SYSTEM-LEVEL VALIDATION OF MULTICORE ARCHITECTURES

By

XIAOKE QIN

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2012

© 2012 Xiaoke Qin

I dedicate this to my family.

ACKNOWLEDGMENTS

First of all, I truly appreciate the effort of my Ph.D. adviser Prof. Prabhat Mishra. He not only guided me to overcome challenging problems, but also taught me how to explore new directions. More importantly, he is always considerate to me and has helped me building my career. He is the person who made this dissertation come true.

I would like to thank my other Ph.D. committee members: Prof. Sartaj Sahni, Prof. Jih-Kwon Peir, Prof. Greg Stitt and Prof. Ann Gordon-Ross for their valuable comments and suggestions. I also thank my lab-mates, Mingsong Chen, Kanad Basu, Weixun Wang, Chetan Murthy, Kartik Shrivastava, Hadi Hajimiri and Kamran Rahmani. It was my great pleasure to work with them. I really enjoyed our friendship and I hope it will last forever.

Last but not least, I sincerely thank my family for their love and support. They encouraged me to pursue my dreams and become a good person. I would like to give the most special thanks to my girlfriend, Jie. Her love and devotion paved the road to my doctoral degree.

This work was partially supported by grants from National Science Foundation (NSF) CAREER Award 0746261.

TABLE OF CONTENTS

page

CHAPTER

LIST OF TABLES

LIST OF FIGURES

Abstract of dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

SYSTEM-LEVEL VALIDATION OF MULTICORE ARCHITECTURES

By

Xiaoke Qin

May 2012

Chair: Prabhat Mishra
Major: Computer Engineering

Multicore processors are widely used in today's servers, desktop and embedded systems. It is a major challenge to verify functional correctness as well as non-functional requirements of multicore architectures. Direct application of existing functional validation approaches usually consumes too much time to reach the coverage goal due to the complexity of multicore designs. Escaped bugs can lead to serious consequences in many scenarios. Due to parallel execution of task sets, existing approaches are also insufficient to validate whether applications in such systems can be scheduled within the given temperature, energy, and timing constraints. If these constraints are violated, it can lead to performance degradation or even catastrophic consequences in safety-critical systems. This dissertation presents novel techniques to address validation challenges of both functional and non-functional requirements in modern multicore architectures. My research has made four major contributions: i) it proposes efficient directed test generation techniques that exploit symmetry in multicore designs; ii) it proposes a novel test generation approach for state- and transition- coverage in a wide variety of cache coherence protocols; iii) it proposes a scalable directed test generation technique based on interleaved concrete and symbolic execution; and iv) it proposes schedulability validation approaches for task sets in multicore architectures under temperature and energy constraints. Extensive experimental results demonstrate significant improvement in overall validation effort.

CHAPTER 1
INTRODUCTION

Multicore architectures are widely used in todays desktop, server, and embedded systems. Due to the existence of power wall, conventional single core architectures can no longer deliver the required performance improvement by increasing frequency. Instead, architects integrate more and more cores into the same chip to boost the throughput. By operating multiple cores at a lower frequency, multicore architectures can achieve the same performance with significantly less power consumption compared with a high clock rate monolithic core. For desktop-based systems and servers, the multicore architectures deliver the required throughput keeping pace with today's applications with increasing computation complexity. Due to successful deployment of dual-core and quad-core processors, the next generation processors will have 32, 64 or even hundreds of cores. For embedded systems, the energy efficiency of multicore architectures allows devices to operate for longer time with the same battery capacity. Besides, since multiple cores are sharing the same die, the Printed Circuit Board (PCB) size is also reduced. With the growing demand for green data-centers, long-life computers and handhold devices, multicore architectures will continue to dominate the design of next generation System-on-Chip (SoC) architectures.

Successful multicore designs must satisfy both functional and non-functional requirements. Functional requirements ensure that the processor performs all logical functions as specified by the design specification. Non-functional requirements are imposed to make the design satisfy various design constraints such as area, power, energy, temperature, and performance. Clearly, functional requirements are important, because a buggy (erroneous) design leads to unreliable systems. Depending on application domains, unreliable systems can cause loss of vital information or even disaster. Non-functional requirements are also equally important, because violation of non-functional requirements can also lead to serious consequences. For example, due to uneven activities on different cores, the die temperature of busy cores can easily reach $120°C$ [16]. If the high die temperature is not well controlled, the transient error occurs more frequently and the device is less reliable. Also, devices that always operate in high temperature usually have much shorter lifespan as shown in industrial studies [82]. To avoid these unwanted scenarios, both functional and non-functional validation must be performed to ensure the success of modern multicore designs.

The rest of this chapter is organized as follows. Section 1.1 and Section 1.2 describe existing validation techniques and associated challenges for validation of functional and non-functional requirements, respectively. Section 1.3 summarizes the contribution of this dissertation. Finally, Section 1.4 outlines the organization of this dissertation.

1.1 Functional Validation of Multicore Architectures

While multicore architectures are very successful to boost the throughput, their increasing complexity also introduces significant validation challenges. Most widely used functional validation techniques are based on simulation using random and constrained-random tests [93] [1] [83]. The multicore design is placed within a simulation environment and a test generator feeds random tests into the design. The behavior of the design under test is compared with the golden reference model to detect any functional errors.

As illustrated in Figure 1-1 [77], the verification complexity has grown tremendously in last two decades. For example, in 2007 a typical SoC design (with 100 million gates) used one trillion test vectors for simulation. Due to the increasing complexity of multicore architectures, even trillions of simulation vectors may not be inadequate to achieve the required coverage goal within ever decreasing time-to-market window. Since simulation vectors are generated randomly, it is quite difficult for random tests to activate coverage holes. Directed tests [22] are promising to address this problem. By analyzing the logical structure of the design, a small number of directed tests can activate the

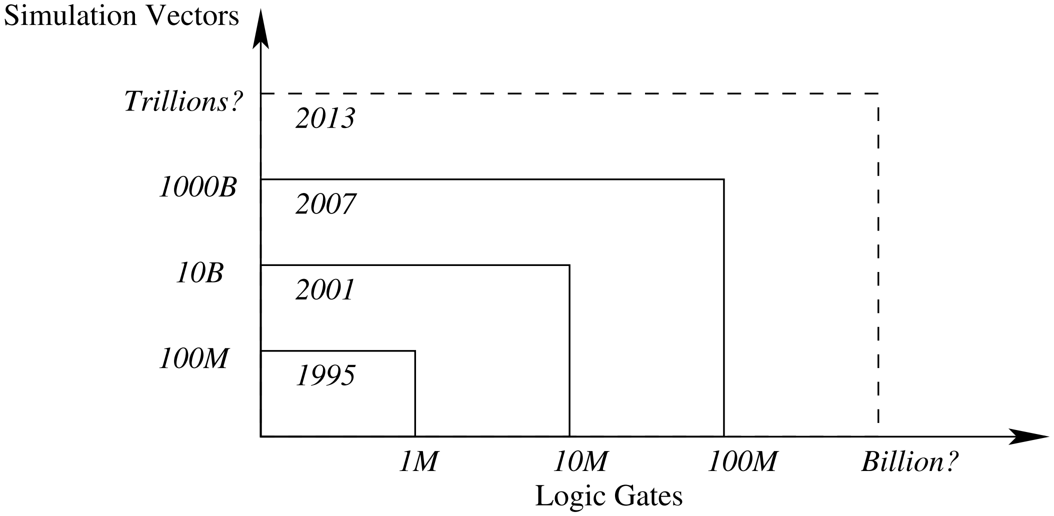

Figure 1-1. Simulation effort growth with design complexity

desired behavior of the system. They can be applied in addition to the random tests to reach the coverage goal with much less time. Unfortunately, most directed tests are manually written, which is time consuming and error-prone. Fully automatic directed test generation schemes are desired to accelerate the verification process of multicore architectures. There are two major objectives in directed test generation. First, the overall validation effort should be minimized by reducing the total number of tests required to achieve the coverage goal. Secondly, test generation time should also be small.

Model checking [13, 28] is promising for automated generation of directed tests. To activate a particular scenario, we can feed the negated version of a property to the model checker, and use the resultant counterexample as a directed test. Due to the state space explosion problem, such a process is usually very time consuming. Since different cores in a multicore design usually contain similar structure, their formal descriptions (such as CNF in SAT-based model checking) also exhibit significant symmetry. We believe such symmetry can be exploited to accelerate the model checking process, because the information we learn from one core may be applied directly to other cores. Unfortunately, this intuitive reasoning is hard to implement because it is very difficult to reconstruct the symmetry from the CNF formula. The high level information is lost during CNF synthesis, and it is inefficient as well as computationally expensive to recover through “reverse engineering” methods.

An important requirement of functional validation is to achieve certain state or transition coverage of the state space of the design. Simulation using random tests is widely used in industry to fulfill this goal. However, due to the symmetric nature of multicore architectures, its state space contains some unique features, which can be utilized to reduce the test length or testing time required to reach the required coverage goal. Although the FSM of each cache controller is easy to understand, the structure of the product FSM for modern cache coherence protocols usually have obscure structures that are hard to analyze. Besides, modern processors usually contain multiple cache levels, which greatly complicates the global state space. Even if the global state space can be described, it is still difficult to find an efficient way to perform traversal in it. In other words, the test generation algorithm must activate all states and transitions with limited number of unnecessary transitions. Moreover, since the state space is very large, the tests usually introduce a large storage overhead. Therefore, it is desirable that the test can be generated on the fly.

