Multi-Core Model Checking and Maximum Satisfiability Applied to Hardware-Software Partitioning

Alessandro Trindade, Federal University of Amazonas
Hussama Ismail, Federal University of Amazonas
Edilson Galvão, Federal University of Amazonas
Helder Silva, Federal University of Amazonas
Lucas Cordeiro, Federal University of Amazonas

We present an alternative approach to solve the hardware and software partitioning problem, which uses Bounded Model Checking (BMC) based on Satisfiability Modulo Theories (SMT) in conjunction with a multi-core support using Open Multi-Processing. In a nutshell, the multi-core SMT-based BMC approach allows initializing many verification instances based on the number of available processing cores. Each instance checks for a different optimum value until the optimization problem is satisfied. We implement our algorithms on top of the Efficient SMT-Based Context-Bounded Model Checker (ESBMC) and also integrate the maximum satisfiability solver νZ tool into ESBMC. We compare all our approaches to another state-of-the-art optimization tool (Matlab). Experimental results show that there is no single optimization tool to solve all hardware-software partitioning benchmarks; however, Matlab and ESBMC-νZ are the most efficient ones to solve hardware-software partitioning problems, although multi-core ESBMC had a significant performance improvement in particular cases.

Additional Key Words and Phrases: hardware-software co-design, hardware-software partitioning, optimization, model checking, multi-core, OpenMP.

1. INTRODUCTION

Nowadays, with the strong development of embedded systems, the design phase plays an important role. At early stages, the design is split into separated flows: hardware and software. The partitioning decision process, which deals with decisions upon which parts of the application have to be designed in hardware (HW) and which in software (SW), must be supported by any well-structured methodology. If not, this leads to a number of issues (design flow interruptions, redesigns, and undesired iterations) which affect the overall development process, the quality, and the lifecycle of the final system.

Starting at the 1990s, intensive research was performed in HW-SW partitioning, and several approaches proposed, as shown in [**?**] and [**?**]. In [**?**, **?**] was shown that it is possible to use Bounded Model Checking (BMC) based on Satisfiability Modulo Theories (SMT), implemented in a tool called Efficient SMT-Based Context-Bounded Model Checker (ESBMC), in order to perform HW-SW partitioning in embedded systems. The present work extends those studies since there is a substantial improve-

A. Trindade, H. Ismail, R. Degelo, E. Galvão
Graduate Program in Electrical Engineering, Federal University of Amazonas, Brazil
{alessandro.b.trindade, hussamaismail, rdegelo, esj.galvao}@gmail.com

and

H. Silva and L. Cordeiro
Federal University of Amazonas, Brazil
prhsilva2012@gmail.com lucascordeiro@ufam.edu.br

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2010 Copyright held by the owner/author(s). 1539-9087/2010/03-ART39 $15.00
DOI: 0000001.0000001

ment in terms of the SMT-based verification method, which has been extended with a multi-core parallel- and binary-search approaches, as well as, the integration of the Maximum SMT (MaxSMT) solver νZ into ESBMC, which is a state-of-art optimization tool based on SMT [**?**].

Here, we exploit the availability of multi-core processors; in particular, SMT-based verification methods are applied to the HW-SW partition problem in three different ways using a multi-core ESBMC approach with OpenMP [**?**]: ESBMC-SS using a sequential-search (SS), ESBMC-PS using a parallel-search (PS), and ESBMC-PB using a binary-search (BS). Experimental results are compared to ILP (integral linear programming), GA (generic algorithms) in a multi-core version, and also to νZ, which supports only a single-core approach [**?**]. The ILP and GA algorithms are implemented with the optimization toolbox of Matlab [**?**], while νZ is a built-in tool to the SMT solver Z3. All multi-core ESBMC approaches, together with νZ, are implemented with the Efficient SMT-based Context-Bounded Model Checker (ESBMC) tool [**?**].

Contributions. The main contribution of the present study is to describe and evaluate a comprehensive SMT-based BMC approach in a multi-core architecture applied to solve HW-SW optimization problems. Additionally, we integrate the MaxSMT solver νZ into an off-the-shelf BMC tool, which is typically used for software verification, in order to formulate and solve optimization problems within the logical context of constraints. Experimental results 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 an SMT-based BMC approach. Although there is no single tool for efficiently solving all HW-SW partitioning problems, we show that the MaxSMT solver νZ is faster than other state-of-the-art optimization tools for small- and medium-size optimization problems. To the best of our knowledge, this is the first work to use a multi-core SMT-based verification and a MaxSMT solver to check for HW-SW partitioning problems in embedded systems.

Availability of Data and Tools. Our experiments are based on a set of publicly available benchmarks. All benchmarks, tools, and results of our evaluation are available on a supplementary web page[1].

