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

## Author A

University A
E-mail: {nonono}@nonono.com

## Author B

University B
E-mail: {nonono}@nonono.com

**Abstract:** We present new alternative approaches to solve the hardware and software (HW-SW) partitioning problem. First, we use Bounded Model Checking (BMC) based on Satisfiability Modulo Theories (SMT) in conjunction with a multi-core support using Open Multi-Processing to create four variants to solve the partitioning problem. Multi-core SMT-based BMC approach allows initializing many verification instances using different approaches based on the number of available processing cores. In particular, 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). Additionally, we integrate the maximum satisfiability solver $\nu$Z tool into ESBMC, which provides a portfolio of approaches for solving linear optimization problems over SMT formulas. We compare all proposed approaches to a state-of-the-art optimization tool (MATLAB). Experimental results show that there is no single optimization tool to solve all HW-SW partitioning benchmarks.

**Keywords:** hardware-software co-design, hardware-software partitioning, optimization, model checking, multi-core, maximum satisfiability.

**Reference** to this paper should be made as follows: No no no. 'Multi-Core Model Checking and Maximum Satisfiability Applied to Hardware-Software Partitioning', *International Journal of Embedded Systems*, Vol. x, No. x, pp.xxx–xxx.

**Biographical notes:** no no no.

No no no.

No no no.

## 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 (HW) and software (SW). The partitioning decision process, which deals with decisions upon which parts of the application have to be designed in hardware and which one in software, 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.

Since the first decade of 2000s, two main paths have been tracked to solve the HW-SW partitioning problem, *i.e.*, to find the exact solution of a optimization problem, as shown in Mann *et al.* (2007); to use heuristics to speed up performance time, as shown in Arato *et al.* (2003) and Arato *et al.* (2005). It is worth mentioning that the using heuristics the final solution is not necessarily an optimal global solution.

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 Ramalho *et al.* (2013) it presents a bounded model checker for C++ programs, which is an evolution of dealing with C programs, and Cordeiro *et al.* (2012) uses ESBMC for embedded ANSI-C software. In Trindade and Cordeiro (2015), and Trindade *et al.* (2015) it was proven that it is possible to use ESBMC to solve HW-SW partitioning in a single- and multi-core way, but the former has performance issues that were improved by the latter, which used only a sequential search to perform multi-core model checking. There are related studies focused on decreasing the verification time of model checkers by applying Swarm Verification, as shown in Holzmann *et al.* (2011), and modifications of internal search engines to add support for parallelism, as shown in Holzmann (2012), but there is still the need for initiatives related to parallel SMT solver, according to Wintersteiger *et al.* (2009).

Recently, the SMT solver Z3 has been extended to pose and solve optimization problems modulo theories, as shown in Bjorner *et al.* (2015). In particular, $\nu$Z tool offers substantial performance improvement in optimization problems, according to Bjorner and Phan (2014) and Bjorner *et al.* (2015). As an application example, Pavlinovi *et al.* (2015) 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 alternative single-core approach as an optimization problem related to SMT and use $\nu$Z to compute an optimal error source in a given ill-typed program.

At this work, we apply SMT-based verification methods 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 (integer linear programming), GA (generic algorithms) in a multi-core version, and also to $\nu$Z, which supports only a single-core approach, according to Bjorner *et al.* (2015). The ILP and GA algorithms are implemented with the optimization toolbox of Matlab, according to MathWorks (2013), while $\nu$Z is a built-in tool to the SMT solver Z3. All multi-core ESBMC approaches, together with $\nu$Z, are implemented with the ESBMC tool. 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.

### 1.1 *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 `http://esbmc.org/`.

### 1.2 *Organization of this Work*

This article is organized as follows: Section 2 gives a background on optimization techniques, $\nu$Z, 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 $\nu$Z. In Section 5, we show the experimental results using several embedded systems applications. In Section **??**, we discuss the related work and we conclude and describe future work in Section 6.

