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
Renato Degelo, 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.

CCS Concepts: •**Hardware** → **Hardware-software codesign; Software tools for EDA;** •**Software and its engineering** → **Model checking; Software verification;**

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 one in software (SW), must be supported by any well-structured methodology. If there is no methodology support, a number of issues, *e.g.*, design flow interruptions, redesigns, and undesired iterations may affect the overall development process, the quality, and the life-cycle of the final system.

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

DOI: 0000001.0000001

Starting at the 1990s, intensive research was performed in HW-SW partitioning, and several approaches proposed, as shown in [Peter Arato 2003] and [Zoltan Adam Mann 2007]. In [Alessandro Trindade 2015a; 2015b] 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 improvement 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 [Nikolaj Bjrner 2015].

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 [Tao Tang 2010]: 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 [Nikolaj Bjrner 2015]. The ILP and GA algorithms are implemented with the optimization toolbox of Matlab [The MathWorks 2013], while νZ is a built-in tool to the SMT solver Z3. All multi-core ESBMC approaches, together with νZ, are implemented with the ESBMC tool [Lucas Cordeiro 2012].

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 4.6 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

[1] http://esbmc.org/

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 [Rao 2009]. There is no single method available for efficiently solving all optimization problems [Rao 2009]. The most well-known technique is linear programming, which is an method 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 [Rao 2009]. The only drawback is that the found solution may not be the global minimum or maximum. Alternatively, tools such as ESBMC and νZ 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 ESBMC and νZ tools.

2.2. Bounded Model Checking with ESBMC

Among the recent model checking techniques, there is one that combines model checking with satisability 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 [Armin Biere 2009]. 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 [Edmund Clarke 2001].

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 [Armin Biere 2009]. To cope with increasing software complexity, SMT solvers can be used as back-ends for solving the generated VCs, as shown in [Lucas Cordeiro 2012], [Alessandro Armando 2009], [Malay Ganai 2006].

In this study, ESBMC has been used as a BMC tool to solve HW-SW partitioning problems [Lucas Cordeiro 2012]. In particular, there are two directives in ESBMC that can be used to guide it 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

[Mikhail Ramalho 2013]. 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 [Leonardo De Moura 2008], Boolector [Robert Brummayer 2009], MathSAT [Clark Barrett 2011], CVC4 [Marco Bozzano 2005], and Yices [Dutertre 2014]. Most of them do provide neither multi-threaded support nor a parallel version to solve the generated SMT equations.

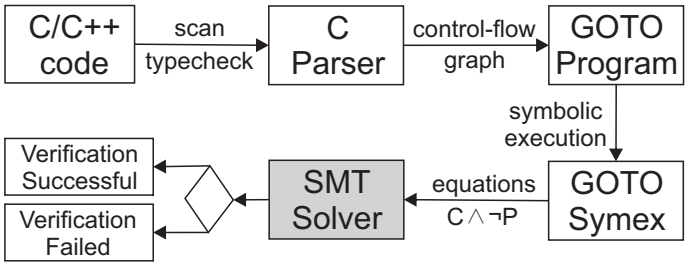

Fig. 1: ESBMC architecture.

2.3. OpenMP

The OpenMP is a set of directives for parallel programming that augments C/C++ and Fortran languages [Muller 2002]. 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 to facilitate the development of parallel applications for a variety of platforms.

In particular, OpenMP uses the *fork-join* model of parallel execution [Muller 2002]. 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 [Minjie Wu 2014].

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

```
1 int k;
2 #pragma omp parallel for
3 for (k = 0; k < 10; k++)
4   a[k] = 2*k;
```

Listing 1: OpenMP basic Example.

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 routinely 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 [Nikolaj Bjrner 2014]. 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) [Federica Paci 2008], in order to achieve Maximum Satisfiability (MaxSAT) [Nina Narodytska 2014] and a Simplex to solve numbers without defined patterns.

