Utilização de hardware reconfigurável para acelerar a *satisfação booleana* *

Iouliia Skliarova, António B. Ferrari

Resumo **– Este artigo apresenta um estudo de possibilidade de aceleração da *satisfação booleana* com a ajuda do hardware reconfigurável. A *satisfação booleana* (SAT) é um problema importante que tem muitas aplicações em CAD e outras áreas. Neste artigo propomos uma técnica de desenvolvimento orientada a problema em geral para acelerar a resolução de SAT formulado sobre matriz discreta. O algoritmo utilizado requer uma unidade de controlo bastante complexa que é implementada inteiramente em hardware reconfigurável. Por fim, são analisadas diferentes possibilidades de resolução de SAT e argumenta-se que os melhores resultados podem ser obtidos com a ajuda da colaboração de uma aplicação de software executada num computador de uso geral com um circuito de solução de SAT implementado em FPGA.**

Abstract **– The paper presents a case study of accelerating Boolean satisfiability in reconfigurable hardware. Boolean satisfiability (SAT) is an important problem having many applications in CAD and other areas. We propose an application-specific approach to accelerate the backtrack search algorithm for the SAT problem formulated over discrete matrix. The algorithm employed involves a quite sophisticated control unit, which is entirely implemented in reconfigurable hardware. Finally, we analyze different possibilities of solving the SAT problem and argue that the best results can be achieved by the use of software, running on a general-purpose computer, together with an FPGA-based reconfigurable SAT solver.**

I. INTRODUCÃO

A computação reconfigurável é uma área bastante nova em que o hardware é programado de acordo com as necessidades de cada aplicação específica. A computação reconfigurável recorre ao uso de componentes lógicos programáveis, tais como FPGAs. Ao contrário dos computadores convencionais que são programados a nível de instruções, os dispositivos reconfiguráveis são programados a nível de componentes funcionais. Isto permite implementar funcionalidades bastante mais ricas e, consequentemente, atingir para certas aplicações um desempenho muito mais elevado que o dos computadores de uso geral. O desempenho elevado é atingido devido aos seguintes factores [1]:

- Largura de banda elevada no acesso à memória;
- Exploração de paralelismo e de *pipelining*;
- Uso de unidades funcionais específicas e optimizadas.

Analisemos estas técnicas em mais detalhe. Tradicionalmente os computadores de uso geral organizam a memória em forma de um conjunto de palavras de tamanho fixo. Os dados de um problema não cabem exactamente numa só palavra, portanto são precisos vários acessos à memoria para os processar. Contudo, em FPGA a organização da memória pode ser feita de acordo com a dimensão dos dados de um problema, portanto estes podem ser processados numa única operação.

Muitas aplicações involvem operações bastante simples mas estas não são bem suportadas por ALUs convencionais. Sendo assim é possível implementar em FPGA a unidade funcional optimizada para certas operações e tamanhos de dados. Esta unidade é normalmente simples e ocupa poucos recursos de hardware portanto é possível reproduzi-la para explorar o paralelismo.

As aplicações que são normalmente implementadas em sistemas reconfiguráveis são as dominadas pelo processamento de dados caracterizadas por estruturas de controlo relativamente simples. Neste artigo propomos a utilização de computação reconfigurável em problemas com estruturas de controlo complexas que surgem em particular na área de optimização combinatória. Para tal escolhemos o problema de *satisfação booleana* que é de grande importância em áreas diversas.

Recentemente foram propostas várias implementações em FPGAs de circuitos de solução de SAT [2, 3, 4]. Todas estas propostas beseiam-se na ideia de geração de um circuito especializado para cada instância do problema a resolver (*instance-specific approach*). A maior vantagem desta estratégia é que o mapeamento directo da função booleana nos componentes funcionais permite incrementar significativamente o desempenho e assugurar uma boa utilização de recursos do hardware. Neste caso o tempo necessário para resolver um

* Trabalho financiado com a bolsa da FCT-PRAXIS XXI/BD/21353/99

problema é composto pela soma do tempo de geração do circuito respectivo, tempo de configuração da FPGA e tempo de execução. Existem vários compiladores especiais que aceleram a geração de configurações de FPGA. Contudo, o tempo de compilação do hardware continua a ser bastante elevado e, como regra, é maior que o tempo da execução do algoritmo. Portanto, este método só pode ser usado para problemas com grandes volumes de dados de entrada, quando o tempo de compilação do hardware é amortizado pelo tempo de execução.

A outra questão que afecta cada algoritmo implementado em hardware reconfigurável é a capacidade da plataforma respectiva. Caso o circuito não possa ser implementado numa só FPGA, é possível ocupar vários dispositivos aplicando os métodos especiais de partição entre FPGAs. Contudo não é garantido que um problema arbitrário possa ser resolvido com os recursos de hardware reconfigurável disponíveis.

