Multi-core model checking and maximum satisfiability applied to hardware-software partitioning

DOI:
10.1504/IJES.2017.088044

Document Version
Final published version

Link to publication record in Manchester Research Explorer

Citation for published version (APA):

Published in:
International Journal of Embedded Systems

Citing this paper
Please note that where the full-text provided on Manchester Research Explorer is the Author Accepted Manuscript or Proof version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version.

General rights
Copyright and moral rights for the publications made accessible in the Research Explorer are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Takedown policy
If you believe that this document breaches copyright please refer to the University of Manchester’s Takedown Procedures [http://man.ac.uk/04Y6Bo] or contact uml.scholarlycommunications@manchester.ac.uk providing relevant details, so we can investigate your claim.
Multi-Core Model Checking and Maximum Satisfiability Applied to Hardware-Software Partitioning

Alessandro Bezerra Trindade*, Renato de Faria Degelo, Edilson Galvão dos Santos Junior, Hussama Ibrahim Ismail, Helder Cruz da Silva and Lucas Carvalho Cordeiro

Federal University of Amazonas
Av. Rodrigo Otávio, 3000, Campus Universitário, Manaus, Brazil, ZIP 69077-000
E-mail: alessandro.b.trindade@gmail.com*
E-mail: rdegelo@gmail.com
E-mail: esj.galvao@gmail.com
E-mail: hussamaismail@gmail.com
E-mail: helder@ufam.edu.br
E-mail: lucascordeiro@ufam.edu.br
* Corresponding author

Abstract: Bounded Model Checking (BMC) based on Satisfiability Modulo Theories (SMT) is well known by its capability to verify software. However, its use as optimization tool, to solve hardware and software (HW-SW) partitioning problem, is something new. In particular, its integration with the Maximum Satisfiability solver $\nu Z$ tool, which provides a portfolio of approaches for solving linear optimization problems over SMT formulas, is unprecedented. We present new alternative approaches to solve the HW-SW partitioning problem. First, we use SMT-based BMC in conjunction with a multi-core support using Open Multi-Processing to create four variants to solve the partitioning problem. The multi-core SMT-based BMC approaches allow initializing many verification instances based on the number of available processing cores, where each instance checks a different optimum value until the optimization problem is satisfied. Additionally, we integrate the $\nu Z$ into the BMC, making it as a specialized solution for optimization in a single-core environment. We implement all five approaches on top of the Efficient SMT-Based Context-Bounded Model Checker (ESBMC) and compare them to a state-of-the-art optimization tool. Experimental results show that there is no single optimization tool to solve all HW-SW partitioning benchmarks, but based on medium-size benchmarks, ESBMC-$\nu Z$ had better performance.

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


Biographical notes: Alessandro Bezerra Trindade received the B.Sc. and M.Sc. degrees in electrical engineering from the Federal University of Amazonas (UFAM), in 1995 and 2015, respectively. Currently, he is pursuing his PhD in informatics at UFAM.

Renato de Faria Degelo, received his software analysis and development technologist degree in 2011 from Uninorte and is pursuing his M.Sc. at UFAM. Currently, he is a software engineer at Samsung Institute (SIDIA).

Edilson Galvo dos Santos Junior, received his computer engineering degree in 2015 from Foundation Center for Analysis, Research and Technological Innovation (FUCAPI) and is pursuing his M.Sc. at UFAM. Currently, he is a software engineer at SIDIA.

Hussama Ibrahim Ismail holds a B.Sc. degree in computer engineering from FUCAPI in 2013 and a M.Sc. degree in Electrical Engineering from UFAM in 2015. His current research interests are formal methods and embedded systems.

Helder Cruz da Silva received the B.Sc. degree and the M.Sc. from the Federal University of Uberlândia (UFU), in 1998 and 2001, respectively. He received the Ph.D. degree from University of Minho in 2013. Currently, he is an assistant professor at UFAM.

Lucas Carvalho Cordeiro received the Ph.D. degree from the University of Southampton in 2011. From 2009 to 2016, he was an adjunct professor at UFAM (he is currently in unpaid leave). Since 2016, he is a researcher at the University of Oxford.