In νZ, MaxSAT is responsible for the restrictions, while OptSMT optimizes linear arithmetic objectives [Nikolaj Bjrner 2015]. 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 $(K + W)$, which is subject to 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 $W = 1$.

```
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)
```

Listing 2: Example of SMT formula using νZ.

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) [Peter Barth 1995; Vasco Manquinho 2005]. 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.

Z3 is available for platforms in C, C++, Java, .NET, and Python; it is possible to download Z3 with νZ from its github repository [Magellan 2015]. In this work, the python API is used to formulate HW-SW partitioning problems using the νZ tool.

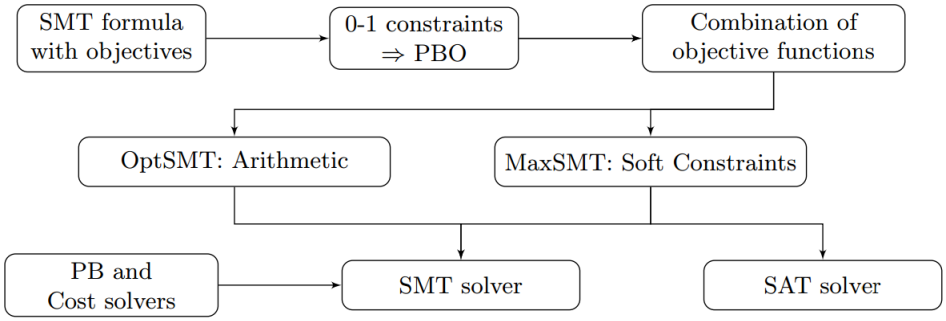

Fig. 2: νZ architecture extracted from [Nikolaj Bjrner 2015].

3. MATHEMATICAL MODELING

The mathematical modeling of the HW-SW partitioning problem was taken from [Peter Arato 2003; Zoltan Adam Mann 2007].

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 [Teich 2012].

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 $s(V_i)$ (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 [Peter Arato 2003], 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, which is thus related to system with hard real-time constraints. Based on Eq. 1 and 3, the constraints can be reformulated as

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

where x represents the decision variable. Concerning the complexity of this problem, [Peter Arato 2003] demonstrate that it is NP-Hard [Thomas Cormen 2009].

4. ANALYSIS OF THE PARTITIONING PROBLEM

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 into ESBMC, in order to speed up the HW-SW partitioning verification. HW-SW partitioning using ILP-based and Genetic Algorithms are also explained.

4.1. Partitioning problem using ILP-based,Genetic Algorithms

The ILP and GA were taken from our previous studies [Alessandro Trindade 2015a; 2015b]. Both use slack variables in order to be possible to represent constraints and to use commercial tools. However, GA had improvements from the parameters of related studies in order to increase the solution accuracy without producing timeout. The tuning was performed by empirical tests and resulted in changing of three parameters, which are passed to the function *ga* of MATLAB [The MathWorks 2013]: the population size was set from 300 to 500, the Elite count changed from 2 (default value) to 50, and the number of generations changed from $100 * NumberOfVariables$ (default) to 75.

4.2. 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 [Alessandro Trindade 2015a].

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 those 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.

```
1  Initialize Variables
2  Declare number of nodes and edges
3  Declare hardware cost of each node as array (h)
4  Declare software cost of each node as array (s)
5  Declare communication cost of each edge (c)
6  Declare the initial software cost of (S0)
7  Declare transposed incidence matrix graph G (E)
8  Define the solutions variable (Xi) as Boolean
9  main {
10  For TipH = 0 to Hmax Do {
11   populate Xi with nondeterministic/test values
12   Calculate s(1-x)+c|Ex| and store at variable
13   Requirement isued by Assume (Variable <= S0)
14   Calculate Hp cost Based on value tested of Xi
15   Violation check with Assert(Hp > TipH)
16  }
17 }
```

Listing 3: Pseudocode describing sequential ESBMC

4.3. 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 [Leonardo Dagum 1998] is used in this present work as a front-end for ESBMC. Fig. 3 shows our first approach called ESBMC sequential-search “ESBMC-SS”.