Tendo em atenção estas questões propomos aplicar a estratégia orientada à aplicação e não à instância do problema. Neste caso o circuito é desenvolvido uma só vez, optimizado e testado. Portanto, o tempo de compilação do hardware pode ser considerado igual a zero, e o tempo de resolução de um problema só é composto por tempo de configuração da FPGA e o tempo de execução. Uma vez que os recursos de FPGA disponíveis são sempre limitados, propomos utilizar uma arquitectura que é baseada na colaboração do software e hardware reconfigurável.

O resto deste artigo está organizado de maneira seguinte. Na secção 2 especifica-se formalmente o problema de *satisfação booleana* e descreve-se o algoritmo utilizado para a sua resolução. Na secção 3 é proposta a arquitecura do circuito que implementa o algoritmo considerado. A seguir, na secção 4, é descrito o modelo de colaboração de software e do hardware reconfigurável. Finalmente, são apresentados os resultados e a sua comparação com os do GRASP [5] – o mais eficiente algoritmo para SAT implementado em software. As conclusões estão na secção 6.

II. PROBLEMA DE SAT E O ALGORITMO UTILIZADO

É conhecido que CNF (*Conjunctive Normal Form*) é a conjunção de um número de cláusulas onde cada cláusula é a disjunção de uma ou mais variáveis ou das suas negações. O problema de *satisfação booleana* (SAT) consiste em determinar se a função em CNF é satisfazível, i.e. se existe tal atribuição de valores às variáveis que faz com que a função tome valor 1. Obviamente, para que a função inteira avalie a 1 é necessário que cada cláusula seja igual a 1. Por exemplo, a função seguinte contem 3 variáveis e 4 cláusulas e é satisfeita quando $x_1=x_2=x_3=1$:

$$(x_1 \vee \overline{x}_3)(\overline{x}_2 \vee x_3)(x_1)(\overline{x}_1 \vee x_2)$$

De outro lado a função seguinte é insatisfazível:

$$(x_1 \vee \overline{x}_3)(x_3)(\overline{x}_1)(\overline{x}_1 \vee x_2)$$

O problema de SAT pode ser especificado sobre vários modelos matemáticos tais como funções booleanas e matrizes discretas. Um modelo pode ser formalmente transformado noutro. Escolhemos a representação em matrizes porque estas são de bastante fácil processamento em FPGA.

Formulemos o problema de SAT sobre a matriz **M**. Para isso vamos fazer corresponder a cada cláusula c_i uma linha da matriz m_i e a cada variável x_j uma coluna da matriz m_j. Se a variável x_j entra na cláusula c_i então o elemento respectivo da matriz m_{ij} é igual a '1', se a variável x_j entra na cláusula c_i com a negação então o elemento respectivo da matriz m_{ij} é igual a '0', e se a variável x_j não entra na cláusula c_i então o elemento respectivo da matriz m_{ij} é igual a '-' (*don't care*). Por exemplo, a função seguinte:

$$(x_1 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2) \wedge (\overline{x}_2 \vee \overline{x}_3) \wedge (x_1 \vee x_2) \qquad (1)$$

pode ser representada com a matriz **M**:

$$\mathbf{M} = \begin{array}{c} \begin{array}{ccc} \mathbf{x_1} & \mathbf{x_2} & \mathbf{x_3} \end{array} \\ \left[\begin{array}{ccc} \mathbf{1} & - & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & - \\ - & \mathbf{0} & \mathbf{0} \\ \mathbf{1} & \mathbf{1} & - \end{array}\right] \end{array} \begin{array}{c} \\ \mathbf{c_1} \\ \mathbf{c_2} \\ \mathbf{c_3} \\ \mathbf{c_4} \end{array}$$

É de notar que resolver um problema de SAT é equivalente ao encontrar um vector ternário **v** que seja ortogonal a cada linha da matrix **M** respectiva. Caso seja impossível arranjar tal vector então a função correspondente é insatisfazível. Por outro lado caso encontremos o vector **v**, devemos invertê-lo. Os zeros e uns no vector invertido apontarão àqueles variáveis que devem ser iguais a 0 e 1 respectivamente para satisfazer a função. Para a matriz **M** apresentada em cima a solução é o vector **v** = -01. Sendo assim, $\overline{\mathbf{v}}$ = -10, i.e. $x_2=1$ e $x_3=0$. É fácil verificar que atribuição destes valores às variáveis satisfaz a função (1).

