG na sintaxe LALP.

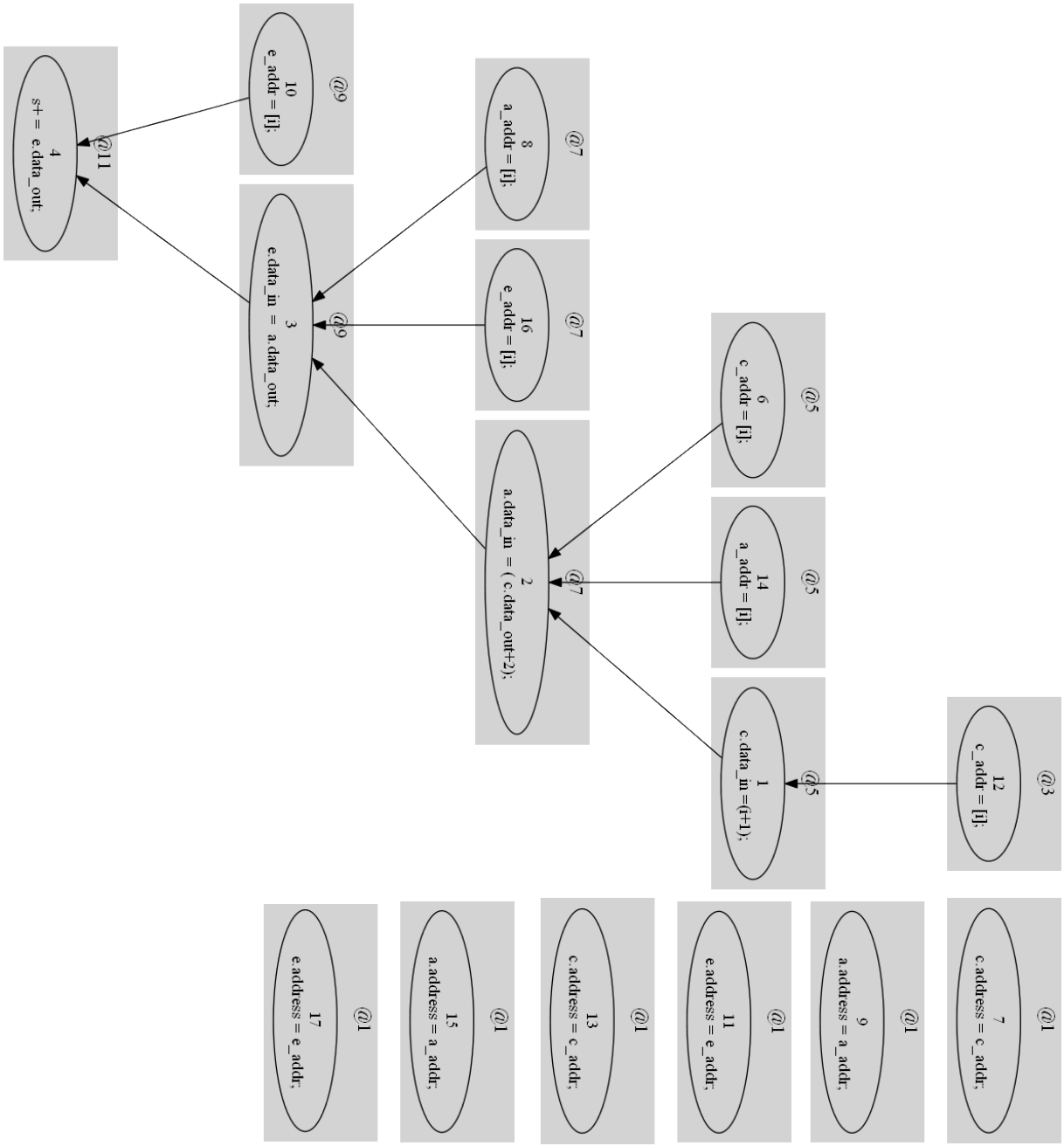

Figura 4.19: DDG LALP após procedimento de escalonamento das instruções.

É possível observar que as instruções 6,8,10,12,14 e 16 do DDG LALP, que correspondem às variáveis auxiliares de endereçamento, são sincronizadas dois ciclos antes dos respectivos vetores, haja visto a característica da linguagem. As instruções 1,2,3 e 4 correspondem as mesmas instruções 1,2,3 e 4 do código C original representado no DDG da Figura 4.17, contudo traduzidas para a sintaxe LALP. Estas instruções possuem a mesma restrição no momento do sincronismo, sendo que o intervalo entre uma instrução e outra respeita dois ciclos de *clock* de distância. O motivo está na disponibilidade do dado na memória, ou seja, como a instrução 2 consome a produção da instrução 1, a instrução 2 deve esperar dois ciclos de *clock* para que este dado esteja disponível na memória e assim a produção da instrução 2 possa ser feita. De forma similar, todo este processo ocorre também ao longo das instruções 3 e 4, pois o consumo da instrução sucessora depende da produção da instrução predecessora.

O próximo passo é gerar o bloco de código na sintaxe LALP, sendo que este bloco corresponde a junção do código fonte e o DDG LALP. O código fonte é avaliado novamente, pois neste estágio são necessárias as declarações das variáveis utilizadas. Desse modo, é resgatado parte do código e traduzido para a sintaxe LALP.

Com as declarações já processadas, outra função é necessária para compor o bloco de código a ser criado. Da mesma forma que no código fonte C, o código LALP deve conter o cabeçalho da estrutura de repetição, contudo também traduzido para a sintaxe adequada, ou seja, o cabeçalho da estrutura de repetição (*loop*) é criado alterando o comando em C, $for(inicializa\_variavel\_iterador; condicional; incremento)$; para o comando na sintaxe LALP, $Counter(inicializa\_variavel\_iterador; condicional; incremento@controle\_de\_iteracao)$. Sendo que a identificação da variável utilizada no cabeçalho, deve ser criada antes do comando de repetição em LALP. Esta estrutura conecta o iterador da repetição ao ciclo de *clock* e é declarada da seguinte forma $variavel\_iterador.clk\_en = init;$. Ao término da criação desta estrutura inicial do bloco de código LALP, o DDG LALP é ordenado e impresso na sequência. O resultado da geração do bloco de código LALP, a partir do DDG mostrado na Figura 4.19 e código fonte apresentado na Listagem 4.18 é apresentado na Listagem 4.19.

Listagem 4.19: Código LALP gerado a partir do DDG mostrado na Figura 4.19 e código fonte apresentado na Listagem 4.18.

```
[FPGA] Code Lalp exported:
 {
  int i;
  int s = 0;
  int n = 5;
  int a[n], c[n], e[n];
  int a_addr;
  int c_addr;
  int e_addr;
 }
 i.clk_en = init;
 counter(i=0; i<n; i++@11);
  a_addr = i;
  c_addr = i;
  e_addr = i;
  a.address = a_addr;
  c.address = c_addr;
  e.address = e_addr;
  c.data_in =(i+1) when i.step@5;
  a.data_in = ( c.data_out+2) when i.step@7;
  e.data_in = a.data_out when i.step@9;
  s+ = e.data_out when i.step@11;
```

Note que algumas das informações existente no DDG LALP não são impressas. Isso acontece, pois para o correto escalonamento de todo o grafo, é necessário que as instruções estejam disponíveis para que o algoritmo possa percorre-las. Contudo para a geração de código LALP, algumas características exigem a re-avaliação e adequação do código. O exemplo mostrado na Listagem 4.19, corresponde a um algoritmo onde o cálculo das operações ocorrem dentro da mesma iteração, ou seja, não existe manipulação de posições da memória diferentes do escopo do iterador. Desse modo, não é necessário dizer às variáveis auxiliares o momento que devem acessar a memória, já que acessam a mesma posição do iterador *i*.

