Aplicando Verificação de Modelos para o Particionamento de Hardware/Software

Alessandro Bezerra Trindade
Universidade Federal do Amazonas (UFAM)
Manaus, Amazonas, Brasil
alessandro.b.trindade@gmail.com

Lucas Carvalho Cordeiro
Universidade Federal do Amazonas (UFAM)
Manaus, Amazonas, Brasil
lucascordeiro@ufam.edu.br

***Resumo*—Quando se realiza um coprojeto de hardware/software para sistemas embarcados, emerge o problema de se decidir qual função do sistema deve ser implementada em *hardware* (HW) ou em *software* (SW). Este tipo de problema recebe o nome de particionamento de HW/SW. Na última década, um esforço significante de pesquisa tem sido empregado nesta área. Neste artigo, são apresentadas duas novas abordagens para resolver o problema de particionamento de HW/SW usando a técnica de verificação de modelos com a ferramenta ESBMC (*Efficient SMT-Based Bounded Model Checker*). São comparados os resultados obtidos com a tradicional técnica de programação linear inteira e com o método moderno de otimização por algoritmo genético. O objetivo é demonstrar, com os resultados empíricos, que as técnicas de verificação de modelo podem ser efetivas, em casos particulares, para encontrar a solução ótima do problema de particionamento HW/SW, quando comparado com técnicas usuais.**

Palavras-chave—coprojeto hardware/software; codesign; sistemas embarcados; particionamento; programação linear inteira; algoritmo genético; verificação formal

I. INTRODUÇÃO

A complexidade dos sistemas computacionais não para de crescer e a demanda para encurtar o tempo de um produto chegar ao mercado causa problemas desafiadores para os projetistas. Sistema computacional neste contexto significa ter partes dos componentes implementados em hardware (HW) e outras partes em software (SW). Um exemplo de HW seria um ASIC ou qualquer outro processador de propósito único. Um exemplo de SW seria um processador de propósito geral, como um microprocessador. Componentes em HW são usualmente muito mais rápidos do que os de SW, entretanto são significativamente mais caros. Os componentes de SW, por outro lado, são mais baratos de criar e manter, mas são mais lentos. Durante o coprojeto de HW/SW (HSCD) o passo mais crítico do HSCD é o particionamento: escolher quais componentes devem ser colocados em HW e quais em SW, para atendimento das restrições de desempenho e preço.

Na chamada primeira geração de coprojeto, a plataforma é a de uma única unidade central de processamento (CPU) e de um único ASIC com ambos se comunicando por um barramento e usando memória compartilhada ou *buffers*, conforme Arató et al. [1]. Neste artigo, será aplicado um método de verificação de modelos baseado em Teorias do Módulo da Satisfabilidade (*Satisfability Modulo Theories* ou SMT em inglês) para resolver o problema do particionamento de HW/SW e então comparar os resultados com as técnicas tradicionais de otimização, como a programação linear inteira (ILP) e o algoritmo genético (GA). Trabalhos similares comparam os resultados de particionamento entre diferentes técnicas de otimização como visto em Arató et al. [1], Mann et al. [2], Wang e Zhang [3], Sapienza et al. [4], Jianliang e Manman [5] e Li et al. [6].

Em qualquer projeto de HW e SW de sistemas complexos, mais tempo é gasto em verificação do que na construção, conforme Baier e Katoen [7]. Métodos formais baseados em verificação de modelos oferecem grande potencial para se obter uma verificação efetiva e rápida no processo de projeto. Matemática aplicada para modelagem e análise de sistemas computacionais caracteriza os métodos formais, objetivando estabelecer corretude do sistema através de rigor matemático.

Trazendo o conceito para o particionamento de HW/SW, um verificador de modelo poderia analisar um código formalmente especificado em uma dada linguagem onde cada componente do sistema, especificado por meio de objetivos claros (minimizar o custo de HW) e requisitos (custo de SW), e explorar sistematicamente todos os seus estados até uma solução ótima ser encontrada. Salvo melhor juízo ou convicção, a proposta aqui apresentada é o primeiro trabalho na qual se usa uma ferramenta de verificação de modelos baseada em Teorias do Módulo da Satisfabilidade para resolver um problema de particionamento. A proposta é implementada com a ferramenta ESBMC (*Efficient SMT-Based Bounded Model Checker*), baseado em Cordeiro et al. [8]. O artigo visa provar que é possível usar ferramenta de verificação de modelos para resolver problemas de particionamento de HW/SW, sem que nenhuma adaptação se faça na ferramenta.

Uma contribuição secundária reside na avaliação do valor inicial do custo de software na definição dos requisitos da modelagem matemática do particionamento HW/SW, o que não foi feito em trabalhos similares. Busca-se definir um valor que possibilite melhorar o desempenho das técnicas.

II. MODELAGEM MATEMÁTICA

A. Modelagem informal

O modelo possui cinco características, conforme definido em Arató [1], Mann [2] e Arató [16]: Há apenas um contexto de software e apenas um contexto de hardware, ou seja, os componentes do sistema só podem ser mapeados para um dos dois contextos; A implementação em software de um componente tem um custo de software, que é o tempo de execução deste componente. A implementação em hardware de um componente tem um custo (área, dissipação de calor ou consumo de energia). Como o hardware é significante mais rápido do que o software, o tempo de execução de um componente em hardware é considerado zero. Se dois componentes são mapeados para o mesmo contexto, então não

há *overhead* (ou custo) de comunicação. Portanto o escalonamento não se faz necessário. Esta configuração, de acordo com Teich [9] descreve o coprojeto de primeira geração. Na segunda geração, múltiplos núcleos tornam a arquitetura mais complexa. E na terceira, a tecnologia se torna heterogênea, com a inclusão de periféricos, por exemplo. Neste artigo, apenas o coprojeto de primeira geração é tratado.