ESBMC-SS obtains the problem specification represented by a ANSI-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,

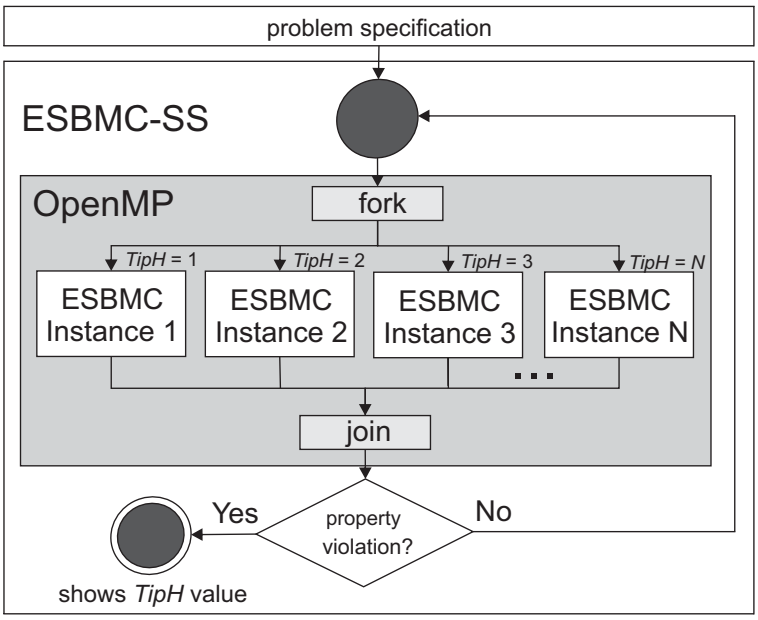

Fig. 3: ESBMC-SS approach.

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 threads. In particular, different threads are managed by the OpenMP API, which is responsible for the thread life-cycle: start, running, and dead states, using different values as condition. After executing N instances, if there is no code violation, then ESBMC-SS starts new instances again; this represents a sequential-search on a multi-core environment. 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.

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

The previous parallelization is implemented by continuously forking ESBMC instances in a sequential manner 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.

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.

4.5. 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

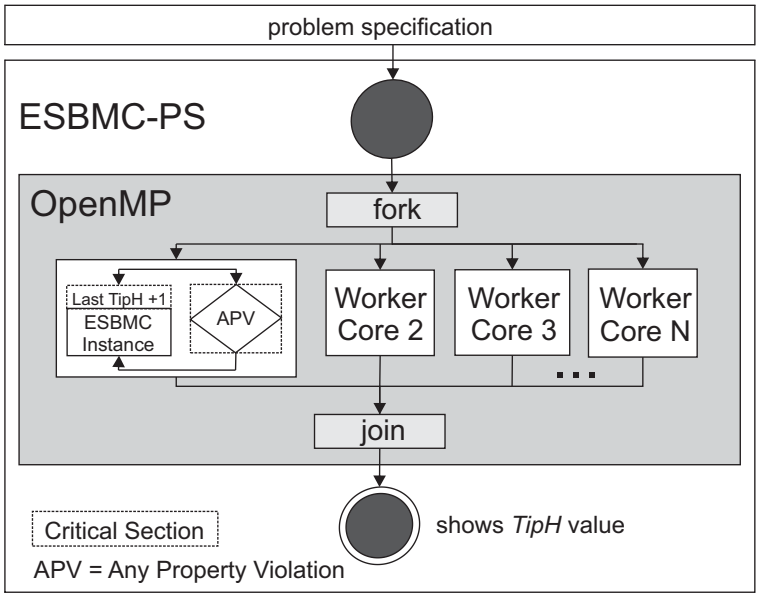

Fig. 4: ESBMC-PS approach.

the domain of possible values into intervals and then by returning the middle of the largest interval so that two new intervals are created.