This article is organized as follows: Section 2 gives a background on optimization techniques, νZ, ESBMC, and OpenMP tools. Section 3 describes the informal and formal mathematical modeling. The SMT-based BMC method is presented in Section 4, and in particular, Section 4.5 presents the partitioning model using νZ. In Section 5, we show the experimental results using several embedded systems applications. In Section 6, we discuss the related work and we conclude and describe future work in Section 7.

2. BACKGROUND

The HW-SW partitioning problem is typically represented as a set of constraints and an objective function in linear programming. We describe the linear programming problem and present related tools that are used to model and solve the HW-SW partitioning problem.

2.1. Optimization

Optimization is the act of obtaining the best result (*i.e.*, the optimal solution) under given circumstances [**?**].

There is no single method available for efficiently solving all optimization problems [**?**]. The most well-known technique is linear programming, which is an method

[1]http://esbmc.org/

applicable for the solution of problems in which the objective function and the constraints appear as linear functions of the decision variables. A particular case of linear programming is ILP, in which the variables can assume just integer values. Eq. 1 shows a typical linear programming problem, where A and b are vectors or matrixes that describe the constraints

$$minf^t x \; such \; that = \begin{cases} A.x \leq b, \\ Aeq.x = beq, \\ x \geq 0. \end{cases} \tag{1}$$

In some cases, the time to find a solution using ILP is impractical. Even with the use of powerful computers, a problem can take hours before an optimal solution is reached. If the optimization problem is complex, some heuristics can be used to solve the same problem faster, *e.g.*, those used in the GA [**?**]. The only drawback is that the found solution may not be the global minimum or maximum. Alternatively, tools such as νZ and ESBMC can be used to solve optimization problems so that the global minimum or maximum solution is found. The following sections describe the main features of νZ and ESBMC tools.

2.2. Bounded Model Checking with ESBMC

Among the recent model checking 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 [**?**]. In particular, BMC based on Boolean Satisfiability (SAT) has been introduced as a complementary technique to binary decision diagrams for alleviating the state explosion problem.

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) ψ such that ψ is satisfiable if and only if ϕ has a counterexample of depth k or less [**?**]. To cope with increasing software complexity, SMT solvers can be used as back-ends for solving the generated VCs, as shown in [**?**], [**?**], [**?**].

In this study, ESBMC has been used as a BMC tool to solve HW-SW partitioning problems [**?**]. In particular, there are two directives in ESBMC that can be used to guide a model checker to solve an optimization problem: ASSUME and ASSERT. The directive ASSUME is responsible for ensuring the compliance of constraints (software costs), and the directive ASSERT controls the halt condition (minimum hardware cost). Then, with some C/C++ code, it is possible to guide ESBMC to solve optimization problems.

2.2.1. ESBMC Architecture. Fig. 1 shows the current ESBMC architecture, which consists of the C/C++ parser, GOTO Program, GOTO Symex, and SMT solver [**?**]. In particular, ESBMC compiles C/C++ code into equivalent GOTO-programs (*i.e.*, control-flow graphs) using a gcc-compliant style. GOTO-programs can then be processed by the symbolic execution engine, called GOTO Symex, where two recursive functions compute the constraints (C) and properties (P); finally, it generates two sets of equations (i.e., $C \wedge \neg P$), which are checked for satisfiability by an 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 [**?**], Boolector [**?**], MathSAT [**?**], CVC4 [**?**], and Yices [**?**]. Most of them do provide neither multi-threaded support nor a parallel version to solve the generated SMT equations.

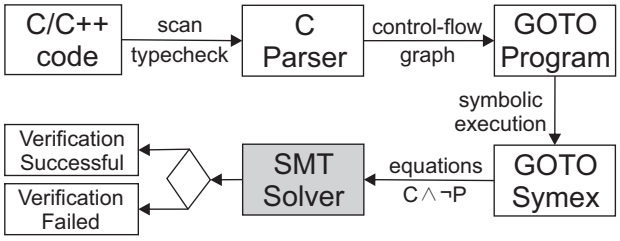

Fig. 1. ESBMC Architecture.

2.3. OpenMP

The OpenMP is a set of directives for parallel programming that augments C/C++ and Fortran languages. OpenMP supports most processor architectures and operating systems, *e.g.*, Solaris, AIX, HP-UX, Linux, Mac OS X, and Windows. OpenMP uses a portable and very robust model that makes easy for programmers to develop parallel applications for a variety of platforms.

In particular, OpenMP uses the fork-join model of parallel execution [**?**]. The main thread executes the sequential parts of the program; if a parallel region is encountered, then 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 returns to the single-threaded execution mode [**?**].

