Síntese comportamental de componentes de um Sistema Operacional em hardware

João Paulo Pizani Flor[1]

[1]Departamento de Informatica e Estatistica – Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 476 – 88.040-900 – Florianopolis – SC – Brazil

joaopizani@inf.ufsc.br

Abstract. *With recent advances on programmable logic devices and the growing need to harness the inherent parallelism in many applications, it is increasingly popular to implement in FPGAs algorithms that were previously implemented in software. On the other hand, the description of hardware realizing these algorithms still has a low level of abstraction (when compared to software). Also, most attempts at high-level synthesis focus on specific applications. This paper aims to describe an Operating System component in a high level of abstraction and to synthesize this component to a FPGA device. The chosen application area and the adopted methodology characterize this work as a feasibility study of automatic implementation of general-purpose algorithms in hardware.*

Resumo. *Com os avanços na tecnologia de lógica programável e a crescente necessidade de se explorar o paralelismo inerente a várias aplicações, é cada vez mais comum implementar em FPGAs algoritmos antes implementados em software. Porém a descrição de hardware para realizar tais algoritmos ainda é feita em baixo nível de abstração (se comparada ao software). Além disso, a maioria dos trabalhos que visam a síntese de alto nível foca em aplicações específicas. Este trabalho visa descrever em alto nível um componente de Sistema Operacional e implementá-lo em um FPGA. A área de aplicação escolhida e a metodologia adotada fazem deste trabalho um estudo de viabilidade da implementação automática em hardware de algoritmos de propósito geral.*

1. Introdução

Apesar de já ser sabido há um bom tempo que todo e qualquer algoritmo pode ser implementado por um circuito, questões essencialmente tecnológicas impediram por muito tempo o uso de circuitos booleanos como um modelo para a implementação de algoritmos em larga escala. O problema fundamental no uso de circuitos booleanos era a falta de *flexibilidade*. Até recentemente, uma vez que um circuito fosse fabricado, ele não poderia mais ser alterado ou reconfigurado para exercer tarefa diferente daquela para qual foi projetado. A forma de se obter uma máquina *universal*, capaz de resolver qualquer problema computável, foi projetando-se um circuito capaz de interpretar instruções codificadas em binário e realizar diferentes operações de acordo com o tipo de instrução. Essa arquitetura, uma implementação física da máquina universal de Turing, é a tão conhecida arquitetura de von Neumann[Von Neumann and Godfrey 1993], e ainda hoje é a base para quase todos os processadores.

Apesar de algumas desvantagens inerentes à arquitetura de von Neumann, a hegemonia por ela conquistada é simples de se explicar. Os avanços na tecnologia de

fabricação de transistores vinham permitindo que o poder de computação dos processadores aumentasse sem grandes remodelagens arquiteturais. Durante muito tempo, a frequência dos processadores cresceu seguindo uma função exponencial.

De fato, o cofundador da Intel®, Gordon E. Moore, previu essa "tendência" da indústria de microprocessadores com um célebre artigo[Moore et al. 1965]. Tal tendência foi verificada quase exatamente por décadas, porém vem perdendo validade gradativamente nos dias atuais. Embora a miniaturização dos transistores continue, o *aumento na frequência* dos processadores perde força devido à grande quantidade de potência dissipada em forma de calor[Kuroda 2001].

Levando em conta essa perda de velocidade no aumento da frequência dos processadores, a comunidade profissional e acadêmica volta-se para modelos de programação paralela. Cresce também cada vez mais o interesse pela implementação de algoritmos diretamente em hardware, dado o seu paralelismo inerente e os recentes avanços na tecnologia de lógica programável. Um fator que dificulta a ampla implementação de algoritmos em hardware é a escassez de técnicas e ferramentas que permitam descrições dos algoritmos em alto nível de abstração, e a síntese automática de tais descrições. Investigar o estado da arte na área de síntese de alto nível, assim como realizar um estudo de caso implementando um algoritmo de uso prático, são as metas deste trabalho.

2. Trabalhos correlatos

Em uma analogia simplificada entre o projeto de hardware e o desenvolvimento de software podemos dizer que atualmente os projetistas de hardware "programam" seus sistemas em linguagem de montagem. As linguagens e ferramentas para síntese de alto nível têm como objetivo transportar a descrição de um algoritmo em hardware para um nível de abstração equivalente à programação de software em linguagens como C++, Java, etc.