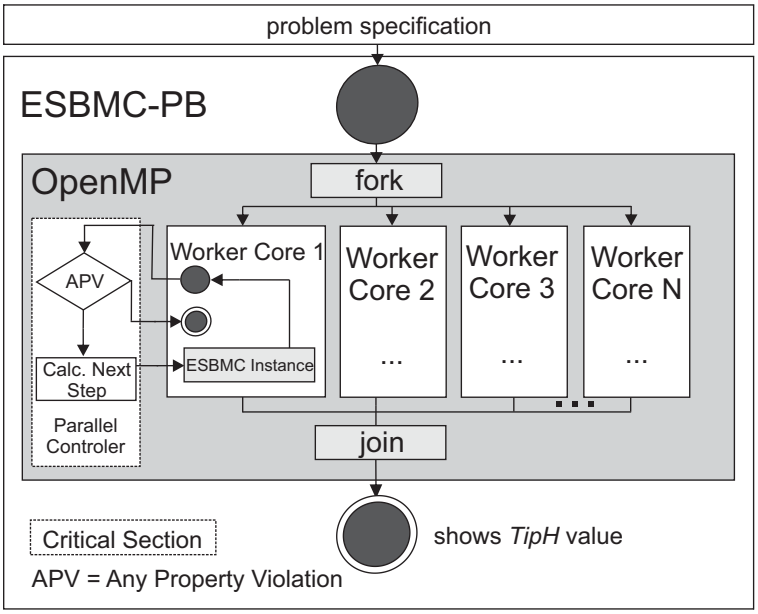

Fig. 5: ESBMC-PB approach.

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 requests 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.

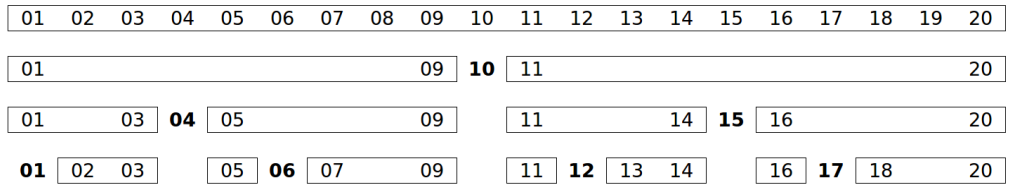

Fig. 6: Binary step calculation.

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 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.

```
1   GetNextStep(){
2   int largestChunk = -1;
3   chunk largest;
4   for each(chunk in chunks){
5      if(chunk.right - chunk.left > largestChunk){
6            largestChunk = chunk.right - chunk.left;
7            largest = chunk;
8      }
9    }
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 }
```

Listing 4: Steps calculation using intervals.

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 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.

```
1 step = controller.GetNextStep();
2 int pid = ExecuteStep(step);
3 while(isRunning(pid)){
4   if(!controller.isNeeded(step))
5    kill(pid);
6 }
```

Listing 5: Worker sample.

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.

4.6. 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 [Nikolaj Bjrner 2014]. 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 ESBMC-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 ESBMC-νZ that *Fobj* should be minimized. Finally, the model is checked by ESBMC-νZ and if there is a solution that meets the constraints, then the *Fobj* value is provided.

```
--Initialize Variables
  Create vZ context
  Create binary vector (x)
  Declare number of nodes, edges and S0
  Declare hardware cost of each node as array (h)
  Declare software cost of each node as array (s)
  Declare communication cost of each edge (c)
  Declare transposed incidence matrix graph G (E)
--Arithmetic Expressions
  SC = s(1-x)
  CMC = c*|EX|
  Fobj =  x[i] * h[i]
--Assert Constraints
  Add constraints (SF + CMC <= S0)
  Add constraints to minimize Fobj
  Check Model
  Print Result
```

Listing 6: Pseudocode describing ESBMC-νZ.

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 [The MathWorks 2013], ESBMC-SS, ESBMC-PS, ESBMC-PB, and ESBMC-νZ using a set of standard HW-SW partitioning benchmarks [Zoltan Adam Mann 2007].