The most basic directive of OpenMP is the "#pragma omp parallel for", which parallelizes the enclosing loop; a basic OpenMP example is shown below:

Listing 1. OpenMP basic sample.

```
01.  int i;
02.  #pragma omp parallel for
03.  for (i = 0; i < 10; i++)
04.       a[i] = 2 * i;
```

In the above example, the for loop is executed in parallel. Each iteration of the loop is executed in a separated thread; and each thread may use an idle processor. There is also a way to specify critical regions, which is a code block that is guaranteed to be executed by a single thread at a time. To create a critical region, the "#pragma omp critical" directive is used.

2.4. Solving Optimization Problems with νZ

In this study, the SMT solver Z3 is used to check for the satisfiability of formulas generated from the HW-SW partitioning problem [**?**]. In particular, we exploit the use of MaxSMT solver νZ, which is implemented on top of the SMT solver Z3, in order to solve optimization problems; νZ base function is to optimize objective functions, which formulate optimized criteria, within the logical context of constraints. νZ also includes an incremental version of the Maximum Resiliency (MaxRes) [**?**], in order to achieve Maximum Satisfiability (MaxSAT) [**?**] and a Simplex to solve numbers without defined patterns.

In νZ, MaxSAT is responsible for the restrictions, while OptSMT optimizes linear arithmetic objectives [**?**]. In summary, νZ provides three main functions that extend Z3 for solving optimization problems, which are: *maximize*, *minimize*, and *assert-soft*.

— **maximize(T)** this function informs to the solver that a given variable T should be maximized, which includes real, integer, or bit-vector variables.
— **minimize(T)** this function informs the solver that a given variable T should be minimized, the accepted types are the same as maximize function.
— **Assert-Soft F : weight n** the function *assert-soft* adds a restriction to F, which can also add a weight n; the default value is 1.

As an example, one can optimize $(K + W)$, with restrictions in $(K < 2)$ and $(W - K < 1)$. The expected result of this optimization problem described in the code below is 2. In fact, the model generated by νZ shows that $K = 1$ and $W = 1$.

Listing 2. Example of SMT formula using νZ.

```
1. (declare-const K Int)
2. (declare-const W Int)
3. (assert (< K 2))
4. (assert (< (- W K) 1))
5. (maximize (+ K W))
6. (check-sat)
```

Fig. 2 shows the νZ architecture. Initially, the SMT formula with objectives is converted to $0-1$ constraints, which leads to a Pseudo-Boolean Optimization (PBO) [**?**, **?**]. If there are many objective functions, νZ invokes OptSAT for arithmetic or MaxSAT for soft constraints. For constraints using real values, νZ combines linear arithmetic objectives and uses only one instance of OptSMT. When "soft constrains" is used in the mode " lexicographic", νZ invokes MaxSAT using multiple calls for its engine.

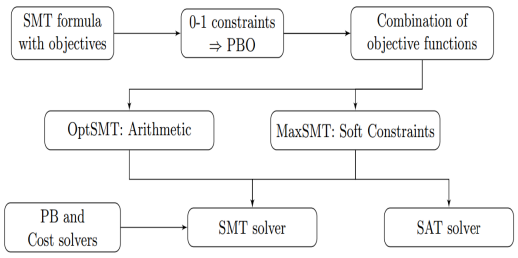

Fig. 2. νZ Architecture extracted from [**?**].

Z3 is available for platforms in C, C++, Java, .NET, and Python; it is possible to download Z3 with νZ from its github repository [**?**]. In this work, the python API is used to formulate HW-SW partitioning problems using the νZ tool.

3. MATHEMATICAL MODELING

The mathematical modeling of the HW-SW partitioning problem was taken from [**?**, **?**].

3.1. Informal Model (or Assumptions)

The informal model 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. 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. 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. 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 is only on the partitioning problem. That configuration describes a first-generation co-design, where the focus is on bipartitioning [**?**].

3.2. Formal Model

The inputs of the problem are: a directed simple graph $G = (V, E)$, called the task graph of the system, is necessary. The vertices $V = \{V_1, V_2, \ldots, V_n\}$ represent the nodes that are the components of the system that will be partitioned. The edges E represent communication between components. Additionally, each node V_i has a cost $h(V_i)$ (or h_i) of hardware if implemented in hardware and a cost $s(s_i)$ (or s_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 Arató *et al.* [**?**], 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 $V_h \cap V_s = 0$. The crossing edges are $E_p = \{(V_i, V_j) : V_i \in V_s, V_j \in V_h$ or $v_i \in V_h, v_j \in V_s$. The hardware cost of P is given by Eq. 2

$$H_p = \Sigma_{v_i \in V_H} h_i \tag{2}$$

and the software cost of P (*i.e.*, software cost of the nodes and the communication cost) is given by Eq. 3

$$S_p = \Sigma_{v_i \in V_s}\, s_i + \Sigma_{(v_i, v_j) \in\, E_p} c(V_i, V_j) \tag{3}$$

Three different optimization and decision problems can be defined. In this paper, the focus is on the case that S_0 is given, i.e., to find a P HW-SW partitioning so that $S_p \leq S_0$ and H_p is minimal (*i.e.*, system with hard real-time constraints). Based on Equations 1 and 3, the constraints can be reformulated as

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

where x represents the decision variable. Concerning the complexity of this problem, Arató *et al.* [**?**] demonstrate that it is NP-Hard [**?**].