## 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 as defined in Rao (2009). There is no single method available for efficiently solving all optimization problems, according to 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 from the objective function, while $Aeq$ and $beq$ are matrices that describe the linear equality constraints

$$min f^t x \text{ such that } = \begin{cases} A.x \leq b, \\ Aeq.x = beq, \\ x \geq 0. \end{cases} \quad (1)$$

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, according to Rao (2009), *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 ESBMC and $\nu$Z 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 $\nu$Z 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 Biere *et al.* (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, according to Clarke *et al.* (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 $\phi$, and a bound $k$, BMC unrolls the system $k$ times and translates it into a verification condition (VC) $\psi$ such that $\psi$ is satisfiable if and only if $\phi$ has a counterexample of depth $k$ or less, as defined in Biere *et al.* (2009). To cope with increasing software complexity, SMT solvers can be used as back-ends for solving the generated VCs, as shown in Cordeiro *et al.* (2012), Armando *et al.* (2009), Ganai and Gupta (2006).

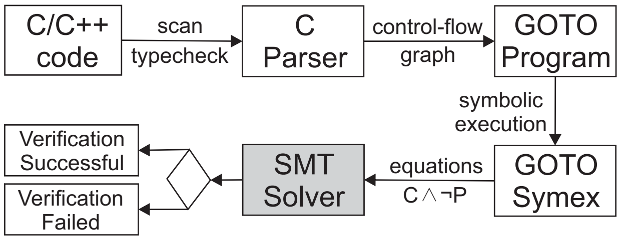
In this study, ESBMC has been used as a BMC tool to solve HW-SW partitioning problems, as shown in Cordeiro *et al.* (2012). In particular, there are two directives in ESBMC that can be used to guide it to solve an optimization problem: ASSUME and ASSERT. The directive ASSUME is responsible for ensuring 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, according to Ramalho *et al.* (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, as shown in Moura and Bjorner (2008); Boolector, as shown in Brummayer and Biere (2009); MathSAT, as shown in Barrett *et al.* (2011); CVC4, as shown in Bozzano *et al.* (2005); and Yices, as shown in Dutertre (2014). Most of them do provide neither multi-threaded support nor a parallel version to solve the generated SMT equations.

**Figure 1** ESBMC architecture.




### *2.3 OpenMP*

The OpenMP is a set of directives for parallel programming that augments C/C++ and Fortran languages, as defined in 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, as shown in 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, as shown in Wu *et al.* (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:

**Figure 2** OpenMP basic Example.

```
int k;
#pragma omp parallel for
```

```
for (k = 0; k < 10; k++)
  a[k] = 2*k;
```

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 $\nu$Z*

In this study, the SMT solver Z3 is used to check for the satisfiability of formulas generated from the HW-SW partitioning problem, as shown in Bjorner and Phan (2014). In particular, we exploit the use of MaxSMT solver $\nu$Z, which is implemented on top of the SMT solver Z3, in order to solve optimization problems; $\nu$Z base function is to optimize objective functions, which formulate optimized criteria, within the logical context of constraints. $\nu$Z also includes an incremental version of the Maximum Resiliency (MaxRes), as shown in Paci *et al.* (2008), in order to achieve Maximum Satisfiability (MaxSAT), as defined in Narodytska and Bacchus (2014) and a Simplex to solve numbers without defined patterns.

In $\nu$Z, MaxSAT is responsible for the restrictions, while OptSMT optimizes linear arithmetic objectives, as shown in Bjorner *et al.* (2015). In summary, $\nu$Z provides three main functions that extend Z3 for solving optimization problems, which are: *maximize*, *minimize*, and *assert-soft*.

- **maximize(T)** this function instructs the solver that a given variable $T$ should be maximized, which includes real, integer, or bit-vector variables.
- **minimize(T)** this function instructs 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.

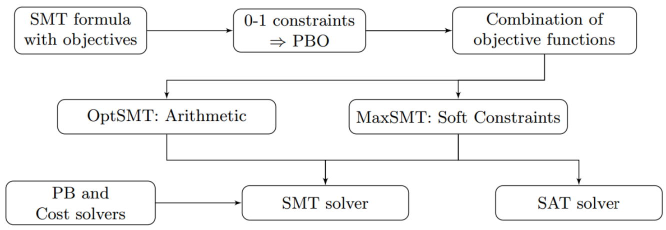
As an example, one can optimize $(K + W)$, which is subject to restrictions in $(K < 2)$ and $(W - K < 1)$. The expected result of this optimization problem described in the code below is 2. In fact, the model generated by $\nu$Z shows that $K = 1$ and $W = 1$.