O bloco de código LALP gerado de forma automática mostrado na Listagem 4.19, não corresponde ao código final do algoritmo na linguagem. Portanto, é necessária a intervenção do programador LALP neste estágio. Intervenção essa, que determinará as estruturas que não foram criadas do bloco automático e as adequações para uma melhor utilização do *hardware*. Haja visto, a preocupação inicial do processo automático ser em gerar um bloco de código

LALP como referência para o programador, garantindo a correta execução das instruções considerando o maior intervalo do conjunto de instruções, e não a geração de um código otimizado.

A intervenção do programador LALP se da em definir o cabeçalho do algoritmo e a alguns refinamentos ao longo do código. A Listagem 4.20 apresenta o código LALP modificado pelo programador, a partir do bloco de código LALP gerado de forma automática.

Listagem 4.20: Intervenção do programador LALP ao código gerado de forma automática como mostrado na Listagem 4.19.

```
1  const DATA_WIDTH = 32;
2  const n = 5;
3  typedef fixed(DATA_WIDTH, 1) int;
4  typedef fixed(1, 0) bit;
5  teste8(in bit init) {
6   {
7    int i;
8    int s = 0;
9    int a[n]={0};
10   int c[n]={0};
11   int e[n]={0};
12   int a_addr;
13   int c_addr;
14   int e_addr;
15  }
16  i.clk_en = init;
17  counter(i=0; i<n; i++@11);
18   a_addr = i;
19   c_addr = i;
20   e_addr = i;
21   a.address = a_addr;
22   c.address = c_addr;
23   e.address = e_addr;
24   c.data_in =(i) + 1 when i.step@5;
25   a.data_in = (c.data_out + 2) when i.step@7;
26   e.data_in = a.data_out when i.step@9;
27   s += e.data_out when i.step@11;
28 }
```

As intervenções do programador podem ser notadas nas linha 1 a 2, onde são declaradas as constantes usadas para o número de bits de cada valor e para o número de iterações, respectivamente. Na linha 3 é definido um tipo de dado usado para 32 bits de ponto fixo com sinal e na

linha 4 um tipo de um único bit para sinais de controle. Por fim, a linha 5 inicia o nome que será usado na criação da entidade em VHDL, seguido de sinais de entrada e saída desta entidade. As demais linhas foram geradas de forma automática e neste caso inalteradas pelo programador.

Desse forma, em continuidade a explicação o bloco que vai da linha 6 até a linha 15 contém as declarações das variáveis escalares e arranjos, sendo que o programador LALP interviu apenas na inicialização dos vetores. A linha 16 corresponde ao sinal externo de inicialização *init* que é usado para habilitar a contagem. A linha 17 é responsável por iniciar as instruções com o contador, sendo que a diretiva @11 indica que o componente irá gerar um novo valor para *i* a cada ciclo de *clock*. As linhas 18 a 20 indicam o armazenamento do endereço de memória às variáveis auxiliares. As linhas 21 a 23 realizam a conexão do endereçamento realizado às variáveis auxiliares aos respectivos vetores. As linha 24 a 27 descrevem as operações principais do código, que por sua vez, inicia com a atribuição após 5 ciclos de *clock* o valor $i+1$ ao vetor *c*, por característica da linguagem o valor desta atribuição só se torna disponível após 2 ciclos de *clock* e por isso o vetor *a* recebe o valor da saída de $c+2$ ($c.data_out+2$) no *step* 7. Como o vetor *e* recebe a saída do valor *a* e a variável *s* a saída do vetor *e*, as mesmas características de acesso à memória são utilizadas, fazendo com que os sincronismos estre estas instruções respeitem o intervalo de dois ciclos de *clock*.

No decorrer dos estudos, foi observado que a diretiva *counter* do LALP possui três funções: controlar a iteração, indexar a memória e definir o sincronismo das instruções. Desse modo, foram definidos 3 grandes perfis de código fonte que se enquadram nas funções desempenhadas pela diretiva:

1. Código fonte que não possui acesso à memoria (uso de vetores).
2. Código fonte que possui acesso a uma única posição de memória durante a iteração (a[i]).
3. Código fonte que possui acesso a mais de uma posição de memória durante a iteração (a[i-1]).

O presente trabalho buscou cobrir as situações em que se enquadram os perfis de código fonte 1 e 2, pois o sincronismo calculado é capaz de controlar a iteração (não de forma ótima) e sincronizar as instruções internas. Já o perfil de código 3 não relaciona apenas a problemática sincronismo e controle da iteração, e sim também a indexação de memória. Caso este, não tratado neste momento, já que foge ao foco do trabalho. Pois a preocupação se concentra no sincronismo das instruções em *modulo scheduling*, desconsiderando parâmetros de acesso à memória como requisitado pelo *framework* LALP.

É importante destacar, que todo o processo de checagem dos códigos LALP gerados de forma automática e as intervenções para a realização dos testes de execução dos códigos completos, que serão apresentados no Capítulo 5, foram realizados em conjunto com o doutorando

Cristiano (OLIVEIRA,) da Universidade de São Paulo em São Carlos. Seu projeto é intitulado: *Combinando Transformações de Computação e de Dados para Arquiteturas Reconfiguráveis de Grão Fino*,consiste em em pesquisar e desenvolver técnicas para geração de códigos otimizados, considerando aspectos relativos ao mapeamento, gerenciamento e acesso à memória em arquiteturas reconfiguráveis, uma vez que a latência para leitura e escrita de dados têm sido um gargalo para aplicações de alto desempenho.

Capítulo 5

RESULTADOS E CONTRIBUIÇÕES

Este capítulo expõe os resultados dos testes realizados para os estudos de caso EC-1 e EC-2, tal como, as contribuições e limitações impostas pelo processo automatizado.

De forma introdutória aos testes realizados, os *benchmarks* podem ser divididos em vários tipos, segundo (BERRY; CYBENKO; LARSON, 1991). Os principais tipos são listados a seguir:

- Sintético: são aqueles em que nenhuma computação útil é realizada no código, ou seja, não representa nenhuma aplicação real, serve somente para exercitar alguns componentes básicos da arquitetura alvo.
- *Kernel*: é a utilização de parte do código que concentra a maior parte da computação como *benchmark*. São bastante interessantes por sua simplicidade e pequeno tamanho.
- Algoritmo: são códigos bem definidos pela literatura, podem implementar métodos conhecidos em computação numérica, resolução de equações lineares, até filtros em imagens.
- Aplicação: são programas completos, que resolvem problemas científicos bem definidos.

Grande parte dos *benchmark* utilizados por este trabalho para testes, correspondem ao tipo Sintético. Pois, em um primeiro momento o objetivo é considerar o fluxo das instruções, as dependências, as iterações e o comportamento em situações diversas. Alguns testes também foram realizados considerando os tipos *Kernel* e Algoritmo, mas não de forma exaustiva e por isso serão tratados como trabalhos futuros.

5.1 Estudo de Caso 1: FSM

Uma FSM é composta por estados e transições, como dito na seção correspondente do Capítulo 2. A identificação destas estruturas a partir da infraestrutura criada, pode ser observada

nos itens seguintes, tal como seu correspondente diagrama de estados na Figura 5.1:

- Número de Estados: Número de ciclos (1/ciclo) do *scheduling*.
- Conteúdo do Estado: As instruções geradas no escalonamento para o ciclo específico.
- Transições:
  - Condição sempre verdadeira do estado 0 ao estado $n-1$.
  - No estado n se condição for $i < n$, volta para o estado 0.
  - No estado n se condição for $i = n$, vai para o estado $final$ (termina o *loop*).