4. ANALYSIS OF THE PARTITIONING PROBLEM USING ESBMC

As computer hardware architecture moves from single- to multi-cores, parallel programming environments should be exploited to take advantage of the ability to run several threads on different processing cores. This section describes the verification algorithm using sequential ESBMC, followed by three multi-core model checking algorithms and the integration of the MaxSMT solver νZ, in order to speed up the HW-SW partitioning verification.

4.1. Verification Algorithm using Sequential ESBMC

Algorithm 3 shows ESBMC pseudocode with the same constraints 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 stops the algorithm.

The ESBMC algorithm starts with the declarations of hardware, software, and communication costs. S_0 must also be defined, as the transposed incidence matrix and the identity matrix, as typically done in MATLAB. Here, matrices A and b are generated. At that point, the ESBMC algorithm starts to differ from the ILP and GA presented in [**?**].

It is possible to inform to ESBMC with which type of values the variables must be tested. Therefore, there is a declaration to populate all decision variables x with nondeterministic Boolean values. Those values that change for each test will generate a possible solution and obey the constraints. If this is achieved, then a feasible solution is found and the ASSUME directive is responsible for ensuring the compliance of constrains (i.e., $A.x \leq b$).

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 (a *true-false* statement). If the predicate is *FALSE*, then the optimization is finished, *i.e.*, the solution is 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 algorithm restarts and a new possible solution is generated and tested until the *ASSERT* generates a *false* condition. When the *false* condition happens at verification-time, the execution code is aborted and ESBMC presents the counterexample that caused the condition to be broken. That is the point in which the solution is presented (minimum HW cost).

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

Listing 3. Pseudocode describing sequential ESBMC.

```
01. Initialize Variables
02. Declare number of nodes and edges
03. Declare hardware cost of each node as array (h)
04. Declare software cost of each node as array (s)
05. Declare communication cost of each edge (c)
06. Declare the initial software cost of (S_0)
07. Declare transposed incidence matrix graph G (E)
08. Define the solutions variable (X_i ) as Boolean
09. main {
```

```
10.  for TipH = 0 to Hmax do {
11.   populate X_i with nondeterministic/test values
12.   Calculate s(1-x)+c|E_x| and store at variable
13.   Requirement isued by Assume (Variable <= S_0)
14.   Calculate Hp cost Based on value tested of X_i
15.   Violation check with Assert(H_p > TipH)
16.  }
17. }
```

4.2. Multi-core ESBMC with OpenMP (ESBMC-SS)

Typically, ESBMC verification runs are performed only in a single-core. If the processor provides 8 processing cores, only one is used for the verification and the others remain idle. Thus, there is a significant unused hardware resource during this process.

To optimize the CPU resources utilization without modifying the underlying SMT Solver, the Open Multi-Processing (OpenMP) library [**?**] is used in this present work as a front-end for ESBMC. Fig. 3 shows our first approach called "Multi-core ESBMC".

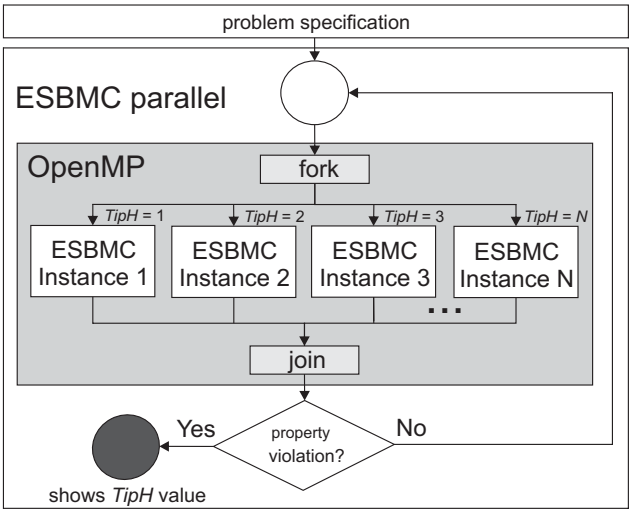

Fig. 3. ESBMC-SS Approach.

ESBMC-SS obtains the problem specification represented by a C program. The HW-SW partitioning is violated, when the correct optimum value (*TipH*) parameter is reached; ESBMC-SS starts a parallel region with different instances of ESBMC, based on the number of available processing cores. All these ESBMC instances run independently of each other, as shown in Fig. 3. Note that there is no shared-memory (or message-passing) mechanism among the threads. In particular, different threads are managed by the OpenMP API, which is responsible for the thread lifecycle: start, running, and dead states, using different values as condition. After executing N instances, if there is no code violation, then multi-core ESBMC starts new instances again. During the parallel region execution, if a violation is found in any running thread, then it presents a counterexample with the violation condition and the verification time. If all threads of the batch processing are terminated, then ESBMC-SS finishes its execution.

