Applying Multi-Core Model Checking to Hardware/Software Partitioning in Embedded Systems

Alessandro Trindade, Hussama Ismail, and Lucas Cordeiro

Federal University of Amazonas, Manaus, Amazonas, Brazil
{alessandro.b.trindade, hussamaismail}@gmail.com
lucascordeiro@ufam.edu.br

Abstract. Hardware/Software partitioning is one of the critical steps in hardware/software co-design flow, and has very important influence on the system performance. It is when the problem of allocating properly which functions of the system should be implemented in hardware (HW) or in software (SW) emerges. In the last ten years, a significant research effort has been carried out in the HW/SW partitioning area. In this paper, we present an alternative approach to solve the HW/SW partitioning problem, which uses SMT-based verification technique in conjunction with a multi-core support using OpenMP. The multi-core approach allows initializing many instances (i.e., based on processor cores numbers) of the model checker, with each one using a different optimum value until the problem is satisfied. The goal is to show that multi-core model checking techniques can be effective, in particular cases, to find the optimal solution of the HW/SW partitioning problem using a context-bounded model checking, based on Satisfiability Modulo Theories (SMT) solvers. We compare the experimental results of our multi-core approach using the Integer Linear Programming (ILP) and the Genetic Algorithm (GA).

Keywords: hardware/software co-design; hardware/software partitioning; optimization; model checking; multi-core; OpenMP.

1 Introduction

The demand for shorter time-to-market and the fact that the product must be functionally correct, cheap, fast, and reliable represent a challenge to designers of embedded systems. Size, heat dissipation, and energy consumption play an important role in system design. Computer systems in this context mean to have components implemented with parts in both hardware and software. Hardware means an application-specific hardware unit. Software means a program running on a general-purpose hardware unit (e.g., a microprocessor).

Hardware components are usually much faster than software, but they are also significantly more expensive. Software components, on the other hand, are cheaper to develop and to maintain, but they are slower. In hardware/software co-design (HSCD), several issues are addressed, but the most critical design step in HSCD is the partitioning. The challenge is to choose which component must be placed in hardware and in software, in order to meet restrictions of performance and price.

In this paper, an example of partitioning will be presented, modeled, and solved. A multi-core Satisfiability Modulo Theories (SMT)-based verification method will be applied to the hardware and software partitioning and we then compared to the results with integer linear programming (ILP) and genetic algorithm (GA).

Most of the related work compares the partitioning results between different optimization tools, such as ILP and some heuristic, as seen, for example, in [1] and [2]. The aim is always to compare the performance among the different solutions.

In any hardware and software design of complex systems, more time is spent on verification than on construction, as shown in [3]. Formal methods based on model checking offer great potential to obtain a more effective and faster verification in the design process. Programs, and more generally computer systems, may be viewed as mathematical objects with behavior that is, in principle, well determined. This makes it possible to specify programs using mathematical logic, which constitutes the intended (correct) behavior. Then, one can try to give a formal proof or otherwise establish that the program meets its specification. This area of research is referred as formal methods as defined in [4]. The aim is to establish system correctness with mathematical rigor.

In the recent decades, research in formal methods has led to the development of very promising verification techniques that facilitate the early detection of bugs in order to ensure the correctness of the system. Model-based verification techniques are based on models that describe the possible system behavior in a mathematically precise and unambiguous manner. Thus, such problems as incompleteness, ambiguities, and inconsistencies, which normally are discovered only in later stages of the design, can be detected in advance. The system models are accompanied by algorithms that systematically explore all the states of the system model in a brute-force manner.

Bringing this idea to hardware/software partitioning, a model checker could analyze a code formally specified in a given language, where each component of a system, specified in clear objectives (such as minimize hardware cost) and requirements (such as software cost), could be explored in all its states until an optimal solution is found (i.e., the act of obtaining the best result among different possible solutions). In [5] was shown that it is possible to use SMT-based verification to perform hardware/software partitioning in embedded systems. The present work extends [5] since there is a substantial improvement in terms of the GA and ESBMC algorithms performance. Additionally, multi-core processors have been used in all segments of industry to implement high-performance computing. It is essentially motivated by limitations of the CMOS technology, which limits the increase of the chip's frequency after it reaches 4 GHz [42]. Hardware platforms, together with multi-processing platforms, have allowed operating systems to distribute tasks executions across multiple processors, which thus generates an increase in performance (i.e., in our case reduced verification time) if compared to a single-core solution. To the best of our knowledge, this is the first work to use a multi-core SMT-based verification to solve a hardware/software partitioning problem in embedded systems.

We implement our ideas with the Efficient SMT-based Bounded Model Checker (ESBMC) tool that builds on the front end of the C Bounded Model Checker (CBMC). As its main contribution, this paper proves that it is possible to take advantage of some characteristics of a formal verification tool in a multi-core environment to solve optimization problems, specifically hardware/software partitioning.

2 Background

2.1 Optimization

Optimization is the act of obtaining the best result (i.e., the optimal solution) under given circumstances [6]. In the design, construction, and maintenance of any engineering system, engineers have to make many technological and managerial decisions at several stages. The ultimate goal of all such decisions is either to minimize the effort required or to maximize the desired benefit. Because the effort required or the benefit desired in any practical situation can be expressed as a function of certain decision variables, optimization can be defined as the process of finding the conditions that give the maximum or minimum value of a function [6].

There is no single method available for solving all optimization problems efficiently [6]. Hence, a number of optimization methods have been developed for solving different types of optimization problems. The most known technique is the linear programming, which is an optimization method applicable for the solution of problems in which the objective function and the constraints appear as linear functions of the decision variables. The constraint equations in a linear programming problem may be in the form of equalities or inequalities.