Para encontrar o vector ortogonal a cada linha de uma matrix ternária aplicámos o algoritmo proposto em [6] apresentando-o em forma de uma árvore de pesquisa [7]. Neste caso a cada vértice da árvore corresponde um vector ternário **v** e uma matriz $\mathbf{M}'$ composta por vários menores da matriz **M**. Inicialmente o vector **v** está totalmente indeterminado, i.e. **v**="-…-", e $\mathbf{M}'$=**M**. A transição de um vértice da árvore de pesquisa a outro efectua-se se reduzirmos a matriz **M** ou atribuirmos valor 0 ou 1 a uma componente do vector **v**. Na fig.1 está apresentado o esquema de blocos do algoritmo utilizado. O esquema mostra que durante a pesquisa aplicam-se vários métodos de redução, depois, quando a redução se torna impossível, efectua-se a decomposição da situação corrente (o que corresponde à ramificação da árvore). A seguir, aplicam-se outra vez os métodos de redução e o algoritmo continua até que encontre a solução ou chegue à conclusão que esta não existe. Os

métodos de redução envolvidos no algoritmo são os seguintes:

- Método 1. Na matriz $\mathbf{M}'$ apagam-se todas as colunas que são totalmente indeterminadas, i.e. não contêm zeros nem uns.
- Método 2. Na matriz $\mathbf{M}'$ apagam-se todas as linhas que são ortogonais ao vector **v** corrente.
- Método 3. Na matriz $\mathbf{M}'$ apagam-se todas as colunas que correspondem às componentes determinadas do vector **v**.
- Método 4. Caso na matriz $\mathbf{M}'$ exista uma linha que tem uma só componente com o valor 0 ou 1 e todas as outras suas componentes iguais a '-', então à componente correspondente do vector **v** atribui-se o valor inverso.
- Método 5. Caso na matriz $\mathbf{M}'$ exista uma coluna que não contem valor 0 (ou 1), então este valor é atribuido à componente correspondente do vector **v**.

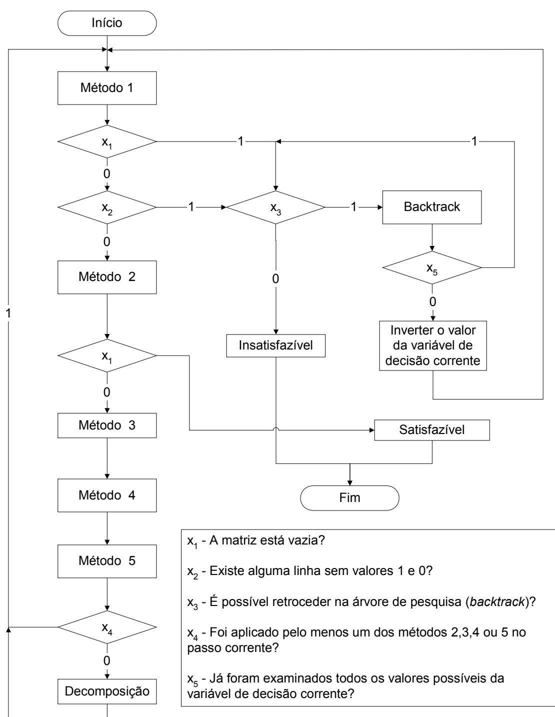

Fig. 1 – O algoritmo de resolução de SAT formulado sobre uma matriz discreta

Na decomposição de uma situação efectua-se o exame sequencial de valores 1 e 0 de alguma componente do vector **v**. Deve ser escolhida a componente que corresponda à coluna mais determinada da matriz $\mathbf{M}'$, i.e. à coluna que tenha o número mínimo de valores '-'. Sendo assim efectua-se a *selecção dinâmica* da variável de decisão seguinte. Para cada variável de decisão o valor 1 é examinado antes do valor 0.

Caso depois de apagar uma linha da matriz $\mathbf{M}'$ esta se torne vazia então o valor corrente do vector **v** representa a solução. Por outro lado se a matriz $\mathbf{M}'$ ficar vazia depois de eliminar uma coluna ou se $\mathbf{M}'$ contiver uma linha sem valores 1 e 0, então é impossível encontrar a solução neste ramo da árvore de pesquisa. Portanto é necessário retroceder até ao último ponto de ramificação no qual o exame da variável de decisão não foi concluido, inverter o valor desta variável e continuar a percorrer a árvore de pesquisa. Caso ao percorrer a árvore completa não seja possível encontrar o vector **v** ortogonal a cada linha da matriz **M** então a função booleana correspondente é insatisfazível.