B. *Modelagem formal*

Um grafo simples direcionado $G = (V, E)$, chamado de grafo de tarefas do sistema ou grafo acíclico orientado (DAG, do inglês *directed acyclic graph*). Os vértices $V = \{v_1, v_2, \ldots, v_n\}$ representam os nodos que são os componentes do sistema a ser particionado. As arestas (E) representam as comunicações entre os componentes. Adicionalmente, cada nodo v_i tem um custo de hardware $h(v_i)$, se for implementado em hardware e um custo de software $s(v_i)$, se for implementado em software, $c(v_i, v_j)$ representam os custos de comunicação entre v_i e v_j se forem implementados em contextos componentes (hardware ou software).

Baseado em Arató et al. [1], P é chamado uma partição de hardware/software se é uma bipartição de V: $P = (V_H, V_S)$, onde $V_H \cup V_S = V$ e $V_H \cap V_S = \emptyset$. As arestas cruzadas são $E_P = \{(v_i, v_j): v_i \in V_S, v_j \in V_H \text{ ou } v_i \in V_H, v_j \in V_S\}$. O custo de hardware de P dado pela Equação (1), e o custo de software P, ou seja, o custo de software dos nodos e o custo de comunicação, é dado pela Equação (2):

$$H_P = \sum_{v_i \in V_H} h_i \qquad (1)$$

$$S_P = \sum_{v_i \in V_S} s_i + \sum_{(v_i, v_j) \in Ep} c(v_i, v_j) \qquad (2)$$

O problema de otimização usado neste artigo será o mais comum encontrado em sistemas de tempo real, no qual o custo de software inicial é dado. Busca-se encontrar uma partição P de HW-SW tal que SP ≤ S0 e o H seja mínimo.

Com relação a complexidade deste tipo de problema, Arató et al. [1] demonstraram que é *NP-Hard*.

C. *Aplicando o modelo a um exemplo*

Exemplifica-se aqui o processo de particionamento de HW/SW de um sistema embarcado de 10 nodos e 13 arestas, conforme Figura 1. O mesmo valerá para mais nodos.

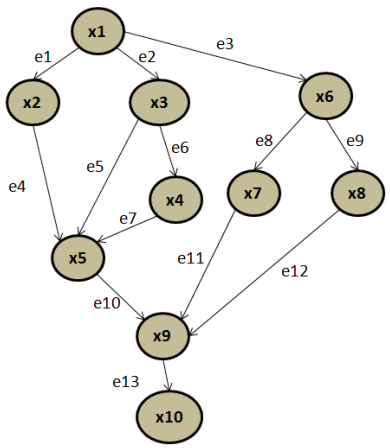

Fig. 1. Grafo direcionado do sistema a ser particionado (Fonte: autoria própria)

Os dados fornecidos para cada nodo são os seguintes: custo de hardware de cada nodo $h = \{5, 10, 3, 4, 8, 5, 10, 3, 4, 8\}$, custo do software de cada nodo $s = \{10, 15, 7, 4, 9, 10, 15, 7, 4, 9\}$, e o custo de comunicação de cada aresta se implementado em contextos diferentes $c = \{5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5\}$.

Para criar o algoritmo, é necessário criar uma matriz chamada $E \in \{-1, 0, 1\}^{13x10}$, que é a matriz incidente transporta de G, ou seja,

$$E[i,j] := \begin{cases} -1 & se\ a\ aresta\ i\ começa\ no\ nodo\ j \\ 1 & se\ a\ aresta\ i\ termina\ no\ nodo\ j \\ 0 & se\ a\ aresta\ i\ não\ é\ incidente\ ao\ nodo\ j \end{cases}$$

Seja $x \in \{0, 1\}^{10}$ um vetor binário indicando a partição, ou seja,

$$x[i] := \begin{cases} 1\ se\ o\ nodo\ i\ for\ particionado\ para\ Hardware \\ 0\ se\ o\ nodo\ i\ for\ particionado\ em\ software \end{cases}$$

Pode ser visto que os componentes do vetor $|Ex|$ indicam as arestas que cruzam o limite entre os dois contextos.

O problema de otimização usado em todas as técnicas demonstradas neste artigo é formulado como:

$$Minimizar\ hx \qquad (3a)$$

Sujeito as restrições:

$$s(1-x) + c|Ex| \leq S_0 \qquad (3b)$$

$$x \in \{0, 1\}^{10} \qquad (3c)$$

A formulação acima pode ser transformada em uma ILP equivalente pela introdução das variáveis auxiliares $y \in \{0, 1\}^{13}$ para eliminar o módulo da Equação 3b.