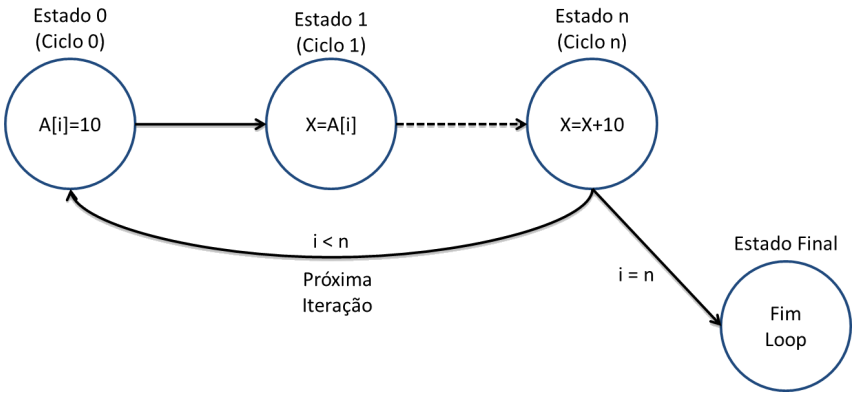

Figura 5.1: Representação em diagrama de estados.

A criação de uma FSM implementando *loop pipelining* é um processo difícil para o projetista de *hardware*, já que existem preocupações como dependências, sincronismos e outros detalhes.

O trabalho desenvolvido para a extensão do compilador Cetus permite a geração automática de estados e transições implementando a técnica *loop pipelining*. Isso é feito a partir da tabela de escalonamento, onde as instruções de cada ciclo correspondem às instruções de um dado estado. Sendo geradas de forma automática as transições entre os ciclos, com exceção do último estado, onde é feito o teste do contador do *loop*.

Utilizando como exemplo o trecho de código fonte já escalonado considerando o II calculado igual a 2, apresentado na Listagem 5.1. É apresentado na sequência, de forma reduzida a Tabela 5.1 com o escalonamento das instruções em função das transições, sendo destacado na cor azul o *kernel* desta execução.

Listagem 5.1: Código C do corpo de um *loop* após o escalonamento, considerando o II calculado igual a 2.

```
1 ...
2 for ( i=0; i<10; ++i ){
3   a[i] = 10; @0
4   b[i] = a[i−2]∗2; @3
5   c[i] = b[i−1]+5; @2
6 }
7 ...
```

	Ciclos	i	i+1	i+2	...
	0	1			
	1				
kernel {	2	3	1		
kernel {	3	2			
repetição do *kernel* para outras interações {	4		3	1	
repetição do *kernel* para outras interações {	5		2		
	:			:	

Tabela 5.1: Funções de transição do algoritmo escalonado apresentado na Listagem 5.1

A partir da tabela com as funções de transição considerando as dependências e sincronismos, é possível construir uma máquina de estados correspondente. Na Listagem 5.2 são apresentadas as instruções por estado, após o cálculo para determinar o número de estados, instruções e iterações correspondentes ao prólogo, *kernel* e epílogo. Sendo apresentado na Figura 5.2 a representação desta FSM na forma de diagrama de estados. Note que no *kernel*, são necessários apenas dois estados para que todas as instruções sejam executadas. É importante ressaltar a necessidade em criar um *buffer*, para guardar valores produzidos e ainda não consumidos, de uma mesma instrução, garantindo a correta execução da maquina de estados finitos.

Listagem 5.2: Instruções por estado, após escalonamento e cálculo do numero de estados.

```
1  ---------Prologue---------
2  Instructions per state
3  State 0:
4  | Interaction || Instruction
5  |       0       || 1 - a[i]=10;
6  ---------Kernel---------
7  Instructions per state
8  State 1:
9  | Interaction || Instruction
10 |    (n + 0)    || 3 - c[i]=(b[(i-1)]+5);
11 |    (n + 1)    || 1 - a[i]=10;
12 |    (n + 0)    || 2 - b[i]=(a[(i-2)]*2);
13 Number of Repetitions: 9
14 ---------Epilogue---------
15 Instructions per state
16 State 2:
17 | Interaction || Instruction
18 |       9       || 3 - c[i]=(b[(i-1)]+5);
19 State 3:
20 | Interaction || Instruction
21 |       9       || 2 - b[i]=(a[(i-2)]*2);
```

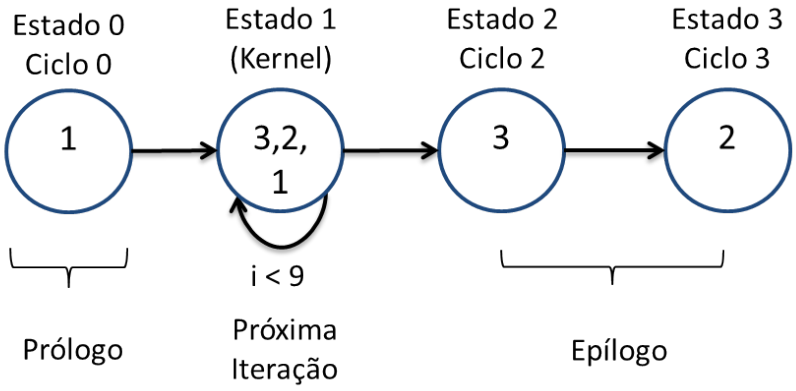

Figura 5.2: Diagrama de estados criado a partir da Tabela 5.1.

Outro exemplo da obtenção de uma FSM a partir das análises e transformações realizadas pela infraestrutura criada, pode ser acompanhado considerando o trecho de código já escalonado e com o II igual a 1, apresentado na Listagem 5.3. E na sequência, de forma reduzida a Tabela 5.2 com o escalonamento das instruções em função das transições, sendo destacado também na cor azul o *kernel* desta execução.

Listagem 5.3: Código C do corpo de um *loop* após o escalonamento, considerando o II calculado igual a 1.

```
...
for (i=n; i<10; i++)
{
 c = a + 10; @0
 a = c * 2; @1
 b = c * 5; @1
 s += c; @1
 k = s; @2
 s += k; @3
}
...
```

	Ciclos	i	i+1	i+2	i+3	...
	0	1				
	1	2,3,4	1			
	2	5	2,3,4	1		
kernel {	3	6	5	2,3,4	1	
	4		6	5	2,3,4	1
repetição do *kernel* para outras interações {	5			6	5	2,3,4
	6				6	5
	7					6
	:			:		

Tabela 5.2: Funções de transição do algoritmo apresentado na Listagem 5.3

Com a tabela das funções de transição criada, é possível construir uma máquina de estados correspondente. Na Listagem 5.4 são apresentadas as instruções por estado, após o cálculo para determinar o número de estados, instruções e iterações correspondentes ao prólogo, *kernel* e epílogo. Sendo apresentado na Figura 5.3 a representação desta FSM na forma de diagrama de estados. Note que no *kernel*, é necessário apenas um estado para que todas as instruções sejam executadas. É importante ressaltar a necessidade em criar um *buffer*, para guardar valores produzidos e ainda não consumidos, de uma mesma instrução, garantindo a correta execução da maquina de estados finitos.

Listagem 5.4: Instruções por estado, após escalonamento e cálculo do numero de estados.

```
---------Prologue---------
Instructions per state
State 0:
| Interaction || Instruction
|       0       || 1 - c=(a+10);
```

```
State 1:
| Interaction || Instruction
|       0       || 2 - a=(c*2);
|       0       || 3 - b=(c*5);
|       0       || 4 - s+=c;
|       1       || 1 - c=(a+10);
State 2:
| Interaction || Instruction
|       0       || 5 - k=s;
|       1       || 2 - a=(c*2);
|       1       || 3 - b=(c*5);
|       1       || 4 - s+=c;
|       2       || 1 - c=(a+10);
---------Kernel----------
Instructions per state
State 3:
| Interaction || Instruction
|    (n + 0)    || 6 - s+=k;
|    (n + 1)    || 5 - k=s;
|    (n + 2)    || 2 - a=(c*2);
|    (n + 2)    || 3 - b=(c*5);
|    (n + 2)    || 4 - s+=c;
|    (n + 3)    || 1 - c=(a+10);
Number of Repetitions: 7
---------Epilogue----------
Instructions per state
State 4:
| Interaction || Instruction
|       7       || 6 - s+=k;
|       8       || 5 - k=s;
|       9       || 2 - a=(c*2);
|       9       || 3 - b=(c*5);
|       9       || 4 - s+=c;
State 5:
| Interaction || Instruction
|       8       || 6 - s+=k;
|       9       || 5 - k=s;
State 6:
| Interaction || Instruction
|       9       || 6 - s+=k;
```