1.2 Validation of Non-functional Requirements

So far we have described the importance of ensuring functional correctness and challenges associated with verifying multicore architectures. It is also equally important to ensure that all the non-functional requirements are met. One of the key challenges is to find whether a given task set can be scheduled on the processor(s) without violating the required temperature and energy constraints. This kind of validation is important to ensure the reliability of multicore designs, because high die temperature leads to more frequent transient errors as well as shorter processor lifespan [82]. Besides, the management of overall energy consumption is also crucial to the success of embedded systems. Since many handheld devices are equipped with multicore processors but still battery-powered, we need to validate that all important tasks are finished with limited energy consumption.

It is usually very costly to perform such validation, because the manufacturer need to build the full system and test the design by executing real task sets. Detection of failures at this stage is expensive, since it will lead to re-design of the system. Since the worst case behavior of real-time systems usually can be obtained by offline analysis, we believe it is possible to predict the system behavior based on the information collected via static analysis of task sets and execution environment. In other words, in various cases, non-functional validation can be performed without running the actual system in real environments. The major challenge in this field comes from the NP-hard nature [103] [100] [86] of the schedulability problem. In fact, it is NP-hard even to verify the schedulability of a task set under temperature and energy constraints in a single core processor. The problem is more complex when the system contains multiple cores.

1.3 Research Contributions

My research proposes novel techniques to address challenges in both functional and non-function validation of multicore systems. The objective of my research is to develop efficient test generation approaches and validation algorithms for modern multicore architectures.

Figure 1-2 presents the scope of this dissertation. The proposed research develops efficient validation techniques to address different functional and non-functional requirements using a wide variety of design models including system-level models, formal models as well as RTL models.

Figure 1-3 outlines the four major research contributions of this dissertation that are summarized as follows. The first three are related to verifying functional correctness whereas the last one ensure that the non-functional requirements are satisfied.

Directed test generation for multicore architectures: This work proposes a novel technique that exploits temporal, structural, and spatial symmetry in multicore designs

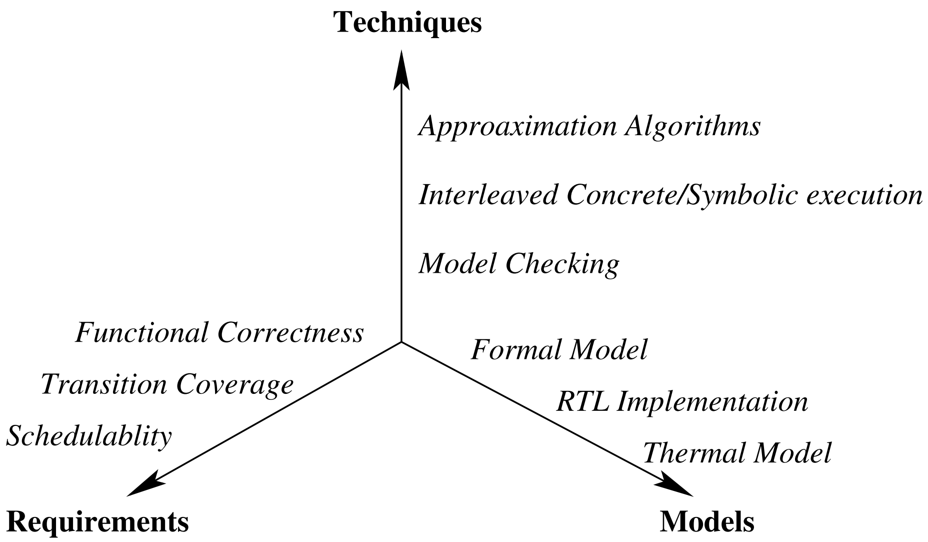

Figure 1-2. Design validation models, requirements and techniques

Multicore System Validation Challenges		
Functional Requirements		Non−functional Requirements
BMC−based Directed Test Generation	*(Chapter 3 and 4)*	Schedulablity Validaition Under Energy and Temperature Constraints
Cache ProtocolTransition Coverage	*(Chapter 5)*	
Scalable Test Generation	*(Chapter 6)*	*(Chapter 7 and 8)*

Figure 1-3. Dissertation outline

at the same time. Our proposed technique enables the reuse of the knowledge learned from one core to the remaining cores in multicore architectures (structural symmetry), from one bound to the next for a given property (temporal symmetry), as well as from one property to other properties (spatial symmetry). Our experimental results on both hardware and software designs demonstrate an order-of-magnitude reduction in overall test generation time.

Efficient test generation for state and transition coverage in cache coherence protocols: This work proposes an efficient test generation approach for a wide variety of cache coherence protocols. Based on detailed analysis of the space structure, our approach creates efficient test sequences for different parts of the global FSM state space to achieve 100% state and transition coverage for each cache coherence protocol. We develop a graphical description of the state space structure of several commonly used cache coherence protocols and present an on-the-fly directed test generation algorithm based on the Euler tour of hypercubes. The experimental results on different cache coherence protocols show the effectiveness of our approach on systems with many cores.

Scalable directed test generation for real HDL designs: This work develops a scalable technique to enable directed test generation of HDL models by incorporating static analysis and simulation based validation. By performing interleaved concrete and symbolic execution, our approach avoids the error-prone design translation process and enables directed test generation for real designs. Compared with existing approaches based on combined concrete and symbolic execution, our approach is capable of analyzing real processor designs with dynamic array references. The experimental results illustrate that our proposed technique is scalable, and enables directed test generation for real designs.

Temperature- and energy-constrained scheduling for multicore architectures: This work explores the DVS scheduling problem on multicore systems under both temperature and energy constraints. We show that this problem is NP-hard even when the steady state temperature is considered. We also present an exact algorithm and a polynomial time approximation scheme for the problem. When the original problem is schedulable, our approximation algorithm is guaranteed to generate a solution, which will not violate the temperature constraint, and consume no more time or energy than a specified approximation bound, e.g., within 1% of the optimal time consumption and energy constraints. The experimental results demonstrate that our technique is able to produce schedules close to optimal solution with reasonable execution time on real benchmarks.

1.4 Dissertation Organization

This dissertation is organized as follows. Chapter 2 introduces relevant existing research works. Chapter 3 and Chapter 4 describe proposed directed test generation for the functional validation of multicore architectures. Chapter 5 discusses proposed test generation approaches for transition coverage in cache coherence protocols. Chapter 6 describes our scalable directed test generation approach for HDL designs. Chapter 7 describes our schedulability validation approaches under energy and temperature constraints. Chapter 8 presents our schedulability validation technique for multicore processors. Chapter 9 concludes this dissertation.

CHAPTER 2
RELATED WORK

This chapter surveys existing system-level validation techniques. For ease of presentation, we have divided the existing approaches into three categories. First, we describe the test generation approaches for architecture validation. Next, we discuss existing techniques for validation of cache coherence protocols. Finally, we present techniques for validation of non-functional requirements.

2.1 Test Generation for Architecture Validation

Model checking techniques are promising for functional verification and test generation of complex systems [39, 50, 51, 64]. Figure 2-1 shows the general framework for directed test generation using model checking. In order to create directed tests, the formal model of the design specification and a suitable fault model are provided as input. Then a set of properties are generated for the desired behaviors (faults) that should be activated in the simulation based validation stage. For example, when a graph model of the design and a functional coverage fault model is provided,

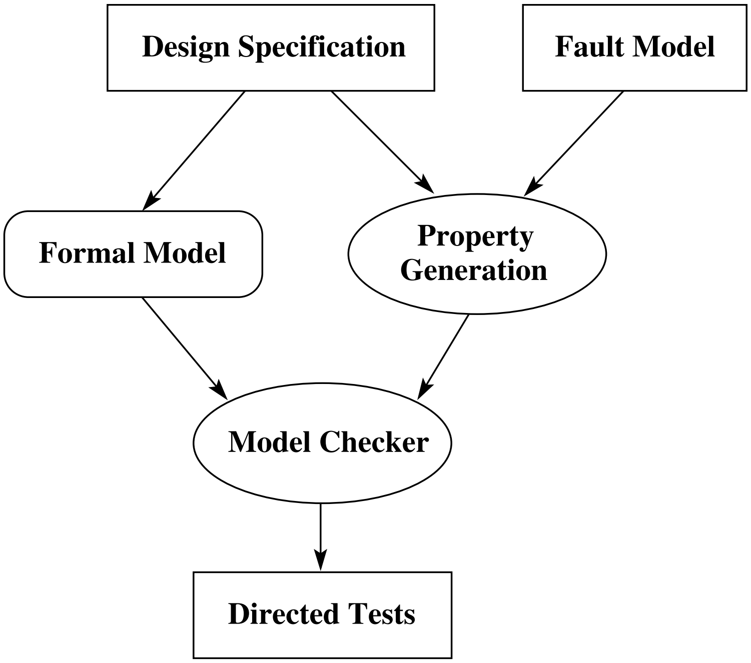

Figure 2-1. Directed test generation flow

a coverage-driven property generation can be used. Similarly, in case of circuits with stuck-at fault model, the property will be in the form of $G(a = 1)$ or $G(a = 0)$. Next, a model checker is employed to check whether there exists some states which violate the negated version of the property. If the model checker finds a violation, it reports a counterexample. This counterexample contains a sequence of input information which will drive the system from an initial state to a state that does not satisfy the negated version of the property, or in other words, which satisfies the original property. Therefore, we can use it as a test to activate the corresponding property or behavior during simulation-based validation.

Although model checking is effective for directed test generation, the capacity of the conventional symbolic model checking is usually limited. Bounded model checking (BMC) was proposed to address this problem by checking whether there is a counterexample for the property within a given bound [13] [28]. Given a design D, a safety property p, and a bound k, BMC will unroll the design k times and encode it using the following formula:

$$BMC(M,p,k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i) \tag{2–1}$$

where $I(s_0)$ is the initial state of the system, $R(s_i, s_{i+1})$ represents the state transition from state s_i to state s_{i+1}, and $p(s_i)$ checks whether property p holds on state s_i. The formula is then transformed to CNF and checked by a SAT solver. If the SAT solver finds some assignment which makes the CNF true, it implies that the property does not hold at bound k, i.e., $M \nvDash_k p$. Otherwise, if no such assignment is found, we conclude that the property holds up to k, or $M \vDash_k p$.