$$Minimizar\ hx \qquad (4a)$$

Sujeito as restrições:

$$s(1-x) + cy \leq S_0 \qquad (4b)$$

$$Ex \leq y \qquad (4c)$$

$$-Ex \leq y \qquad (4d)$$

$$x \in \{0, 1\}^{10} \qquad (4e)$$

Ferramentas comerciais não aceitam a forma das Equações (4b), (4c), (4d) e precisam ser reescritas na representação:

$$\min_x f^T x\ tal\ que\ A.x \leq b \qquad (5)$$

Demanda-se transformar as Equações (4b), (4c) e (4d) em uma única matriz: decompondo as três equações em termos das variáveis de decisão x e y, e depois as colocando de volta em uma única matriz e isolando qualquer variável para o lado esquerdo da equação. Começando pela Equação (4b):

$$[10\ 15\ 7\ 4\ 9\ 10\ 15\ 7\ 4\ 9] \begin{bmatrix} 1-x_1 \\ 1-x_2 \\ 1-x_3 \\ 1-x_4 \\ 1-x_5 \\ 1-x_6 \\ 1-x_7 \\ 1-x_8 \\ 1-x_9 \\ 1-x_{10} \end{bmatrix} + [5\ 5\ 5\ 5\ 5\ 5\ 5\ 5\ 5\ 5\ 5\ 5\ 5] \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \\ y_{11} \\ y_{12} \\ y_{13} \end{bmatrix} \leq S_0 \qquad (6)$$

Ou ainda, em notação algébrica:

$$-10x_1 - 15x_2 - 7x_3 - 4x_4 - 9x_5 - 10x_6 - 15x_7 - 7x_8 - 4x_9 - 9x_{10} + 5y_1 + 5y_2 + 5y_3 + 5y_4 + 5y_5 + 5y_6 + 5y_7 + 5y_8 + 5y_9 + 5y_{10} + 5y_{11} + 5y_{12} + 5y_{13} \leq S_0 - 90 \qquad (7)$$

E de volta na representação de matriz, com todas as variáveis do lado esquerdo:

$$[10\ 15\ 7\ 4\ 9\ 10\ 15\ 7\ 4\ 9\ 10\ 15\ 7\ 4\ 9\ 5\ 5\ 5\ 5\ 5\ 5\ 5\ 5\ 5\ 5\ 5\ 5].\begin{bmatrix} x_1 \\ \vdots \\ x_{10} \\ y_1 \\ \vdots \\ y_{13} \end{bmatrix} \leq S_0 - 90 \quad (8)$$

Fazendo o mesmo para as Equações (4c) e (4d), e colocando-as, todas juntas, resulta na Equação (9).

$$\begin{bmatrix} -S_{1\,xV} & C_{1\,x\,E} \\ E_{E\,x\,V} & -I_{ExE} \\ -E_{ExV} & -I_{ExE} \end{bmatrix}_{(1+2*E)x\,(V+E)} \cdot \begin{bmatrix} x_{V\,x\,1} \\ y_{E\,x\,1} \end{bmatrix}_{(V+E)x1} \leq \begin{bmatrix} S_0 - 9 * V_{1x1} \\ 0_{E\,x\,1} \\ 0_{E\,x\,1} \end{bmatrix}_{(1+2*E)x\,1} \quad (9)$$

III. Algoritmo baseado em Programação Linear Inteira: Programação Inteira Binária

Para a formulação ILP foi usada a ferramenta matemática MATLAB na versão R2013a e com o toolbox de Otimização MathWorks [10].

As Equações (4a) e (9) foram usadas, bem como a função *bintprog* do MATLAB, que resolve problemas de programação inteira binária. Esta função usa um algoritmo de *branch-and-bound* que busca uma solução ótima através de uma série de problemas de relaxação, nas quais os requisitos das variáveis de serem inteiras binárias são substituídas pela limitação $0 \leq x \leq 1$. De acordo com Daskalaki et al. [11], o algoritmo pode ser descrito como um processo de 3 etapas: Primeiro ele busca por uma solução inteira factível (que atende os requisitos). Então ele atualiza o ponto de melhor inteiro binário factível encontrado até então à medida que a busca em árvore cresce. E finalmente, verifica se nenhuma solução factível melhor é possível.

A Figura 2 apresenta o esqueleto do programa no MATLAB. As etapas do método descrito aqui são realizadas na linha 11, com a chamada da função *bintprog*.

1	Declarar o número de nodos e arestas
2	Declarar custo de hardware de cada nodo como um vetor (h)
3	Declarar o custo de software de cada nodo como um vetor (s)
4	Declara o custo de comunicação de cada aresta como um vetor (c)
5	Declarar o custo do software inicial (S0)
6	Declarar a matriz incidental transposta do grafo G (E)
7	Criar matriz identidade I (dimensão = num. de arestas x num. de arestas)
8	Criar a matriz A
9	Criar a matriz b
10	Iniciar cronômetro
11	Resolver ILP: função *bintprog*, passando h, A e b como parâmetros
12	Parar cronômetro
13	Apresentar a solução

Fig. 2. Esqueleto do programa MATLAB para ILP (Fonte: autoria própria)

IV. Algoritmo genético (GA)

Para o GA também foi usada a ferramenta matemática MATLAB, versão R2013 com o toolbox de otimização [10].

GAs fazem parte dos chamados métodos modernos ou não tradicionais de otimização, de acordo com Rao [12]. Alguns autores, como Belegundu [13], nomeiam GA não como uma ferramenta de otimização, mas como uma busca heurística, pois não garante a melhor solução global do problema.

Os elementos da genética natural, como reprodução, cruzamento e mutação, são usados nesta técnica. GAs seguem passos bem definidos, conforme Mitchell [14]: Uma codificação inicial de indivíduos (zeros e uns) deve ser definida. O problema de particionamento é muito feliz porque a possível solução é representada por um vetor binário. Depois, uma população de indivíduos é gerada randomicamente; O 3º passo é a criação de uma nova população, ou geração: cada membro da população atual recebe uma pontuação através do cálculo do seu valor de *fitness* (custo de hardware). A partir daí, os membros com melhor pontuação são classificados como elite e serão os pais usados para produzir a população da iteração seguinte. Os filhos serão produzidos através de mudanças randômicas dos bits de um único pai (mutação) ou pela combinação dos valores dos vetores de um par de pais (cruzamento). Dessa forma, o GA substitui a população atual com os filhos que formarão a próxima geração. O passo final define como o algoritmo será parado: convergência ou número de iterações.