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 [Robert Brummayer 2009] (freely available) was used as the default solver for ESBMC. ESBMC-νZ as a built-in tool to Z3 was also used [Nikolaj Bjrner 2014]. For ILP and GA formulations, MATLAB R2013a from MathWorks with Parallel Computing Toolbox was used [The MathWorks 2013]. MATLAB is a dynamically typed high-level language, known as the state-of-the-art mathematical software [Tranquillo 2011] and is widely used by the engineering community [Luhe Hong 2010].

All experiments were conducted on an otherwise idle Intel Core i7-2600 (8-cores), with 3.4 GHz and 15 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 91.7% to ESBMC (sequential, SS, PB and νZ), 95.9% to ESBMC-PS, and 92.0% to ILP and GA. A timeout condition (TO) is reached when the verification time is longer than 3600 seconds. A memory-out (MO) occurs when the tool reaches 15 GB of memory.

5.2. Description of Benchmarks

To perform the experiments, some benchmarks provided by [Zoltan Adam Mann 2007] 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 [Guthaus M. 2001]. The clustering and fuzzy benchmarks are designed from [Zoltan Adam Mann 2007] 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.

Name	Nodes	Edges	Description
CRC32	25	32	32-bit cyclic redundancy check [Guthaus M. 2001]
Patricia	21	48	Routine to insert values in Patricia Tree [Guthaus M. 2001]
Dijkstra	26	69	Computer shortest paths in a graph [Guthaus M. 2001]
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 three TOs (timeouts) related to RC6, Fuzzy and Mars benchmarks.

In contrast to ESBMC-νZ, ILP solves 5 out of 7 benchmarks. When ILP produces a result, it 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 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.

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 timeout 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 timeouts and memory-outs in all tools that aim to find the exact solution. No tool was capable to solve Mars in less than 3600 seconds, while GA solved all benchmarks, but mostly 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, but limited to 329 nodes.

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 partitioning, *i.e.*, to find the exact solution [Zoltan Adam Mann 2007], to use heuristics to speed up performance time [Peter Arato 2003], and hybrid ones [Peter Arato 2005].

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 [Mikhail Ramalho 2013] it presents a bounded model checker for C++ programs, which is an evolution of dealing with C programs and [Lucas Cordeiro 2012] uses ESBMC for embedded ANSI-C software. In [Alessandro Trindade 2015a] and [Alessandro Trindade 2015b] 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 [Gerard Holzmann 2011], and modifications of internal search engines to add support for parallelism [Holzmann 2012], but there is still the need for initiatives related to parallel SMT solvers [Christoph Wintersteiger 2009].

Recently, the SMT solver Z3 has been extended to pose and solve optimization problems modulo theories [Nikolaj Bjrner 2015]. In particular, νZ tool offers substantial performance improvement in optimization problems [Nikolaj Bjrner 2014; 2015]. As an application example, [Zvonimir Pavlinovi 2015] propose an approach which con-

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)	1.6	1.3	1.6	648.9	1806.2	TO	TO
	Hp	15	47	31	241	692	-	-
GA	T(s)	6.7	7.4	8.8	340.4	2050.0	1371.9	TO
	Error %	13.3	0.0	29.0	1.7	-6.5	-37.6	-
ESBMC	T(s)	30.3	313.7	324.7	MO	MO	MO	MO
	Hp	15	47	31	-	-	-	-
ESBMC-SS	T(s)	2.2	5.8	7.0	1609.3	TO	TO	TO
	Hp	15	47	31	241	-	-	-
ESBMC-PS	T(s)	3.7	10.0	12.0	2468.0	TO	TO	TO
	Hp	15	47	31	241	-	-	-
ESBMC-PB	T(s)	4.3	4.7	6.3	218.7	TO	TO	TO
	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	-	-	-

siders 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 [Zvonimir Pavlinovi 2015], which uses MaxSMT solver νZ, shows a sig-

nificant 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 [Roberto Sebastiani 2015], 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.