**Figure 3** OpenMP basic Example.

```
(Declare-Const K Int)
(Declare-Const W Int)
(assert (< K 2))
(assert (< (- W K) 1))
(maximize (+ K W))
(check-sat)
```

Fig. 4 shows the $\nu$Z architecture. Initially, the SMT formula with objectives is converted to $0 - 1$ constraints, which leads to a Pseudo-Boolean Optimization (PBO), as shown in Barth and Putnam (1995) and Manquinho and Marques-Silva (1995). If there are many objective functions, $\nu$Z invokes OptSAT for arithmetic or MaxSAT for soft constraints. For constraints using real values, $\nu$Z combines linear arithmetic objectives and uses only one instance of OptSMT. When "soft constrains" is used in the mode " lexicographic", $\nu$Z invokes MaxSAT using multiple calls for its engine.

**Figure 4** $\nu$Z architecture extracted from Bjorner *et al.* (2015).




Z3 is available for platforms in C, C++, Java, .NET, and Python; it is possible to download Z3 with $\nu$Z from its github repository in Microsoft Research (2015). In this work, the python API is used to formulate HW-SW partitioning problems using the $\nu$Z tool.

## 3 Mathematical modeling

The mathematical modeling of the HW-SW partitioning problem was taken from Arato *et al.* (2003) and Mann *et al.* (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, as shown in Teich (2012).

### *3.2 Formal Model*

A directed simple graph $G = (V, E)$, called the task graph of the system, is given. Where 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).

$P$ is called a hardware-software partition if it is a bipartition of $V : P = (V_h, V_s)$, where $V_h \cup V_s = V$ and $V_h \cap V_s = 0$. The crossing edges are $E_p = \{(V_i, V_j) : V_i \in V_s, V_j \in V_h$ or $V_i \in V_h, V_j \in V_s$, Arato *et al.* (2003). 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)

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

In particular, different optimization (and decision) problems can be defined for partitioning HW-SW, as described by Arato *et al.* (2003). In this paper, however, the focus is on systems with hard real-time constraints: $S_0$ is given (initial cost of software), *i.e.*, the goal is to find a $P$ HW-SW partitioning so that $S_p \leq S_0$ and $H_p$ is minimal, which is thus related. Consequently, the constraints can be reformulated based on Equations (1) and (3) as

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

where $s$ and $c$ are the vectors representing the cost functions, $E$ is the transposed incidence matrix of $G$ (indicating which edges cross the boundary between the contexts of hardware and software), and $x$ represents the decision variable (a binary vector indicating the partition: 1 if the node is realized in hardware and 0 if the node is realized in software). Concerning the complexity of this problem, Arato *et al.* (2003) demonstrate that it is NP-Hard, as defined in Cormen *et al.* (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 $\nu$Z 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, from Trindade and Cordeiro (2015) and Trindade *et al.* (2015). Both use slack variables in order to eliminate the modulus operator of Eq. (4) 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, as shown in 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*

Figure 5 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.

Remembering that, as defined by formal model, there is a index of each decision variable, indicated by letter "i", which ranges from zero to the number of nodes of the problem to be solved.

The ESBMC algorithm starts with the declarations of hardware, software, and communication costs. $S_0$ must also be defined, as the transposed incidence matrix (used in Eq. (4)) and the identity matrix (necessary to work with the matrices), 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 Trindade and Cordeiro (2015).

**Figure 5** Pseudocode describing sequential ESBMC.

```
Initialize variables
Declare number of nodes and edges
Declare the maximum hardware cost (H_max)
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 the initial sw cost of (S_0)
Declare transposed incidence matrix graph G(E)
Define the decision variables (x_i) as Boolean
main {
 For TipH = 0 to H_max do {
  Populate x_i with non-deterministic values
```

```
    Calculate s_i(1 − x_i) + c|Ex_i| and store in variable
    Requirement enforced by assume(variable ≤ S_0)
    Calculate H_p cost based on value of x_i
    Violation check with assert(Hp > TipH)
  }
}
```

It is possible to instruct ESBMC with which type of values the variables must be tested. Therefore, there is a declaration to populate all decision variables $x$ with non-deterministic Boolean values. As a result, the Boolean value that is assigned to each decision variable $x_i$ is actually selected by the SMT Solver, during its solving phase, which checks once all possible combinations to yield a feasible solution, *e.g.*, by handling the terms in the given background theory using a decision procedure Moura and Bjorner (2008); Brummayer and Biere (2009). If this is achieved, the ASSUME directive ensures the compliance of the constraint $A.x \leq b$.