A Figura 3 mostra o esqueleto do programa de particionamento com GA. Todas as etapas detalhadas aqui são realizadas na chamada da função *ga* na linha 16.

1	Declarar o número de nodos e arestas
2	Declarar custo de hardware de cada nodo como um vetor (*h*)
3	Declarar o custo de software de cada nodo como um vetor (*s*)
4	Declara o custo de comunicação de cada aresta como um vetor (*c*)
5	Declarar o custo do software inicial (S_0)
6	Declarar a matriz incidental transposta do grafo *G (E)*
7	Criar matriz identidade *I* (dimensão = núm. de arestas x núm. de arestas)
8	Criar a matriz A
9	Criar a matriz b
10	Definir limite inferior da solução como zero para todas as variáveis (*lb*)
11	Definir limite superior da solução como um para todas as variáveis (*ub*)
12	Definir a função *fitness* como o custo de hardware
13	Declarar que todas as variáveis deverão assumir valores inteiros (*intcon*)
14	Mudar o valor padrão da população para 300
15	Iniciar cronômetro
16	Resolver GA: função *ga*, passando *h, A, b, lb, ub* e *intcon* como parâmetros
17	Parar Cronômetro
18	Apresentar a solução

Fig. 3. Esqueleto do programa MATLAB para GA (Fonte: autoria própria)

V. Análise do problema de particionamento com ESBMC

Inicia-se o processo criando-se um programa em Linguagem C/C++. Em seguida, o ESBMC realiza um *scan* (ou *lexer*) no programa, transformando o código em uma lista de fichas (ou *tokens*) para poder identificar e separar os comandos, operadores lógicos, números e parênteses. Em seguida, um *parse* transforma as fichas em uma árvore de sintaxe.

Para se chegar ao grafo de fluxo de controle (ou programa *goto*), deve-se informar as propriedades que serão verificadas. No problema de particionamento, todas as verificações padrões podem ser suprimidas, mas incluindo-se uma assertiva que irá controlar a solução do problema e causar o contraexemplo, conforme linha 18 das Figuras 5 e 6. O resultado desta etapa é um código que substitui os *loops* por *gotos*. Em seguida ocorre uma execução simbólica do programa *goto*, gerando-se o que se conhece por SSA (*static single assignment form* em inglês), ou designação única estática.

Em seguida, fórmulas que correspondem às condições de verificação (as restrições e as propriedades) são geradas a partir do SSA e passadas para o solucionador, que por sua vez irá verificar a satisfabilidade destas fórmulas. A visão geral de todas as etapas descritas aqui para o método de verificação baseado em Teorias do Módulo da Satisfabilidade (ESBMC) está apresentada na Figura 4.

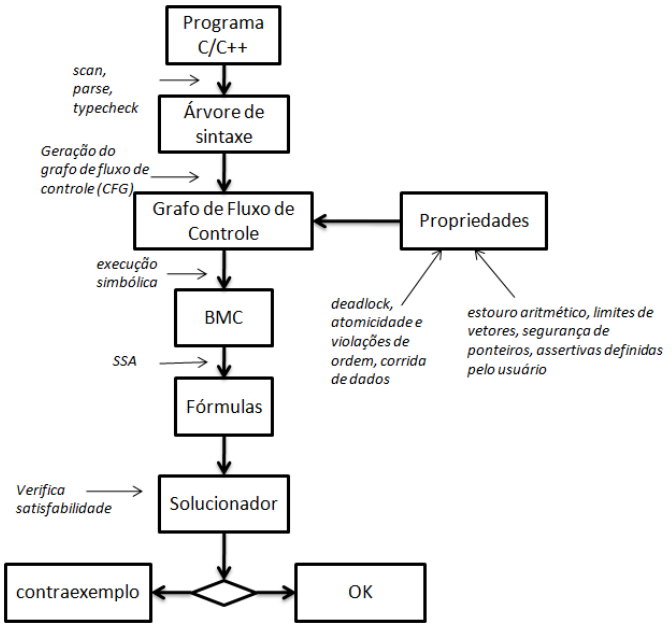

Fig. 4. Visão geral do ESBMC (Fonte: autoria própria)

A Figura 5 mostra o uso da técnica ESBMC com as mesmas restrições e condições impostas ao ILP e ao GA.. Até a linha 11 o código é praticamente igual. Pela sua característica de verificador de modelos, é possível definir com que tipo de valores as variáveis serão testadas. Um *loop* inicia com a declaração para povoar as variáveis com valores booleanos não-determinísticos. Estes valores mudarão a cada verificação, buscando uma possível solução do problema, com o ASSUME garantindo o atendimento das restrições ($Ax \leq b$).

No fim do algoritmo, um predicativo ASSERT controla a condição de parada. Este testa a função objetivo. Se o ASSERT retornar uma condição de VERDADEIRO, ou seja, o custo do hardware encontrado for maior que a solução ótima, então o verificador de modelos reinicia e uma nova possível solução é gerada e testada até que ASSERT gere uma condição de FALSO. Uma condição de FALSO no tempo de execução aborta a execução do ESBMC e este apresenta o contraexemplo que causou a parada. Neste ponto a solução é apresentada (o custo do hardware).