BMC cannot prove the validity of a safety property to hold globally when no counterexample is found within a specific bound, but it is quite effective to falsify a design when the bound is not large. The reason is that SAT solvers usually require less space and time than conventional Binary Decision Diagram (BDD) based model checkers [65]. Therefore, SAT-based BMC is suitable for directed test generation [64], where a counterexample typically exists within a relatively small bound. To generate the directed test, the negated version of the property is checked by BMC. The SAT solver will find an assignment of all input and state variables, which satisfies Equation (2–1). As a result, we can extract the assignment sequence of input variables and use it as a test to activate the desired property in the system.

A great deal of work has been done to reduce the SAT solving time during BMC [22–25, 43, 52, 79, 91]. The basic idea is to exploit the regularity of the SAT instances between different bounds. For example, incremental SAT solvers [43, 91] reduce the solving time by employing the previously learned conflict clauses. Generated conflict clauses are kept in the database as long as the clauses which led to the conflicts are not removed. Strichman [79] proposed that if a conflict clause is deduced only from the transition part of a SAT instance, it can be safely forwarded to all instances with larger bounds, because the transition part of the design will still be in the SAT instance when we unroll the design for more times. Besides, the learned conflict clauses can also be replicated across different time steps. However, the existing approaches did not exploit the symmetric structure within the same time step. *In directed test generation for multicore architectures, same knowledge about the core structure needs to be re-discovered for each core independently, which can lead to significant wastage of computational power.*

When BMC is applied in circuits, Kuehlmann [53] proposed that the unfolded transition relation can be simplified by merging vertices that are functionally equivalent under given input constraints. In this way, the complexity of transition relation is greatly reduced. Since this technique is based on the AIG representation of logic designs, it is difficult to use for accelerating the solving process of CNF instances, which are directly created from high level specifications. Functional validation based on high level specification is very effective in many scenarios. For example, Bhadra et al. [45] used executable specification to validate multiprocessor systems-on-chip designs. Chen et al. [22] proposed directed test generation based on high level specification. To accelerate the test generation process, conflict clauses learned during checking of one property are forwarded to speed up the SAT solving process of other related properties, although the bound is required as an input. Similarly, the simultaneous SAT solver [49] enabled the learned clauses to be reused by properties. Decision ordering was also studied in [23] to reduce the SAT solving time. *These approaches did not take the advantage of structural symmetry in multicore architectures.*

When SAT instance contains symmetric structure, symmetry breaking predicate [3, 5, 30, 62, 80] can be used to speed up the SAT solving by confining the search to non-symmetric regions of the space. By adding symmetry breaking predicates to the SAT instance, the SAT solver is restricted to find the satisfying assignments of only one representative member in a symmetric set. *However, this approach cannot effectively accelerate the directed test generation for multicore processors, because the properties for test generation are usually not symmetric with respect to each core.* Thus, the symmetric regions in the entire space are usually small despite the fact that the structure of each core is identical. Biere et al. [14] proposed that each component can be solved individually to accelerate the solving process. *However, the symmetric structure is not used at the same time for further speedup.*

During the validation process, it is also very important to generate assertions effectively. One important work in this direction is GoldMine [81], which automatically uses data mining and formal verification to generate assertions for real hardware designs. Using the simulation trace of RTL designs, GoldMine employs decision tree based supervised learning algorithms to mine potential assertions from the simulation data. Liu et al. [54] also proposed a methodology, which utilizes GoldMine to achieve coverage closure during design validation. Once the assertion is generated, automatic test generation approaches can be employed to generate the tests, which can be used to activate the desired behavior of the system. For example, test generation tools based on interleaved concrete and symbolic execution, such as DART [40], CUTE [72], and Apollo [7], are promising in capturing important bugs in large software systems. STAR [55] and HYBRO [56] are proposed to generate tests by combining static and dynamic analysis for hardware validation. Due to the effective utilization of the CFG, HYBRO [56] demonstrated remarkable improvement over previous path-based test generation technique [55]. However, HYBRO cannot be applied on real-life designs containing dynamic array references.

2.2 Validation of Cache Coherence Protocols

Verification of cache coherence protocols for multicore and multiprocessor systems has been widely studied in both academia and industry. Existing studies can be broadly grouped into two categories: formal verification [27, 33, 36] and simulation based validation [2, 83, 93]. Formal methods using model checking can prove mathematically whether the description of certain cache coherence protocol violates the required property. For example, Murφ [33] was designed and used to verify various cache coherence protocols based on explicit model checking. Counter-example guided refinement [27] is employed to verify complex protocols with multilevel caches. Besides, symbolic model checking tools are also developed for coherence verification. For example, Emerson et al. [36] investigated the verification problem with parameterized cache coherence protocol using BDDs. Although formal methods can guarantee the correctness of a design, they usually require that the design should be described in certain input languages. As a result, model checking usually cannot be applied to implementations directly.

Simulation based approaches, on the other hand, are able to handle designs at different abstraction levels and therefore more widely used in practice. For example, Wood et al. [93] used random tests to verify the memory subsystem of SPUR machine. Successive loads and stores to the same location are employed as test template to expose possible errors. Genesys Pro test generator [2] from IBM extended this direction with more complex and sophisticated test templates. To reduce the search space, Abts et al. [1] introduced space pruning technique during their verification of the Cray processor. Wagner et al. [83] designed the MCjammer tool which can get higher state coverage than normal constrained random tests. Existing random test generation tools are proven to be effective to discover potential bugs. However, due to their random nature, it is very hard to achieve full state and transition coverage in a reasonable time. Since an uncovered transition can only be visited by taking a unique action at a particular state, it may not be feasible for a random test generator to eventually cover all possible states and transitions. To address this problem, some random testers are equipped with small amount of memory, so that the future search can be guided to the uncovered regions. Unfortunately, unless the memory is large enough to hold the entire state space, it is still quite hard to achieve full coverage by such guided random testing.

2.3 Task Schedulability under Constraints

Energy-aware scheduling techniques for real-time systems have been widely studied to reduce energy consumption. While several works employed dynamic cache reconfiguration [87] [85], most of them are based on Dynamic Voltage Scaling (DVS). Aydin et al. [9] addressed both static and dynamic slack allocation problems for periodic task sets, while Shin et al. [73] also considered aperiodic tasks. Jejurikar et al. focused on energy-aware scheduling for non-preemptive task sets [47] and leakage power minimization [48]. Zhong et al. [103] solved a system-wide energy minimization problem with consideration of other components. Wang et al. [85] proposed a leakage-aware energy saving technique based on DVS as well as cache reconfiguration. As shown in [100], applying DVS in real-time systems is a NP-hard problem. Optimal and approximation algorithms are given in [103] [100] [86], while other works proposed heuristics. A survey on recent works can be found in [21]. However, these techniques are not aware of controlling the operating temperature.

Temperature-aware scheduling in real-time systems has drawn significant research interests in recent years. Wang et al. [84] introduced a simple reactive DVS scheme aiming at meeting task timing constraints and maintaining processor safe temperature. Zhang et al. [101] proved the NP-hardness of temperature-constrained performance optimization problem in real-time systems and proposed an approximation algorithm. Yuan et al. [97] considered both temperature and leakage power impact in DVS problem for soft real-time systems. Chen et al. [20] explored temperature-aware scheduling for periodic tasks in both uniprocessor and homogeneous multiprocessor DVS-enabled platforms. Liu et al. [57] proposed a design-time thermal optimization framework which is able to solve problem variants energy-aware (EA), temperature -aware (TA) and temperature-constrained energy-aware (TCEA) scheduling in embedded system with task timing constraints. Jayaseelan et al. [46] exploited different task execution orders, in which each task has distinct power profile, to minimize peak temperature. However, none of these techniques solves temperature-constrained and energy-constrained (TCEC) problem. Moreover, they all make certain assumptions on system characteristics that limits their applicability.

Existing research formulated the voltage/frequency assignment problems in different models. For example, Integer Linear Programming (ILP) has been widely applied to many voltage/frequency assignment problems without the temperature constraint [94, 102]. Chantem et al. [19] also used ILP to model scheduling problem with steady-state temperature constraints. Unfortunately, when transient temperature is considered, the full expansion of the temperature constraint introduces a large number of product terms, which prevent us to solve the problem efficiently using ILP solvers. Coskun et al. [29] circumvented this problem using an iterative ILP and thermal simulation approach, although the convergence to the optimal solution is not guaranteed.

Another important modeling technique is timed automata [6]. Norstorm et al. [66] first extended timed automata with the notion of real-time tasks and showed that the traditional schedulability analysis can be transformed to a decidable reachability problem in timed automata, which can be solved using model checking tools. Fersman et al. [37] further generalized this approach with asynchronous processes and preemptive tasks in continuous-time model. However, none of these techniques considered energy or temperature related issues.

There are several studies on Dynamic Power Management (DPM) using formal verification methods for embedded systems [74] and multiprocessor platforms [58]. Shukla et al. [74] provided a preliminary study on evaluating DPM schemes using an off-the-shelf model checker. Lungo et al. [58] tried to incorporate verification of DPM schemes in the early design stage. They showed that tradeoffs can be made between design quality and verification efforts. None of these approaches considers temperature management in such systems. Moreover, they did not account for energy and timing constraints, which are important in real-time embedded systems. Wang et al. [88] discussed the application of timed automata in schedulability problem with both energy and temperature constraints. Nevertheless, due to the capacity limit of model checker, the proposed technique can only be applied to small task sets.