4.3. Multi-core ESBMC with OpenMP using Workers (ESBMC-PS)

The previous parallelization is implemented by continuously forking ESBMC instances until the first violation is found. However, since OpenMP only returns from a parallelized loop, when every forked thread finishes, some processing cores could remain idle for some period of time.

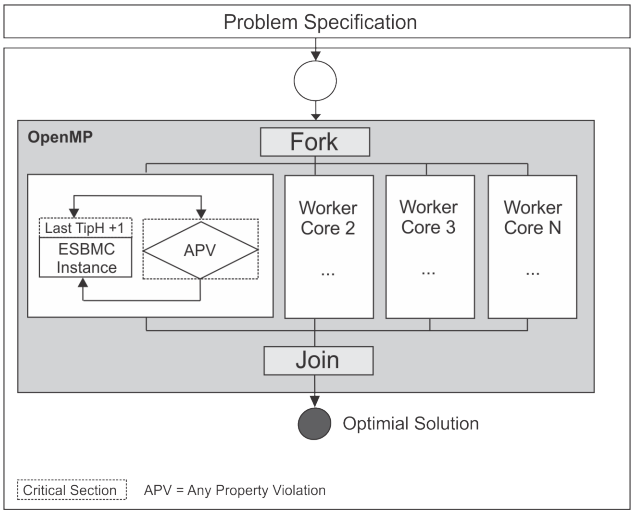

Fig. 4. ESBMC-PS Approach.

Consequently, the second approach aims to remove the idle time from the parallel loops, by creating workers inside threads so that the next step is immediately executed if there is a processing core available, as shown in Fig. 4. This approach could potentially lead to great performance improvements, but as ESBMC checks for each step almost at the same rate, the processor does not remain idle for a longer period and thus there is almost no optimization.

4.4. Multi-core ESBMC with OpenMP using Binary Search (ESBMC-PB)

The most optimized approach applies a parallelized binary search to reduce the amount of steps to be executed in order to find the optimal solution. A controller is designed to return the step to be executed so that the number of verification runs are substantially reduced. The parallelized binary search accomplishes this by splitting the domain of possible values into intervals and then by returning the middle of the largest interval so that two new intervals are created.

As an example, given a problem of domain from 1 to 20 (see Fig. 6), we firstly create an initial interval from 1 to 20. When the next available core asks for a step to be executed, the controller obtains the largest interval, *i.e.*, $[1, 20]$, divides it by two, which creates two new intervals (*i.e.*, $[1, 9]$ and $[11, 20]$), and returns the middle of the original interval (*i.e.*, 10). The controller also checks whether an interval has less than two elements to avoid creating empty or invalid intervals.

Note that there might gaps between steps, which are produced by the customized binary search. For instance, in the example shown in Fig. 6, if step 10 returns *false*, then one can conclude that all steps after 10 is *false* as well. However, if the same step 10 returns *true*, we can assume that all steps before 10 is *true* as well. As a result, an auxiliary method to remove unnecessary steps is implemented in the controller by

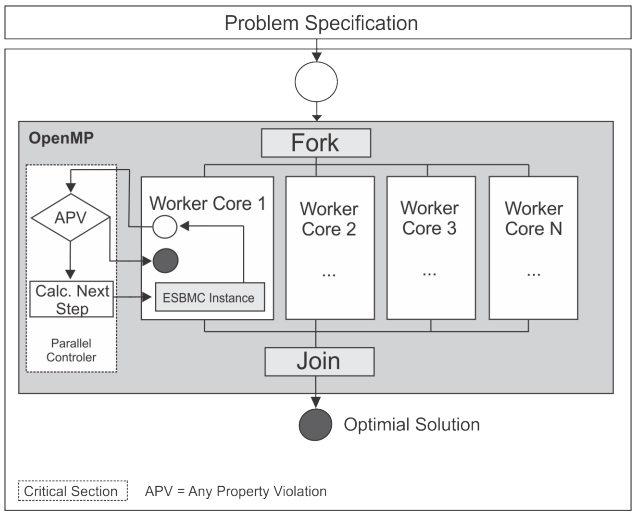

Fig. 5. ESBMC-PB Approach.

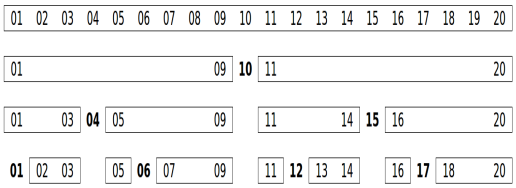

Fig. 6. Binary Step Calculation.

removing or shrinking existing intervals. This approach leads to a high impact in the verification time. However, if a step is running and is not needed anymore, the worker kills the forked process and starts a new one.