Uma segunda versão do algoritmo do ESBMC foi criada, conforme Figura 6. No ESBMC-2 não se faz necessário adicionar variáveis auxiliares, usa-se as Equações (3).

```
1  Inicializar variáveis
2  Declarar o número de nodos e arestas
3  main {
4      Declarar custo de hardware de cada nodo como um vetor (h)
5      Declarar o custo de software de cada nodo como um vetor (s)
6      Declarar o custo de comunicação de cada aresta como um vetor (c)
7      Declarar o custo do software inicial (S0)
8      Declarar a matriz incidental transposta do grafo G (E)
9      Criar matriz identidade I (dim = núm. de arestas x núm. de arestas)
10     Criar a matriz A
11     Criar a matriz b
12     Definir as variáveis de solução (xi) como Booleanas
13     Realizar verificação {
14         Povoar xi com valores não determinísticos
15         Armazenar na matriz "temp" o resultado da matriz A(i x j)
           multiplicada por xi
16         Fazer o teste ESBMC (comando ASSUME) para o requisito
           "temp" ≤ b (i.e. Ax ≤ b)
17         Calcular a função objetivo baseada no valor factível de xi
           encontrado pelo ESBMC
18         ASSERT (função objetivo > Hp)
19     }
20     Verificação formal realiza testes sistemáticos com diferentes valores
       de xi
21 }
```

Fig. 5. Esqueleto do ESBMC Algoritmo 1 (ESBMC-1) (Fonte: autoria própria)

```
1  Inicializar variáveis
2  Declarar o número de nodos e arestas
3  main {
4      Declarar custo de hardware de cada nodo como um vetor (h)
5      Declarar o custo de software de cada nodo como um vetor (s)
6      Declarar o custo de comunicação de cada aresta como um vetor (c)
7      Declarar o custo do software inicial (S0)
8      Declarar a matriz incidental transposta do grafo G (E)
9      Criar matriz identidade I (dim. = num de arestas x num de arestas)
10     Criar a matriz A
11     Criar a matriz b
12     Definir as variáveis de solução (xi) como Booleanas
13     Realizar verificação {
14         Povoar xi com valores não determinísticos
15         Calcular s(1-x)+c*|E*x| e armazenar na variável "restrição"
16         Realizar teste ESBMC (comando ASSUME) para o requisito
           "restrição" ≤ S0
17         Calcular a função objetivo baseada no valor factível de xi
           encontrado pelo ESBMC
18         ASSERT (função objetivo > Hp)
19     }
20     Verificação formal realiza testes sistemáticos com diferentes valores
       de xi
21 }
```

Fig. 6. Esqueleto do ESBMC Algoritmo 2 (ESBMC-2) (Fonte: autoria própria)

VI. Resultados Empíricos

Os resultados são divididos em duas partes. A 1ª realizada com vetores de teste de autoria própria para avaliar o impacto da mudança do valor inicial de S_0. Na 2ª fase de testes, S_0 é fixo e se utilizou *benchmarks*.

Foi usada a versão 1.21.1 de 32 bits do ESBMC. O solucionador de SMT foi o Z3 na versão 3.2. O código foi escrito em ANSI-C. Utilizou-se MATLAB R2013a da MathWorks [10] para ILP e o GA. Um laptop HP ProBook 4430s com com MS-Windows 7 de 64-bits, 8GB de RAM e processador i5 da Intel com *clock* de 2,3GHz foi a plataforma dos testes.

Cada tempo foi medido três vezes, e uma média aritmética foi calculada. Uma condição de *time out* (TO) representa um tempo de execução maior que 7200 segundos sem se obter uma solução. Uma solução não factível (NFS, em inglês) é alcançada quando um algoritmo termina porque alcançou seu limite de tempo de execução ou iterações, encontrando uma solução, mas sem satisfazer as restrições. E o *memory out* (MO) ocorre quando não consegue mais alocar memória.

De uma forma geral, os resultados aqui apresentados são dependentes do tipo de máquina e da configuração usada nos testes empíricos. Eventuais mudanças no código da ferramenta ESBMC também impactam nos resultados desta técnica. E uma possível mudança do MATLAB para outra ferramenta, no que concerne os algoritmos ILP e GA, pode trazer mudanças nos tempos medidos destas técnicas.

A. Mudando o Valor de S_0

A Tabela 1 mostra os resultados obtidos. Para cada número de nodos, cinco diferentes valores de S_0 foram selecionados, indo desde zero (tudo particionado para hardware) a um valor onde todos os nodos são alocados em software. No meio de cada faixa foram escolhidos valores de S_0 equidistantes.

Com 10 nodos, o ESBMC-2 foi o mais rápido. A técnica GA teve o pior desempenho, e com um erro de 18,20%. A influência de S_0 foi significante na maioria das técnicas: ILP teve uma diferença de cinco vezes entre a solução mais rápida e a mais lenta; e a ESBMC teve uma diferença de desempenho de duas vezes entre os extremos de tempo medidos.

No grafo de 25 nodos, ILP teve o melhor desempenho, sendo 6,4 vezes mais rápido que ESBMC-2 e 12,99 vezes mais rápido que ESBMC-1. GA apresentou um desempenho intermediário, mas com um erro médio de cerca de 42%. S_0 teve alto impacto: na técnica ILP, a diferença entre as soluções foi de 4,69 vezes; no GA o tempo manteve-se praticamente constante, enquanto no ESBMC a diferença de desempenho foi de cerca de 21,9 vezes.