A particular case of linear programming is ILP, in which the variables can assume just integer values. Another particular case of linear programming is when all the design variables of an optimization problem are allowed to take on values of either zero or one. This kind of problem is known as a zero–one programming problem or binary integer programming [6]. In some cases, the time needed to find a solution using ILP is infeasible. Even with the use of powerful computers, a problem can take hours running before an optimal solution is reached. If the optimization problem is very complex, some heuristics can be used to solve the problem [6]. The only drawback is that the found solution may not be the global minimum or maximum.

2.2 Bounded Model Checking with ESBMC

Model checking refers to algorithms for exploring the state space of a transition system to determine if it obeys a specification of its intended behavior [7]. These algorithms can perform exhaustive exploration in a highly automatic way and, thus, have attracted much interest in industry. However, model checking has been held back by the state explosion problem, in which the number of states in a system grows exponentially in the number of system components [7]. Much research has been devoted to mitigate this problem.

Among the recent techniques, there is one that combines model checking with satisfiability solving. This technique, known as bounded model checking (BMC), does a very fast exploration of the state space, and for some types of problems it offers large performance improvements over previous approaches, as shown in [7].

BMC based on Boolean Satisfiability (SAT) has been introduced as a complementary technique to binary decision diagrams for alleviating the state explosion problem, as described by [7]. The basic idea of BMC is to check the negation of a given property at a given depth: given a transition system M, a property ϕ, and a bound k, BMC unrolls the system k times and translates it into a verification condition (VC) φ such that φ is satisfiable if and only if ϕ has a counterexample of depth k or less [7], [8]. Standard SAT checkers can be used to check whether φ is satisfiable. In BMC of software, the bound k limits the number of loop iterations and recursive calls in the program [7].

One well-known tool that implements BMC for ANSI-C/C++ programs using SAT solvers is CBMC, as defined in [8]. It can process C/C++ code using the goto-cc tool, which compiles the C/C++ code into equivalent GOTO-programs (i.e., control-flow graphs) using a gcc-compliant style. The GOTO-programs can then be processed by the symbolic execution engine. CBMC uses two recursive functions that compute the constraints and properties, as defined in [7], and [8].

To cope with increasing software complexity, SMT solvers can be used as back-ends for solving the generated VCs, as shown in [8], [9], and [10].

According to [11], SMT-based model checking can be used to verify the correctness of embedded ANSI-C software. Reference [12] shows that ESBMC can be used to verify multi-threaded software. According to [13], ESBMC can also be used to model check C++ software based on SMT solvers. In [5] it was proven that it is possible to use a BMC tools (and in particular, ESBMC) as an optimization tool.

There are two directives in C/C++ code that will be used to guide a model checker to solve an optimization problem: ASSUME and ASSERT. The directive ASSUME will be responsible for ensuring the compliance of constraints (software costs), and ASSERT will control the halt condition or code violation (minimum hardware cost). Then, with some C/C++ code it is possible to guide ESBMC to solve optimization problems. In addition, with OpenMP, a multi-core optimization tool based in formal verification can be implemented.

2.3 Multi-core ESBMC with OpenMP

Nowadays, although the hardware used to perform tests usually have a modern multi-core architecture, with the ability to run several threads on different processing cores, ESBMC verification runs are still performed only in a single-core (i.e., if the processor has 8 processing cores, only one is used for the verification and the others remain idle). Therefore, there is a significant unused hardware resource during this process.

Fig.1 shows the current ESBMC architecture, which consists of the C parser, GOTO Program, GOTO Symex, and SMT solver. The main factor for ESBMC to use only a single-core relies on its back-end (i.e., SMT Solver). Currently, the SMT solvers supported by ESBMC are: Z3 [37], Boolector [38], MathSAT [39], CVC4 [40], and Yices

[41]. Most of them do not have multi-threaded support or a parallel version to solve the generated SMT equations.

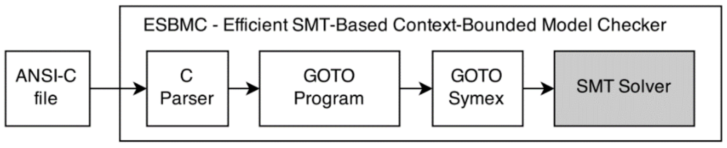

Fig. 1. ESBMC architecture

To improve the hardware resources usage and to decrease the verification time, without modifying the SMT Solver, we adopt in this present work the OpenMP (Open Multi-Processing) library [36] as a front-end for ESBMC. OpenMP is a standard Application Programming Interface (API) for shared memory programming, which has been very successful for structural parallelism in applications. The API provides a directive-based programming approach to write parallel versions of C/C++ programs [18]. In OpenMP, the implementation is based on the fork-join model. The main thread executes the sequential parts of the program and when a parallel region is encountered, it forks a team of worker threads. After the parallel region finishes (i.e., the API waits until all threads terminate), then the main procedure get back to the single-threaded execution mode [43]. Fig. 2 shows our approach called "Multi-core ESBMC".