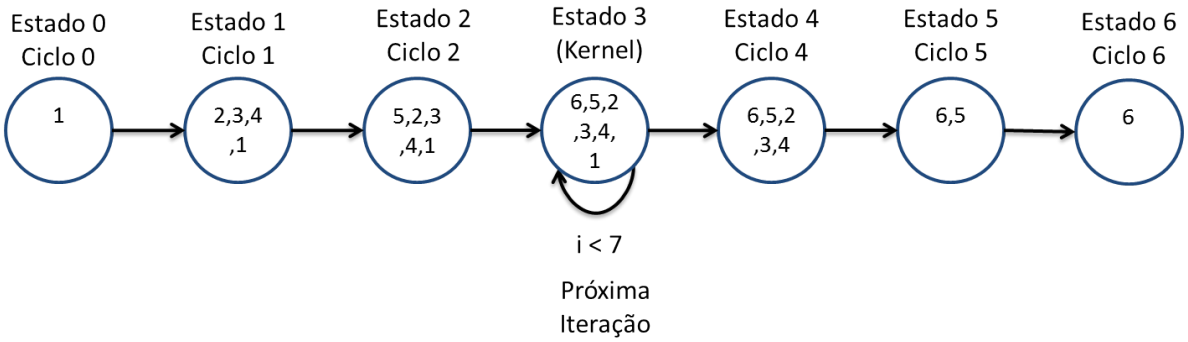

Figura 5.3: Diagrama de estados criado a partir da Tabela 5.2.

A característica que denota melhor desempenho na criação das FSM em ambos os exemplos, está diretamente ligada com o mínimo intervalo de iniciação (II) calculado para a realização dos escalonamentos. Ou seja, a máquina de estados gerada a partir da infraestrutura desenvolvida, terá o modelo com o melhor desempenho possível, já que é considerado para o cálculo do escalonamento das instruções, o menor intervalo de iniciação entre elas sem que ocorra violações.

5.1.1 Limitações na Geração de uma FSM

A geração automática de uma máquina de estados finitos a partir do código fonte escrito na linguagem C, impõem alguns limites para a melhor exploração de cada estado. Desse modo, a seguir são apresentados pontos que impedem a geração eficiente de uma FSM, tal como, pontos não tratados neste momento:

- Estruturas com ponteiros não são utilizadas.
- Necessária a criação de um *buffer*, para armazenar os valores produzidos e ainda não consumidos, de uma mesma instrução.
- *Loops* aninhados requerem mais testes e podem apresentar resultados inesperados.
- Necessário identificar uma função que contenha a estrutura (*loop*) que será processada.
- Para melhor exploração de uma FSM a partir de código C, recomenda-se simplificar as instruções em código de três endereços.

5.2 Estudo de Caso 2: LALP

O segundo estudo de caso, trata da utilização do compilador como auxiliar na geração de código LALP. Os detalhes sobre estes procedimentos, encontram-se descritos na Subseção 4.2.6

do Capítulo 4. Com o processo criado e testado, é iniciada a fase de obtenção dos resultados e avaliação da infraestrutura desenvolvida. Tendo dito no inicio deste capítulo os tipos de *benchmarks* e as justificativas de uso ou não por este trabalho, são apresentados os resultados dos testes realizados em três exemplos de códigos fontes.

5.2.1 Exemplo de código: Algoritmo Fibonacci

A seguir é apresentado o Algoritmo de Fibonacci escrito na linguagem C na Listagem 5.5 e seu correspondente em LALP, com os devidos ajustes realizados pelo programador, na Listagem 5.6. Vale enfatizar que a geração automática de bloco de código LALP calculou e inseriu todos os @, como detalhado na Seção 4.2.6 do Capítulo 4.

Listagem 5.5: Algoritmo de Fibonacci em C.

```
1  #define N 32
2  int main() {
3    int v[N] = {0, 1};
4    int i;
5    int temp1, temp2, temp3;
6    temp2= v[0];
7    temp1= v[1];
8    temp3= temp1 + temp2;
9    for (i=2; i<N; i++) {
10     v[i] = temp3;
11     temp2= temp1;
12     temp1= temp3;
13     temp3= temp1 + temp2;
14   }
15   return v[i];
16 }
```

Listagem 5.6: Algoritmo de Fibonacci criado a partir de bloco de código LALP gerado de forma automática.

```
const DATA_WIDTH = 32;
typedef fixed(DATA_WIDTH, 1) int;
typedef fixed(1, 0) bit;
fibonacciComReuso(in bit init) {
 {
  int v[32] = { 0, 1 } ;
  int i;
  int temp1 = 1;
  int temp2 = 0;
  int temp3 = 0;
  int v_addr;
 }
 i.clk_en = init;
 counter(i=2; i<32; i++@5);
  v_addr = i;
  v.address = v_addr;
  temp2=temp1 when i.step@1;
  v.data_in =temp3 when i.step@2;
  temp1=temp3 when i.step@3;
  temp3=(temp1+temp2) when i.step@5;
}
```

Como forma de verificar a correta execução do código gerado, é apresentado na Figura 5.4 o resultado da sintetização do código em forma de ondas. Com a execução deste processo, a linguagem LALP é analisada e sintetizada para a linguagem de descrição de *hardware* VHDL e em seguida a prova da execução do código é tirada na simulação utilizando o programa GTKWave, *software* este, já integrado ao *framework* LALP para depuração do código VHDL gerado pela ferramenta.

É possível acompanhar uma sequência de redimensionamentos até a seleção de cada saída de variável em forma de ondas e seus respectivos valores em cada ciclo de *clock*. Sendo que para cada variável analisada, a referida instrução que a envolve é apresentada em cima de sua representação em forma de ondas. Desse modo, as variáveis temporárias e o vetor são inicializados respectivamente com 0 e 1 e as atribuições às operações são realizadas. Note que até o inicio da atribuição de $temp1$ e $temp2$ à variável $temp3$ o intervalo é maior do que as demais operações. Isso acontece pois com as dependências entre as instruções, são necessários 5 ciclos de *clock*, para que os devidos valores estejam acessíveis e o cálculo seja realizado. Outro