Temperature- or energy-constrained scheduling problems are also related to the multi-constrained path (MCP) problem for Quality of Service (QoS). MCP was extensively studied by network community. For example, Chen et al. [26] designed an approximation algorithm for MCP with two constraints. [76] and [98] studied the efficient heuristics for MCP problems. Xue et al. [96] proposed polynomial time approximation algorithms, which can be applied for more than two constraints. However, since the QoS costs are usually modeled as additive constants, these existing methods cannot be applied directly to solve TCEC problem due to the fact that the computation of the temperature is not additive.

CHAPTER 3
SYNCHRONIZED GENERATION OF DIRECTED TESTS

Model checking is promising for automatic generation of directed tests [39, 64], because the counterexample of the negated version of a property can be used as a test to activate the property. Existing test generation techniques using SAT-based bounded model checking (BMC) [67] can be divided into two categories based on whether it addresses one property or multiple properties. The first category is applicable for test generation for one design and one property with varying bounds [78, 79]. However, the knowledge obtained are not shared when solving for other properties on the same design. In contrast, the methods in the second category tries to accelerate the test generation for multiple properties with known bounds [63]. They first group similar properties into clusters. Then, the knowledge are shared by all properties in the same cluster. This approach exploit the fact that although each test generation instance is created for a different property, these instances still have a large overlap, because the design remains unchanged. The major drawback of this solution is that it assumes that the bound is known. In general, it is very difficult to determine the bound upfront without actually solving the SAT instance, which limits the applicability of this solution.

In this work [68], we combine the advantages of both approaches by developing a novel BMC based test generation technique for multiple properties of the same design, which enables the reuse of learned knowledge across different bounds as well as across properties in the same cluster. The basic idea of our approach is to *synchronize the solving process of multiple properties for different bounds, so that the utilization of learned knowledge can be maximized*. One may think that solving many SAT instances together can be dramatically complex than solving one instance, and therefore may be impractical. On the contrary, since all these instances are generated by unrolling the same design for several times, we successfully developed a simple but effective approach to significantly reduce the overall SAT solving time by forwarding knowledge among different solving processes. Our experimental results demonstrate an order-of-magnitude reduction in overall test generation time.

The rest of the chapter is organized as follows. Section 3.1 briefly discusses the background on SAT-based BMC. Section 3.2 describes our test generation methodology for multiple properties and bounds. Section 3.3 presents our experimental results. Finally, Section 3.4 concludes this chapter.

3.1 Background

This section briefly describes the basic concepts of existing acceleration techniques for directed test generation using BMC.

3.1.1 Conflict clause forwarding

Many techniques and heuristics are employed in SAT solvers to accelerate the solving process. Modern SAT solvers such as zChaff [38] and GRASP [59] adopt the Davis-Putnam-Logemann-Loveland (DPLL) [31, 32] algorithm and conflict-driven non-chronological backtracking. The basic idea behind these techniques is to save the knowledge learned during resolving current conflict to avoid the same conflict in the future [99]. A conflict occurs, when the current assignment of some variables, through a set of clauses, implies that one variable must be true and false at the same time. In this case, conflict analysis will trace back along the implication relations and find the closest assignment of variables that led to the conflict. We can forbid such assignment from occurring again by adding a carefully designed clause, i.e., conflict clause, to the original CNF. Generally, conflict clauses are only meaningful within the same SAT instance. However, when the set of clauses that led to the conflict clause are shared by multiple SAT instances, we can also forward conflict clauses across instances.

3.1.2 Property clustering

Property clustering is another important technique to reduce the total test generation time with BMC. As indicated in Figure 2-1, for a given design and fault model, we first generate a set of properties, which can be used to activate all the faults.

Then, different SAT instances for these properties are solved to obtain the tests. Since sharing of knowledge among similar properties usually reduces the overall solving time, we can cluster properties into different groups and solve all the properties in the same group together.

Although the intention behind property clustering is intuitive, the challenge here is to determine the proper number of clusters and which properties should be in the same cluster. On one extreme, one can group all properties into a single cluster and solve them together. Although this approach maximizes the sharing of knowledge among properties, it also increases the possibility that too many conflict clauses are accumulated in the solver's database, which hampers the overall performance of the SAT solver. On the other extreme, we can also let each cluster have only one property, which is actually the approach adopted by Strichman et al. [79]. In this way, only knowledge relevant to the properties will be explored. However, it is also possible that the same knowledge will be discovered again and again for different properties, which is a significant overhead, when the number of similar properties is large. Therefore, it is desirable to strike a balance between the knowledge sharing and overhead introduced by irrelevant conflict clauses.

Our property clustering approach for designs given in graph models is similar to [63]. The properties are grouped together by their similarity on structural or textual overlap. The properties in the same cluster are describing behaviors of the same functional unit or component. In this way, it is likely that the knowledge or conflict clauses that we obtained during solving one property will be helpful to other properties in the same cluster. For circuits with stuck-at fault model, we perform the clustering of properties based on the cone-of-influence (COI). Output signals with large overlap in their COIs are grouped into the same cluster.

3.2 Synchronized Test Generation

Figure 3-1 shows the framework of our synchronized test generation approach. In order to create directed tests, the formal model of the design, a set of properties for the desired behaviors (faults) that should be activated, and the corresponding cluster information are accepted as input. Next, the SAT instances for each property are grouped into different clusters based on their similarity and then solved simultaneously to create the test suite, which can be used to trigger the desired behaviors during simulation-based validation. Algorithm 1 outlines the key steps in our directed test generation framework. The contribution is the synchronized test generation for properties in a cluster, which will be explained in details in Algorithm 2.

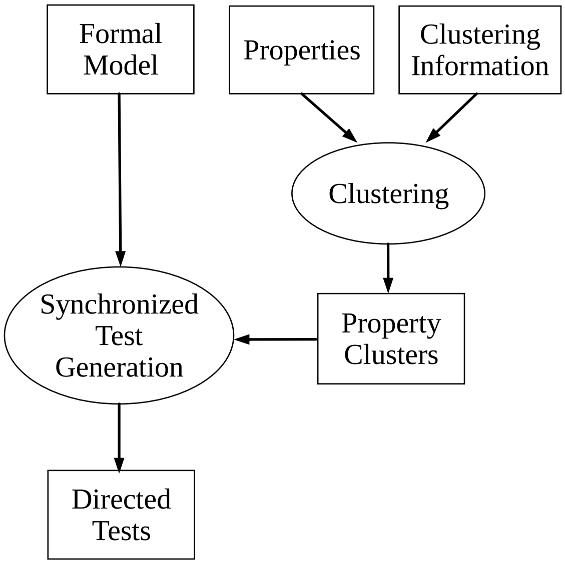

Figure 3-1. Synchronized Test Generation

To highlight the contribution of our work, Figure 3-2 compares our approach with two closely related techniques: i) incremental SAT for single property with unknown bound [79] and ii) test generation for multiple properties with known bounds [63]. In this example, there are three properties p_1, p_2, and p_3 with bounds 3, 2, and 1 respectively. We use solid dots to represent different SAT instances and lines to indicate the conflict clause forwarding paths. Strichman et al. [79] solved each property separately, and

```
Algorithm 1: Test generation framework
Input: i) Design $D$, ;
        ii) Properties $P$ for fault activation ;
Output: Tests for corresponding faults
Cluster similar properties into groups.;
$TestSuite = \emptyset$;
for each property cluster $PC$ do
    Perform Synchronized Test Generation on $PC$.;
    Add generated tests into TestSuite.;
end
return TestSuite
```

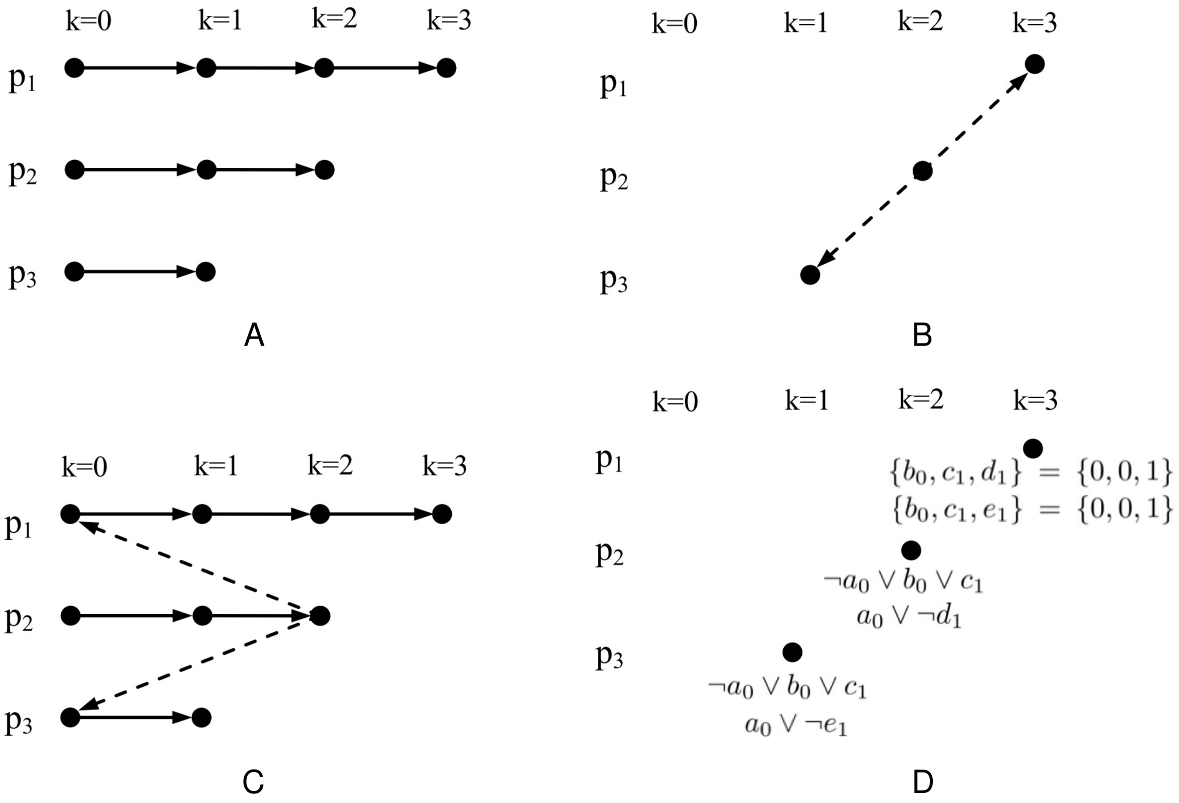
k=0
k=1
k=2
k=3
p1
p2
p3
A
B
C
D
$\{b_0, c_1, d_1\} = \{0, 0, 1\}$
$\{b_0, c_1, e_1\} = \{0, 0, 1\}$
$\neg a_0 \vee b_0 \vee c_1$
$a_0 \vee \neg d_1$
$\neg a_0 \vee b_0 \vee c_1$
$a_0 \vee \neg e_1$