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 ($H_{max}$). 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, there is no need for adding slack variables in Eq. (4), which reduces the number of variables to be solved if compared to ILP and GA.

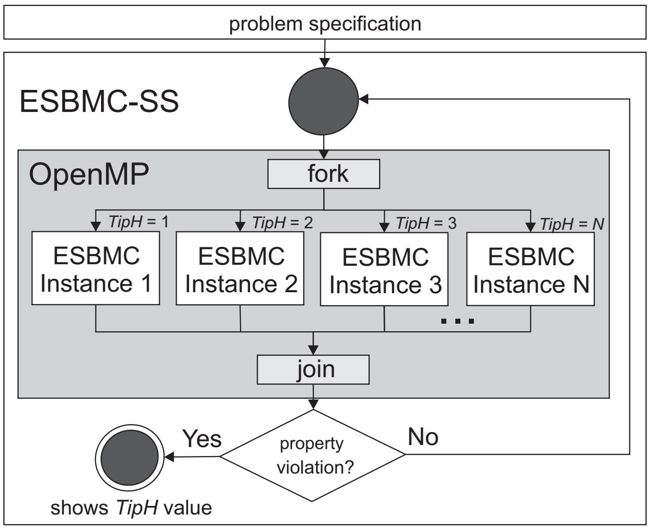
### *4.3 Multi-core ESBMC with OpenMP (ESBMC-SS)*

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, according to Dagum and Menon (1998), is used in this present work as a front-end for ESBMC. Fig. 6 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

**Figure 6** ESBMC-SS approach.



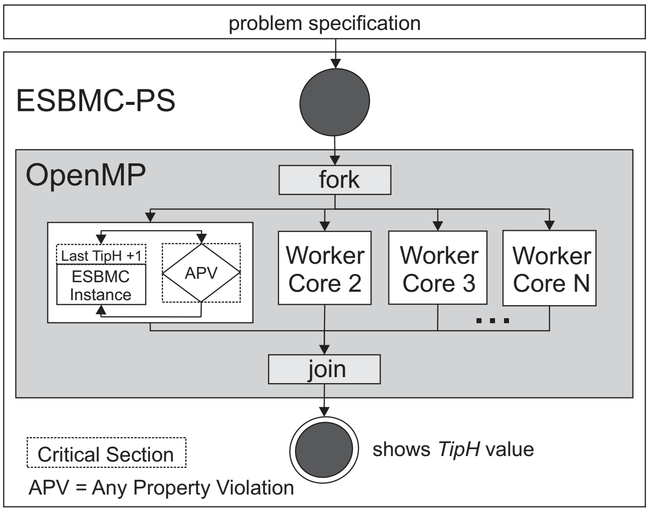

partitioning is violated, when the correct optimum value (*TipH*) parameter is reached; ESBMC-SS starts a parallel region with different instances of ESBMC, based on the number of available processing cores. All these ESBMC instances run independently of each other, as shown in Fig. 6. Note that there 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. 7. 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.

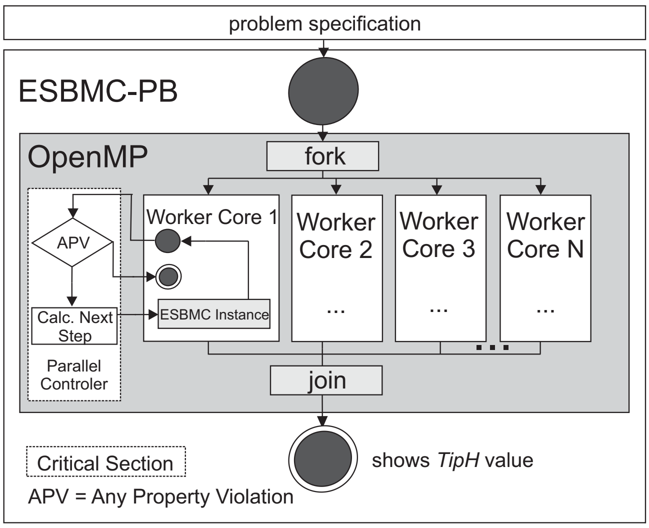
**Figure 7** ESBMC-PS approach.