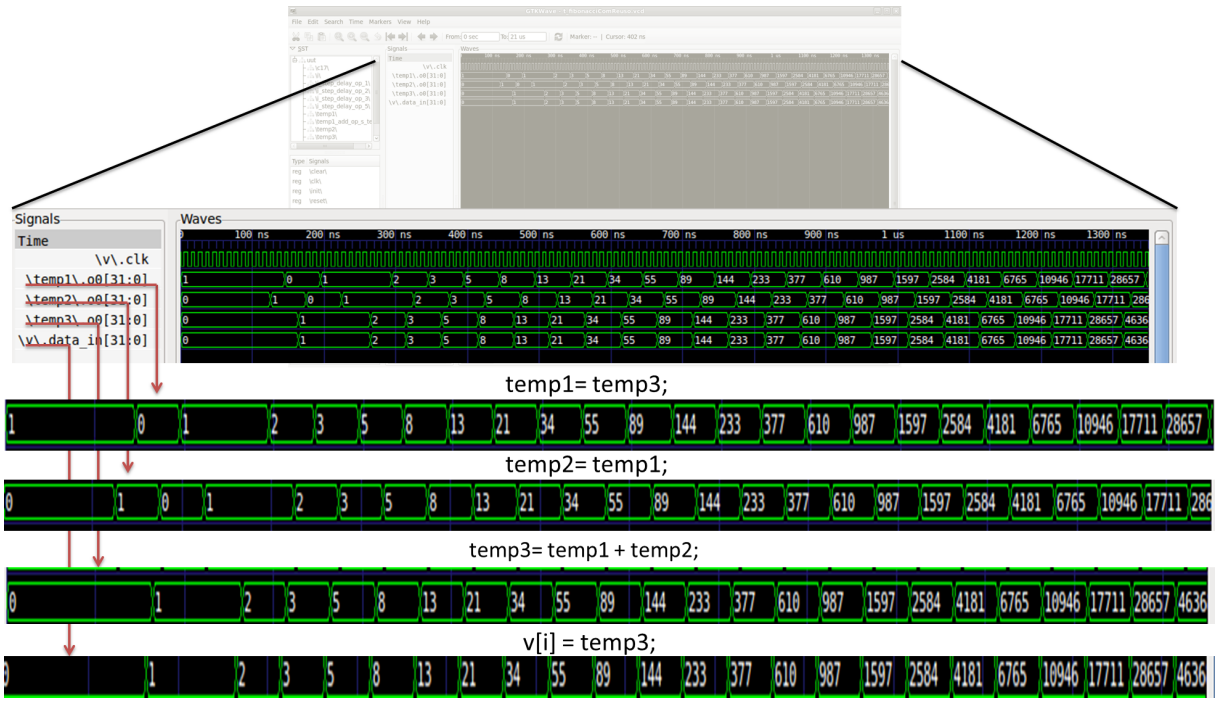

Figura 5.4: Resultado da sintetização do Algoritmo Fibonacci gerado de forma automática a partir da Listagem 5.6.

ponto a ser observado, são os resultados armazenados em $temp3$ e replicados para $v.data\_out$ ($v[i] = temp3$). Este procedimento permite verificar que o vetor está andando de forma correta segundo o escalonamento criado.

5.2.2 Exemplo de código: Algoritmo para Manipulação de Vetores

Outro exemplo de código executado e sintetizado, demostrando a situação onde é feita a manipulações de dados entre vetores, é apresentado na Listagem 5.7 em linguagem C e seu correspondente em LALP, com os devidos ajustes realizados pelo programador, na Listagem 5.8.

Listagem 5.7: Exemplo de código com manipulação de vetores em C.

```
1  int main()
2  {
3      int i;
4      int s=0;
5      int n = 10;
6      int a[n], b[n], c[n], k[n];
7      for ( i=0; i<n; ++i )
8      {
9          a[i]= 2*i;
10         c[i]= a[i];
11         b[i]= a[i]*c[i];
12         k[i] = c[i] + b[i];
13     }
14 }
```

Listagem 5.8: Código criado a partir de bloco de código LALP gerado de forma automática.

```
const DATA_WIDTH = 32;
const n = 10;
typedef fixed(DATA_WIDTH, 1) int;
typedef fixed(1, 0) bit;
teste4(out int result, in bit init) {
 {
  int i;
  int s = 0;
  int a[n]={0};
  int b[n]={0};
  int c[n]={0};
  int k[n]={0};
  int a_addr;
  int b_addr;
  int c_addr;
  int k_addr;
 }
 i.clk_en = init;
 counter(i=0; i<n; i++@2); //@11−Detectado de forma automática
  a_addr = i;
  b_addr = i;
  c_addr = i;
  k_addr = i;
  a.address = a_addr;
  b.address = b_addr;
  c.address = c_addr;
  k.address = k_addr;
  a.data_in = (2∗i) when i.step@5;
  c.data_in = a.data_out when i.step@7;
  b.data_in = (a.data_out ∗ c.data_out) when i.step@9;
  k.data_in = (c.data_out + b.data_out) when i.step@11;
  result = k.data_out when i.step@13;
}
```

É possível observar que a partir do bloco de código LALP gerado de forma automática, o programador não apenas completou o código como demonstrado na Subseção 4.2.6 do Capítulo 4. Mas sim otimizou o código, ao perceber que o *counter* poderia segurar a iteração por 2 ciclos ao invés de 11 como determinado de forma automática, sem que o resultado sofresse alguma

alteração. Desse modo, a sintetização do código considerando a otimização realizada pelo programador, é apresentada na Figura 5.5.

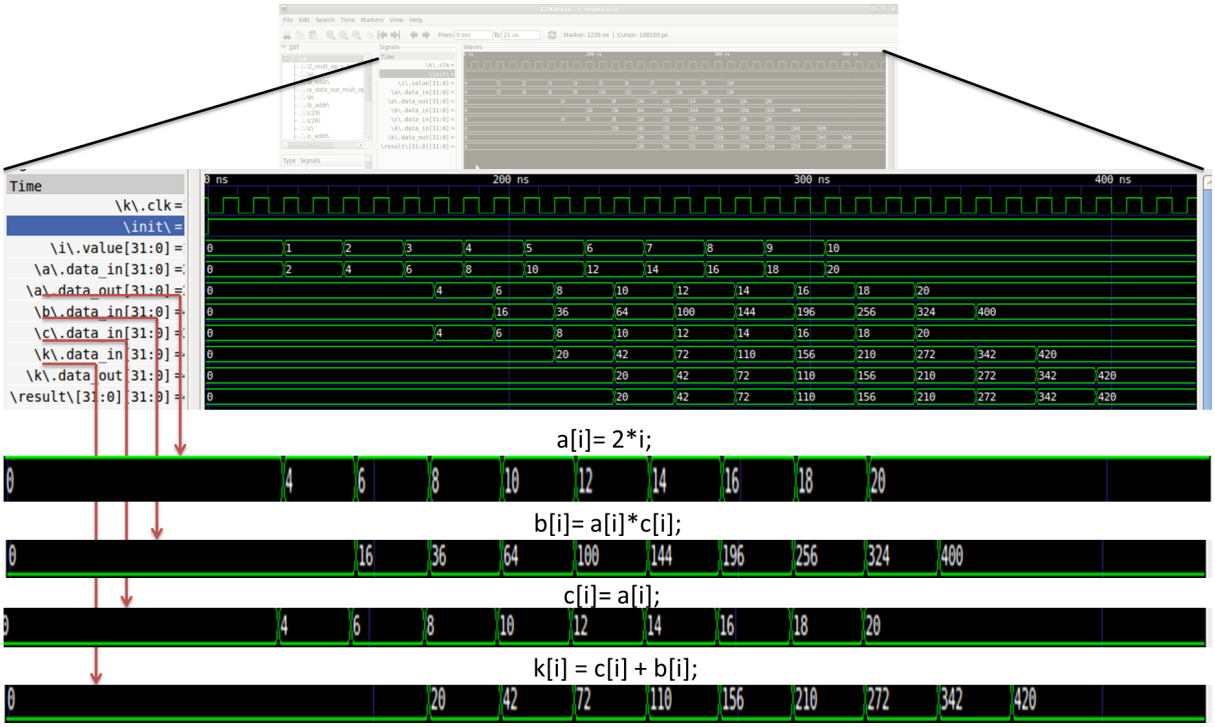

Figura 5.5: Resultado da sintetização do código gerado de forma automática a partir da Listagem 5.8.

5.2.3 Exemplo de código: Algoritmo para Manipulação de Vetores e Escalares

Outro exemplo de código executado e sintetizado, demostrando a situação onde é feita a manipulações de dados entre vetores e variáveis escalares é apresentado na Listagem 5.9 em linguagem C e seu correspondente em LALP, com os devidos ajustes realizados pelo programador, na Listagem 5.10.