Uma das vertentes mais promissoras para atingir esse objetivo é a utilização do *framework* SystemC[OSCI 2005]. O SystemC é um grupo padronizado de classes C++ que permite a modelagem e simulação de componentes de hardware. Em SystemC é possível descrever o comportamento de componentes de hardware usando a sintaxe C++, e utilizando-se de várias abstrações para a comunicação inter-componentes.

A OSCI, organização responsável pela manutenção e avanço do framework SystemC, disponibiliza também sob licença livre uma biblioteca para simulação de modelos SystemC. O projetista compila seu projeto descrito em SystemC com a cadeia usual de ferramentas (*g++*, *ld*, *as*), e como resultado da compilação é gerado um arquivo executável. Ao ser executado, ele *simula* o circuito descrito e pode produzir vários tipos de saída para depuração (incluindo arquivos com formas de onda). Não há ainda, porém, nenhum projeto de software livre para a síntese de hardware a partir de SystemC.

2.1. Abordagens baseadas em SystemC

Dentre os produtos comerciais para síntese de alto nível, um dos que utiliza modelos em SystemC como entrada é o *Cynthesizer*®, da Forte Design Systems. os modelos são escritos em SystemC de nível TLM e a ferramenta faz a tradução para componentes RTL, os quais podem então ser fabricados como ASIC, ou sintetizados em FPGA (utilizando as ferramentas dos fornecedores).

Muitos dos detalhes necessários à implementação do modelo em hardware são omitidos na descrição TLM. por isso, a ferramenta *Cynthesizer*®incorpora também um módulo para exploração de espaço de projeto, permitindo que o projetista defina restrições e métricas sobre o circuito, buscando otimizar tais métricas.

2.2. Síntese de ANSI C++

Além do SystemC, existem também abordagens que buscam realizar a síntese a partir de descrições em ANSI C++. Algumas ferramentas trabalham com C++ padrão ANSI, e permitem utilização de construções avançadas da linguagem, tais como ponteiros, classes e structs, *arrays* multidimensionais e metaprogramação via *templates*. Já outras ferramentas trabalham com subconjuntos mais restritos da linguagem, e usam extensões que tornam as descrições de hardware incompatíveis com compiladores tradicionais de C++ para software.

Algumas construções de C++, porém, são invariavelmente inadequadas à uma implementação em hardware e, portanto, não são suportadas por nenhuma das ferramentas de síntese pesquisadas. Em geral a inadequação dessas construções advém da necessidade de conhecer todos os detalhes do algoritmo em tempo de síntese. Tal necessidade proíbe:

Alocação dinâmica de memória A quantidade de memória a ser utilizada pelo algoritmo precisa ser um valor conhecido em tempo de síntese. Alocar memória dinamicamente em um FPGA significaria reconfigurar partes de um FPGA dinamicamente, uma tarefa ainda complexa e ineficiente.

Recursões não limitadas Funções recursivas sem limite explícito de profundidade de chamadas recaem no uso de alocação dinâmica e são, portanto, inadequadas à implementação em hardware. Isso pois, na chamada de uma nova instância de função recursiva, novo espaço teria de ser alocado para as variáveis no escopo local *daquela* instância. Algumas ferramentas dão suporte à *recursão com profundidade limitada*, implementada utilizando-se de metaprogramação estática via *templates*.

Uma cadeia de ferramentas de grande destaque em HLS é o *Catapult-C*®, do fabricante Mentor Graphics®. Uma peculiaridade do Catapult-C é que ele aceita como linguagens de entrada no projeto de um sistema tanto C++ (ANSI) como SystemC. Enquanto a entrada em C++ modela basicamente o *comportamento* do sistema sendo projetado, a entrada em SystemC permite a especificação de mais detalhes arquiteturais, tais como a configuração de barramentos e NoC.

Em um processo de refinamento interativo, o Catapult-C oferece ao projetista comparativos entres as várias opções de parâmetros deixados livres pela descrição em alto nível, e através de estatísticas e gráficos o projetista faz uma decisão informada levando em consideração as prioridades do projeto em questão. o Catapult-C também é capaz de realizar a síntese automática de interfaces entre componentes, e possui um módulo que faz a otimização do consumo de energia nos componentes produzidos, levando em conta as características próprias dos dispositivos de vários fabricantes.