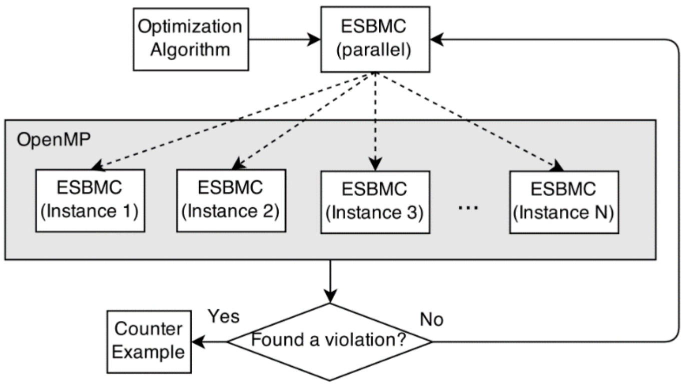

Fig. 2. Multi-core ESBMC Approach

Multi-core ESBMC obtains the problem specifications, the number of cores available in the processor, and starts a parallel region with *N* different instances of ESBMC, based on the number of available cores. All these ESBMC instances, as shown in Fig. 2, run in different threads managed by the OpenMP API, i.e., it manages the thread lifecycle: start, running, and dead states. After *N* instances execution, if there is no code violation, then multi-core ESBMC starts *N* new instances again. If during the parallel region execution, a violation is found in any running thread, it presents the counterexample with the violation condition and the verification time. After all threads of the batch processing are dead, multi-core ESBMC finishes its execution.

3 Mathematical modeling

3.1 Informal Model

Based on [1] and [2] the informal model used in this paper can be described by five characteristics. First, there is only one software context, i.e., there is just one general-purpose processor, and there is only one hardware context. In other words, the components of the system must be mapped to either one of these two contexts. Second, the software implementation of a component is associated with a software cost, which is the running time of the component if implemented in software. Third, the hardware implementation of a component has a hardware cost, which can be area, heat dissipation, and energy consumption. Fourth, based on the premise that hardware is significantly faster than software, the running time of the components in hardware is considered as zero. Finally, if two components are mapped to the same context, then there is no overhead of communication between them; otherwise, there is an overhead.

The consequence of these assumptions is that scheduling does not need to be addressed in this work, as mentioned in [1], [2]. Hardware components do not need scheduling, because the running time is assumed to be zero. Because there is only one processor, software components do not need to be scheduled as well. Therefore, the focus here is only on the partitioning problem. That configuration, according to [14], describes a first-generation kind of co-design where the focus is on bipartitioning.

3.2 Formal Model

According to [1] and [2] the inputs of the problem are: A directed simple graph $G = (V, E)$, called the task graph of the system, is necessary. The vertices $V = \{v_1, v_2, ..., v_n\}$ represent the nodes that are the components of the system that will be partitioned. The edges (E) represent communication between the components. Additionally, each node v_i has a cost $h(v_i)$ of hardware (if implemented in hardware) and a cost $s(v_i)$ of software (if implemented in software). Finally, $c(v_i, v_j)$ represents the communication cost between v_i and v_j if they are implemented in different contexts (hardware or software).

Based on [1], P is called a hardware/software partition if it is a bipartition of V: $P = (V_H, V_S)$, where $V_H \cup V_S = V$ and $V_H \cap V_S = \emptyset$. The crossing edges are $EP=\{(v_i, v_j): v_i \in V_S, v_j \in V_H \text{ or } v_i \in V_H, v_j \in V_S\}$. The hardware cost of P is given by Equation (1), and the software cost of P (i.e., software cost of the nodes and the communication cost) is given by Equation (2):

$$H_P = \sum_{v_i \in V_H} h_i \tag{1}$$

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

Even based on [1], three different optimization and decision problems can be defined. In this paper, the focus is on the case that S_0 is being given. Find a P HW-SW partitioning so that $S_P \leq S_0$ and H is minimal (system with hard real-time constraints).

Concerning the complexity of this problem, reference [1] demonstrates that it is NP-Hard.

4 Applying the Formal Model to an Example

The idea here is to explain the process of performing a hardware/software partitioning of an embedded system with 10 nodes and 13 edges, as represented in Fig. 3. The same idea is valid for a greater number of nodes.

Based in section 3.2, the hardware cost of each node was chose as h = [5 10 3 4 8 5 10 3 4 8], the software cost was s = [10 15 7 4 9 10 15 7 4 9], and finally, the communication cost to each edge was c = [5 5 5 5 5 5 5 5 5 5 5 5 5].

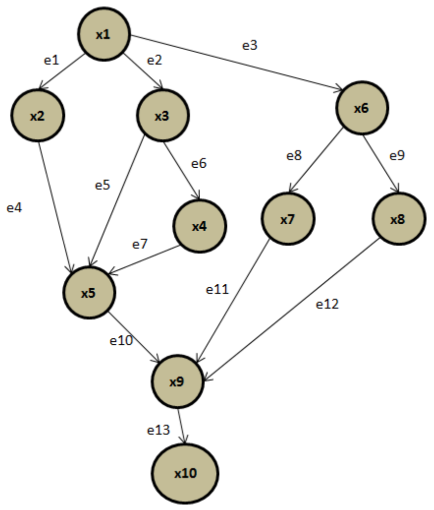

Fig. 3. Directed graph of the system to be partitioned (Source: [5])

To create the algorithm, it is necessary to use a matrix called E ∈ {-1, 0, 1}^{13x10}, which is the transposed incidence matrix of G, that is,

$$E[i,j] := \begin{cases} -1 & if\ edge\ i\ starts\ in\ node\ j \\ 1 & if\ edge\ i\ ends\ in\ node\ j \\ 0 & if\ edge\ i\ is\ not\ incident\ to\ node\ j \end{cases}$$

Let x ∈ {0, 1}^{10} be a binary vector indicating the partition, i.e.,

$$x[i] := \begin{cases} 1\ if\ node\ i\ is\ realized\ in\ hardware \\ 0\ if\ node\ i\ is\ realized\ in\ software \end{cases}$$

It can be seen that the components of the vector |Ex| indicate which edges cross the boundary between the two contexts.