Figure 3-2. Different incremental SAT solving techniques. A) Strichman [79]. B) Mishra and Chen [63]. C) A naive combination of [79] and [63]. D) Tentative assignment of variables during checking p_1 at $k = 3$.

passed the knowledge (deduced conflict clauses) “horizontally” within instances for the same property (Figure 3-2A). In contrast, Mishra et al. [63] solved one “base” property first, (e.g., p_2 in this case), then forward the learned clause “vertically” between other SAT instances for different properties, as shown in Figure 3-2B.

Clearly, it should be profitable if we can appropriately forward conflict clauses “vertically” between properties while solving for each property “horizontally”. In this way, the knowledge learned during checking a property for a specific bound can benefit itself with larger bounds as well as across other properties. One intuitive way to combine the two approaches, as shown in Figure 3-2C, is to choose some property as based property (p_2 in Figure 3-2C), check this property for different bounds, and then forward the learned conflict clauses to other SAT instances for other properties. Unfortunately, this naive combination has three problems. First, it is very hard to choose the base property, that should yield a large number of conflict clauses which can be shared by other properties. Unlike [63], where each property has only one SAT instance, we do not know how many SAT instances we have to solve. As a result, it is impossible to apply the clustering technique proposed in [63], to determine the base property. Secondly, even if we correctly find the optimal base property, it is still difficult to choose the suitable bound of the receiving property to forward clauses, because SAT instances with inappropriate bounds may be solved trivially. Moreover, the learning during checking non-base properties is wasted. For example, in Figure 3-2D, suppose $(\neg a_i \vee b_i \vee c_{i+1})$, $(a_i \vee \neg d_{i+1})$ and $(a_i \vee \neg e_{i+1})$ are clauses within the transition constraint of the system at time step $i+1$.

In the SAT solving process of p_2 with bound $k = 2$, a conflict clause $(b_0 \vee c_1 \vee \neg d_1)$ is deduced based on $(\neg a_0 \vee b_0 \vee c_1)$ and $(a_0 \vee \neg d_1)$ to prevent the assignment $\{b_0, c_1, d_1\} = \{0, 0, 1\}$, which will result in a conflict on a_0. During the solving process of p_1 with bound $k = 2$, the SAT solver may explore the assignment $\{b_0, c_1, d_1\} = \{0, 0, 1\}$ if Strichman’s approach [79] is employed. Such assignment can be avoided by using [63] (as shown in Figure 3-2B and Figure 3-2C), because the learned conflict clause $(b_0 \vee c_1 \vee \neg d_1)$ is forwarded to p_1.

However, learned clauses are only allowed to be forwarded from the base property (p_2 in this case). The knowledge learned during solving non-base properties will not be reused. As indicated in Figure 3-2D, conflict clause $(b_0 \vee c_1 \vee \neg e_1)$ is deduced based on $(\neg a_0 \vee b_0 \vee c_1)$ and $(a_0 \vee \neg e_1)$ during the solving process of p_3 with bound $k = 1$. Since p_3 is not a base property, this information will not be reused by p_1. Therefore, during the solving process of p_1 with bound $k = 2$, the SAT solver will still try to make the assignment $\{b_0, c_1, e_1\} = \{0, 0, 1\}$. When the number of properties is large, this may cause a great waste of computational power, because we have to explore the same search space for many times, if the space is not visited during the solving process of the base property.

Our approach to solve this problem is based on the effective identification of conflict clauses that can be shared by other SAT instances across properties and bounds. *In fact, for any bound $k_0 \geq 0$, all SAT instances generated during BMC (Equation 2–1) with $k \geq k_0$ clearly share the transition clauses $I(s_0) \wedge \bigwedge_{i=0}^{k_0-1} R(s_i, s_{i+1})$, although their property terms $\bigvee_{i=0}^{k} \neg p(s_i)$ are different. This observation implies that all conflict causes deduced based on these common clauses during solving process of any SAT instance can be forwarded to any other SAT instances with $k \geq k_0$, because all of them have the same set of clauses that led to the conflict clause.* Therefore, if we check all properties together for $k = 0, 1, 2, ...$, i.e., “synchronously”, all conflict clauses can be safely shared by all subsequent SAT instances.

Algorithm 2 outlines our synchronized test generation method for clustered properties. It accepts each property cluster and the design of the system as input and produces corresponding tests. As indicated before, this algorithm will check all properties synchronously for each bound. In each iteration, we first generate the transition clause set CS_T^k (corresponding to $I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})$) using BMC(D,true,k),

Algorithm 2: Synchronized Test Generation For Properties in a Cluster

```
Input: i) Design $D$, ;
       ii) Properties $P$, ;
       iii) Maximum bound $K_{max}$
Output: Test Set $TS$
Bound $k \longleftarrow 0$;
Common Conflict Clause Set $CCS \longleftarrow \emptyset$;
$TS \longleftarrow \emptyset$;
while $P \neq \emptyset$ and $k \leq K_{max}$ do
    Clause Set $CS_T^k \longleftarrow BMC(D, true, k)$;
    for $p \in P$ do
        Clause Set $CS_p^k \longleftarrow BMC(D, p, k)$;
 Step1: In $CS_p^k$, mark all clauses that also exist in $CS_T^k$ ;
 Step2: $(ConflictC, test_p) \longleftarrow$ SAT($CCS \bigcup CS_p^k$);
 Step3: $CCS \longleftarrow CCS \bigcup$ CheckMark ($ConflictC$);
        if $test_p \neq null$ then
            remove $p$ from $P$;
            $TS \longleftarrow TS \bigcup test_p$;
        end
    end
    $k \longleftarrow k + 1$;
end
return TS
```

then randomly choose a property p from the property set P, and create its own clause set CS_p^k (corresponding to $I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i)$). Next, we perform following 3 steps.

1. Mark all clauses in CS_p^k which are also in CS_T^k. Since CS_T^k remains same for all properties at k, this step can be implemented efficiently by table lookup, as described in Section 3.2.2.

2. Use a SAT solver to solve the CNF formula $CCS \bigcup CS_p^k$, which contains not only CS_p^k, but also all previously learned conflict clauses in CCS.

3. For new conflict clauses $ConflictC$ learned by SAT solver, merge the clauses deduced purely by marked clauses into CCS. This step is similar to the isolation technique proposed in [78] and [63].

If the satisfied assignment, or a counterexample $test_p$ is found in step 2, we record it in test set TS and remove p from P. This process repeats until tests for all properties are found or the maximum bound K_{max} is reached. Finally, the algorithm returns all generated tests.

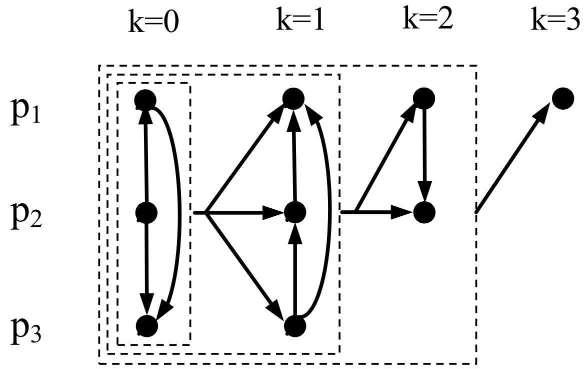

Figure 3-3. Synchronized test generation for multiple properties

We use the same example in Figure 3-2 to illustrate the flow of Algorithm 2. The clause forwarding path are shown in Figure 3-3. In the first iteration for $k = 0$, suppose we randomly pick p_2 from the property set. At the beginning, the common conflict clause set CCS is empty. Thus, p_2 is solved directly. Since the bound of p_2 is 2, the SAT instance is not satisfiable and no test is generated. However, all conflict clauses deduced based on clauses in CS_T^0 are now recorded in CCS, and will be used to accelerate the solving process of both p_1 and p_3 at bound 0. Similarly, the conflict clauses generated during solving p_1 at $k = 0$ will be used to speed up p_3 at $k = 0$ (assumes p_3 is solved last). In the next iteration, all instances will be solved with the help of conflict clauses learned by all three SAT instances at $k = 0$, because all conflict clauses are recorded in CCS. Eventually, three tests will be generated at bound 3, 2, and 1 for p_1, p_2 and p_3 respectively. In the case of Figure 3-2D, since both $(\neg a_0 \vee b_0 \vee c_1)$, $(a_0 \vee \neg d_1)$ and $(a_0 \vee \neg e_1)$ are clauses from the transition constraint of the system, both $(b_0 \vee c_1 \vee \neg d_1)$ and $(b_0 \vee c_1 \vee \neg e_1)$ will be recorded in CCS based on Algorithm 2. Therefore, during the solving process of p_1 with bound $k = 2$, the SAT solver will skip the assignment $\{b_0, c_1, d_1\} = \{0, 0, 1\}$ and $\{b_0, c_1, d_1\} = \{0, 0, 1\}$. In this way, the unnecessary waste of time is avoided.