References

Lorenzo Platania Alessandro Armando, Jacopo Mantovani. 2009. Bounded model checking of software using SMT solvers instead of SAT solvers. *Intl. Journal on Software Tools for Technology Transfer* 11, 1 (2009), 69–83.

Lucas Cordeiro Alessandro Trindade. 2015a. Applying SMT-based verification to hardware/software partitioning in embedded systems. *Design Automation for Embedded Systems (to appear)* (2015).

Lucas Cordeiro Alessandro Trindade, Hussama Ismail. 2015b. Multi-Core Model Checking to Hardware-Software Partitioning in Embedded Systems (short paper). *V Brazilian Symp. on Computing Systems Engineering (to appear)* (2015).

Hans van Maaren Toby Walsh Armin Biere, Marijn Heule. 2009. Bounded Model Checking. *Handbook of Satisfiability* (2009).

Leonardo de Moura Christoph Wintersteiger, Youssef Hamadi. 2009. A Concurrent Portfolio Approach to SMT Solving. *Proc. of the Intl. Conf. on Computer-Aided Verification* (2009), 715–720.

Morgan Deters Liana Hadarean Dejan Jovanovic Tim King Andrew Reynolds Cesare Tinelli Clark Barrett, Christopher Conway. 2011. CVC4. *Proc. of the Intl. Conf. on Computer-Aided Verification* 6806 (2011), 171–177.

Bruno Dutertre. 2014. Yices 2.2. *Proc. of the Intl. Conf. on Computer-Aided Verification* 8559 (2014), 737–744.

Richard Raimi Yunshan Zhu Edmund Clarke, Armin Biere. 2001. Bounded model checking using satisfiability solving. *Formal Methods in System Design* 19, 1 (2001), 7–34.

Yuqing Sun Elisa Bertino Federica Paci, Rodolfo Ferrini. 2008. Authorization and User Failure Resiliency for WS-BPEL business processes. *Proc. of the Intl. Conf. on Service-Oriented Computing* (2008), 116–131.

Alex Groce Gerard Holzmann, Rajeev Joshi. 2011. Swarm Verification Techniques. *IEEE Transactions on Software Engineering* 37, 6 (2011), 845–857.

Ernst D. Austin T. Mudge T. Brown R. Guthaus M., Ringenberg J. 2001. Matthew Guthaus, Jeffrey Ringenberg, Dan Ernst, Todd Austin, Trevor Mudge, Richard Brown. *Proc. of the annual workshop on workload characterization* (2001), 3–14.

Gerard Holzmann. 2012. Parallelizing the spin model checker. *Proc. of the Intl. Symp. on Model Checking Software* 7385 (2012), 155–171.

Ramesh Menon Leonardo Dagum. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *Proc. of the Intl. Conf. on Computational Science Engineering* (1998), 46–55.

Nikolaj Bjrner Leonardo De Moura. 2008. Z3: An Efficient SMT Solver. *Proc. of the Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems* (2008), 337–340.

Joao Marques-Silva Lucas Cordeiro, Bernd Fischer. 2012. SMT-based Bounded Model Checking for Embedded ANSI-C Software. *IEEE Transactions on Software Engineering* 38, 4 (2012), 957–974.

Jianli Cai Luhe Hong. 2010. The application guide of mixed programming between MATLAB and other programming languages. *Proc. of the Intl. Conf. on Computer and Automation Engineering* (2010), 185–189.

Magellan. 2015. Z3 API - Source code and documentation. *Accessed 18th August 2015* (2015).

Aarti Gupta Malay Ganai. 2006. Accelerating high-level bounded model checking. *Proc. of the IEEE/ACM Intl. Conf. on Computer-Aided Design* (2006), 794–801.

Alessandro Cimatti1-Tommi Junttila Peter Van Rossum Stephan Schulz Roberto Sebastiani Marco Bozzano, Roberto Bruttomesso. 2005. MathSAT: Tight integration of SAT and mathematical decision procedures. *Journal of Automated Reasoning* 35, 1-3 (2005), 265–293.