Consideremos um exemplo. Vamos verificar se a função booleana seguinte é satisfazível:

$$\begin{aligned} f(x_1,...,x_8) &= (x_1 \vee \overline{x}_8) \wedge (x_1 \vee x_3) \wedge \\ &\wedge (x_1 \vee \overline{x}_2 \vee x_7 \vee x_8) \wedge (x_1 \vee \overline{x}_7 \vee x_8) \wedge \\ &\wedge (x_1 \vee x_2 \vee \overline{x}_3 \vee x_8) \wedge (\overline{x}_1 \vee x_2 \vee x_8) \wedge \\ &\wedge (\overline{x}_1 \vee \overline{x}_7 \vee x_8) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee x_7) \end{aligned} \quad (2)$$

A função pode ser transformada na seguinte matriz **M**:

$$\begin{array}{c} \\ \mathbf{M} = \end{array} \begin{array}{c} \begin{array}{cccccccc} \mathbf{x_1} & \mathbf{x_2} & \mathbf{x_3} & \mathbf{x_4} & \mathbf{x_5} & \mathbf{x_6} & \mathbf{x_7} & \mathbf{x_8} \end{array} \\ \left[\begin{array}{cccccccc} 1 & - & - & - & - & - & - & 0 \\ 1 & - & 1 & - & - & - & - & - \\ 1 & 0 & - & - & - & - & 1 & 1 \\ 1 & - & - & - & - & - & 0 & 1 \\ 1 & 1 & 0 & - & - & - & - & 1 \\ 0 & 1 & - & - & - & - & - & 1 \\ 0 & - & - & - & - & - & 0 & 1 \\ 0 & 0 & - & - & - & - & - & - \end{array}\right] \begin{array}{c} 1\\2\\3\\4\\5\\6\\7\\8 \end{array} \end{array}$$

A aplicação do algoritmo considerado à matriz **M** pode ser representada com a árvore de pesquisa seguinte (fig. 2).

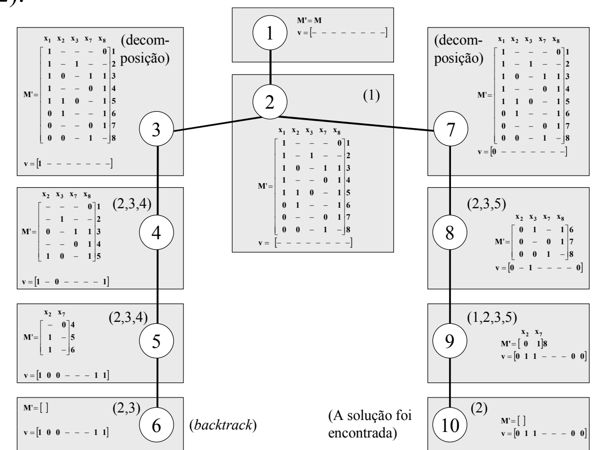

Fig. 2 – Árvore de pesquisa para encontar o vectro **v** ortogonal a cada linha da matriz **M**

Esta árvore é bastante simples e contem um só ponto de ramificação. A numeração de vértices reflecte a ordem por que são examinadas as situações intermédias. A fig. 2 mostra também a evolução do vector **v** e das matrizes $\mathbf{M}'$ no processo de pesquisa. Entre parenteses estão indicados os métodos aplicados em cada passo do algoritmo. É fácil verificar que o vector **v** encontrado é

ortogonal a cada linha da matriz **M**. Sendo assim a função (2) é satisfazível com a seguinte atribuição de valores às variáveis: x_1=1, x_2=0, x_3=0, x_7=1 e x_8=1.

III. ARQUITECTURA

Como foi mencionado na introdução, o tempo de compilação do circuito de hardware limita significativamente a gama de problemas para os quais a utilização da FPGA pode ser considerada razoável em comparação com a solução em software. Tendo isto em atenção tentámos criar um circuito universal que possua uma estrutura predefinida que pode ser reutilizada de problema a problema. Como resultado foi proposta a arquitectura representada na fig. 3.

A arquitectura contem uma unidade de controlo central que executa o algoritmo da fig. 1. A matriz é realizada com base em RAM. Cada elemento da matriz m_{ij} é codificado de maneira seguinte:

- 00 – significa zero lógico;
- 01 – significa um lógico;
- 10 – denota *don't care*;
- 11 – indica que o elemento respectivo não é utilizado, por exemplo este foi eliminado da matriz.

Note-se que esta realização permite recolher uma linha da matriz em um só ciclo de relógio. Por outro lado para ler uma coluna precisamos de aceder a cada linha, extrair dela uns determinados bits e finalmente compor destes bits a coluna.