Algorithm 4 describes how the customized binary search calculates and returns the step to be executed.

Listing 4. Steps Calculation using Intervals.

```
01. GetNextStep(){
02.  int largestChunk = -1;
03.  chunk largest;
04.  for each(chunk in chunks){
05.     if(chunk.right - chunk.left > largestChunk){
06.           largestChunk = chunk.right - chunk.left;
07.           largest = chunk;
08.     }
```

```
09.    }
10.    chunks.remove(largest);
11.    int median = largest.left + floor((largest.right − largest.left) / 2)
12.    if(median > 0){
13.       if(largest.right − largest.left > 1)
14.          chunks.add(new chunk(largest.left, median − 1));
15.       if(largest.right != largest.left)
16.          chunks.add(new chunk(median + 1, largest.right);
17.     }
18.    return median;
19. }
```

Note that Algorithm 4 is called from each worker in order to get the next step to execute if it exists; otherwise, either zero or a negative number is returned. From lines 4 to 9, the algorithm finds the largest interval. Then, from the line 10 the largest interval is removed and the median is calculated in line 11. After that, two new intervals are created, the left side (in line 14) and the right side (in line 16). At the end, the median is returned.

Listing 5. Worker sample.

```
01. step = controller.GetNextStep();
02. int pid = ExecuteStep(step);
03. while(isRunning(pid)){
04.    if(!controller.isNeeded(step))
05.     kill(pid);
06. }
```

Algorithm 5 describes how the worker starts and monitors ESBMC instances. The algorithm starts by retrieving the step to be executed from the controller (line 1), then initiates the ESBMC instance and obtains the process *id* from the forked process (line 2). While the step is being executed, the controller checks whether this step is still needed (line 4). If not, then the ESBMC instance is killed (line 5) and the worker is free to initiate another step.

4.5. Analysis of the partitioning problem using νZ (ESBMC-νZ)

Algorithm 6 encodes the objective function and constraints related to the HW-SW partitioning problem using νZ functions [**?**]. A νZ logical context must firstly be created (line 2), in order to add constraints and to check whether a given model exists to the set of constraints. Note that the number of nodes and edges, software, hardware, and communications costs as well as the incidence matrix E must also be declared.

The arithmetic expressions from lines 10 to 12 represent the constraints described in Eq. 4. Here, variable *SC* refers to the software cost, while *CC* denotes the communication cost. In line 12, the *Fobj* (objective function) is declared, which denotes the product between the hardware cost and the decision variables vector, which contains only Boolean values. *Fobj* should be minimized to obtain the optimal hardware solution. To achieve this, two constraints are imposed to νZ: the first one refers to the sum of the software and communication costs, where the result should be less than S_0; and the second one informs to νZ that *Fobj* should be minimized. Finally, the model is checked by νZ and if there is a solution that meets the constraints, then the *Fobj* value is provided.

Listing 6. Pseudocode describing νZ.

```
01.# Initialize Variables
02.  Create vZ context
03.  Create binary vector (x)
04.  Declare number of nodes, edges and S_0
05.  Declare hardware cost of each node as array (h)
06.  Declare software cost of each node as array (s)
07.  Declare communication cost of each edge (c)
08.  Declare transposed incidence matrix graph G (E)
09.# Arithmetic Expressions
10.  SC = s(1-x)
11.  CMC = c*|EX|
12.  Fobj =  x[i] * h[i]
13.# Assert Constraints
14.  Add constraints (SF + CMC <= S_0)
15.  Add constraints to minimize Fobj
16.  Check Model
17.  Print Result
```

5. EXPERIMENTAL EVALUATION

This section is split into three parts. The setup is described in Section 5.1, while Section 5.2 describes all benchmarks that were used for performing the experimental evaluation. Section 5.3 reports a comparison among Matlab [**?**], Multi-Core ESBMC, and MaxSMT νZ [**?**] using a set of standard HW-SW partitioning benchmarks [**?**].

5.1. Experimental Setup

ESBMC 1.24 running on a 64-bit Ubuntu $14.04.1$ LTS operating system was used. A parallel approach of the ESBMC-SS, ESBMC-PS, ESBMC-PB were implemented in C++11. Version $2.0.1$ of Boolector SMT-solver [**?**] (freely available) was used as the default solver for ESBMC. For ILP and GA formulations, MATLAB R2013a from MathWorks with Parallel Computing Toolbox was used [**?**]. MATLAB is a dynamically typed high-level language, known as the state-of-the-art mathematical software [**?**] and is widely used by the engineering community [**?**]. ESBMC-νZ as a built-in tool to Z3 was also used [**?**].