1 Introduction

Nowadays, with the tight development time 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 might affect the overall development process, the quality, as well as the final system life-cycle.

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

There is also the effort to create a hardware-software (HW/SW) co-simulation environment in order to help designers obtain an appropriate HW/SW partitioning that satisfy specified tradeoffs, as described by Jan et al. (2005) and by Hau and Khalil-Hani (2009).

In terms of SMT-based verification, most related studies are restricted to present the model, its modification to specific 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) use 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 mentioned in Holzmann et al. (2011), and modifications of internal search engines to add support for parallelism, as presented by Holzmann (2012), but there is still the need for initiatives related to parallel SMT solving, as described by Wintersteiger et al. (2009).

Recently, the SMT solver Z3 has been extended to pose and solve optimization problems modulo theories, as presented by Björner et al. (2015). In particular, νZ tool offers substantial performance improvement in optimization problems, as described by Björner and Phan (2014). Additionally, 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 νZ to compute an optimal error source in a given ill-typed program.

1.1 Contributions

Here, 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). Our experimental results are compared to ILP (integer linear programming) and GA (generic algorithms) in a multi-core version, and also to νZ, which supports only a single-core approach, as described by Björner et al. (2015). The ILP and GA algorithms are implemented with the optimization toolbox of Matlab, as described in 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. 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.

1.2 Availability of Data and Tools

Our experiments are based on a set of publicly available benchmarks. All tools, benchmarks, and results of our evaluation are available on a supplementary web page http://esbmc.org and http://esbmc.org/benchmarks.

1.3 Organization of this Work

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 4.4 presents the partitioning model using νZ. In Section 5, we show the experimental results using several embedded systems applications. We conclude and describe future work in Section 6.

2 Preliminaries

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, as described by Rao (2009). The most well-known technique is linear programming, which is a 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 $A_{eq}$ and $b_{eq}$ are matrices that describe the linear equality constraints.

$$\min_{x \in \mathbb{R}^n} f(x) \text{ such that } \begin{cases} Ax \leq b, \\ A_{eq} x = b_{eq}, \\ x \geq 0. \end{cases}$$

(1)

Here, we refer to the relevant work of Mann et al. (2007) in optimization, who modified a branch-and-bound algorithm to speed up the execution time. Sapienza et al. (2013) used multiple criteria decision analysis to solve the HW-SW partitioning problem.

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, as described by 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. As relevant studies, Jiang et al. (2010) used a GA mixed with simulated annealing and Huang and Binh (2012) designed a modified Pareto optimization using GA. In the other hand, there is some research done to reduce the complexity of computation in order to improve the performance of some techniques, as described by Boucheneb and Hadjijd (2006).

Alternatively, tools such as ESBMC and vZ can be used to solve optimization problems so that the global minimum or maximum solution is found, as described by Trindade and Cordeiro (2015) and Bjorner et al. (2015). The following sections describe the main features of ESBMC and vZ tools.

2.2 Bounded Model Checking with ESBMC

Among 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 specific types of problems, it offers large performance improvements over previous approaches, as presented by 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-space explosion problem, as described by 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 presented by Ganai and Gupta (2006), Armando et al. (2009), Cordeiro et al. (2012).

In this study, however, ESBMC has been used as a BMC tool to solve HW-SW partitioning problems, as mentioned 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 ESBMC architecture, which consists of the C/C++ parser, GOTO Program, GOTO Symex, and SMT solver, as described by Cordeiro et al. (2012). 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., $C \land \neg P$), 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 presented by Moura and Bjorner (2008); Boolector, as mentioned in Brummayer and Biere (2009); MathSAT, as presented by Barrett et al. (2011); CVC4, as presented by Bovozano et al. (2005); and Yices, as presented by Dutertre (2014). Most of them do provide neither multi-threaded support nor a parallel version to solve the generated SMT equations.