Note that our algorithm does not require the SAT instances to be preprocessed using Cone-Of-Influence (COI) optimization as in [79] and [63], because original SAT instances have more overlapped clauses, which are effectively exploited by our approach to accelerate the overall solving process. Our experimental results in Section 3.3 show that our approach (without COI) outperforms [79] and [63] that use COI optimization.

In the remainder of this section, we prove the correctness of our approach and discuss the implementation details of our synchronized test generation algorithm.

3.2.1 Correctness of the Proposed Approach

To show the correctness of our test generation approach, we need to show that in Algorithm 2, solving $CCS \bigcup CS_p^k$ is equivalent to solving CS_p^k. Formally, let φ_p^k and ψ be the CNF formulae formed by clause set CS_p^k and CCS respectively, we need to prove that φ_p^k is satisfiable iff $\varphi_p^k \wedge \psi$ is satisfiable using the following lemma.

Lemma 1. *$\varphi_p^k \vdash \psi$ for all $p \in P$ and $k \geq 0$.*

Proof. Let φ_T^k be the CNF formula formed by CS_T^k. We first show that

$$\varphi_T^k \vdash \psi \tag{3–1}$$

for $k \geq 0$ by induction on the size of ψ. In the basis step, formula 3–1 obviously holds because ψ is empty.

Considering the moment before a new conflict clause π is added to ψ in some iteration when the bound $k' \leq k$, π must be deduced from $\varphi_T^{k'} \wedge \psi$, i.e., $\varphi_T^{k'} \wedge \psi \vdash \pi$. By induction hypothesis, $\varphi_T^k \vdash \psi$ before π is added into ψ. We also know that $\varphi_T^k \vdash \varphi_T^{k'}$, because their original forms satisfy

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \;\vdash\; I(s_0) \wedge \bigwedge_{i=0}^{k'-1} R(s_i, s_{i+1})$$

Hence, $\varphi_T^k \vdash \varphi_T^{k'} \wedge \psi$. As a result, we have $\varphi_T^k \vdash \pi$ and $\varphi_T^k \vdash \psi \wedge \pi$, which means formula 3–1 still holds, after any new clause is added to ψ, as long as $k' \leq k$.

On the other hand, we notice that

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i) \ \vdash \ I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})$$

or

$$\varphi_p^k \vdash \varphi_T^k$$

Therefore, we conclude that

$$\varphi_p^k \vdash \psi$$

for all $p \in P$ and $k \geq 0$. □

Since $\varphi_p^k \vdash \psi$, we have $\varphi_p^k \leftrightarrow \varphi_p^k \wedge \psi$. In other words,

Theorem 3.1. $\forall p \in P$ φ_p^k *is satisfiable iff* $\varphi_p^k \wedge \psi$ *is satisfiable.*

The correctness of our approach is therefore justified.

3.2.2 Implementation Details

Our synchronized test generation algorithm is built around zChaff SAT solver [38], which provides clause management scheme to support incremental SAT solving. zChaff maintains all input clauses and generates conflict clauses within an internal clause database DB. When invoked, it will solve the CNF formed by all clauses currently in DB. The management of clauses within database DB is based on "group". For each clause, zChaff assigns a 32-bit group ID. Each bit identifies whether that clause belongs to a certain group or not. When a conflict clause is deduced by clauses from multiple groups, its group ID is a "OR" product of the group ID of all its parent clauses, i.e., this clause belongs to multiple groups. zChaff also allows user to add or remove clauses by group ID between successive solving processes. If one clause belongs to multiple groups, it is removed when any of these groups are removed.

With these utilities, the step 1 and 3 in Algorithm 2 can be implemented efficiently as follows:

1. In the clause marking step, add all clauses in $CS_T^k \bigcap CS_p^k$ into DB with group ID 1.
2. Add other clauses in CS_p^k into DB with group ID 2.
3. After solving all clauses in DB with zChaff, remove clauses with group ID 2.

In this way, CCS is implicitly maintained within DB, because only conflict clauses generated purely based on clauses in $CCS \bigcup CS_T^k$ are kept after each iteration.

There is another potential overhead in step 1. Before we mark it in CS_p^k, we have to identify whether it is in CS_T^k. Since CS_T^k remains same for all properties at k, we build a hash table to record all clauses in CS_T^k. It takes $O(1)$ time to determine whether a clause from CS_p^k is in CS_T^k. Therefore, the overall time consumption of steps 1 and 3 in Algorithm 2 is negligible compared to the SAT solving time.

3.3 Experiments

We have evaluated our test generation approach using different software and hardware designs. In this section, we compare our approach with existing methods [79] and [63] in three scenarios: a stock exchange system, a VLIW implementation of the MIPS architecture, and ISCA'89 benchmark circuits. In the first two scenarios, the systems and properties are described in SMV language and converted to CNF clauses (DIMACS files) using NuSMV [18]. We used zChaff [38] as our SAT solver to implement our test generation algorithm. The experiments were performed on a PC with 3.0GHz AMD64 CPU and 4GB RAM.

3.3.1 A Stock Exchange System

The design in our first case study simulates the behavior of a common online stock exchange system (OSES). It can accept, check and execute the customers orders (market order and limit order). The system is specified using UML activity diagram and implemented in JAVA. Its UML behavior specification has 27 activities, 29 transitions and 18 key paths. The specification is translated into NuSMV input to generate corresponding SAT instances. Then we apply our synchronized SAT solving approach to find the satisfiable assignments, which can be used as tests. We compared our approach with Strichman's approach [79] and a naive combination of [79] and [63] on different properties with unknown bounds. For Strichman's approach [79], we use it to solve a sequence of SAT instances for the same property with varying bounds until a satisfiable instance is found. The naive combination of [79] and [63] is developed as described in Section 3.2. After SAT instance generation, we applied cone of influence (COI) to speed up Strichman's approach. When our approach was applied, we did not use COI as indicted in Section 3.2.

Table 3-1. Test generation time comparison for OSES

Prop.	Bound	Our	[79] vs ours		[79] + [63]* vs ours	
		Approach	[79]	Speed-	[79] + [63]	Speed-
		Time (s)	Time (s)	up	Time (s)	up
1	15	2.94	180.31	61.24	67.58	22.96
2	14	2.55	150.49	59.06	57.70	22.64
3	14	3.12	149.89	48.04	61.11	19.59
4	15	10.54	139.56	13.25	42.53	4.04
5	14	19.38	130.58	6.74	55.74	2.88
6	14	2.97	107.13	36.09	61.66	20.77
7	16	6.61	101.67	15.39	35.86	5.43
8	16	3.54	89.31	25.20	3.76	1.06
9	15	1.73	84.19	48.72	38.97	22.55
10	12	1.96	84.07	42.80	5.51	2.80
11	13	1.21	83.94	69.48	22.54	18.66
12	15	2.83	83.80	29.59	39.77	14.04
13	15	5.60	83.01	14.81	23.49	4.19
14	14	1.34	80.25	59.88	22.60	16.86
15	14	11.16	79.79	7.15	22.53	2.02
16	15	0.85	78.72	92.39	10.94	12.85
17	15	0.88	78.28	88.95	14.51	16.49
18	15	0.86	78.19	90.49	12.60	14.58
19	12	79.40	74.96	0.94	75.10	0.95
20	12	1.38	73.46	53.23	5.43	3.93
Total	-	160.87	2011.62	**12.50**	679.93	**4.23**

* This is an intuitive combination of [79] and [63] (Figure 3-2C). We have shown these results to demonstrate how our approach is superior than any naive combination of existing methods [79] and [63].

Table 1 shows the results of 20 most time consuming properties using Strichman's approach [79]. The first column shows the properties used for test generation. The second column indicates corresponding bounds of each property. The third column shows the test generation time (in seconds) for each property using our approach. The time consumed by steps 1 and 3 in Algorithm 2 is also counted in this column. The fourth column indicates the time required by Strichman's approach [79] to generate the test for the same property. The time is calculated as the summation of the time to solve all the SAT instances from $k = 0$ to the bound of the property. The fifth column shows the speedup[1] of our approach over [79]. The last two columns present the test generation time using the naive combination of [79] and [63] and the speedup of our approach. It can be seen that our approach can produce more than 10 times improvement compared to [79], because many more conflict clauses are reused by subsequent iterations. This is especially important for "hard" SAT instances, which have to explore a potentially large assignment space. For example, the "hardest" property p_1 for [79] actually consumes less than 3 seconds in our approach. Clearly, the time consumption for solving multiple SAT instances using our approach is significantly smaller than the summation of time to solve each instances independently. The overall time consumption is reduced by knowledge sharing during solving all properties synchronously.

One interesting observation is that the most time consuming property p_{19} in our approach has a bound of only 12. The reason for this is that the clauses learned during the solving process of easier properties like p_{19} eliminated some useless searching attempts for the solution of harder properties like p_1. More importantly, these clauses are more effective than the conflict clauses learned during solving SAT instances of the same property with smaller bounds. Although p_{19} itself, which was solved first, did not

[1] calculated as (previous column / third column)

benefit from other properties, the overall time consumption was dramatically reduced. As a result, our approach outperforms [79], which only forwards clauses within SAT instances of the same property.

For the naive combination of [79] and [63], we chose p_{19} as the base property and forwarded the clauses learned during solving it to other properties at bound 11. These parameters are selected to illustrate the best possible performance of the combination. It is remarkably faster compared to Strichman's approach [79], although it is still 4 times slower than our approach. It should be noted that in reality, it is impossible to choose the optimal parameter for this combination because the bounds are unknown for all properties. In other words, the performance of the naive combination of [79] and [63] will be much worse than we illustrated here. Thus, our approach will outperform it more significantly in practical scenarios.