A ALU é utilizada para calcular o número de zeros, uns e *don't cares* num vector ternário (i.e. em várias linhas e colunas da matriz – para os métodos 1, 4 e 5 na fig. 1) e verificar se dois vectores ternários são ortogonais (i.e. se alguma linha da matriz é ortogonal ao vector **v** – para o método 2 na fig. 1).

O bloco *Registos* guarda a informação seguinte:

- *cur_row* – a linha activa da matriz;
- *cur_col* – a coluna activa da matriz;
- *cur_v* – o valor actual do vector **v**;
- *cur_d* – o valor actual da variável de decisão corrente;
- *del_rows* – vector que indica as linhas apagadas da matriz;
- *del_cols* - vector que indica as colunas apagadas da matriz.

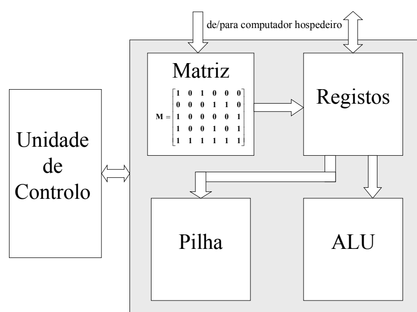

Fig. 3 – Arquitectura do circuito

A pilha é utilizada para suportar o processo de retrocesso na árvore de pesquisa (*backtrack*). Quando decompomos alguma situação, os valores de todos os registos são escritos na pilha e quando retrocedemos na árvore de pesquisa estes são recuperados da pilha para os registos.

Durante a execução do algoritmo a matriz não é modificada. Todas as alterações possíveis (eliminação de linhas e colunas) são armazenadas em registos correspondentes. Sendo assim, não é necessário guardar na pilha as matrizes intermédias.

As dimensões máximas da matriz são fixas: m_{max}*n_{max}. Para o caso de necessidade do processamento de uma matriz com dimensões menores existem dois registos especiais que guardam as dimensões actuais da matriz m_{act}*n_{act}. De acordo com estes valores a unidade de controlo só vai forçar o processamento da área da matriz pretendida.

IV. INTERACÇÃO DE SOFTWARE E HARDWARE RECONFIGURÁVEL

A arquitectura descrita na secção III satisfaz a restrição imposta: esta não é alterada de instância a instância e só precisa de receber vários dados para a matriz **M**. Um outro problema importante é que é impossível guardar e processar em FPGA uma matriz de dimensões arbitrárias. Portanto propomos utilizar a estratégia seguinte. Implementamos o algoritmo considerado numa aplicação de software. No processo de construção da árvore de pesquisa são empregues vários métodos de decomposição e de redução. Como resultado, ao se mover de cima para baixo ao longo de um dos ramos da árvore, as dimensões iniciais da matriz vão diminuindo. Logo que as dimensões novas satisfaçam as restrições do circuito (tais como o número máximo de linhas e colunas da matrix m_{max}*n_{max}) a matriz poderá ser transferida para a FPGA para o processamento posterior.

Esta estratégia está representada na fig. 4. Inicialmente, a aplicação de software deve configurar a FPGA com o circuito respectivo. A seguir, se as dimensões da matriz original satisfazem as restrições predefinidas, então esta poderá ser transferida para a FPGA e o problema inteiro será resolvido em hardware. Em caso oposto a aplicação de software vai resolvendo o problema até que as dimensões da matriz intermédia (que deve ser construída no passo corrente do algoritmo) satisfaçam as restrições. A partir daí a FPGA ficará responsável por todos os passos seguintes. Se o hardware reconfigurável encontrar a solução o problema está resolvido e o resultado será transferido para o computador hospedeiro. Por outro lado, se o ramo da árvore de pesquisa considerado não permitir encontrar a solução, o controlo regressará à aplicação de software que vai continuar a percorrer a árvore de pesquisa até atingir um outro ponto em que a matriz intermédia satisfaz as restrições. Os dados da matriz serão então transferidos para a FPGA e

esta vai tentar resolver o sub-problema. Os passos considerados são repetidos até encontrar a solução ou concluir que não existe solução nenhuma.