### *4.5 Multi-core ESBMC with OpenMP using Binary Search (ESBMC-PB)*

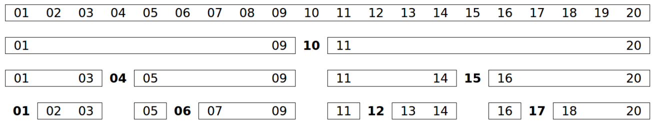
The most optimized approach applies a parallelized binary-search to reduce the amount of steps to be executed in order to find the optimal solution. A controller is designed to return the step to be executed so that the number of verification runs are substantially reduced. The parallelized binary search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

**Figure 8** ESBMC-PB approach.




As an example, given a problem of domain from 1 to 20 (see Fig. 9), 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.

**Figure 9** 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. 9, 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 in Figure 10 describes how the customized binary search calculates and returns the step to be executed.

Note that algorithm of Figure 10 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 18). At the end, the median is returned.

**Figure 10** Steps calculation using intervals.

```
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 +
12    floor((largest.right - largest.left) / 2);
13  if(median > 0){
14   if(largest.right - largest.left > 1)
15    chunks.add(
16      new chunk(largest.left, median - 1)
17    );
18   if(largest.right != largest.left)
19    chunks.add(
20     new chunk(median + 1, largest.right)
21    );
22  }
23  return median;
24 }
```

Algorithm of Figure 11 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.

**Figure 11** Worker sample.

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

### *4.6 Analysis of the partitioning problem using νZ (ESBMC-νZ)*

Algorithm of Figure 12 encodes the objective function and constraints related to the HW-SW partitioning problem using $\nu$Z functions, as seen in Bjorner and Phan (2014). A $\nu$Z 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-$\nu$Z: 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 instructs to ESBMC-$\nu$Z that *Fobj* should be minimized. Finally, the model is checked by ESBMC-$\nu$Z and if there is a solution that meets the constraints, then the *Fobj* value is provided.