We also investigated the impact of cluster size on the overall solving time. The total 135 properties are clustered into groups with different size. The results are shown in Figure 3-4. Figure 3-4A presents the overall solving time with respect to different average cluster size. Figure 3-4B shows the corresponding average number of forwarded clauses per cluster (solid curve) and the total number of conflicts encountered for different cluster size (dotted curve). Their values can be found on the left and right Y-axis respectively. The result suggests that larger clustering is generally helpful to reduce the overall solving time. The reason is that the number of forwarded clauses usually increases with the average cluster size, which can effectively reduce the total number of conflict encountered during the solving process.

3.3.2 A VLIW MIPS Processor

We also applied our test generation approach to a single-issue MIPS processor [42], [64]. There are five pipeline stages: fetch, decode, execute, memory access, and writeback. The execute stage has four parallel execution paths: integer ALU, 7

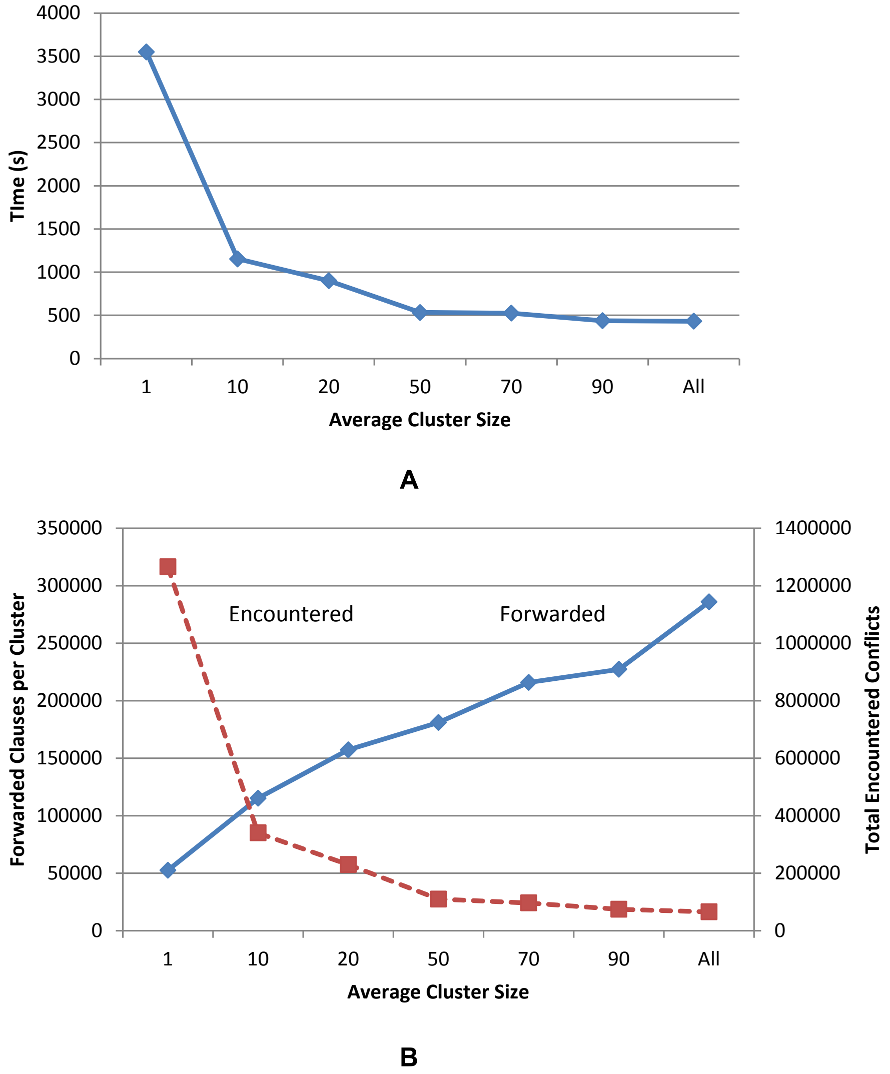

Figure 3-4. Test generation for OSES using different cluster size. A) Time consumption. B) Forwarded clauses and encountered conflicts.

stage multiplier (MUL1 -MUL7), four stage floating-point adder (FADD1 - FADD4), and multi-cycle divider (DIV).

We translated the design into the NuSMV input and used the three approaches to solve the generated SAT instances for different properties and bounds. For the combination of [79] and [63], we chose p_{17} as the base property and forwarded learned clauses to bound 7. The results are given in Table 2. We only show the results on 20 most time consuming properties using Strichman's approach. It can be seen that our approach outperforms both Strichman's approach [79] and the naive combination of [79] and [63] by 15 and 3 times respectively.

Table 3-2. Test generation time comparison for MIPS

Prop.	Bound	Our	[79] vs ours		[79] + [63]* vs ours	
		Approach	[79]	Speed-	[79] + [63]	Speed-
		Time (s)	Time (s)	up	Time (s)	up
1	8	0.78	139.29	179.48	18.66	24.04
2	8	0.74	132.07	178.46	19.45	26.29
3	8	0.76	125.18	164.70	18.18	23.93
4	8	0.76	120.02	158.74	18.45	24.40
5	8	0.76	115.84	151.61	27.14	35.53
6	9	0.86	111.13	129.81	58.26	68.06
7	8	0.81	108.09	133.76	26.63	32.95
8	9	0.95	104.56	110.29	53.59	56.52
9	8	0.75	96.25	128.67	16.77	22.41
10	8	0.77	87.24	113.00	16.47	21.33
11	8	0.76	87.23	114.77	17.37	22.85
12	8	0.77	84.98	110.64	16.45	21.42
13	7	0.65	81.08	125.11	13.35	20.60
14	9	32.31	80.25	2.48	31.61	0.98
15	8	0.76	75.47	99.30	7.25	9.54
16	8	0.76	72.05	94.30	20.63	26.99
17	7	76.54	71.72	0.94	72.30	0.94
18	8	1.00	70.05	70.33	19.46	19.53
19	8	0.76	69.85	91.90	6.98	9.19
20	8	0.76	65.80	87.03	11.08	14.65
Total	-	122.99	1898.13	**15.43**	490.06	**3.98**

The impact of cluster size on the overall solving time are shown in Figure 3-5. There are 170 properties in total. It can be observed that the overall solving time becomes constant after the average cluster size is more than 50 (Figure 3-5A). At the same time, the number of forwarded clauses per cluster is not increasing, as indicated by the dotted curve in Figure 3-5B. This phenomenon can be explained by the fact that once the clusters are large enough to include all the similar properties, the overall solving time will not be further improved and the number of forwarded clauses becomes stable.

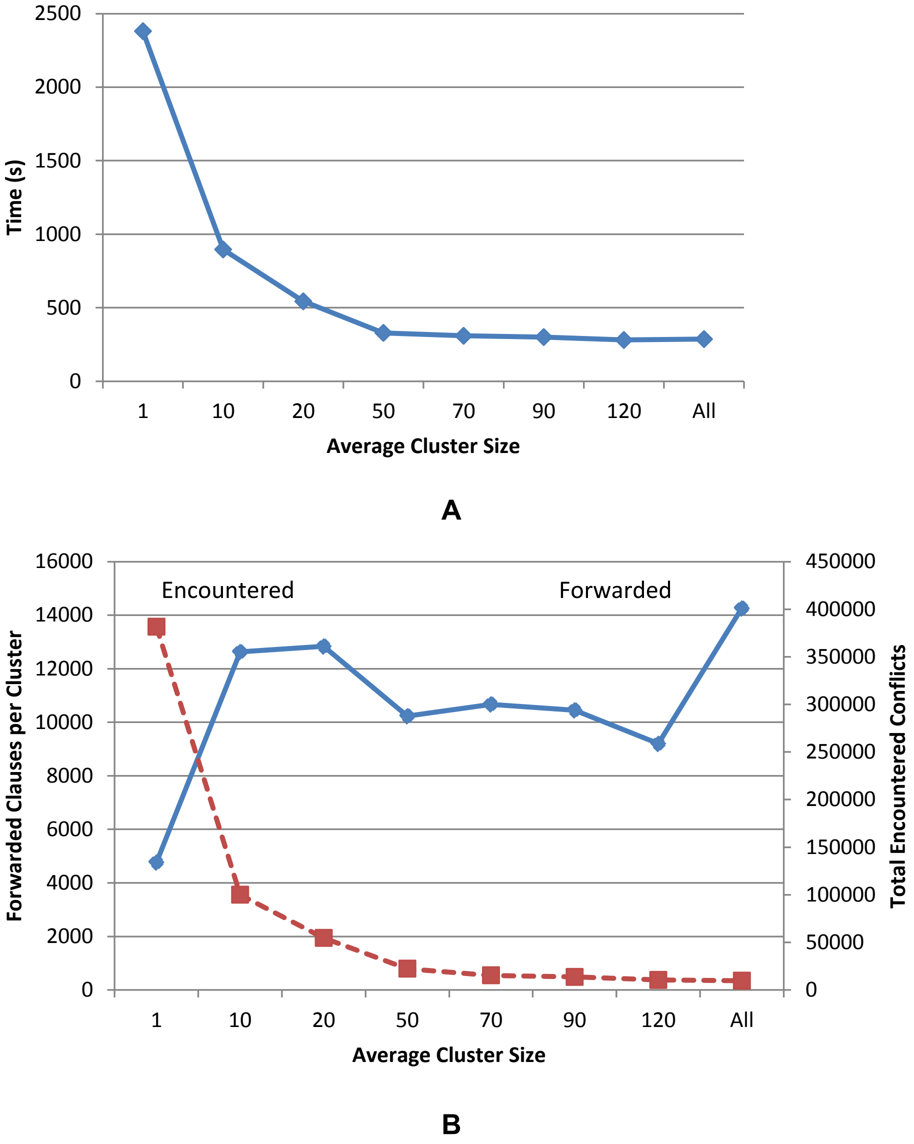

Figure 3-5. Test generation for MIPS using different cluster size. A) Time consumption. B) Forwarded clauses and encountered conflicts.