The optimization can be formulated as defined by [1], and [2]:

$$Minimize\ h_x \tag{3}$$

Subject to the restrictions

$$s(1 - x) + c|Ex| \leq S_0 \tag{4}$$

$$x \in \{0,1\}^{10} \tag{5}$$

The above formulation can be transformed into an ILP equivalent by introducing the slack variables y ∈ $\{0, 1\}^{13}$ to eliminate the modulus as shown in [1], and [2].

$$Minimize\ h_x \tag{6}$$

Subject to the restrictions

$$s(1 - x) + cy \leq S_0 \tag{7}$$

$$Ex \leq y \tag{8}$$

$$-Ex \leq y \tag{9}$$

$$x \in \{0,1\}^{10} \tag{10}$$

Equations (3, 4, and 5) and Equations (6, 7, 8, 9, and 10) are equivalent. However, when using commercial off-the-shelf tools to solve the optimization problem, the form presented by Equations (3, 4, and 5) are not suitable. Those commercial tools use the following representation, with only one restriction to inequalities:

$$min\ f^T x\ such\ that\ Ax \leq b \tag{11}$$

Thus, the basic idea is to transform Equations (7), (8), and (9) into only one matrix. This is carried out by decomposing those three equations in terms of decision variables *x* and *y*, then putting back to matrix form by isolating any variable to the left side of the equation. The following example shows the process to transform the algebraic formulation of Equation (7) into a matrix based on the 10 nodes of the running example:

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

In algebraic notation:

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

And back in matrix representation, with all the variables at the left side:

$$[-10\ -15\ -7\ -4\ -9\ -10\ -15\ -7\ -4\ -9\ 5\ 5\ 5\ 5\ 5\ 5\ 5\ 5\ 5\ 5\ 5\ 5\ 5]\begin{bmatrix} x_1 \\ \vdots \\ x_{10} \\ y_1 \\ \vdots \\ y_{13} \end{bmatrix} \leq S_0 - 90 \quad (14)$$

The same procedure applied to Equations (8) and (9), then putting on matrix form (all together), it results is Equation (15), that is independent of the number of variables. The indexes show the dimension of each matrix (where V represents the total number of vertices and E represents the total number of edges).

$$\begin{bmatrix} -S_{1xV} & C_{1xE} \\ E_{ExV} & -I_{ExE} \\ -E_{ExV} & -I_{ExE} \end{bmatrix}_{(1+2*E)x(V+E)} \begin{bmatrix} x_{Vx1} \\ y_{Ex1} \end{bmatrix}_{(V+E)x1} \leq \begin{bmatrix} S_0 - 9 * V_{1x1} \\ 0_{Ex1} \\ 0_{Ex1} \end{bmatrix}_{(1+2*E)x1} \quad (15)$$

Equation (15) will be used by ILP and GA in this paper. However, according to [5], because ESBMC can handle Equations (4) and (15) in the same manner, this paper will perform tests with ESBMC using just Equation (4) (i.e., without slack variables).

5 ILP-based Algorithm: Binary Integer Programming

Equations (3) and (15), and the Matlab [15] function bintprog, which solves binary integer programming problems, were used. This function uses a linear programming (LP)-based branch-and-bound algorithm that searches for an optimal solution by solving a series of LP-relaxation problems, in which the binary integer requirement on the variables is replaced by the weaker constraint $0 \leq x \leq 1$. ILP Matlab Pseudocode shows the high level code created to solve the partitioning problem.

```
ILP Matlab Pseudocode
  Declare number of nodes and edges
  Declare hardware cost of each node as an array (h)
  Declare software cost of each node as an array (s)
  Declare communication cost of each edge as an array (c)
  Declare the initial software cost (S0)
  Declare the transposed incidence matrix of graph G (E)
  Create Identity Matrix I (num. edges x num. edges)
  Construct the matrix A
  Construct the matrix b
  Start chronometer
  Solve ILP with bintprog (h, A and b as parameters)
  Stop chronometer
  Present the results
```

6 Genetic Algorithm

In recent years, some optimization methods that are conceptually different from the traditional mathematical programming techniques have been developed. These methods are labeled as modern or nontraditional methods of optimization, according to [6], and they are normally faster than ILP when dealing with complex problems, which involve a large number of variables. GAs emerge from these methods and, philosophically speaking, are based on Darwin's theory of survival of the fittest. Some authors, such as [19], label GA not as an optimization tool, but as a heuristic search tool, because the algorithm is not guaranteed to reach the problem's best global solution.

According to [6] and [20], GAs are based on the principles of natural genetics and natural selection. The basic elements of natural genetics – reproduction, crossover, and mutation – are used in the genetic search procedure, as shown in [21].

The following GA Matlab pseudocode was created to solve the partitioning problem.

```
GA Matlab Pseudocode
  Declare number of nodes and edges
  Declare hardware cost of each node as an array (h)
  Declare software cost of each node as an array (s)
  Declare communication cost of each edge as an array (c)
  Declare the initial software cost (S0)
  Declare the transposed incidence matrix of graph G (E)
  Create Identity Matrix I (num. edges x num. edges)
  Construct the matrix A
  Construct the matrix b
  Define lower bound as zero to all variables (lb)
  Define upper bound as one to all variables (ub)
  Define fitness function as the hardware cost
  Declare all solution variables as integer (intcon)
  Change the default size of population
  Start chronometer
  Call ga function (parameters h, A, b, lb, ub, intcon)
  Stop chronometer
  Present the results
```

7 Analysis of the partitioning problem using ESBMC

ESBMC pseudocode shows the algorithm with the same restrictions and conditions placed on ILP and GA. Two values must be controlled to obtain the results and to perform the optimization. One is the initial software cost, as defined in Section 3.2. The other is the halting condition (code violation) that breaks the algorithm.