3. Desenvolvimento

3.1. Estruturas de dados

A estrutura de dados fundamental do escalonador é uma fila de escalonamento, objeto da classe System::Scheduling_Queue<T,Q>, com os parâmetros:

T: O tipo de objeto a ser escalonado. Como a fila é genérica, não apenas *Threads* são passíveis de escalonamento. Qualquer tipo pode ser utilizado como T, devendo satisfazer apenas aos seguintes requisitos:

- Declarar um tipo chamado "Criterion", que será usado como critério de ordenamento da fila.
- Declarar um método "link", para satisfazer à interface de elemento de lista.

Q: parâmetro de tipo inteiro sem sinal, indica *quantas* filas de escalonamento devem ser utilizadas.

Sobre a hierarquia de listas e filas do EPOS cabem ainda alguns comentários adicionais:

List<T, El> É um template de lista duplamente encadeada, onde o parâmetro El encapsula o encadeamento.

- El deve implementar *getters* e *setters* para os atributos "next" e "prev".
- El é um parâmetro **opcional** do template, e seu valor padrão é List_Elements::Doubly_Linked<T>

Ordered_List<T, R, El, relative> Subclasse de List<T, El>, este é um template de lista ordenada, que recebe dois parâmetros adicionais:

R: um tipo que encapsula o conceito de *rank* de um elemento. Qualquer tipo de dados *isomórfico* aos números inteiros pode ser usado como R, devendo implementar o método de *cast* para int. Este parâmetro é **opcional** e seu valor padrão é List_Element_Rank.

relative: um parâmetro booleano indicando se a lista é *relativa*. Diz-se que uma lista ordenada é relativa quando cada elemento armazena apenas a diferença entre seu rank e o de seu antecessor. Este parâmetro é **opcional** e seu valor padrão é false.

Seguindo a filosofia de implementação do EPOS, o comportamento de filas de escalonamento está isolado dos seus detalhes de implementação (como alocação e armazenamento) e implementado na classe *Scheduling_Queue<T, Q>*. O parâmetro T é o tipo de entidade a ser escalonado, enquanto o parâmetro Q define o número de filas a serem usadas no escalonamento.

Qualquer tipo T de entidade pode ser escalonado, desde que satisfaça ao seguinte requisito:

- T deve declarar um tipo "Criterion". O tipo T::Criterion será utilizado como critério de ordenação das filas subjacentes ao escalonador, e define qual é o algoritmo de escalonamento a ser utilizado. Alguns critérios já implementados no EPOS são Round-Robin, Rate monotonic e EDF.

O diagrama 1 detalha os métodos públicos da classe Scheduling_Queue, assim como um exemplo de entidade escalonável, uma *Thread*

3.2. Implementação RTL de referência

Para avaliar a eficácia do processo de síntese de alto nível nós realizamos uma análise comparativa (em termos de área ocupada e atraso) com uma implementação de referência; os detalhes dessa análise encontram-se na seção de resultados. Nossa referência é a implementação de um escalonador em nível RT (escrito na linguagem VHDL), descrito em [Marcondes and Fröhlich 2009].

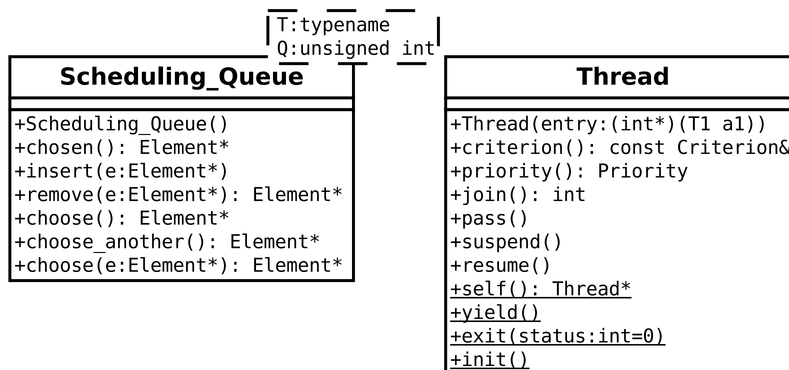

Figura 1. Diagrama de classe UML da fila de escalonamento utilizada no trabalho e da classe Thread, um exemplo de entidade escalonável

O bloco escalonador contém vários pequenos sub-blocos, cada um encapsulando um dos serviços fornecidos. Além desses blocos algorítmicos, ainda compõem o escalonador uma memória que armazena a fila de escalonamento, um bloco responsável pelo gerenciamento de recursos e uma interface que faz a interpretação dos comandos recebidos do mundo externo e ativa o bloco algorítmico correspondente.