TABLE I. RESULTADOS EMPÍRICOS DE S_0 COM 10, 25, 50 E 100 NODOS (FONTE: AUTORIA PRÓPRIA)

Nodos (arestas)	S0	Solução exata		ILP		GA		ESBMC-1		ESBMC-2	
		Hp	Sp	T(s)	Hp	T(S)	Hp	T(S)	Hp	T(S)	Hp
10 (13)	0	60	0	1,20	60	3,06	60	0,26	60	0,28	60
	23	52	14	1,71	52	2,15	56	0,33	52	0,29	52
	45	38	43	1,00	38	2,23	51	0,45	38	0,52	38
	67	30	55	0,54	30	2,20	39	0,33	30	0,32	30
	90	0	90	0,34	0	2,19	26	0,26	0	0,58	0
25 (33)	0	150	0	0,45	150	3,63	150	1,84	150	1,21	150
	56	120	50	0,82	120	3,05	142	6,97	120	5,88	120
	112	87	111	1,16	87	2,90	136	33,55	87	12,90	87
	168	48	166	1,69	48	2,95	95	14,42	48	5,88	48
	225	0	225	0,36	0	2,84	74	1,53	0	2,90	0
50 (64)	0	300	0	0,46	300	6,24	NFS	20,25	300	17,46	300
	112	236	112	0,92	236	4,58	264	65,19	236	52,72	236
	224	173	223	21,04	173	4,20	223	220,42	173	86,51	173
	336	92	333	77,71	92	4,34	197	1375,33	92	233,39	92
	450	0	450	0,38	0	4,12	155	19,65	0	4,18	0
100 (132)	0	600	0	0,58	600	33,55	NFS	786,09	600	99,69	600
	225	462	224	143,16	462	17,65	NFS	2696,68	462	TO	462
	450	311	448	29,01	311	10,85	436	3668,07	311	TO	311
	675	165	674	225,14	165	7,62	410	TO	165	TO	165
	900	0	900	0,40	0	7,48	305	405,07	0	4929,68	0

LEGENDA: TO = *time out* NFS = solução não factível

Para 50 nodos GA foi o mais rápido. Entretanto, o erro médio da solução apresentada pelo GA foi de cerca de 52,4%. ESBMC-2 foi 3,9 vezes mais lento que a ILP, e ESBMC-1 foi 16,92 vezes mais lento que a solução pela ILP. Com relação a S_0, na ILP a diferença entre os desempenhos foi de 202 vezes, na técnica com ESBMC a diferença ficou em 70 vezes.

Com 100 nodos, GA novamente foi o mais rápido, mas como um erro médio de 94,65%. ILP foi o segundo colocado com relação a desempenho, sendo cinco vezes mais lento que GA, mas 245 vezes mais rápido que ESBMC-1 e 524 vezes mais rápido que ESBMC-2. Desta vez, o algoritmo ESBMC-1 foi mais rápido que o ESBMC-2. S_0 causou uma diferença no tempo de cerca de 562 vezes no algoritmo ILP, 28 vezes no ESBMC-1 e 140 vezes no ESBMC-2.

A Tabela 1 mostra que se o problema de particionamento tiver 10 ou menos nodos, então ESBMC-1 e ESBMC-2 são as melhores opções. Iniciando com 25 nodos e maiores, ILP foi a melhor em termos de desempenho e corretude. Especificamente com relação ao ESBMC, se o problema de particionamento tiver menos de 100 nodos, então o algoritmo ESBMC-2 é melhor do que o ESBMC-1, por ser mais rápido. A técnica com GA apresentou o melhor desempenho com mais de 25 nodos, mas sem uma corretude aceitável.

B. Testes de *Benchmark*

Para realizar a Parte 2 dos testes, alguns vetores de teste fornecidos por Mann et al. [2] foram usados. A Tabela 2 lista esses *benchmarks*. Os vértices nos grafos correspondem a instruções de linguagem de alto nível. Custos de software e de hardware são dimensionais em relação ao tempo, e os custos de hardware representam a área ocupada no *chip*.

TABLE II. DESCRIÇÃO DOS *BENCHMARKS* USADOS (FONTE: AUTORIA PRÓPRIA)

Nome	Nodos	Arestas	Descrição
CRC32	25	32	Verificação de redundância cíclica de 32-bits. Do MiBench [15]
Patricia Insert	21	48	Rotina de inserção de valores. Usado para armazenar tabelas. Do MiBench [15]
Dijkstra	26	69	Calcula o caminho mais curto de um grafo. Da categoria de rede do MiBench[15]
Clustering	150	331	Algoritmo de segmentação de imagens de uma aplicação médica
RC6	329	448	Grafo de criptografia RC6
Fuzzy	261	422	Algoritmo de clustering com lógica fuzzy
Mars	417	600	Código MARS da IBM
Random1	400	400	Grafo de comunicação randômico

TABLE III. TESTES EMPÍRICOS COM OS *BENCHMARKS* (FONTE: AUTORIA PRÓPRIA)