As the previous pseudocodes, the ESBMC algorithm starts with the declarations of hardware, software, and communication costs. S_0 also must be defined, as the transposed incidence matrix and the identity matrix, as done for the MATLAB. Here, the matrices A and b are generated. At that point, the ESBMC algorithm starts to differ from the others presented here.

As a model checker to verify correctness, it is possible to tell the ESBMC with which type of values the variables must be tested. Therefore, there is a declaration to populate all the decision variables with non-deterministic Boolean values. Those values that change for each test will generate a possible solution and to obey the restrictions. If this is achieved, then a feasible solution is found and the ASSUME directive is responsible for ensuring the compliance of constrains (i.e., $Ax \leq b$).

A loop controls the cost of hardware hint, starting with zero and reaching the maximum value considering the case where all nodes are partitioned to hardware. To every test performed, the hardware hint is compared to the feasible solution. This is accomplished by an ASSERT statement at the end of the algorithm, a predicate that controls the halt condition (true-false statement). If the predicate is FALSE, then the optimization is finished, i.e., the solution was found. The ASSERT statement tests the objective function, i.e., the hardware cost, and will stop if the hardware cost found is lower than or equal to the optimal solution. However, if ASSERT returns a TRUE condition, i.e., the hardware cost is higher than the optimal solution, then the model checking restarts and a new possible solution is generated and tested until the ASSERT generates a FALSE condition. When the FALSE condition happens at run-time, the execution code is aborted and ESBMC presents the counterexample that caused the condition to be broken. That is the point where the solution is presented (minimum hardware cost).

In the ESBMC algorithm, which is shown below, it is not necessary to add slack variables because the modulus operation is kept in Equation (4), which reduces the number of variables to be solved.

```
ESBMC Pseudocode
   Initialize variables
   Declare number of nodes and edges
   Declare hardware cost of each node as an array (h)
   Declare software cost of each node as an array (s)
   Declare communication cost of each edge (c)
   Declare the initial software cost ($S_0$)
   Declare transposed incidence matrix of graph G (E)
   Define the solutions variables ($x_i$) as Boolean
   main{
     For TipH=0 to Hmax do {
          Populate $x_i$ with nondeterministic/test values
          Calculate s(1-x)+c*|E*x| and store at variable
          Requirement insured by ASSUME (variable $\leq S_0$)
          Calculate Hp cost based in value tested of $x_i$
          Violation check with ASSERT (Hp > TipH)
     }
}
```

In the multi-core ESBMC algorithm, the only difference is the fact that the value of TipH and its range is not declared in the algorithm, as show above. The proposed approach is invoked for each test problem, as follows:

```
esbmc-parallel <filename> <hmin_value> <HMAX>
```

Where <*filename.c*> is the optimization problem described in ANSI-C format, <*hmin_value*> is the minimum (zero to HW/SW partitioning problem) and <*HMAX*> is the maximum hardware cost for the specified problem.

Therefore, the algorithm starts *N* different instances of ESBMC using the different optimization values for HMAX in order to find a violation. If all instances finish and no violation is found, multi-core ESBMC starts new *N* instances. When a violation is found, it reports time and hardware cost. If multi-core ESBMC tests all the possibilities for the hardware cost and has not found a violation, it shows: “Violation not found”.

8 Experimental Evaluation

ESBMC 1.24 running on a 64-bit Ubuntu 14.04.1 LTS operating system was used. Version 2.0.1 of Boolector SMT-solver [38], freely available from Institute for Formal Models and Verification (FMV) was used as well. For the ILP and GA formulations, MATLAB R2013a from MathWorks was used [15]. MATLAB is a dynamically typed high-level language known as the state-of-the-art mathematical software [16] and is widely used by the electrical and computer engineering community [17]. However, it does not support parallel execution (ILP and GA algorithms). The ESBMC algorithm was implemented in C++ language[1]. A desktop with 24GB of RAM and i7 (8-cores) from Intel with clock of 3.40 GHz was used. Each time was measured 3 times in GA (average taken) and just once in ESBMC and ILP. The reason is that GA times are not so close as ESBMC and ILP. A time out condition (TO) is reached when the running time is longer than 7200 seconds. A memory out (MO) happens when no additional memory can be allocated for the algorithm. Table 1 lists the benchmarks[1].

Table 1. Description of Benchmarks

Name	Nodes	Edges	Description
CRC32	25	32	32-bit cyclic redundancy check [22]
Patricia Insert	21	48	Routine to insert values into Patricia tries [22]
Dijkstra	26	69	Computer shortest paths in a graph [22]
Clustering	150	331	Image segmentation algorithm in a medical application
RC6	329	448	RC6 cryptography graph
Fuzzy	261	422	Clustering algorithm based on fuzzy logic
Mars	417	600	MARS cipher from IBM

[1] Available at: http://www.esbmc.org/benchmarks/

The vertices in the graphs correspond to high-level language instructions. Software and communication costs are time dimensional, and hardware costs represent the occupied area.

The overall performance (Table 2) shows that ILP is the best solution of all techniques, even if we consider that the Fuzzy benchmark reached time out with ILP. Thus, the maximum limit to use ILP is around 329 nodes or less.