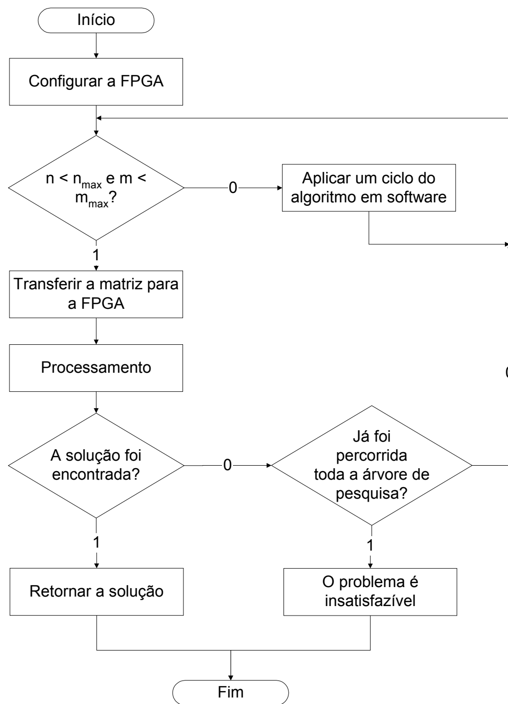

Fig. 4 – Colaboração de software e hardware reconfigurável

V. EXPERIÊNCIAS

Para as experiências foi utilizada a placa com interface PCI ADM-XRC da AlphaData [8] que contém uma FPGA XCV812E [9]. Esta FPGA é composta por um *array* de 56*84 CLBs e incorpora uma memória adicional organizada em blocos distribuidos por todo o dispositivo. Portanto, esta FPGA serve muito bem para a aplicação considerada porque podemos aproveitar a grande quantidade da memória disponível para guardar a matriz e organizá-la de maneira a assegurar a largura de banda necessária no acesso às linhas da matriz. A interacção com a FPGA é feita com a ajuda da biblioteca de interface com a placa ADM-XRC que suporta a inicialização e configuração da FPGA, transferência de dados, processamento de interrupções e erros, gestão de relógio, etc.

Foram implementados 3 circuitos com as seguintes dimensões máximas da matriz em FPGA: 64*64, 128*128 e 256*256 (vamos referenciar estes circuitos como *c64*, *c128* e *c256* respectivamente). Devido ao facto de cada elemento da matriz ser codificado com dois bits, podemos processar matrizes ternárias com as dimensões ≤64*32, ≤128*64 e ≤256*128. Na tabela 1 está representada a informação sobre a área ocupada e a frequência de relógio de cada um dos circuitos implementados. A área está expressada em número de *slices*, sendo cada CLB composto por dois *slices*.

Circuito	Área ocupada (*slices*)	% dos recursos da XCV812E	Frequência do relógio (MHz)
c64	1694	18	40.0
c128	3213	34	40.6
c256	6376	67	33.7

Tabela 1 – Parâmetros de circuitos implementados

Para estimar a eficiência da abordagem proposta realizamos uma série de experiências. Para tal escolhemos o problema *pigeon hole* do DIMACS [10]. Este problema consiste em determinar se é possível pôr *n+1* bolas em *n* buracos sem ter duas bolas no mesmo buraco. Obviamente, isto é impossível, portanto, todas as instâncias são insatisfazíveis. Os parâmetros principais das cinco instâncias disponíveis no conjunto de *benchmarks* do DIMACS estão representados na tabela 2.

Instância	Nº de variáveis	Nº de cláusulas	t_{GRASP} (s)
Hole6	42	133	0.14
Hole7	56	204	4.31
Hole8	72	297	51.574
Hole9	90	415	531.961
Hole10	110	561	5685.6

Tabela 2 – Parâmetros de várias instâncias do problema *pigeon hole*

Este problema requer consumos significativos de tempo quando resolvido em software. As tabelas 3, 4 e 5 contêm os resultados da resolução do problema com a ajuda de uma aplicação desenvolvida em C++ que colabora de acordo com a estratégia descrita na secção IV com os circuitos *c64*, *c128* e *c256* respectivamente implementados em FPGA. Podemos ver que as dimensões iniciais da matriz para cada instância excedem as capacidades dos circuitos *c64* e *c128*. Portanto, nestes casos cada problema é primeiro processado em aplicação de software. Logo que uma matriz intermédia satisfaça as restrições predefinidas do circuito respectivo, esta será transferida para a FPGA e processada lá. As colunas direitas das tabelas 3, 4, e 5 indicam quantas vezes é necessário utilizar a FPGA para cada instância e circuito. Como podemos ver, à medida que a área reservada para a matriz em hardware cresce, aumenta a parte da árvore de pesquisa que é processada em hardware (ver fig. 5) e o número de transferências da matriz para a FPGA diminui.

No nosso caso o tempo de resolução de um problema é:

$$t_{total} = t_{config} + t_{sw} + t_{hw}$$

t_{config} é o tempo de configuração da FPGA com o circuito pretendido. Este varia de 0.31 s para circuito *c64* a 0.35 s para circuito *c256*, e começando com a instância *hole7* torna-se insignificante comparando-o com o tempo de execução em software "puro".