3.3. Escalonador em hardware

Como já foi mencionado, nossa implementação de um escalonador a ser implementado em hardware foi guiada pelo desejo de se realizar o menor número possível de modificações em relação ao escalonador que executa em software. Em particular, quanto às alterações que tiveram de ser realizadas, dois princípios foram seguidos:

- As alterações devem permitir que o componente possa *continuar* a executar em software sem alteração de semântica, mesmo que com penalidade de performance.
- Deve-se, sempre que possível, encapsular as alterações em classes, evitando alterar diretamente o código-fonte do componente a ser sintetizado.

Durante o fluxo de síntese de alto nível, estruturas do tipo array[1] presentes no código são mapeadas para bancos de registradores ou memórias, e as variáveis do tipo ponteiro são mapeadas para índices dos arrays aos quais se referem.

Em um código C++ sintetizável, todo ponteiro deve se referir a uma estrutura de dados definida no código. Uma análise estática do código é realizada pela ferramenta de síntese para garantir que em nenhum dos possíveis fluxos de execução um ponteiro inválido seja desreferenciado. Em particular, são proibidas atribuições de literais inteiras a variáveis do tipo ponteiro.

As interfaces das estruturas de dados do EPOS em geral, e a da classe Scheduling_Queue, em particular, trabalham com ponteiros, e em várias situações ponteiros nulos (inválidos), são retornados para sinalizar situações de falha. Para contornar essa inadequação aos requisitos de síntese introduzimos o *tipo opção*[2] Maybe<T>.

O tipo Maybe<T>pode ser conceitualmente definido da seguinte maneira:

[1]uma sequência de elementos de dados residentes em uma faixa contínua do espaço de endereçamento
[2]do inglês "Option Type". `http://en.wikipedia.org/wiki/Option_type`

```
Maybe T = Nothing | Just T
```

Ou seja, há dois construtores possíveis para obter-se um valor do tipo Maybe<T>: Um construtor vazio (chamado de *Nothing*) e um construtor que toma um parâmetro (chamado de *Just*).

Outra das limitações da implementação de algoritmos em hardware é a impossibilidade de se utilizar alocação dinâmica de memória.

Por outro lado, todas as estruturas de dados do EPOS são independentes da forma com que seus elementos estão alocados. Isso permitiu que não fizéssemos qualquer alteração no código das estruturas nesse sentido. Devido, porém, justamente a essa independência, tivemos que implementar separadamente um gerenciador de alocação para as filas de escalonamento.

A classe Scheduler_PreAlloc implementa esse gerenciador. Um objeto da classe Scheduler_PreAlloc é um *invólucro* para um objeto da classe Scheduling_Queue, e exporta publicamente uma interface bastante semelhante à do objeto envolvido. Duas grandes diferenças podem ser notadas: as operações possuem semântica de passagem *por valor* e nas inserções e remoções são feitas alocações e liberações de espaço no armazenamento subjacente. A figura 2 mostra um diagrama de blocos do gerenciador.

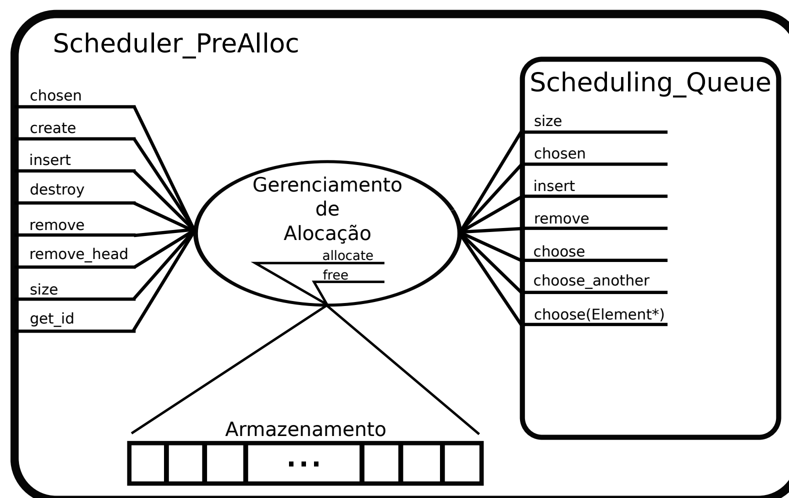

Figura 2. Diagrama de blocos do gerenciador de alocação para a fila de escalonamento Scheduling_Queue

Uma das restrições mais importantes da ferramenta de síntese de alto nível utilizada se refere à forma como é feita a inferência das portas de entrada e saída do bloco de hardware sendo modelado. O projetista deve designar *uma única* função em todo o seu código como a função “raiz” do bloco. Isso é feito colocando-se o *pragma*[3] “hls_design top” sobre a assinatura da função.