GA was the only technique that could solve all benchmarks, but the error from the exact solution varied from -37.6% to 29%, even with adjusts in parameters as population size, number of generation and the number of individuals with best fitness values in the current generation that are guaranteed to survive to the next generation.

Multi-core ESBMC had a better performance than that of pure ESBMC. The relative speedup obtained ranged from 1.9 to 60.3, which shows a reasonable improvement. Until the number of 150 nodes is reached, the ESBMC technique, mainly Multi-core ESBMC, has shown itself to be a good choice to solve hardware/software partitioning. This is because the exact solution was found and the execution time was mostly closer to ILP (from the same performance to 4.7 times faster). If the complexity of test vectors increases, then pure ESBMC algorithm has the drawback of creating an even more complex problem to be solved, because it increases the states created with the LOOP that controls the hardware cost hint.

Table 2. Results of the benchmarks

		CRC32	**Patricia**	**Dijkstra**	**Clustering**	**RC6**	**Fuzzy**	**Mars**
	Nodes	25	21	26	150	329	261	417
	Edges	32	48	69	331	448	422	600
	S_0	20	10	20	50	600	4578	300
Exact Solution	Hp	15	47	31	241	692	13820	876
	Sp	19	4	19	46	533	4231	297
ILP	Time(s)	2	1	2	649	1806	TO	5429
	Hp	15	47	31	241	692	-	876
GA	Time(s)	7	7	9	340	2050	1372	5000
	Error	13,3%	0,0%	29,0%	1,7%	-6,5%	-37,6%	-27,5%
ESBMC	Time(s)	31	362	292	3010	TO	MO	MO
	Hp	15	47	31	241	-	-	-
Multi-core ESBMC	Time(s)	2	6	7	1615	TO	TO	TO
	Hp	15	47	31	241	-	-	-
ESBMC Relative Speedup		**15.4**	**60.3**	**41.7**	**1.9**	-	-	-

Time outs are represented with TO and examples that exceed available memory are represented with MO

With the RC6 benchmark (329 nodes), ESBMC was unable to present a solution without exceeding the time limit of 7200 seconds. Pure ESBMC had even a worse performance with Fuzzy and Mars, because the execution presented memory out. This is a clear indication that the prune method adopted by the ILP's search tree solver is more efficient than that adopted by ESBMC solver.

9 Related work

The most relevant initiatives related to the hardware/software partitioning problem are presented by [1], [2], and [23], where the hardware/software partitioning problem is properly formalized and some algorithms are employed to solve it. Since the second half of the first decade of the 2000s, three main paths have been tracked to improve or to present alternative solutions to the optimization of hardware/software partitioning, i.e., to find the exact solution, to use heuristics to speed up performance time, and hybrid ones.

In the first group, the exact solution to the hardware/software partitioning problem is found. Here can be cited the relevant work of [2], who modified a branch-and-bound algorithm to speed up the execution time. Reference [24] presents a greedy algorithm that reaches the global optimal solution as well. In [25], the authors use multiple criteria decision analysis. The use of SMT-based verification presented in this paper can be grouped into this category, because the exact solution is found with the algorithm. The difference is based only in terms of the technique chosen to solve the problem.

Another path followed in past initiatives and which has had more researches is the creation of heuristics to speed up the running time of the solution. These heuristics are mainly based on modern methods of optimization. In this category can be cited the relevant work of [1] that uses a genetic algorithm to solve the optimization problem. In [26] a solution is reached through a particle swarm optimization algorithm. Reference [27] uses an algorithm that employs ant colony optimization and [28] uses a particle swarm optimization and an immune clone algorithm. Reference [29] creates a modified particle swarm algorithm and in [30] a genetic algorithm mixed with simulated annealing. Reference [31] proposes a 1D search algorithm to decrease the complexity of the problem. The difference between this kind of solutions and SMT-based verification is based on two facts: ESBMC is guaranteed to find the exact solution, but the heuristics are faster when the complexity is greater.

Finally, there are approaches that mixes heuristics with exact solution tools is created. The idea is to use a heuristic to speed up some phase of an exact solution tool. It worth mentioning that the final solution is not necessarily an optimal global solution. In counterpart, the running time is lower than the pure exact solution. In this category can be cited the relevant work of [23]. In [32] a branch-and-bound algorithm is modified with particle swarm optimization. In [33], a Pareto optimization is modified using genetic algorithm. In [34], a multi-integer linear programming model is modified using a new kind of heuristic. Comparing this category with the SMT-based verification methodology the same difference is observed: only SMT-based verification is guaranteed to find the exact solution, but hybrid algorithms are faster when complexity rises.

In terms of SMT-based verification, most work is restricted to present the model, its modification to language C/C++, and the application to multi-thread algorithms or to embedded systems to check program correctness. Reference [13] presents a bounded model checker for C++ programs, which is an evolution of dealing with C programs and [11] uses the ESBMC model checker for embedded ANSI-C software. In [5] it was proven that it is possible to use ESBMC to solve hardware/software partitioning, but in a single core way.

There are related studies focused on decreasing the verification time of model checkers by applying techniques like the Swarm Verification [44], and modifications of internal search engines to support parallelism [45], but there is still the need for initiatives related to parallel SMT solvers [46]. Recently, the SMT solver Z3 has been extended to pose and solve optimization problems modulo theories [47].