Listagem 5.9: Exemplo de código com manipulação de vetores e escalares em C.

```
1  int main()
2  {
3      int A[5] = {0,2,4,6,8};
4      int B[5] = {1,3,5,7,9};
5      int C[5] = {0,0,0,0,0};
6      int x1, x2,x3,x4,x5,x6;
7      int i;
8      for (i=0; i<5; i++)
9      {
10         x1= A[i];
11         x2= B[i];
12         x3= x1 + x2;
13         x4= x3 * 4;
14         x5= x3 * x4;
15         x6= x5 * 2;
16         C[i]= x6;
17     }
18 }
```

Listagem 5.10: Código criado a partir de bloco de código LALP gerado de forma automática.

```
const DATA_WIDTH = 32;
typedef fixed(DATA_WIDTH, 1) int;
typedef fixed(1, 0) bit;
teste10(in bit init) {
 {
  int A[5] = {1,3,5,7,9};
  int B[5] = {0,2,4,6,8};
  int C[5] = {0,0,0,0,0};
  int x1=0;
  int x2=0;
  int x3=0;
  int x4=0;
  int x5=0;
  int x6=0;
  int i;
  int A_addr;
  int B_addr;
  int C_addr;
 }
 i.clk_en = init;
 counter(i=0; i<5; i++@13); //@2-Sugestão de otimização do programador
  A_addr = i + 1;
  B_addr = i + 1;
  C_addr = i + 1;
  A.address = A_addr;
  B.address = B_addr;
  C.address = C_addr;
  x1 = A.data_out;
  x2 = B.data_out;
  x3 = (x1+x2);
  x4 = (x3*4);
  x5 = (x3*x4);
  x6 = (x5*2);
  C.data_in =x6;
}
```

Como o código possui apenas uma dissipação dos dados entre as variáveis e só realiza um *store* no final, os sincronismos das instruções podem ser desprezados já que o próprio *counter*

é capaz de controlar as iterações até que a atribuição ao vetor $C.data\_in = x6$ ($C[i] = x6$) seja realizada. Neste exemplo o programador detectou uma possível otimização do código, ao perceber que o *counter* poderia segurar a iteração por 2 ciclos ao invés de 13 como determinado de forma automática, sem que o resultado sofresse alguma alteração. Contudo o valor foi mantido para demostrar que o resultado não é alterado e sim o prolongamento do sinal inicial. Desse modo, a sintetização do código considerando o *counter* superestimado é apresentada na Figura 5.6.

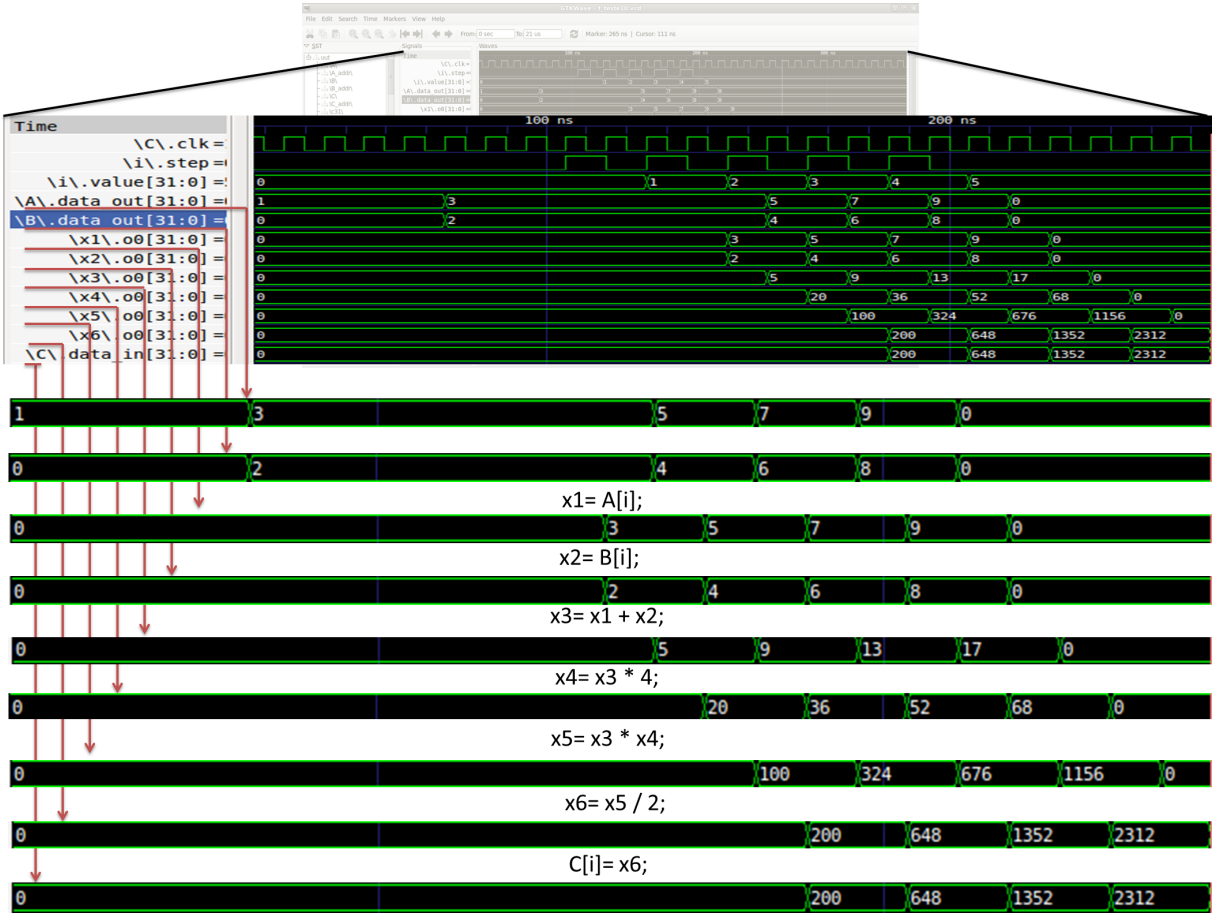

Figura 5.6: Resultado da sintetização do código gerado de forma automática a partir da Listagem 5.10.

As duas primeiras instruções da figura, mostram as saídas dos vetores $A.data\_out$ e $B.data\_out$ ($A[i]$ e $B[i]$) definidos estaticamente no inicio do código, e em seguida as demais instruções são executadas normalmente. O que se deve notar na execução destas instruções, é o intervalo de tempo para seu início. Como o contador definido de forma automática considerou o controle de iteração igual a 13, as instruções demoram 13 ciclos para que as atribuições sejam iniciadas. É portando, que a intervenção do programador para ajustes e otimizações no código se fazem necessários, para a obtenção de um melhor desempenho do *hardware* gerado.

5.2.4 Limitações na Geração de Código LALP

A geração automática de código LALP completo, não caracteriza o objetivo deste trabalho e sim o estudo de caso da linguagem e *framework* LALP, fornecendo facilitadores para sua utilização. Desse modo, a seguir são apresentados pontos que impedem a geração automática de código LALP, tal como, pontos não tratados neste momento:

- Estruturas com ponteiros não são utilizadas.
- *Loops* aninhados requerem mais testes e podem apresentar resultados inesperados.
- Necessário identificar uma função que contenha a estrutura (*loop*) que será processada.
- LALP apresenta algumas características específicas para tratar o acesso à memória. Estas não foram abordadas a priori por este trabalho.
- Código fonte que possui acesso a mais de uma posição de memória durante a iteração (a[i-1]), não foram tratados.
- A diretiva *counter* do LALP possui três funções: controlar a iteração, indexar a memória e definir o sincronismo das instruções. Este trabalho se prestou a solucionar o sincronismo das instruções e parcialmente o controle de iteração.

Vale enfatizar que algumas dessas restrições são comuns em ferramentas que objetivam a geração de *hardware* a partir de código fonte na linguagem C.

5.3 Contribuições

As contribuições relacionadas aos estudos de caso EC-1 e EC-2, são consideradas a seguir:

- Estudo de Caso 1: FSM
  - Mecanismo para auxiliar a geração de FSM para *hardware* customizado.
  - FSM criada a partir desta infraestrutura, tende a ter desempenho otimizado, já que o mínimo II é calculado com precisão e geralmente atingido pelo processo e escalonamento implementado.
  - FSM criada a partir desta infraestrutura, pode ser aplicada em projetos diversos de *hardware* customizado.
- Estudo de Caso 2: LALP

- Ferramenta para facilitar/automatizar a programação LALP a partir de código C, respeitando as limitações impostas pela linguagem LALP e as características do *hardware* "*target*" considerado por essa linguagem.
- Apesar de as análises e transformações não gerarem o código exatamente como em LALP, o ponto crítico de sua programação foi resolvido. Sendo a determinação automática e precisa das dependências de dados e o escalonamento de instruções na modalidade de *loop pipelining*. O que antes exigia do programador conhecimento aprofundado na linguagem e arquitetura alvo.
- A viabilidade de integração do compilador Cetus modificado ao *framework* LALP, mostrou-se promissora.