Instância	Dimensões iniciais da matriz	t_{congif} (s)	t_{sw} (s)	t_{hw} (s)	t_{total} (s)	Aceleração	FPGA
Hole6	133 * 42	0.31	0.0985	0.2205	0.629	0.223	60
Hole7	204 * 56		0.79	1.554	2.654	1.624	420
Hole8	297 * 72		7.54	12.419	20.269	2.544	3360
Hole9	415 * 90		73.912	111.442	185.664	2.865	30240
Hole10	561 * 110		879.056	1035.468	1914.834	2.969	302400

Tabela 3 – Resultados das experiências com a arquitectura *c64*

Instância	Dimensões iniciais da matriz	t_{congif} (s)	t_{sw} (s)	t_{hw} (s)	t_{total} (s)	Aceleração	FPGA
Hole6	133 * 42	0.31	0.011	0.103	0.424	0.330	4
Hole7	204 * 56		0.125	0.728	1.163	3.706	28
Hole8	297 * 72		1.195	5.838	7.343	7.024	224
Hole9	415 * 90		11.678	52.819	64.807	8.208	2016
Hole10	561 * 110		146.125	501.988	648.423	8.768	20160

Tabela 4 – Resultados das experiências com a arquitectura *c128*

Instância	Dimensões iniciais da matriz	t_{congif} (s)	t_{sw} (s)	t_{hw} (s)	t_{total} (s)	Aceleração	FPGA
Hole6	133 * 42	0.35	0.00204	0.09276	0.4448	0.315	1
Hole7	204 * 56		0.00278	0.16302	0.5158	8.356	1
Hole8	297 * 72		0.14842	2.41928	2.9177	17.676	14
Hole9	415 * 90		1.62572	21.79388	23.7696	22.379	126
Hole10	561 * 110		17.36361	218.48619	236.1998	24.071	1260

Tabela 5 – Resultados das experiências com a arquitectura *c256*

t_{sw} inclui o tempo de resolução da parte do problema em software (a parte superior da árvore de pesquisa na fig. 5), o tempo necessário para transferência da matriz para a FPGA e para ler o resultado da FPGA. t_{hw} é o tempo de resolução da parte do problema em hardware (a parte inferior da árvore de pesquisa na fig. 5). A aplicação de software foi executada num AMD Athlon/1GHz/256MB com o sistema operativo Windows2000.

Efectuamos uma comparação dos nossos resultados com os conseguidos em GRASP [5] que é um dos mais eficientes algoritmos de resolução de SAT. O GRASP foi também executado num AMD Athlon/1GHz/256MB com o sistema operativo Windows2000 e o tempo necessário para resolver cada instância do *pigeon hole* está indicado na tabela 2. A aceleração conseguida é dada por expressão seguinte: t_{GRASP}/t_{total}. Devido ao facto do circuito em FPGA ser especializado para as dimensões dos dados utilizados pelo algoritmo, as operações são executadas muito mais rápida e eficientemente do que em software. Como resultado, com o crescimento das dimensões máximas da matriz em FPGA, observa-se o aumento da aceleração da nossa implementação em comparação com o GRASP (fig. 6).

Na fig. 7 está demonstrado como dependem t_{sw}, t_{hw} e t_{total} das dimensões da matriz em FPGA. O gráfico foi preparado para a instância *hole9* mas a mesma tendência é válida para todos os outros exemplos (tabelas 3, 4 e 5).

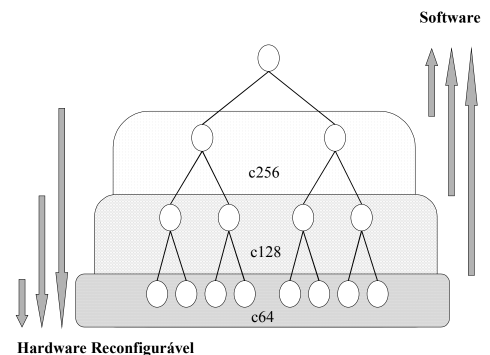

Fig. 5 – Processamento da árvore de pesquisa em software e em hardware

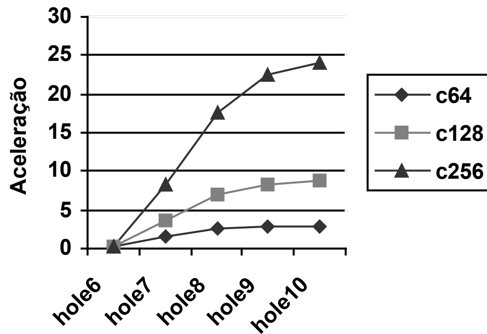

Fig. 6 – Aceleração conseguida com as três arquitecturas implementadas em comparação com o GRASP