10 Conclusions

Concerning the comparative tests, with the four techniques presented in this paper to solve hardware/software partitioning, it was evident that none of them is indicated to partition problems with more than 400 nodes. The computing time to solve the optimization problem reached some hours of execution on a standard desktop computer. If we consider less than 400 nodes, then it is possible to use ILP as the best solution provider. If the problem to be solved has 150 nodes or less, then ESBMC represents a feasible alternative. GA had an intermediate result in terms of performance, but the error presented from exact solution made it not acceptable to that kind of application. This error may be reduced by changing some parameters.

If considering off-the-shelf solutions, as MATLAB to ILP and GA, the coding is simpler. However, ESBMC has a BSD-style license and can be downloaded and used for free. So that is a pro for ESBMC.

Considering the two versions of ESBMC, it is possible to conclude that Multi-core ESBMC had better performance results than pure ESBMC. Thus, considering that nowadays the processors have more and more cores, when modeling the problem, it is possible to consider multi-core ESBMC as an alternative to solve the hardware/software partitioning problem. Future work can be done to decrease the processing time of ESBMC (solver included).

A final question is related to how realistic a 150 nodes is, since ESBMC can solve a problem of 150 nodes and 331 edges in 26 minutes. The answer has to be based on the modeling granularity of the problem to be solved. Some researchers propose fine-grained models, in which each instruction can be mapped to either hardware or software. This may lead to tens of thousands of nodes, maybe even more. Others are advocating coarse-grained models, where decisions are made for bigger components, thus even complex systems may consist of just some dozens of nodes to partition. Therefore, it really depends on the problem model. In principle, a fine-grained approach may allow to obtain better partitions, but at the cost of an exponential increase of the size of the search space.

Future work, we will address specifically more complex types of architectures, including more than one CPU, and the assumption of just singled-threaded program execution will be extended to multiprogramming and multiprocessing.

Acknowledgment

The authors thank Dr. Z.A. Mann for the contribution with benchmarks and Dr. Patricia Nascimento Pena from UFMG for the paper reviewing.

References

1. Arató, P., Juhász, S., Mann, Z.A., Orbán, A., Papp, D.: Hardware/software partitioning in embedded system design. In: Proceedings of the IEEE International Symposium of Intelligent Signal Processing, pp. 192-202 (2003)
2. Mann, Z.A., Orbán, A., Arató, P.: Finding optimal hardware/software partitions. In: Formal Methods in System Design, vol. 31, pp. 241-263. Springer Science+Business Media (2007)
3. Baier, C., Katoen, J-P.: Principles of Model Checking. The MIT Press, London (2008)
4. Clarke, E., Emerson, E., Sifakis, J.: Model checking: algorithmic verification and debugging. In: Communications of the ACM, vol. 52, i. 11, pp. 74-84 (2009)
5. Trindade, A., Cordeiro. L.: Applying SMT-based verification to hardware/software partitioning in embedded systems. In: Design Automation for Embedded Systems. Springer Science+Business Media, New York (2015)
6. Rao, S.: Engineering Optimization: Theory and Practice. 4th edition. John Wiley & Sons, Hoboken (2009)
7. Biere, A.: Bounded model checking. In: Biere, A., Heule, M., van Marren, H., Walsh, T. (org.) Handbook of Satisfiability, IOS Press, pp. 457–481, Amsterdam (2009)
8. Clarke, E., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS 2988, pp. 168-176 (2004)
9. Armando, A., Mantovani, J., Platania, L.: Bounded model checking of software using SMT solvers instead of SAT solvers. In: International Journal on Software Tools for Technology Transfer, Vol. 11, n. 1, pp. 69-83 (2009)
10. Ganai, M., Gupta, A.: Accelerating high-level bounded model checking. In: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, pp. 794–801 (2006)
11. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. In: IEEE Transactions on Software Engineering, vol. 38, ed. 4, pp. 957-974 (2012)
12. Cordeiro, L., Fischer, B.: Verifying multi-threaded software using SMT-based context-bounded model checking. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE), pp. 331-340 (2011)
13. Ramalho, M., Freitas, M., Souza, F., Marques, H., Cordeiro, L.: SMT-Based Bounded Model Checking of C++ Programs. In: International Conference and Workshops on the Engineering of Computer-Based Systems, pp. 147-156, IEEE (2013)
14. Teich, J.: Hardware/Software Codesign: The Past, the Present, and Predicting the Future. In: Proceedings of the IEEE, vol. 100, pp. 1411-1430 (2012)
15. The MathWorks, Inc: MATLAB (version R2013a). Natick, MA (2013)