Capítulo 6

CONCLUSÃO

Soluções computacionais baseadas em microprocessadores podem não ser a melhor alternativa em determinadas situações. Isso se deve a questões como desempenho, custo, consumo de energia e área física utilizada. Outro fator importante, são que os limites no aumento do *clock* de processadores levam a busca por soluções alternativas para o aumento de desempenho. Portanto, uma das alternativas ao uso dos microprocessadores é a utilização total ou parcial de *hardware* customizado.

Ao decorrer dos estudos realizados ao longo deste trabalho, foi observada a necessidade de sistemas de compilação que auxiliem os programadores a desenvolver *hardware* customizado mais facilmente. Isso é necessário pois a utilização direta de linguagens de descrição de *hardware* como VHDL e Verilog, demandam conhecimento específico em sistemas digitais, que por sua vez apresentam uma curva de aprendizado relativamente acentuada, o que justifica todos os esforços na busca por facilitar a programação destes dispositivos.

Foi portanto neste contexto que o presente trabalho dedicou esforços. Tendo como objetivo a criação de uma infraestrutura de compilação para a geração automática (ou semi-automática) de aceleradores em *hardware* customizado, facilitando de alguma forma a programação nestes dispositivos.

Dentro dos objetivos propostos por este trabalho, criou-se um módulo de compilação capaz de tratar as análises e transformações a partir de código C. É possível afirmar que o uso do compilador Cetus para a implementação de uma infraestrutura de auxílio à compilação de *hardware* customizado, mostrou-se bem sucedido. Já que os estudos de caso realizados, apresentaram resultados satisfatórios e viabilizarão o desenvolvimento de pesquisas futuras.

Contudo, a partir dos estudos realizados, dificuldades também foram apontadas na utilização do compilador Cetus, já que para a manipulação do DDG gerado é necessário entender a lógica computacional deste processo, tal como suas estruturas internas. Algumas das análises do compilador foram consideradas insuficientes, e portanto modificações visando a comple-

mentação destas análises tiveram de ser desenvolvidas para dar continuidade ao trabalho.

A utilização da infraestrutura criada para a geração completa de uma FSM, como apresentado no EC-1, será um novo trabalho de mestrado já iniciado. A utilização do EC-2 mostrou-se complexa o bastante para que a geração completa de código LALP, não fosse realizada também por este trabalho. Mas as questões chaves que antes ficavam a cargo do programador, agora são resolvidas automaticamente pelo compilador Cetus modificado. Pois o bloco de código fonte gerado, considerando os perfis tratados, apresentam poucas diferenças do código LALP criado manualmente, simplificando e agilizando a tarefa do programador.

6.1 Trabalhos Futuros

A infraestrutura de compilação desenvolvida neste trabalho oferece oportunidades para a realização de outras pesquisas na mesma linha, pois as análises e transformações realizadas no código fonte fornecem informações relevantes para sua posterior manipulação e adequação para *hardware* customizado. A seguir, são relacionadas algumas sugestões para trabalhos futuros que podem aproveitar e estender os avanços obtidos nesta pesquisa.

- Continuar os testes para avaliar a robustez do compilador modificado e propor melhorias como:
  - Criação de procedimento para suporte de *loop unrolling* e geração de *loop pipelining* e *loop unrolling* também para *loops* aninhados.
  - Simplificação de forma automática das instruções do código fonte em C para código de três endereços.
  - Utilizar a infraestrutura criada como base para uma ferramenta específica para a geração de FSMs a partir de código C.
- Integração total ao *framework* LALP possibilitando a transformação automática de código C para código LALP.
  - Avaliar possíveis modificações de sintaxe, inclusão de novos recursos ao *framework* LALP.

REFERÊNCIAS

AHO, A. *Compilers: principles, techniques, & tools*. [S.l.]: Addison-wesley, 2007.

AIKEN, A.; NICOLAU, A. Perfect pipelining: A new loop parallelization technique. In: SPRINGER. *ESOP'88*. [S.l.], 1988. p. 221–235.

ALLAN, V. et al. Software pipelining. *ACM Computing Surveys (CSUR)*, ACM, v. 27, n. 3, p. 367–432, 1995.

ASSUMPÇÃO, J. J. *State of the art in FPGA technology*. 2010. Acessado em 02/2012. Disponível em: <http://www.merlintec.com/download/jecel_osso.pdf>.

BAXTER, R. et al. High-performance reconfigurable computing–the view from edinburgh. In: *Proc. AHS2007 Conf., Second NASA/ESA Conference on Adaptive Hardware and Systems*. [S.l.: s.n.], 2007.

BERRY, M.; CYBENKO, G.; LARSON, J. Scientific benchmark characterizations. *Parallel Computing*, Elsevier, v. 17, n. 10, p. 1173–1194, 1991.

BLAKE, G.; DRESLINSKI, R.; MUDGE, T. A survey of multicore processors. *Signal Processing Magazine, IEEE*, IEEE, v. 26, n. 6, p. 26–37, 2009.

BOBDA, C. *Introduction to Reconfigurable Computing: Architectures, algorithms and applications*. [S.l.]: Springer Verlag, 2007.

BRANDÃO, G.; ROSÁRIO, D. *Desenvolvimento e aplicações de sistemas embarcados em FPGA*. Tese (Doutorado) — Universidade Federal do Rio Grande do Norte, 2010.

CABRERA, D. et al. OpenMP extensions for FPGA Accelerators. In: IEEE. *Systems, Architectures, Modeling, and Simulation, 2009. SAMOS'09. International Symposium on*. [S.l.], 2009. p. 17–24.

CARTER, N. *Arquitetura de Computadores*. [S.l.]: Bookman, 2002.

CASTILLO, R. et al. Towards a versatile pointer analysis framework. *Euro-Par 2006 Parallel Processing*, Springer, p. 323–333, 2006.

CHAN, P.; MOURAD, S. *Digital system design using field programmable gate arrays*. [S.l.]: PTR Prentice Hall, 1994.

COMPTON, K.; HAUCK, S. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (CSUR)*, ACM, v. 34, n. 2, p. 171–210, 2002.

CONG, J.; ROSENSTIEL, W. The Last Byte: The HLS tipping point. *Design & Test of Computers, IEEE*, IEEE, v. 26, n. 4, p. 104, 2009.

COSTA, K. A. P. D. *Módulo de Conversão, C. and de Dados, F.* Tese (Doutorado) — Universidade de São Paulo, 2007.

COUSSY, P. et al. An Introduction to High-Level Synthesis. *Design & Test of Computers, IEEE*, IEEE, v. 26, n. 4, p. 8–17, 2009.

COUSSY, P.; MORAWIEC, A. *High-level synthesis: from algorithm to digital circuit*. [S.l.]: Springer Publishing Company, Incorporated, 2008.

DAVE, C. et al. Cetus: A Source-to-Source Compiler Infrastructure for Multicores. *Computer*, IEEE, v. 42, n. 12, p. 36–42, 2009.

DAVE, C.; EIGENMANN, R. Automatically tuning parallel and parallelized programs. In: *Proceedings of the 22nd international conference on Languages and Compilers for Parallel Computing*. Berlin, Heidelberg: Springer-Verlag, 2010. (LCPC'09), p. 126–139.

DEHON, A. The density advantage of configurable computing. *Computer*, IEEE, v. 33, n. 4, p. 41–49, 2002.

DESCHAMPS, J. et al. *Synthesis of arithmetic circuits: FPGA, ASIC, and embedded systems*. [S.l.]: John Wiley, 2006.