Felipe Sousa-Hendrio Marques Lucas Cordeiro Bernd Fischer Mikhail Ramalho, Mauro Freitas1. 2013. SMT-Based Bounded Model Checking of C++ Programs. *Intl. Conf. and Workshops on the Engineering of Computer-Based Systems* (2013), 147–156.

Ning Tai-Hongyu Zhao Jiawu Fan Naichang Yuan Minjie Wu, Weiwei Wu. 2014. Research on OpenMP model of the parallel programming technology for homogeneous multicore DSP. *Proc. of the Intl. Conf. on Software Engineering and Service Science* (2014), 921–924.

Matthias Muller. 2002. OpenMP Architecture Review Board, OpenMP C and C++ Application Program Interface, Version 2.0. *OpenMP Architecture Review Board* (2002).

Anh-Dung Phan Nikolaj Bjrner. 2014. vZ - Maximal Satisfaction with Z3. *Proc. of the Intl. Symp. on Symbolic Computation in Software Science* (2014), 1–10.

Lars Fleckenstein Nikolaj Bjrner, Anh-Dung Phan. 2015. vZ - An Optimizing SMT Solver. *Proc. of the Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems* 9035 (2015), 194–199.

Fahiem Bacchus Nina Narodytska. 2014. Maximum Satisfiability Using Core-Guided MAXSAT Resolution. *Proc. of the Conf. on Artificial Intelligence* (2014), 2717–2723.

Andras Orban Peter Arato, Zoltan Adam Mann. 2005. Algorithmic aspects of hardware/software partitioning. *ACM Transactions on Design Automation of Electronic Systems* 10, 1 (2005), 136–156.

Zoltan Adam Mann Andras Orban David Papp Peter Arato, Sandor Juhasz. 2003. Hardware/software partitioning in embedded system design. *Intl. Symp. on Intelligent Signal Processing* (2003), 192–202.

Davis Putnam Peter Barth. 1995. Enumeration Algorithm for Linear Pseudo-Boolean Optimization. *Technical Report MPI-I-95-2-003, Max Plank Institute for Computer Science* (1995).

Singiresu Rao. 2009. Engineering Optimization: Theory and Practice. 4th edition. (2009).

Armin Biere Robert Brummayer. 2009. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. *Proc. of the Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems* 5505 (2009), 174–177.

Patrick Trentin Roberto Sebastiani. 2015. OptiMathSAT: A Tool for Optimization Modulo Theories. *Proc. of the Intl. Conf. on Computer Aided Verification* 9206 (2015), 447–454.

Xiaoguang Ren Tao Tang, Yisong Lin. 2010. Mapping OpenMP concepts to the stream programming model. *Proc. of the Intl. Conf. on Computer Science Education* (2010), 1900–1905.

Jurgen Teich. 2012. Hardware/Software Codesign: The Past, the Present, and Predicting the Future. *Proc. of the IEEE, Special Centennial Issue* 100 (2012), 1411–1430.

Inc. The MathWorks. 2013. MATLAB (version R2013a). (2013).

Ronald Rivest Clifford Stein Thomas Cormen, Charles Leiserson. 2009. Introduction to Algorithms. *MIT Press* I-XIX (2009), 1–1292.

Joseph Tranquillo. 2011. Matlab for Engineering and the Life Sciences. *Synthesis Lectures on Engineering. Morgan Claypool* (2011).

Joao Marques-Silva Vasco Manquinho. 2005. Effective Lower Bounding Techniques for Pseudo-Boolean Optimization. *IST/INESC-ID, Technical University of Lisbon, Portugal* (2005).

Peter Arato Zoltan Adam Mann, Andras Orban. 2007. Finding optimal hardware/software partitions. *Formal Methods in System Design* 31, 3 (2007), 241–263.

Thomas Wies Zvonimir Pavlinovi, Tim King. 2015. Practical SMT-Based Type Error Localization. *Proc. of the Intl. Conf. on Functional Programming* (2015), 412–423.