16. Tranquillo, J.: Matlab for Engineering and the Life Sciences. Synthesis Lectures on Engineering. Morgan & Claypool (2011)
17. Hong, L., Cai, J.: The application guide of mixed programming between MATLAB and other programming languages. In: Proceedings of the 2nd International Conference on Computer and Automation Engineering (ICCAE), pp. 185-189 (2010)
18. Daskalaki, S., Bribas, T., Housos, E.: An integer programming formulation for a case study in university timetabling. In: European Journal of Operational Research, vol. 153, pp. 117-135 (2004)
19. Belegundu, A., Chandrupatla, T.: Optimization Concepts and Applications in Engineering. 2nd Edition. Cambridge University Press, New York (2011)
20. Haupt, R., Haupt, S.: Practical genetic algorithms. 2nd edition. John Wiley & Sons, New York (2004)
21. Mitchell, M.: An introduction to genetic algorithm. 5th Edition. MIT Press, Cambridge (1999)
22. Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T., Brown, R.: MiBench: a free, commercially representative embedded benchmark suite. In: Proceedings of the IEEE 4th annual workshop on workload characterization, pp. 3-14 (2001)
23. Arató, P., Mann, Z.A., Orbán, A.: Algorithmic aspects of hardware/software partitioning. In: ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 10, pp. 136–156 (2005)
24. Wang, H., Zhang, H.: Improved HW/SW partitioning algorithm on efficient use of hardware resource. In: 2nd International Conference on Computer and Automation Engineering (ICCAE), pp. 682-685 (2010)
25. Sapienza, G., Seceleanu, T., Crnknovic, I.: Partitioning Decision Process for Embedded Hardware and Software Deployment. In: IEEE 37th Annual Conference and Workshops on Computer Software and Applications (COMPSACW), pp. 674-680 (2013)
26. Bhattacharya, A., Konar, A., Das, S., Grosan, C., Abraham, A.: Hardware software partitioning problem in embedded system design using particle swarm optimization algorithm. In: Proceedings of the International Conference on Complex, Intelligent and Software Intensive Systems, pp. 171-176 (2008)
27. Wang, G., Gong, W., Kastner, R.: Application partitioning on programmable platforms using the ant colony optimization. In: Journal of Embedded Computing – Embedded Processors and Systems: Architectural Issues and Solutions for Emerging Applications, vol. 2, pp. 119-136 (2006)
28. Luo, L., He, H., Liao, C., Dou, Q.: Hardware/Software partitioning for heterogeneous multicore SOC using particle swarm optimization and immune clone (PSO-IC) algorithm. In: Proceedings of the IEEE International Conference on Information and Automation (ICIA), pp. 490-494 (2010)
29. Jianliang, Y., Manman, P.: Hardware/Software partitioning algorithm based on wavelet mutation binary particle swarm optimization. In: Proceedings of the IEEE 3rd International Conference on Communication Software and Network, pp. 347-350 (2011)
30. Jiang, Y., Zhang, H., Jiao, X., Song, X., Hung, W., Gu, M., Sun, J.: Uncertain Model and Algorithm for Hardware/Software Partitioning. In: Proceedings of the IEEE Computer Society Annual Symposium on VLSI, pp. 243-248 (2012)
31. Jigang, W., Srikanthan, T., Chen, G.: Algorithmic Aspects of Hardware/Software Partitioning: 1D Search Algorithms. In: IEEE Transactions on Computers, vol. 59, issue 4, pp. 532-544 (2010)

32. Eimuri, T., Salehi, S.: Using DPSO and B&B Algorithms for Hardware/Software Partitioning in Co-design. In: Proceedings of the Second International Conference on Computer Research and Development, pp. 416-420 (2010)
33. Huong, P., Binh, N.: An approach to design embedded systems by multi-objective optimization. In: Proceedings of the International Conference on Advanced Technologies for Communications, pp. 165-169 (2012)
34. Li, S., Liu, Y., Hu, S., He, X., Zhang, Y., Zhang, P., Yang, H.: Optimal partition with block-level parallelization in C-to-RTL synthesis for streaming applications. In: Proceedings of the 18th Asia and South Pacific Design Automation Conference, pp. 225-230 (2013)
35. Qawasmeh, A., Malik, A., Chapman, B.: OpenMP task scheduling analysis via OpenMP runtime API and tool visualization. In: Parallel & Distributed processing symposium workshop (IPDPSW), 2014 IEEE International, pp. 1049-1058, IEEE (2014)
36. Dagum, L., Menon, R.: OpenMP: an industry-standard API for shared-memory programming. In: Computational Science & Engineering, vol. 5, issue 1, pp. 46-55, IEEE (1998)
37. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS 2008, LNCS 4963, pp. 337–340, DOI 10.1007/978-3-540-78800-3_24, Springer (2008)
38. Brummayer, R., Biere, A.: Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In: TACAS 2009, LNCS 5505, pp. 174–177, Springer (2009)
39. Barrett, C., Conway, C. Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV 2011, LNCS 6806, pp. 171-177, Springer (2011)
40. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., Rossum, P., Schulz, S., Sebastiani, R.: MathSAT: Tight integration of SAT and mathematical decision procedures. In: Journal of Automated Reasoning , Springer (2005)
41. Dutertre, B.: Yices 2.2. In: Computer-Aided Verification (CAV 2014), LNCS 8559, pp. 737–744, Springer (2014)
42. Tang, T., Lin, Y., Ren, X.: Mapping OpenMP concepts to the stream programming model. In: Int. Conference on Computer Science Education (ICCSE), pp. 1900-1905, IEEE (2010)
43. Wu, M., Wu, W., Tai, N., Zhao, H., Fan, J., Yuan, N.: Research on OpenMP model of the parallel programming technology for homogeneous multicore DSP. In: International Conference on Software Engineering and Service Science (ICSESS), pp. 921,924, IEEE (2014)
44. Holzmann, G.J., Joshi, R., Groce, A.: Swarm Verification Techniques. In: IEEE Transactions on Software Engineering, vol. 37, issue 6, pp. 845, 857, IEEE (2011)
45. Holzmann, G.: Parallelizing the spin model checker. In: Proceedings of the 19th International Conference on Model Checking Software (SPIN 2012), LNCS 7385, pp. 155-171, Springer (2012)
46. Wintersteiger, C., Hamadi, Y., De Moura, L.: A Concurrent Portfolio Approach to SMT Solving. In: International Conference on Computer-Aided Verification (CAV 2009), LNCS 5643, pp. 715-720, Springer (2009)
47. Bjørner, N., Phan, A-D., Fleckenstein, L.: νZ - An Optimizing SMT Solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015), LNCS 9035, pp. 194-199, Springer (2015)