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 mentioned 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 presented by 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:

```c
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 νZ

In this study, the SMT solver Z3 is used to check for the satisfiability of formulas generated from the HW-SW partitioning problem, as presented by Bjorner and Phan (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; the νZ main 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), as mentioned 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 νZ, MaxSAT is responsible for the restrictions, while OptSMT optimizes linear arithmetic objectives, as presented by Bjorner et al. (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 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. Indeed, the model generated by νZ shows that $K = 1$ and $W = 1$.

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

Fig. 4 shows the νZ architecture. Initially, the SMT formula with objectives is converted to 0-1 constraints, which leads to a Pseudo-Boolean Optimization (PBO), as mentioned in Barth and Putnam (1995) and Manquinho and Marques-Silva (1995). 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.
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 system’s components 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 execution time of the component. Third, the hardware implementation of a component has a hardware cost, which can be area, heat dissipation, or energy consumption; the decision is typically made by the engineer or from the project requirement. Fourth, based on the assumption that hardware is significantly faster than software, the execution time of the components in hardware is considered to be zero. Finally, if two components are mapped to the same context, then there is no communication overhead between them; otherwise, there is an overhead. The consequence of these assumptions is that scheduling does not need to be addressed in this study. Hardware components do not need scheduling, because the execution time is assumed to be zero. Since 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, with a simplified HW-SW model, where the focus is on bipartitioning, as presented by 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 $E_p = \{(V_i, V_j) : V_i \in V_h, V_j \in V_s \text{ or } V_i \in V_h, V_j \in V_s\},$ Arato et al. (2003). The hardware cost of $P$ is given in Eq. (2)

$$H_p = \Sigma_{V_i \in V_h} h_i,$$

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

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

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 objective is to find a $P$ HW-SW partitioning so that $S_p \leq S_0$ and $H_p$ is minimal, which is thus related. Therefore, the objective function is:

$$\text{minimize } h_x.$$ (4)

Consequently, the constraints can be reformulated based on Equations (1) and (3) as:

$$s \left(1 - x \right) + c |Ex| \leq S_0,$$ (5)

$$x \in \{0, 1\}^n,$$ (6)

where $h$, $s$ 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), $n$ is the number of nodes, 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-core, and more recently to heterogeneous computing as described by Mittal et al. (2015), 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. (5) and consequently to use commercial tools. However, GA had improvements from
the parameters of related studies 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 mentioned 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. All ILP and GA parameters were presented and discussed in Trindade and Cordeiro (2015) and Trindade et al. (2015). In particular, the ILP and GA algorithms were chosen since they were also used in another related work, as Arato et al. (2003) and Mann et al. (2007), thus facilitating the comparison to the present techniques (that we propose in this paper).

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.

Note that, as defined by the formal model, there is an index for each decision variable, indicated by the letter “i”, which ranges from 0 to the number of nodes of the (particular) problem to be solved.

The ESBMC algorithm starts with the declarations of hardware, software, and communication costs. S0 must also be defined, as the transposed incidence matrix (used in Eq. (5)) 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 ILP and GA presented in Trindade and Cordeiro (2015).

Figure 5 Pseudocode describing sequential ESBMC.

```
1 Initialize variables
2 Declare number of nodes and edges
3 Declare the maximum hardware cost (Hmax)
4 Declare hardware cost of each node as array (h)
5 Declare software cost of each node as array (s)
6 Declare communication cost of each edge (c)
7 Declare the initial sw cost of (S0)
8 Declare transposed incidence matrix graph G(E)
9 Define the decision variables (xi) as Boolean
10 main {
11   For TipH = 0 to Hmax do {
12       Populate xi with non-deterministic values
13       Calculate s_i(1 - x_i) + c_i|Ey| and store in variable
14       Requirement enforced by assume(variable ≤ S0)
15       Calculate H_p cost based on value of x_i
16       Violation check with assert(H_p > TipH)
17   }
18 }
```

It is possible to instruct ESBMC with which type of values the variables must be evaluated. 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, as described by Moura and Bjorner (2008); Brummayer and Biere (2009). If this is achieved, then the ASSUME directive ensures the compliance of the constraint A.x ≤ 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 (which is represented by Hmax). To every performed test, 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 concluded, 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 occurs, the verification is concluded 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. (5), which reduces the number of variables to be solved if compared to ILP and GA.

4.3 Multi-core ESBMC

4.3.1 Multi-core ESBMC with OpenMP (ESBMC-SS)

Typically, ESBMC verification runs are performed only in a single-core. If the processor provides n 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, as described in 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 an ANSI-C program. The HW-SW partitioning is violated, when the correct optimum value parameter (represented by TipH ∈ N) 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 managing 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.

4.3.2 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 time period.

In contrast to ESBMC-SS, where a new block of threads will be executed only if all threads in the previous execution were concluded, ESBMC-PS aims at removing 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. Note that there is no communication among workers, but each worker notifies the controller if a violation is found for its $TipH$ value.

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

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

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

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 the algorithm 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. Furthermore, this algorithm is synchronized, which ensures a unique step value for a worker request. 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.