A ferramenta de síntese irá então inferir, a partir dos parâmetros presentes na assinatura e da forma como eles são utilizados, as portas de entrada e saída do bloco de

[3]anotação não-padronizada feita no código que dá ao compilador informações extras sobre o software, não presentes na linguagem fonte

hardware sendo modelado, sendo que os tipos dos parâmetros definirão os tamanhos das portas.

Nosso escalonador é um objeto, e sua interface possui um método para cada operação oferecida, com parâmetros diferentes para cada método. A figura 3 mostra a solução adotada para adaptar a interface da classe *Scheduler_PreAlloc* à função que define a interface do bloco em hardware.

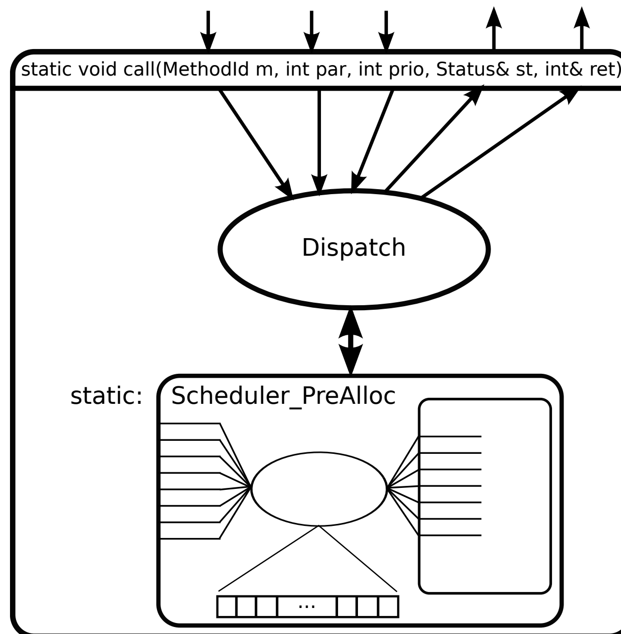

Figura 3. Diagrama do bloco raiz do escalonador sintetizado, enfatizando as portas de entrada e saída e a lógica de *dispatch* das operações

4. Resultados

Nas próximas seções detalhamos o testbench de verificação e todos os cenários de síntese. São também exibidas e analisadas as métricas resultantes da síntese (tanto pré como pós-RTL). A avaliação de nossa implementação é feita em comparação com uma implementação de referência do escalonador em nível RT.

Em todos os cenários, o dispositivo-alvo para a síntese pós-RTL foi um FPGA da família Virtex6®, fabricado pela Xilinx®.

4.1. Cenários

Em geral, a escolha dos cenários de síntese para nossa proposta foi guiada pelo interesse em avaliar microarquiteturas que se situassem em extremos opostos do espectro de área/latência, ou seja, optamos por realizar a síntese de microarquiteturas que tivessem grande área e baixa latência e vice-versa. Uma descrição mais detalhada de cada um dos cenários de síntese segue:

Cenário 1 Neste cenário os arrays usados para armazenar os elementos da fila de escalonamento foram todos mapeados para memórias (RAMs) dedicadas no FPGA. Nenhuma paralelização de loops foi realizada.

Cenário 2 Propositadamente, neste cenário de execução não houve qualquer intervenção do projetista no processo. Todas as diretivas de síntese foram mantidas em seus valores *default*, visando avaliar uma espécie de caso "otimista", em que a ferramenta de síntese tem independência total para inferir a microarquitetura do bloco sendo sintetizado. Neste cenário os arrays acabaram sendo mapeados para registradores e os loops também foram mantidos intactos (sem nenhuma paralelização).

Cenário 3 Cenário com paralelismo total. Todos os loops foram paralelizados, o que diminuiu drasticamente o número de ciclos necessários (no pior caso) para realizar uma operação do escalonador. Naturalmente, tal paralelização acarretou um aumento correspondente na área consumida e também no atraso máximo do bloco.