Nome	Nodos (arestas)	S0	Solução Exata		ILP		GA		ESBMC-1		ESBMC-2	
			Hp	Sp	T(S)	Hp	T(S)	Erro	T(S)	Hp	T(S)	Hp
CRC32	25 (32)	20	20	0	0,51	20	2,96	10,0%	1,63	20	0,94	20
Patricia	21 (48)	10	47	4	0,49	47	3,57	18,4%	5,45	47	2,28	47
Dijkstra	26 (69)	20	40	0	0,62	40	5,00	10,8%	17,17	40	3,10	40
Clustering	150 (331)	50	256	3	14,81	256	347,93	50,2%	MO	-	1201,31	256
RC6	329 (448)	600	703	535	1029,67	703	2439,88	18,7%	MO	-	TO	-
Fuzzy	261 (422)	4578	13579	2129	TO	-	1873,99	38,6%	MO	-	TO	-
Mars	417 (600)	300	887	300	1240,90	887	6078,49	37,8%	MO	-	TO	-
Random1	400 (400)	9383	15560	9366	TO	-	2245,81	11,5%	MO	-	TO	-

LEGENDA: MO = *memory out* TO = *time out*

De uma forma geral o desempenho demonstrado pela técnica ILP nos *benchmarks* foi o melhor. GA foi a única técnica que conseguiu resolver todos os *benchmarks*, mas o erro da solução apresentada ficou entre 10,8% e 50,2%. Mais uma vez, o ESBMC-2 teve um desempenho melhor do que o ESBMC-1, chegando na solução exata do problema em menos tempo (de 1,7 a 5,7 vezes mais rápido).

Até se chegar a 150 nodos, a técnica ESBMC-2 se mostrou uma boa escolha para resolver o problema de particionamento. Quando a complexidade dos vetores de teste aumentou, o ESBMC-1 apresentou o problema de *memory out*. Iniciando em 261 nodos, o algoritmo ESBMC-2 também foi incapaz de achar a solução, pois excedeu o limite de tempo.

VII. TRABALHOS RELACIONADOS

Em Arató et al.[1], Mann et al. [2] e Arató et al. [16], o problema de particionamento foi formalizado e algoritmos foram empregados para resolvê-lo. A partir da 2ª metade da década de 2000, três caminhos foram trilhados.

Há o grupo da solução exata do problema de particionamento. Destaca-se o trabalho de Mann et al. [2], que desenvolveu um algoritmo *branch-and-bound* próprio. Wang e Zhang [3] apresentaram um algoritmo guloso. Sapienza et al. [4] usam múltiplos critérios para análise de decisão. A proposta deste artigo faz parte deste grupo, pois encontra a solução exata. A diferença está apenas no método.

Há o caminho da aplicação de heurísticas para acelerar a resposta do problema. Arató et al. [1] usa algoritmo genético para resolver o problema de otimização. Em Bhattacharya et al. [17] a solução é obtida por meio de um algoritmo de enxame de partículas. Wang et al. [18] usaram um algoritmo de otimização por colônia de formigas. Luo et al. [19] usaram um algoritmo que mistura enxame de partículas e clonagem imune. Jianliang e Manman [5] criaram um algoritmo modificado de enxame de partículas. Jiang et al. [20] usaram

um algoritmo genético adicionado de recozimento simulado. Jigang et al. [21] propuseram um algoritmo 1D para diminuir a complexidade do problema. Comparativamente com a proposta do artigo, apenas ESBMC encontra a solução exata, entretanto as heurísticas são mais rápidas.

Existem ainda pesquisas híbridas que usam uma heurística para acelerar alguma fase da ferramenta de solução exata. Mas a solução não é necessariamente a solução ótima global. Nesta categoria está a de Arató et al. [16] que emprega uma heurística de biparticionamento gráfico. Em Eimuri e Salehi [22] um algoritmo *branch-and-bound* é modificado com otimização por enxame de partículas. Em Huong e Binh [23] uma otimização usando Pareto é modificada com GA. Em Li et al. [6] um modelo de ILP mista é modificado com uma heurística própria. Comparativamente com a proposta do artigo: chega-se no mesmo caso do grupo anterior.

Quanto à verificação baseada em Teorias do Módulo da Satisfabilidade, Ramalho et al. [24] apresenta um verificador de modelos de contexto limitado para programas escritos em C++, que é a evolução da versão original em ANSI-C. Cordeiro et al. [8] usam um verificador de modelos ESBMC para software embarcado em ANSI-C. Entretanto, nenhum dos trabalhos relacionados trabalhou com particionamento de HW/SW ou para resolver otimização.

VIII. CONCLUSÕES

Foi demonstrada a viabilidade de se usar a técnica de verificação formal de software para resolver problemas de particionamento de HW/SW, sem a necessidade de se adaptar a ferramenta ESBMC, bastando usar os algoritmos descritos nas Figuras 5 e 6. No entanto, nenhuma das técnicas empregadas conseguiu particionar mais do que 400 nodos, pois o tempo de computação ficou muito grande. Abaixo de 400 componentes, a técnica ILP apresentou melhor resultado no geral. Com relação ao ESBMC, este se mostrou uma ótima alternativa para resolver problemas com até 150 nodos. O GA apresentou resultado intermediário, mas com erro elevado.

Se considerarmos as soluções comerciais, como o MATLAB, então ILP e GA são mais simples de serem usadas. Entretanto, são soluções pagas. Assim como outros verificadores de modelo, o ESBMC apresenta uma licença do estilo BSD (*Berkeley Software Distribution*) e pode ser baixado e utilizado sem custo. O ESBMC-2 apresentou desempenho melhor que o ESBMC-1. Portanto, quando modelando um problema de particionamento, é possível manter a versão com o módulo da matriz incidente transposta. Isto implica em menos trabalho para preparar as restrições e na redução no número de variáveis a serem solucionadas.