**Figure 12** Pseudocode describing ESBMC-$\nu$Z.

```
1  -Initialize Variables
2   Create vZ context
3   Create binary vector (x)
4   Declare number of nodes, edges and S0
5   Declare hardware cost of each node as array (h)
6   Declare software cost of each node as array (s)
7   Declare communication cost of each edge (c)
8   Declare transposed incidence matrix graph G(E)
9  -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 <= S0)
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, ESBMC-SS, ESBMC-PS, ESBMC-PB, and ESBMC-$\nu$Z using a set of standard HW-SW partitioning benchmarks, as shown in Mann *et al.* (2007).

### *5.1 Experimental Setup*

ESBMC v2.0 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, as shown in Brummayer and Biere (2009) (freely available) was used as the default solver for ESBMC. ESBMC-$\nu$Z as a built-in tool to Z3 was also used, as shown in Bjorner and Phan (2014). For ILP and GA formulations, MATLAB R2013a from MathWorks with Parallel Computing Toolbox was used, as shown in MathWorks (2013). MATLAB is a dynamically typed high-level language, known as the state-of-the-art mathematical software, as shown in Tranquillo (2011) and is widely used by the engineering community, as shown in Hong and Cai (2010).

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 91.7% to ESBMC (sequential, SS, PB and $\nu$Z), 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. The TO was defined based on previous empirical tests, where a larger TO (*e.g.*, 5000 seconds) did not produce substantial differences in the experimental results.

### *5.2 Description of Benchmarks*

**Table 1** Description of Benchmarks.

| Name | Nodes | Edges | Description |
|---|---|---|---|
| CRC32 | 25 | 32 | 32-bit cyclic redundancy check, as shown in Guthaus *et al.* (2011) |
| Patricia | 21 | 48 | Routine to insert values in Patricia Tree, as shown in Guthaus *et al.* (2011) |
| Dijkstra | 26 | 69 | Computer shortest paths in a graph, as shown in Guthaus *et al.* (2011) |
| 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 |

To perform the experiments, some benchmarks provided by Mann *et al.* (2007) were used, as shown in Table 1. 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, as shown in Guthaus *et al.* (2011). The clustering and fuzzy benchmarks are designed from Mann *et al.* (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).

### *5.3 Experimental Results*

Table 2 shows the experimental results using Matlab (ILP and GA) and ESBMC (ESBMC-SS, ESBMC-PS, ESBMC-PB, ESBMC-$\nu$Z) tools.

There is no single tool for efficiently solving all HW-SW partitioning benchmarks. In particular, the best (proposed) solution is ESBMC-$\nu$Z, which solves 4 out of 7 benchmarks; ESBMC-$\nu$Z 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-$\nu$Z, 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 $\nu$Z in all benchmarks, which are supported by ESBMC-$\nu$Z. 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 2.5 times slower than ESBMC-$\nu$Z. However, when the amount of nodes is around 30, ESBMC-PB does not outperform ESBMC-$\nu$Z 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 $\nu$Z 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 timeouts for all implementations of ESBMC; GA did not produce the correct answer, and ILP solves correctly most benchmarks, except for Mars and 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 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-$\nu$Z 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 $\nu$Z 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-

**Table 2** 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 |
| | Hp | 17 | 47 | 40 | 245 | 647 | 8619 | - |
| | 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-$\nu$Z** | T(s) | 0.3 | 0.3 | 0.7 | 86.4 | TO | TO | TO |
| | Hp | 15 | 47 | 31 | 241 | - | - | - |

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

Arato, P., Juhasz, S., Mann, A., Orban, A. and Papp, D. (2003) 'Hardware/software partitioning in embedded system design', *International Symposium on Intelligent Signal Processing*, pp.192–202

Mann, A., Orban, A. and Arato, P. (2007) 'Finding optimal hardware/software partitions', *Formal Methods in System Design*, Vol. 31, No. 3, pp.241–263

Trindade, A. and Cordeiro, L. (2015) 'Applying SMT-based verification to hardware/software partitioning in embedded systems', *Design Automation for Embedded Systems*, Vol. 20, No. 1, pp. 1–19

Trindade, A., Ismail, H. and Cordeiro, L. (2015) 'Applying Multi-Core Model Checking to Hardware-Software Partitioning in Embedded Systems (short paper)', *V Brazilian Symposium on Computing Systems Engineering*, pp. 102–105

Bjorner, N., Phan, A. and Fleckenstein, L. (2015) 'vZ - An Optimizing SMT Solver', *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Vol. 9035, pp.194–199

Tang, T., Lin, Y. and Ren, X. (2010) 'Mapping OpenMP concepts to the stream programming model', *Proceedings of the International Conference on Computer Science Education*, pp.1900–1905

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

Cordeiro, L. Fischer, B. and Marques-Silva, J. (2012) 'SMT-based Bounded Model Checking for Embedded ANSI-C Software', *IEEE Transactions on Software Engineering*, Vol. 38, No. 4, pp.957–974

Rao, S. (2009) 'Engineering Optimization: Theory and Practice. 4th edition'

Biere, A., Heule, M., Maaren, H. and Walsh, T. 2009 'Bounded Model Checking', *Handbook of Satisfiability*