É importante ainda citar que em todos os cenários a síntese pós-RTL foi realizada objetivando-se uma frequência de clock de 50 MHz, portanto o limite máximo de atraso era de 20 ns. Caso algum cenário tivesse uma atraso que ultrapassasse tal limite, sua implementação seria inviável.

4.2. Resultados de síntese

Dados de cada um dos três cenários definidos foram coletados em duas etapas do fluxo de projeto. A primeira delas, pré-RTL, corresponde ao resultado da síntese de alto nível, um arquivo VHDL que contém a descrição do bloco em nível de transferência de registradores. Já a segunda etapa, pós-RTL, corresponde ao resultado do mapeamento do modelo RTL para os blocos funcionais dispositivo FPGA utilizado. A tabela 1 mostra os resultados na etapa pré-RTL.

Cenario	Area ocupada	Ciclos por operacao (pior caso)
Cenario 1	4301.698	79
Cenario 2	6496.576	43
Cenario 3	9847.783	4

Tabela 1. Dados de area e vazao pre-sintese RTL de nossa proposta, em alguns cenarios de parametros. Tais dados foram reportados apos a sintese de alto nÃvel, sobre o modelo RTL gerado

A coluna "Área ocupada" se refere à pontuação de área dada pela ferramenta de síntese, sendo que tal pontuação leva em conta a quantidade de operadores de cada tipo usados no blocos, assim como o peso de cada um dos tipos de operadores.

É notável a diferença entre a quantidade de ciclos necessários por operação nos cenários 1 e 2. Essa diferença pode ser explicada pelo uso de uma memória para o armazenamento, resultando em um gargalo no sistema e impedindo a leitura paralela de dados que não possuem dependências entre si. Já no cenário 3 o consumo de área aumenta em mais de 2 vezes, porém o número de ciclos necessários por operação diminui em proporção bem maior. De fato, os 4 ciclos são provenientes sobretudo do bloco de controle do sistema (máquina de estados finitos).

A tabela 2, a seguir, mostra a utilização de recursos e os atrasos máximos para cada cenário na etapa de síntese pós-RTL.

Cenario	LUTs	LUTs (% do FPGA)	RAMs	Atraso maximo
Cenario 1	1654	1.10%	26	6.672 ns
Cenario 2	3392	2.25%	0	7.341 ns
Cenario 3	5121	3.40%	0	10.597 ns

Tabela 2. Dados de area (consumo de recursos) e atraso maximo de nossa proposta. Os dados foram obtidos apos a sintese RTL do escalonador, com a netlist ja mapeada para o FPGA utilizado

Salienta-se a utilização de blocos de memória RAM da FPGA no cenário 1. Com a utilização de memórias, o número de LUTs[4] ocupadas caiu significativamente.

Outro ponto a ser destacado é que a diferença entre o número de LUTs ocupadas nos cenários 2 e 3 chega a quase 100%. No entanto, quando se compara as porcentagens de utilização do FPGA, a diferença já não é tão significativa (2.25% vs. 3.40%). Isso nos leva a crer que em um futuro ainda próximo, à medida que os FPGAs ganharem mais blocos funcionais, a síntese de alto nível se tornará cada vez mais viável em termos de área.

Como última observação sobre a etapa de síntese pós-RTL, gostaríamos de chamar a atenção para os atrasos máximos obtidos. Mesmo no cenário 3 o atraso máximo previsto ainda é bem menor do que nosso limite superior (20 ns), e portanto a frequência de clock do bloco poderia ser maior, caso fosse necessário.

A comparação com a implementação RTL de referência foi realizada (como já mencionado) somente na etapa de síntese pós-RTL. O mesmo dispositivo-alvo e os mesmos níveis de otimização foram utilizados em todos os cenários. A tabela 3 demonstra essa comparação.

Cenario	LUTs	LUTs (% do FPGA)	Atraso maximo
RTL	1250	0.83%	5.598 ns
Cenario 1	1654	1.10%	6.672 ns
Cenario 2	3392	2.25%	7.341 ns
Cenario 3	5121	3.40%	10.597 ns

Tabela 3. Comparacao entre dados de area (consumo de recursos) e atraso maximo entre nossa proposta e a implementacao RTL de referencia

5. Conclusões

O objetivo principal deste trabalho foi avaliar a viabilidade de se utilizar síntese de alto nível para a implementação de algoritmos de propósito geral em hardware reconfigurável. Tendo em vista os resultados alcançados podemos dizer que, com uma escolha razoável de configurações, uma ferramenta de síntese de alto nível pode chegar a resultados bastante próximos dos alcançados por um experiente projetista de hardware.