Quanto a contribuição secundária na avaliação do valor inicial do custo de software S_0, este apresentou grande impacto nos resultados de desempenho. Se for escolhido um valor extremo para S_0, com todos os componentes em hardware ou em software, então o tempo de solução do problema é menor. Um trabalho futuro é a aplicação de ESBMC para resolver um problema de particionamento para arquiteturas mais complexas, com mais de uma CPU e com multiprogramação e multiprocessamento (coprojeto de segunda geração).

REFERÊNCIAS

[1] P. Arató, S. Juhász, Z. Mann, A. Orbán, and D. Papp, "*Hardware/software partitioning in embedded system design*," in Proceedings of the IEEE international symposium of intelligent signal processing, 2003, pp. 192-202.

[2] Z. Mann, A. Orbán, P. Arató, "*Finding optimal hardware/software partitions*," in Form Methods Syst Dcs 31: pp. 241-263. Springer Science+Business Media, 2007.

[3] H. Wang, H. Zhang, "*Improved HW/SW partitioning algorithm on efficient use of hardware resource*," 2nd Int. Conf. on Computer and Automation Engineering (ICCAE), pp. 682-685, 2010.

[4] G. Sapienza, T. Seceleanu, I. Crnknovic, "*Partitioning Decision Process for Embedded Hardware and Software Deployment*," Comp Soft and Applications Conf Workshops, IEEE 37th Annual, pp. 674-680, 2013.

[5] Y. Jianliang, P. Manman, "*Hardware/Software partitioning algorithm based on wavelet mutation binary particle swarm optimization*," IEEE 3rd Int. Conf. on Comm. Software and Network, pp. 347-350, 2011.

[6] S. Li, Y. Liu, X. S. Hu, X. He, Y. Zhang, P. Zhang, H. Yang, "*Optimal partition with block-level parallelization in C-to-RTL synthesis for streaming applications*," 18th Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 225-230, 2013.

[7] C. Baier, J-P Katoen, "*Principles of Model Checking*," The MIT Press, London, 2008.

[8] L. Cordeiro, B. Fischer, J. Marques-Silva, "SMT-based bounded model checking for embedded ANSI-C software," IEEE Transactions on Software Engineering, 38, (4), pp. 957-974, 2012.

[9] J. Teich. "*Hardware/Software Codesign: The Past, the Present, and Predicting the Future*," Proceedings of the IEEE, Vol. 100, May 13th, pp. 1411-1430, 2012.

[10] The MathWorks, Inc. (2013). MATLAB (version R2013a). Natick, MA.

[11] S. Daskalaki, T. Bribas, E. Housos, "*An integer programming formulation for a case study in university timetabling*," in European Journal of Operational Research 153 2004: pp. 117-135.

[12] S. Rao. *Engineering Optimization: Theory and Practice*. 4th edition. Wiley, 2009.

[13] A. Belegundu and T. Chandrupatla. *Optimization Concepts and Applications in Engineering*. 2nd Edition. Cambridge University Press, 2011.

[14] M. Mitchell. *An introduction to genetic algorithm*. 5th Edition. MIT Press: 1999.

[15] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, R. Brown. "*MiBench: a free, commercially representative embedded benchmark suite*," in Proceedings IEEE 4th workshop workl characterization, 1997.

[16] P. Arató, Z Mann, A Orbán, "*Algorithmic aspects of hardware/software partitioning*". ACM Trans Des Aut Electron Syst 10, pp. 136–156, 2005.

[17] A. Bhattacharya, A. Konar, S. Das, C. Grosan, and A. Abraham, "*Hardware software partitioning problem in embedded system design using particle swarm optimization algorithm*," in Intl. Conf. on Complex, Intelligent and Soft Intensive Systems: pp. 171-176, 2008.

[18] G. Wang, W. Gong and R. Kastner, "*Application partitioning on programmable platforms using the ant colony optimization*," Journal of Embedded Computing, vol. 11, no. 19, pp. 119-136, 2006.

[19] L. Luo, H. He, C. Liao, Q. Dou, "*Hardware/Software partitioning for heterogeneous multicore SOC using particle swarm optimization and immune clone (PSO-IC) algorithm*," IEEE International Conference on Information and Automation (ICIA), pp. 490-494, 2010.

[20] Y. Jiang, H. Zhang, X. Jiao; X. Song, W. Hung, M. Gu, J. Sun, "*Uncertain Model and Algorithm for Hardware/Software Partitioning*," VLSI , IEEE Computer Society Annual Symposium (CSAS), pp. 243-248, 2012.

[21] W. Jigang, T. Srikanthan, G. Chen, "*Algorithmic Aspects of Hardware/Software Partitioning: 1D Search Algorithms*," Computers, IEEE Transactions on, pp. 532 - 544 Volume: 59, April 2010.

[22] T. Eimuri, S. Salehi, "*Using DPSO and B&B Algorithms for Hardware/Software Partitioning in Co-design*," 2nd Int. Conf. on Computer Research and Development, pp. 416-420, 2010.

[23] P. Huong, N. Binh, "*An approach to design embedded systems by multi-objective optimizatio*n," Int. Conf. on Advanced Technologies for Communications (ATC), pp. 165-169, 2012.

[24] M. Ramalho, M. Freitas, F. Sousa, H. Marques, L. Cordeiro, and B. Fischer, "*SMT-based bounded model checking of C++ programs*," in Intl. Conf. and Workshops on the Engineering of Computer-Based Systems (ECBS), pp. 147-156, IEEE, 2013.