```
GetNextStep() {
    int largestChunk = -1;
    chunk largest;
    for each(chunk in chunks) {
        if (chunk.right - chunk.left > largestChunk) {
            largestChunk = chunk.right - chunk.left;
            largest = chunk;
        }
    }
    chunks.remove(largest);
    int median = largest.left +
        floor((largest.right - largest.left) / 2);
    if (median > 0) {
        if (largest.right > median + 1) {
            new chunk(largest.left, median + 1)
        } else {
            new chunk(largest + 1, largest.right)
        }
    }
    return median;
}
```

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); otherwise, the ESBMC instance is killed (line 5) and the worker is free to initiate another step. Note that there is no interchange communication among workers, but they are responsible for notifying the controller if a given violation is found.

Figure 11 Worker sample.

```
step = controller.GetNextStep();
int pid = ExecuteStep(step);
while (!controller.isNeeded(step)) {
    if (control.isRunning(pid)) {
        kill(pid);
    }
}
```

4.3.4 Time Complexity of ESBMC-SS, ESBMC-PS, and ESBMC-PB

With respect to the time complexity of ESBMC-SS, ESBMC-PS, and ESBMC-PB algorithms, they can all be described in two parts, which include the parallelism and the optimization solving. In the first part, the ESBMC-SS, ESBMC-PS, and ESBMC-PB time complexity is considered to be linear (i.e., they are denoted by $O(n)$), taking into account a sequential time since each algorithm runs all possible solutions at once. However, each execution instance of ESBMC-SS, ESBMC-PS, and ESBMC-PB solves a specific optimization problem that is considered to be NP-Hard, as described by Arato et al. (2003). Thus, even a parallel execution being implemented, including (possible) overheads due to the use of the OpenMP library, the time complexity of ESBMC-SS, ESBMC-PS, and ESBMC-PB is still NP-Hard.

4.4 Analysis of the partitioning problem using $\nu Z$ (ESBMC-$\nu Z$)

Algorithm of Figure 12 encodes the objective function and constraints related to the HW-SW partitioning problem using $\nu Z$ functions, as described in Björner 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 constraints set. 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. (5). Here, variable $SC$ refers to the software cost, while $CMC$ 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, while 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 \( F_{obj} \) value is provided.

**Figure 12** Pseudocode describing ESBMC-\( \nu \)Z.

1. **Initialize Variables**
2. Create \( \nu \) context
3. Create binary vector (\( x \))
4. Declare number of nodes, edges and \( S_0 \)
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**
   - \( SC = s(1 - x) \)
   - \( CMC = c \cdot \| EX \| \)
   - \( F_{obj} = x[i] \cdot h[i] \)
10. **Assert Constraints**
    - Add constraints (\( SF + CMC \leq S_0 \))
    - Add constraints to minimize \( F_{obj} \)
11. Check Model
12. **Print Result**

In general, it is worth mentioning that the complexity for ESBMC-SS, ESBMC-PS, and ESBMC-PB in a sequential time are linear (i.e., \( O(n) \)), that is because each algorithm runs all the possible solutions once. However, each execution runs an optimization problem, which is solved by the model checker, that is NP-Hard as described by Arato et al. (2003). Thus, even a parallel time being considered, including possible overheads in OpenMP, the general complexity is simplified by NP-Hard.

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, ESBMC-SS, ESBMC-PS, ESBMC-PB, and ESBMC-\( \nu \)Z using a set of standard HW-SW partitioning benchmarks, as presented by 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 described 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 presented in Björner and Phan (2014). For ILP and GA formulations, MATLAB R2013a from MathWorks with Parallel Computing Toolbox was used, as described in MathWorks (2013). MATLAB is a dynamically typed high-level language, known as the state-of-the-art mathematical software, as described in Tranquillo (2011) and is widely used by the engineering community, as described in Hong and Cai (2010).