[4]LUT: Look-Up Table, o bloco funcional mais comum (numeroso) em FPGAs

Apesar de, em termos absolutos, a solução sintetizada automaticamente ter ocupado uma área bastante maior (cerca de 30% de acréscimo), em termos de porcentagem de ocupação do dispositivo FPGA a diferença não foi significativa. Cremos que essa é uma tendência, e que com o aumento do poder computacional dos FPGAs a síntese de alto nível se tornará de fato cada vez mais viável.

Nosso estudo de caso, um escalonador de recursos, normalmente seria considerado um mau candidato a beneficiar-se de síntese de alto nível, pois é relativamente pobre em computação e possui um estado interno significativo. Mesmo assim conseguimos demonstrar que, com uma boa engenharia de domínio e separação de responsabilidades entre os componentes de um sistema, é possível se ter código genérico suficiente a ponto de executar com poucas modificações tanto em software quanto em hardware.

5.1. Principais contribuições

Compreendemos que a modelagem do problema estudado e a utilidade dos artefatos desenvolvidos não se restringe ao algoritmo que desenvolvemos (um escalonador). Em particular, ressaltamos as seguintes principais contribuições deste trabalho:

Camadas de adaptação de um objeto para implementação em hardware Em nosso desenvolvimento do escalonador proposto, fomos guiados sobretudo pelas restrições da ferramenta de síntese de alto nível. Essas restrições nos levaram a modelar uma arquitetura *em camadas*, que encapsula um objeto, responsabilizando-se pelo gerenciamento de recursos necessários à sua execução e pela comunicação com o mundo externo. Os componentes são genéricos, e podem encapsular outras classes de objetos além do escalonador.

Adaptador Plasma / AMBA Como ambiente de execução do escalonador desenvolvido, escolhemos o processador *softcore* Plasma MIPS. O principal motivo dessa escolha foi o fato de que o Plasma MIPS é um processador de código aberto, livremente modificável. Já nossa escolha de barramento para o SoC onde o escalonador executaria recaiu sobre a família AMBA®, da fabricante ARM®. Como o Plasma MIPS não possuía interface AMBA®, nós desenvolvemos um componente adaptador, o qual torna um processador Plasma mestre em um barramento AXI4Lite (barramento AMBA®simplificado). Este adaptador foi desenvolvido em linguagem VHDL e seu código-fonte, documentação e testes foram publicados no repositório de hardware livre *OpenCores*[5].

5.2. Trabalhos futuros

Como principais trabalhos futuros que poderiam explorar avanços na mesma temática deste trabalho ressaltamos:

- Fazer do escalonador proposto um protótipo de componente de SO efetivamente híbrido, segundo a visão de longo prazo do EPOS, como descrita em [Marcondes and Fröhlich 2009]. Isto é, através de alguns avanços no sistema de compilação do EPOS toda a camada de adaptação para hardware deveria ser acrescentada a um componente, sempre que fosse escolhido pelo usuário sua implementação em hardware.

[5] `http://www.opencores.org`

- Modelar outras estruturas de dados e algoritmos em hardware, utilizando-se também de síntese de alto nível. Seria particularmente interessante uma pesquisa sobre estruturas de dados paralelas, sua descrição em alto nível de abstração e implementação em hardware.
- Partir das contribuições deste trabalho em direção a um protocolo de comunicação entre objetos independentes de implementação. Objetos em hardware teriam um protocolo padronizado para troca de mensagens entre si, assim como com objetos em software. De certa forma, essa linha de trabalho está relacionada à temática de invocação remota e objetos distribuídos.

Referências

Kuroda, T. (2001). Cmos design challenges to power wall. In *Microprocesses and Nanotechnology Conference, 2001 International*, pages 6–7. IEEE.

Marcondes, H. and Fröhlich, A. A. (2009). A hybrid hardware and software component architecture for embedded system design. *Analysis, Architectures and Modelling of Embedded Systems*, pages 259–270.

Moore, G. et al. (1965). Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85.

OSCI (2005). IEEE 1666: SystemC Language Reference Manual, 2005.

Von Neumann, J. and Godfrey, M. (1993). First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75.