All experiments were conducted on an otherwise idle Intel Core i7-2600 (8-cores), with 3.4 GHz and 24 GB of RAM, running Ubuntu 64-bits. Each time was measured 3 times (average taken). Based on standard deviation and tolerance interval to each set of time sample, it was obtained a statistical confidence of $XX\%$ to ESBMC, 92% to νZ, and $YY\%$ to ILP. A time out condition (TO) is reached when the verification time is longer than 7200 seconds. A memory out (MO) occurs when the tool reaches 24GB of memory.

5.2. Description of Benchmarks

To perform the experiments, some benchmarks provided by Mann *et al.* [**?**] were used, as shown in Table I. The nodes in the graphs correspond to high-level language instructions. Software and communication costs are time dimensional, and hardware

costs represent the occupied area. The first three benchmarks are extracted from MiBench [**?**]. The clustering and fuzzy benchmarks are designed from Mann *et al.* [**?**] and are significantly large benchmarks. From the same authors, very complex benchmarks to test the limits of the applicability of techniques were used (RC6 and Mars).

Table I. Description of Benchmarks.

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

5.3. Experimental Results

Table II shows the experimental results using Matlab (ILP and GA) and ESBMC (ESBMC-MC, ESBMC-PS, ESBMC-PB, ESBMC-νZ) tools.

There is no single tool for efficiently solving all HW-SW partitioning benchmarks. In particular, the best (proposed) solution is ESBMC-νZ, which solves 4 out of 7 benchmarks, but it does not find the optimal solution for the RC6 benchmark; however, ESBMC-νZ is faster than ILP in all supported benchmarks (*i.e.*, CRC32, Patricia, Dijkstra, Clustering), but it returns two TOs (time-outs) related to Fuzzy and Mars benchmarks.

In contrast to ESBMC-νZ, ILP solves 6 out of 7 benchmarks. When ILP produces a result, it typically provides the optimal solution. On the one hand, ILP execution time is slower than νZ in all benchmarks, which are supported by νZ. On the other hand, ILP is faster than ESBMC-SS, ESBMC-PS, and ESBMC-PB in all benchmarks, except for the clustering.

Note further that all multi-core ESBMC implementations produce better results than the sequential one. In particular, ESBMC-PB implementation outperforms all other multi-core ESBMC approaches, where its performance improves as the number of nodes and edges increase. One notable case is the clustering benchmark, when verified by ESBMC-PB, it executes 3 times faster than ILP and 1.89 times slower than ESBMC-νZ. However, when the amount of nodes is around 30, ESBMC-PB does not outperform ESBMC-νZ and ILP tools. When analyzing all benchmarks, ESBMC-PB produces TO or MO for RC6, Fuzzy, and Mars; however, the results are still promising if we take into consideration that νZ and Matlab are state-of-the-art tools with respect to optimization problems.

Table II. Experimental results of the HW-SW partitioning benchmarks.

		CRC32	Patricia	Dijkstra	Clustering	RC6	Fuzzy	Mars
	Nodes	25	21	26	150	329	261	417
	Edges	32	48	69	331	448	442	600
	S0	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	T(s)	2	1	2	649	1806	TO	5.42
	Hp	15	47	31	241	692	-	876
GA	T(s)	7	7	9	340	2050	1.37	5000
	Error %	13.3	0	29	1.7	-6.5	-37.6	-27.5
ESBMC	T(s)	31	362	292	3010	TO	MO	MO
	Hp	15	47	31	241	-	-	-
ESBMC-SS	T(s)	2	6	7	1615	TO	TO	TO
	Hp	15	47	31	241	-	-	-
ESBMC-PS	T(s)	4	10	11	2238	TO	MO	MO
	Hp	15	47	31	241	-	-	-
ESBMC-PB	T(s)	6	5	6	207	TO	MO	MO
	Hp	15	47	38	241	-	-	-
ESBMC-νZ	T(s)	0.3	0.3	0.7	86.4	TO	TO	TO
	Hp	15	47	31	241	-	-	-

The only technique that is able to solve all benchmarks is GA; however, its precision is not satisfactory since it produces an error rate between -37.6% and 29.0%.

Note that RC6 produced time out for all implementations of ESBMC; νZ e GA did not produce the correct answer, and ILP solves correctly all benchmarks, except for Fuzzy, which produced time-outs and memory-outs in all tools that aim to find the exact solution. ILP was the only one to solve Mars, while GA solved all benchmarks incorrectly.

The clustering benchmark seems to be the limit to test the ESBMC (described) implementations; note, however, that more than 150 nodes lead to TO and MO. ILP shows robustness and produces results even for a high number of nodes and edges.