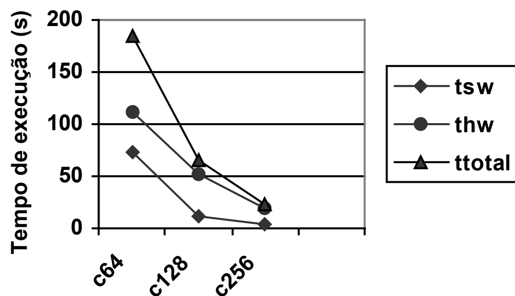

Fig. 7 – Dependência do tempo de resolução da instância *hole9* do tamanho da matriz em FPGA

Na fig. 8 está apresentada a comparação da nossa arquitectura *c256* com a arquitectura descrita em [2, 11] que usa a abordagem optimizada para cada instância do problema.

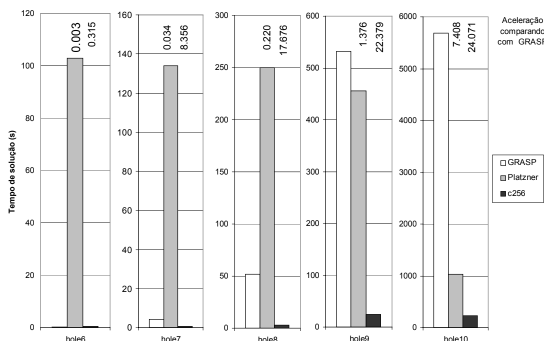

Fig. 8 – Comparação com a arquitectura proposta em [2, 11]

A frequência do relógio do circuito em [2, 11] é de 20 MHz. O tempo de resolução de cada tarefa é igual a:

$$t = t_{hc} + t_{he}$$

onde t_{hc} é o tempo de compilação do circuito de hardware (este domina em todas as instâncias consideradas) e t_{he} é o tempo de execução em FPGA. O tempo t e a aceleração em comparação com o GRASP foram reproduzidos de [11]. Contudo, em [11] o GRASP foi executado num P-II/300MHz/128MB com o sistema operativo Linux. Parece-nos no entanto não ser possível efectuar uma comparação mais exacta porque ao mudar de plataforma de software, o tempo de compilação do hardware também será modificado.

VI. CONCLUSÕES

Neste artigo apresentamos uma arquitectura para resolução de problemas de SAT baseada na colaboração de software e hardware reconfigurável. A técnica proposta permite resolver problemas com dimensões maiores do que a capacidade disponível na plataforma de hardware. Como resultado conseguimos obter uma aceleração até 24x em comparação com o GRASP. Isto é explicado pelas razões seguintes:

- o algoritmo considerado requer a execução de operações simples sobre dados regulares, permitindo a construção de uma ALU optimizada para a aplicação.
- a organização da memória onde é guardada a matriz é personalizada para os tamanhos de dados do problema.

Note-se que uma desvantagem da arquitectura proposta é que para ler uma coluna da matriz são necessários muitos acessos à memória, como é explicado na secção III. Portanto, o trabalho futuro será concentrado em resolver este problema e, em consequência, melhorar significativamente o desempenho dos circuitos.

REFERÊNCIAS

[1] D.Abramson et al., “FPGA Based Custom Computing Machines for Irregular Problems”, Proc. of the HPCA98, Fevereiro de 1998, Las Vegas, Nevada.

[2] M.Platzner, “Reconfigurable Accelerators for Combinatorial Problems”, IEEE Computer, pp. 58-60, April de 2000.

[3] P.Zhong, et al, “Solving Boolean Satisfiability with Dynamic Hardware Configurations”, Proc. of the FPL’1998.

[4] M.Abramovici, J.T.de Sousa, “A SAT Solver Using Reconfigurable Hardware and Virtual Logic”, Journal of Automated Reasoning, Fevereiro de 2000.

[5] J.M.Silva, K.A.Sakallah, “GRASP – A New Search Algorithm for Satisfiability”, Proc. of the Int. Conference on CAD, pp. 220-227, Novembro de 1996.

[6] A.D.Zakrevski, “Logical Synthesis of Cascade Networks”, Moscow: Nauka, 1981 (em russo).

[7] I.Skliarova, A.B.Ferrari, "Modelos matemáticos e problemas de optimização combinatória”, Electrónica e Telecomunicações, vol.3, Nº3, Janeiro de 2001, pp. 202-208.

[8] http://www.alphadata.co.uk

[9] Xilinx, “The programmable Logic Data Book”, San Jose, 2000.

[10] http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html

[11] O.Mencer, M.Platzner, “Dynamic Circuit Generation for Boolean Satisfiability in an Object-Oriented Design Environment”, Proc. of HICSS-32, 1999.