Clarke, E., Biere, A., Raimi, R. and Zhu, Y. (2001) 'Bounded model checking using satisfiability solving', *Formal Methods in System Design*, Vol. 19, No. 1, pp.7–34

Armando, A., Mantovani, J. and Platania, L. (2009) 'Bounded model checking of software using SMT solvers instead of SAT solvers', *International Journal on Software Tools for Technology Transfer*, Vol. 11, No. 1, pp.69–83

Ganai, M. K. and Gupta, A. (2006) 'Accelerating high-level bounded model checking', *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp.794–801

Ramalho, M., Freitas, M., Sousa, F., Marques, H., Cordeiro, L. and Fischer, B. (2013) 'SMT-Based Bounded Model Checking of C++ Programs', *International Conference and Workshops on the Engineering of Computer-Based Systems*, pp.147–156

Moura, L. and Bjorner, N. (2008) 'Z3: An Efficient SMT Solver', *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp.337–340

Brummayer, R. and Biere, A. (2009) 'Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays', *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Vol. 5505, pp.174–177

Barrett, C., Conway, C., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A. and Tinelli, C. (2011) 'CVC4', *Proceedings of the International Conference on Computer-Aided Verification*, Vol. 6806, pp.171–177

Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., Rossum, P., Schulz, S. and Sebastiani, R. (2005) 'MathSAT: Tight integration of SAT and mathematical decision procedures', *Journal of Automated Reasoning*, Vol. 35, No. 1-3, pp.265–293

Dutertre, B. (2014) 'Yices 2.2', *Proceedings of the International Conference on Computer-Aided Verification*, Vol. 8559, pp.737–744

Wu M., Wu, W., Tai, N., Zhao, H., Fan, J. and Yuan, N. (2014) 'Research on OpenMP model of the parallel programming technology for homogeneous multicore DSP', *Proceedings of the International Conference on Software Engineering and Service Science*, pp.921–924

Bjorner, N. and Phan, A (2014) 'vZ - Maximal Satisfaction with Z3', *Proceedings of the International Symposium on Symbolic Computation in Software Science*, pp.1–10

Paci, F., Ferrini, R., Sun, Y. and Bertino, E. (2008) 'Authorization and User Failure Resiliency for WS-BPEL business processes', *Proceedings of the International Conference on Service-Oriented Computing*, pp.116–131

Narodytska, N. and Bacchus, F. (2014) 'Maximum Satisfiability Using Core-Guided MAXSAT Resolution', *Proceedings of the Conference on Artificial Intelligence*, pp.2717–2723

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

Manquinho, V. and Marques-Silva, J. (1995) 'Effective Lower Bounding Techniques for Pseudo-Boolean Optimization', *IST/INESC-ID, Technical University of Lisbon, Portugal*

Microsoft Research (2015) 'Z3 API - Source code and documentation' [online] *https://z3.codeplex.com/* (Accessed 18 August 2015)

Cormen, T., Leiserson, C., Rivest, R. and Stein, C. (2009) 'Introduction to Algorithms', *MIT Press*, Vol. I-XIX, pp.1–1292

Dagum, L. and Menon, R. (1998) 'OpenMP: An Industry-Standard API for Shared-Memory Programming', *Proceedings of the International Conference on Computational Science Engineering*, pp.46–55

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

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

Hong, L. and Cai, J. (2010) 'The application guide of mixed programming between MATLAB and other programming languages', *Proceedings of the International Conference on Computer and Automation Engineering*, pp.185–189

Cordeiro, L. and Fischer, B. (2011) 'Verifying multi-threaded software using SMT-based context-bounded model checking', *Proceedings of the International Conference on Software Engineering*, pp.331–340

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

Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T. and Brown, R. (2011) 'MiBench: A free, commercially representative embedded benchmark suite', *Proceedings of the annual workshop on workload characterization*, pp.3–14

Arato, P., Mann, A., Orban, A. (2005) 'Algorithmic aspects of hardware/software partitioning', *ACM Transactions on Design Automation of Electronic Systems*, Vol. 1, No. 1, pp.136–156

Holzmann, G., Joshi, R. and Groce, A. (2011) 'Swarm Verification Techniques', *IEEE Transactions on Software Engineering*, Vol. 37, No. 6, pp.845–857

Wintersteiger, C., Hamadi, Y. and Moura, L. (2009) 'A Concurrent Portfolio Approach to SMT Solving', *Proceedings of the International Conference on Computer-Aided Verification*, Vol. 5643, pp.715–720

Qawasmeh, A., Malik, A. and Chapman, B. (2014) 'OpenMP task scheduling analysis via OpenMP runtime API and tool visualization', *Proceedings of the International Symposium on Parallel Distributed processing*, pp.1049–1058

Pavlinovi, Z., King, T. and Wies, T. (2015) 'Practical SMT-Based Type Error Localization', *Proceedings of the International Conference on Functional Programming*, pp.412–423

Sebastiani, R. and Trentin, P. (2015) 'OptiMathSAT: A Tool for Optimization Modulo Theories', *Proceedings of the International Conference on Computer Aided Verification*, Vol. 9206, pp.447–454

Holzmann, G., Joshi, R. and Groce, A. (2011) 'Swarm Verification Techniques', *IEEE Transactions Software Engineering*, Vol. 9206, pp.845–857

Holzmann, G. (2012) 'Parallelizing the spin model checker', *Proceedings of the International Symposium on Model Checking Software*, Vol. 7385, pp.155–171