6. RELATED WORK

Since the second half of the first decade of 2000s, three main paths have been tracked to improve or to present alternative solutions to the optimization of HW-SW parti-

tioning, *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 HW-SW partitioning problem is found. The use of SMT-based verification presented in this paper can be grouped into this category, because the exact solution is found with the given algorithms. The difference is based only on the technique chosen to solve the problem. Another path followed in past initiatives and which has had more studies is the creation of heuristics to speed up the running time of the solution. The difference of this kind of solution to SMT-based verification and maximum satisfiability is based on two facts: all ESBMC implementations guarantee to find the exact solution, but heuristics are faster, when the complexity is greater.

Finally, there are approaches that mixes heuristics with exact solution tools. 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. Only the 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 related studies are restricted to present the model, its modification to programming languages (*e.g.*, C/C++ and Java), and the application to multi-thread algorithms or to embedded systems to check for program correctness. In [**?**] it presents a bounded model checker for C++ programs, which is an evolution of dealing with C programs and [**?**] uses ESBMC for embedded ANSI-C software. In [**?**] and [**?**] it was proven that it is possible to use ESBMC to solve HW-SW partitioning in a single- and multi-core way. There are related studies focused on decreasing the verification time of model checkers by applying Swarm Verification [**?**], and modifications of internal search engines to add support for parallelism [**?**], but there is still the need for initiatives related to parallel SMT solvers [**?**].

Recently, the SMT solver Z3 has been extended to pose and solve optimization problems modulo theories [**?**]. In particular, νZ tool offers substantial performance improvement in optimization problems [**?**, **?**]. As an application example, Pavlinovic *et al.* [**?**] propose an approach which considers all possible compiler error sources for statically typed functional programming languages and reports the most useful one subject to some usefulness criterion. The authors formulate this approach as an optimization problem related to SMT and use νZ to compute an optimal error source in a given ill-typed program. The approach described by Pavlinovic *et al.*, which uses MaxSMT solver νZ, shows a significant performance improvement if compared to previous SMT encodings and localization algorithms.

The problem addressed in this present paper uses a single objective function for minimization. In Partrick *et al.* [**?**], OptiMathSAT and νZ are compared for software optimization problems. OptiMathSAT, using multiple objective functions, works better than νZ; however, for problems with a single objective function, the performance of νZ is better than OptiMathSAT.

7. CONCLUSIONS

We presented five approaches to solve the HW-SW partitioning problem and compared them to other state-of-the-art techniques. Experimental results showed that for a number of nodes larger than 300, the best solution for the HW-SW partitioning problem is ILP. Below that, the best solution turns out to be ESBMC-νZ since its execution time is faster and notorious. ESBMC-PB is a viable alternative for a number of nodes lower than 150. 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.

If considering off-the-shelf tools, as MATLAB to ILP and GA, the coding is simpler. However, ESBMC and νZ have BSD-Style and MIT licenses, respectively and can be

downloaded and used for free. Experimental results also pointed to an improvement of ESBMC, when using a parallel approach. In particular, all three parallel approaches described in this paper produced expressive results. The fastest ESBMC approaches is ESBMC-PB, which produces good results for an intermediate amount of edges and nodes. Thus, considering that nowadays processors have more and more cores, when modeling the problem, it is possible to consider multi-core model checking as an alternative to solve the HW-SW partitioning problem.

Finally, there is an issue about 150 nodes problem, since it seems to be the limit of multi-core ESBMC. However, it really depends on the modeling granularity of the problem. Some researchers propose fine-grained models, in which each instruction can be mapped to either HW or SW. This may lead to thousands of nodes or even more. Others defend coarse-grained models, where decisions are made for larger components, thus even complex systems may consist of just some dozens of nodes to partition. In principle, a fine-grained approach may allow to obtain better partitions, but at the cost of an exponential increase of the search space size. In future work, we will address improvements in ESBMC to remove the parallel layer on top of ESBMC and implement it during symbolic execution so that we can optimize the overall verification time.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Maura Turolla of Telecom Italia for providing specifications about the application scenario.

8. THIS IS AN EXAMPLE OF APPENDIX SECTION HEAD

Channel-switching time is measured as the time length it takes for motes to successfully switch from one channel to another. This parameter impacts the maximum network throughput, because motes cannot receive or send any packet during this period of time, and it also affects the efficiency of toggle snooping in MMSN, where motes need to sense through channels rapidly.

By repeating experiments 100 times, we get the average channel-switching time of Micaz motes: 24.3 μs. We then conduct the same experiments with different Micaz motes, as well as experiments with the transmitter switching from Channel 11 to other channels. In both scenarios, the channel-switching time does not have obvious changes. (In our experiments, all values are in the range of 23.6 μs to 24.9 μs.)

9. APPENDIX SECTION HEAD

The primary consumer of energy in WSNs is idle listening. The key to reduce idle listening is executing low duty-cycle on nodes. Two primary approaches are considered in controlling duty-cycles in the MAC layer.