ELLSON, J. et al. Graphviz open source graph drawing tools. In: MUTZEL, P.; JüNGER, M.; LEIPERT, S. (Ed.). *Graph Drawing*. Springer Berlin / Heidelberg, 2002, (Lecture Notes in Computer Science, v. 2265). p. 594–597. 10.1007/3-540-45848-4_57. Disponível em: <http://dx.doi.org/10.1007/3-540-45848-4_57>.

ERCEGOVAC, M.; LANG, T.; MORENO, J. *Introdução aos sistemas digitais*. [S.l.]: Bookman, 2000.

GOOSSENS, G.; VANDEWALLE, J.; MAN, H. D. Loop optimization in register-transfer scheduling for DSP-systems. In: IEEE. *Design Automation, 1989. 26th Conference on*. [S.l.], 2006. p. 826–831.

GRÖTKER, T. et al. *System design with SystemC*. [S.l.]: Springer Netherlands, 2002.

GUO, Z.; NAJJAR, W.; BUYUKKURT, B. Efficient hardware code generation for FPGAs. *ACM Transactions on Architecture and Code Optimization (TACO)*, ACM, v. 5, n. 1, p. 1–26, 2008.

GUPTA, S. et al. *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. [S.l.]: Kluwer Academic Pub, 2004.

HALL, M. et al. Maximizing multiprocessor performance with the SUIF compiler. *Computer*, IEEE, v. 29, n. 12, p. 84–89, 1996.

HAMBLEN, J.; HALL, T.; FURMAN, M. *Rapid prototyping of digital systems*. [S.l.]: Springer Verlag, 2008.

HAUCK, S.; DEHON, A. *Reconfigurable computing: the theory and practice of FPGA-based computation*. [S.l.]: Morgan Kaufmann Pub, 2008.

HENNESSY, J.; PATTERSON, D.; GOLDBERG, D. *Computer architecture: a quantitative approach*. [S.l.]: Morgan Kaufmann, 2003.

HERBORDT, M. et al. Achieving high performance with FPGA-based computing. *Computer*, IEEE, v. 40, n. 3, p. 50–57, 2007.

IMPULSE, C. *Impulse Accelerated Technologies*. 2010.

KUON, I.; TESSIER, R.; ROSE, J. Fpga architecture: Survey and challenges. *Foundations and Trends® in Electronic Design Automation*, Now Publishers Inc., v. 2, n. 2, p. 135–253, 2008.

LEE, S.; JOHNSON, T.; EIGENMANN, R. Cetus–an extensible compiler infrastructure for source-to-source transformation. *Languages and Compilers for Parallel Computing*, Springer, p. 539–553, 2004.

LIMA, F. D. *Projeto com Matrizes de Células Lógicas Programáveis*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Sul, 1999.

LIPSETT, R.; MARSCHNER, E.; SHAHDAD, M. Vhdl-the language. *Design & Test of Computers, IEEE*, IEEE, v. 3, n. 2, p. 28–41, 1986.

MARCUS, M. *A theory of syntactic recognition for natural language*. [S.l.]: MIT Press (Cambridge, Mass.), 1980.

MARTIN, G.; SMITH, G. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, Published by the IEEE Computer Society, p. 18–25, 2009.

MCFARLAND, M.; PARKER, A.; CAMPOSANO, R. The high-level synthesis of digital systems. *Proceedings of the IEEE*, IEEE, v. 78, n. 2, p. 301–318, 2002.

MENEZES, P. *Linguagens formais e autômatos*. [S.l.]: Sagra-Luzzato, 2002.

MENOTTI, R. *LALP: uma linguagem para exploração do paralelismo de loops em computação reconfigurável*. Tese (Doutorado) — Universidade de São Paulo, 2010.

MENOTTI, R. et al. On using LALP to map an audio encoder/decoder on FPGAs. In: IEEE. *Industrial Electronics (ISIE), 2010 IEEE International Symposium on*. [S.l.]. p. 3063–3068.

MENOTTI, R. et al. Automatic generation of FPGA hardware accelerators using a domain specific language. In: IEEE. *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. [S.l.], 2009. p. 457–461.

MENOTTI, R. et al. LALP: A Novel Language to Program Custom FPGA-Based Architectures. In: IEEE. *Computer Architecture and High Performance Computing, 2009. SBAC-PAD'09. 21st International Symposium on*. [S.l.], 2009. p. 3–10.

MENOTTI, R. et al. LALP: A Language to Program Custom FPGA-based Acceleration Engines. *International Journal of Parallel Programming*, Springer, v. 40, n. 3, p. 262–289, 2012.

MENOTTI, R.; MARQUES, E.; CARDOSO, J. Aggressive Loop Pipelining for Reconfigurable Architectures. In: IEEE. *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*. [S.l.], 2007. p. 501–502.

MEYER-BAESE, U. *Digital signal processing with field programmable gate arrays*. [S.l.]: Springer Verlag, 2007.

MOONEN, M. S.; CATTHOOR, F. (Ed.). *Algorithms and Parallel VLSI Architectures III: Proceedings of the International Workshop, Algorithms and Parallel VLSI Architectures III, Leuven, Belgium, August 29-31, 1994*. New York, NY, USA: Elsevier Science Inc., 1995.

MUCHNICK, S. *Advanced compiler design and implementation*. [S.l.]: Morgan Kaufmann, 1997.

NAVEH, B. et al. Jgrapht. *Internet: http://jgrapht.sourceforge.net*, 2008.

NEUMANN, J. V.; ASPRAY, W.; BURKS, A. *Papers of John von Neumann on computing and computer theory*. [S.l.]: The MIT Press, 1987.

NIOS, I. C2H Compiler Users Guide. *Altera, May*, 2007.

OLIVEIRA, C. B. *Combinando transformações de computação e de dados para arquiteturas reconfiguráveis de grão fino*. Qualificação de Doutorado. Universidade de São Paulo, 2012.

PATTERSON, D.; HENNESSY, J. *Computer organization and design: the hardware/software interface*. [S.l.]: Morgan Kaufmann Pub, 2009.

PELO, E.; TERROSO, E. *Dispositivos Lógicos Programáveis(FPGA) e linguagem de descrição de hardware (VHDL)*. 1998. VII-SEMANA DA ENGENHARIA PUCRS.

QUARTUS, I. Version 6.0 handbook. *San Jose, Ca: Altera. Available at< http://www. altera. com/>.*, 2007.

QUINLAN, D. et al. Rose user manual: A tool for building source-to-source translators draft user manual (version 0.9. 5a). 2011.

RANGEL, J. *Compiladores Otimizações Otimização de Código*. 2000. PUC-Rio.

RAU, B. Iterative modulo scheduling: An algorithm for software pipelining loops. In: ACM. *Proceedings of the 27th annual international symposium on Microarchitecture*. [S.l.], 1994. p. 63–74.

ROTEM, N. *[Online]: http://www.c-to-verilog.com/*. 2010.

SASSA, M. et al. Static single assignment form in the coins compiler infrastructure. *Proc. SSGRR 2003w*, 2003.

SKLIAROVA, I.; FERRARI, A. Introdução à computação reconfigurável. *Potugal: Revista do DETUA*, 2003.

SMITH, M. *Application-specific integrated circuits*. [S.l.]: Addison-Wesley, 1997.

THOMAS, D.; MOORBY, P. *The Verilog hardware description language*. [S.l.]: Springer Netherlands, 2002.

TINEO, A. et al. Shape analysis for dynamic data structures based on coexistent links sets. *13th International Workshop on Compilers for Parallel Computers (CPC'2006)*, 2006.

VILLARREAL, J. et al. Designing Modular Hardware Accelerators in C With ROCCC 2.0. In: IEEE. *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. [S.l.], 2010. p. 127–134.

WATSON, D. High Level languages and their compilers. Addison-Wesley Longman Publishing Co., Inc., 1989.

XILINX. *[Online]: http://www.xilinx.com/*.

ZHOU, P. et al. Accmon: Automatically detecting memory-related bugs via program counter-based invariants. In: *In 37th International Symposium on Microarchitecture (MICRO*. [S.l.: s.n.], 2004. p. 269–280.