Note that the experimental results presented here depend on the computer’s processor and memory, the version of each tool (MATLAB and ESBMC), and the partitioning problem to be solved. Additionally, any change from MATLAB to another tool or to GA/ILP libraries for different programming languages can influence the measured time of the algorithms.

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). Empirical tests performed by the authors demonstrated that a higher number of measurements, for each technique, did not produce significant differences in the experimental results (which were always below 10% and mostly around 3%).

Based on the mean, standard deviation and tolerance interval to each set of time sample, it was obtained a confidence level 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 as well, 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Nodes</th>
<th>Edges</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC32</td>
<td>25</td>
<td>32</td>
<td>32-bit cyclic redundancy check, as presented by Guthaus et al. (2011)</td>
</tr>
<tr>
<td>Patricia</td>
<td>21</td>
<td>48</td>
<td>Routine to insert values in Patricia Tree, as presented by Guthaus et al. (2011)</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>26</td>
<td>69</td>
<td>Computer shortest paths in a graph, as presented by Guthaus et al. (2011)</td>
</tr>
<tr>
<td>Clustering</td>
<td>150</td>
<td>331</td>
<td>Image segmentation algorithm in a medical application</td>
</tr>
<tr>
<td>RC6</td>
<td>329</td>
<td>448</td>
<td>RC6 cryptography graph algorithm</td>
</tr>
<tr>
<td>Fuzzy</td>
<td>261</td>
<td>422</td>
<td>Clustering algorithm based on fuzzy logic</td>
</tr>
<tr>
<td>Mars</td>
<td>417</td>
<td>600</td>
<td>MARS cipher from IBM algorithm</td>
</tr>
</tbody>
</table>

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, i.e., software execution time and communication time; and hardware costs represent...
the occupied area. The first three benchmarks are extracted from MiBench, as presented by Guthaus et al. (2011). The fuzzy benchmark was created and presented by Mann et al. (2007) and is a significantly large benchmark. And also from Mann et al. (2007), were used RC6 and Mars as very complex benchmarks to test the limits of each technique. Note that we used the same number (and types) of benchmarks that were previously used in the literature, in order to enable a fair comparison to other existing techniques presented in this study.

5.3 Experimental Results

Table 2 shows the experimental results using Matlab (ILP and GA) and ESBMC (ESBMC-SS, ESBMC-PS, ESBMC-PB, ESBMC-νZ) tools. The number of nodes, edges, and the initial software cost (S0) are given to each benchmark. Hp and Sp represent the results of partitioning process for each technique, i.e., the optimized value of partitioned hardware and software. T(s) means the average time taken from the experiments. Particularly about GA there is an additional row informing how far from the exact answer is the found solution.

As observed in the experimental evaluation, 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; 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 ESBMC-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 2.5 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.

Regarding the amount of MOs, the sequential ESBMC approach has to explore all (possible) states until it finds the HW-SW partitioning solution; it starts from an extreme, where all variables are selected as software, and then incrementally tests one by one to check whether a given node will be implemented in software or in hardware. In contrast, all multi-core ESBMC approaches and ESBMC-νZ, are (heuristically) optimized to reach faster the HW/SW partitioning solution than the sequential one, without the need for exploring all states as the sequential ESBMC approach does. As a result, if the HW/SW partitioning problem grows in complexity, then the sequential ESBMC approach tends to easily reach MO due to the state-space explosion problem.

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; although more benchmarks with similar complexity to clustering should be included in future work for a more precise conclusion. 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 RC6 benchmark with 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 limit of nodes, the best solution turns out to be ESBMC-νZ since its execution time is 4.3 to 7.5 times faster than ILP, faster than any other ESBMC approach (until 462 times faster) and its result is precise (when compared to GA). 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. ESBMC and νZ have BSD-Style and MIT licenses, respectively and can be downloaded and used for free. Similarly, it is also possible to use free GA and ILP libraries (BSD-Style and MIT licenses) for different programing languages.

Experimental results also pointed to an improvement of ESBMC, when using a parallel approach. The fastest ESBMC approach is ESBMC-PB, which produces good results for an intermediate amount of edges and nodes.
Table 2  Experimental results of the HW-SW partitioning benchmarks.

<table>
<thead>
<tr>
<th></th>
<th>CRC32</th>
<th>Patricia</th>
<th>Dijkstra</th>
<th>Clustering</th>
<th>RC6</th>
<th>Fuzzy</th>
<th>Mars</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Nodes</strong></td>
<td>25</td>
<td>21</td>
<td>26</td>
<td>150</td>
<td>329</td>
<td>261</td>
<td>417</td>
</tr>
<tr>
<td><strong>Edges</strong></td>
<td>32</td>
<td>48</td>
<td>69</td>
<td>331</td>
<td>448</td>
<td>442</td>
<td>600</td>
</tr>
<tr>
<td><strong>S0</strong></td>
<td>20</td>
<td>10</td>
<td>20</td>
<td>50</td>
<td>600</td>
<td>4578</td>
<td>300</td>
</tr>
<tr>
<td><strong>Exact Solution</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hp</td>
<td>15</td>
<td>47</td>
<td>31</td>
<td>241</td>
<td>692</td>
<td>13820</td>
<td>876</td>
</tr>
<tr>
<td>Sp</td>
<td>19</td>
<td>4</td>
<td>46</td>
<td>533</td>
<td>4231</td>
<td>297</td>
<td></td>
</tr>
<tr>
<td><strong>ILP</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T(s)</td>
<td>1.6</td>
<td>1.3</td>
<td>1.6</td>
<td>648.9</td>
<td>1806.2</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>Hp</td>
<td>15</td>
<td>47</td>
<td>31</td>
<td>241</td>
<td>692</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>GA</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T(s)</td>
<td>6.7</td>
<td>7.4</td>
<td>8.8</td>
<td>340.4</td>
<td>2050.0</td>
<td>1371.9</td>
<td>TO</td>
</tr>
<tr>
<td>Hp</td>
<td>17</td>
<td>47</td>
<td>40</td>
<td>245</td>
<td>647</td>
<td>8619</td>
<td>-</td>
</tr>
<tr>
<td>Error %</td>
<td>13.3</td>
<td>0.0</td>
<td>29.0</td>
<td>1.7</td>
<td>-6.5</td>
<td>-37.6</td>
<td>-</td>
</tr>
<tr>
<td><strong>ESBMC</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T(s)</td>
<td>30.3</td>
<td>313.7</td>
<td>324.7</td>
<td>MO</td>
<td>MO</td>
<td>MO</td>
<td>MO</td>
</tr>
<tr>
<td>Hp</td>
<td>15</td>
<td>47</td>
<td>31</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>ESBMC-SS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T(s)</td>
<td>2.2</td>
<td>5.8</td>
<td>7.0</td>
<td>1609.3</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>Hp</td>
<td>15</td>
<td>47</td>
<td>31</td>
<td>241</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>ESBMC-PS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T(s)</td>
<td>3.7</td>
<td>10.0</td>
<td>12.0</td>
<td>2468.0</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>Hp</td>
<td>15</td>
<td>47</td>
<td>31</td>
<td>241</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>ESBMC-PB</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T(s)</td>
<td>4.3</td>
<td>4.7</td>
<td>6.3</td>
<td>218.7</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>Hp</td>
<td>15</td>
<td>47</td>
<td>38</td>
<td>241</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>ESBMC-νZ</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T(s)</td>
<td>0.3</td>
<td>0.3</td>
<td>0.7</td>
<td>86.4</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>Hp</td>
<td>15</td>
<td>47</td>
<td>31</td>
<td>241</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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. And so that the limits of the techniques can be defined more precisely, new benchmarks will be included in future work, with their complexity ranging between 100 and 400 nodes. Furthermore, more complex types
of architectures of HW-SW model will be addressed, including more than one CPU, and the assumption of just singled-threaded program execution will be extended to multiprogramming and multiprocessing (second generation of co-design).

References


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.