3.3.3 Circuit Test Generation

We applied our test generation approach to activate stuck-at fault using ISCA 89 benchmark. For each circuit, we search for input sequences, which can generate 0 and 1 on each output port. Benchmark circuits are translated into CNF using standard formulae in zChaff. The results are given in Table 3. The solving time limit for each property is 100 seconds. We only show the results on 5 circuits with maximum total test generation time using Strichman's approach. It can be seen that our approach outperforms both Strichman's approach [79] especially for complex circuits like s38584.

Table 3-3. Test generation time comparison for circuits

Circuit	#Prop.	Our	[79] vs ours	
		Approach	[79]	Speed-
		Time (s)	Time (s)	up
s13207	304	481	568	1.18
s15850	300	241	270	1.13
s35932	640	220	232	1.05
s38417	212	167	210	1.25
s38584	606	2543	3377	1.32
Total	-	3652	4657	1.28

In order to investigate the impact of cluster size on the overall solving time, we also applied our test generation technique on circuit s3854 with different cluster size. There are 606 properties to be checked on the design. The results are shown in Figure 3-6. It can be seen that although the solving time still decreases at the beginning when larger cluster size is used, it might not always be optimal to cluster all properties into a single group. The reason is that too large cluster size may cause many forwarded clauses to be accumulated in the SAT solver's database, as indicated by the solid curve in Figure 3-6B. Too many forwarded clauses can mislead the searching process of the SAT solver, which will eventually increase the overall solving time.

3.4 Summary

Automatic generation of directed tests is promising for simulation based functional validation because it requires less number of test vectors to achieve the same coverage requirement. However, its applicability is limited due to the capacity restriction of current model checking tools. Existing incremental SAT approaches are suitable only for a single property with unknown bound or for multiple properties with known bounds. We presented an efficient technique for test generation by reusing learned knowledge across multiple properties and different bounds. To enable knowledge sharing among properties as well as bounds, we presented a synchronized test generation technique

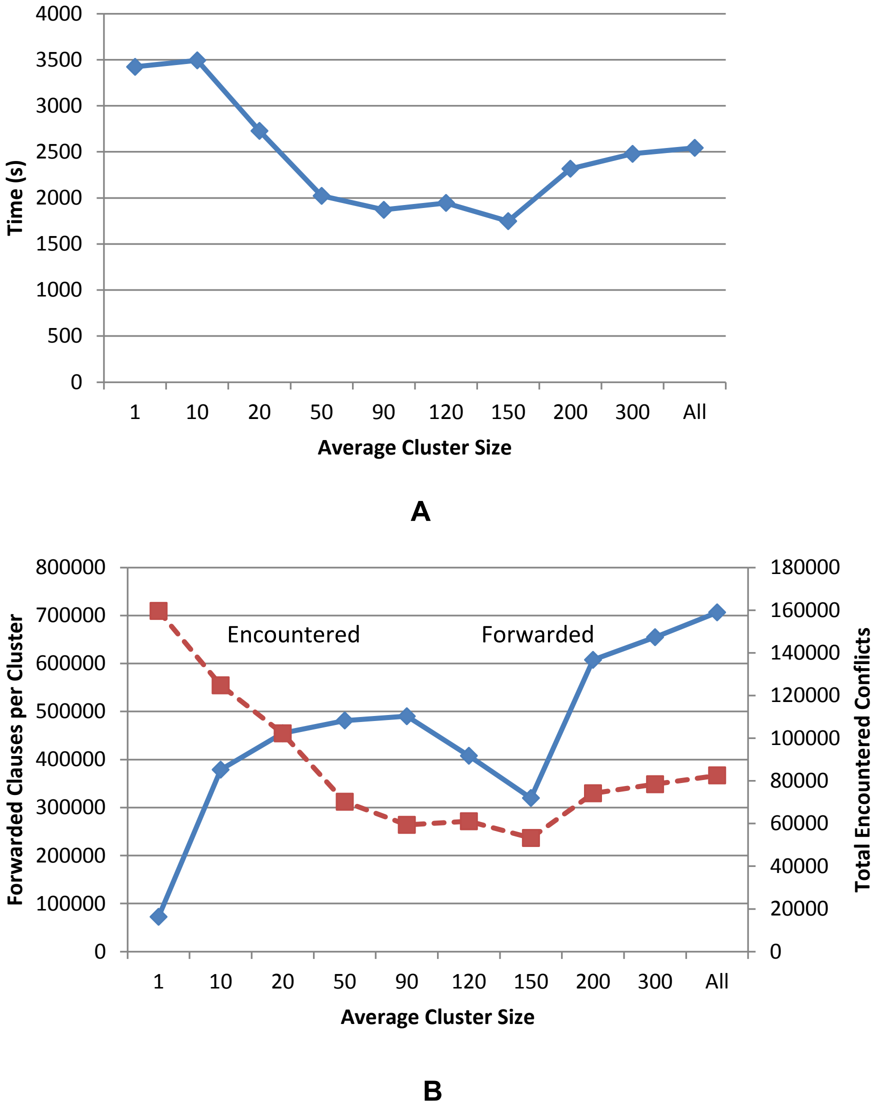

Figure 3-6. Test generation for circuits using different cluster size. A) Time consumption. B) Forwarded clauses and encountered conflicts.

for multiple properties with different bounds. SAT instances for different properties are solved together, so that the discovery and utilization of the common conflict clauses can be maximized. The overall time consumption of checking multiple properties using our approach is remarkably smaller than the summation of time to check each property independently. Our experimental results on both hardware and software designs demonstrated an order-of-magnitude reduction in overall test generation time.

CHAPTER 4
EFFICIENT TEST GENERATION FOR MULTICORE ARCHITECTURES

Chapter 3 explored how to reuse the knowledge during test generation in single-core deisgns. When SAT-based BMC is applied to generate directed tests for multicore architectures, there are two different categories of symmetry in the corresponding SAT instances. The first category is the “temporal” symmetry. It occurs because the SAT instance is encoded by unrolling the same architecture for multiple times. This regularity has already been exploited by existing research [79] to accelerate the SAT solving process. On the other hand, the structural similarity of multiple cores also introduces a second category of symmetry or “spatial” symmetry. This symmetry appears among the CNF clauses for different cores at the same time step. Intuitively, we can also exploit spatial symmetry by reusing the knowledge obtained from one core to other cores. Unfortunately, this intuitive reasoning is hard to implement because it is very difficult to reconstruct the symmetry from the CNF formula. The high level information is lost during CNF synthesis, and it is inefficient as well as computationally expensive to recover through “reverse engineering” methods.

In this work [69], we address the directed test generation for multicore architectures by developing a novel BMC based test generation technique, which enables the reuse of learned knowledge from one core to the remaining cores in the multicore architecture. Instead of direct synthesis of the CNF for the multicore design, we compose the CNF description of the entire design using CNF formulae for cores and the memory subsystem. Since the CNF representation of cores are generated by performing variable substitution of the CNF for one of them, the correct mapping information is easily obtained. In this way, we are able to translate and reuse the conflict clauses learned on any core to other cores. We prove that the CNF description generated by our approach has the same satisfiability as original methods. Our experimental results demonstrate that our approach can remarkably reduce the overall test generation time.

The rest of the chapter is organized as follows. Section 2 describes our test generation methodology for multicore architectures. Section 3 presents our experimental results. Finally, Section 4 summarizes the chapter.

4.1 Test Generation for Multicore Architectures

Our work is motivated by previous works on incremental SAT-based BMC [79]. Based on the temporal symmetry between different bounds, these methods accelerate the SAT solving process by passing the knowledge (deduced conflict clauses) in the temporal direction. Nevertheless, the SAT instances generated by multicore designs also exhibit remarkable spatial symmetry. Figure 4-1 depicts the high level structure of a system with 2 cores. Both cores are identical[1] and connected to memory subsystem with a bus. Figure 4-2 shows the SAT solving process when we perform BMC for bounds 0, 1, 2, and 3 on this multicore architecture using the technique proposed in [79]. We use solid dots to represent different SAT instances and lines to indicate the conflict clause forwarding paths. Although different cores have identical structures, this spatial symmetry is not exploited.

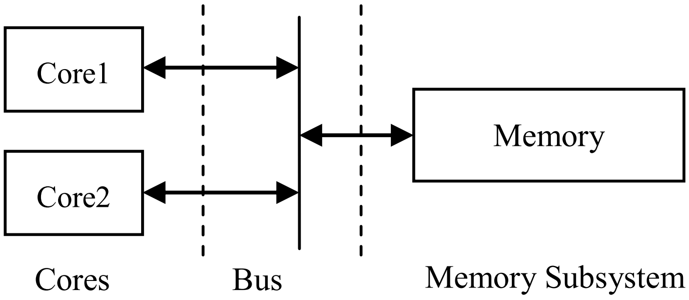

Figure 4-1. Abstracted architecture of a two core system

Intuitively, it should be beneficial if the knowledge or conflict clauses can also be shared “vertically” among different cores as shown in Figure 4-3, because the

[1] We first discuss our approach in the context of homogeneous cores. The application of our approach on heterogeneous cores will be presented in Section 4.1.3.

solving effort spent on a single core can be reused by other cores to save overall time consumption. Unfortunately, the spatial symmetry is difficult to recover from the CNF representation of the SAT instance. The reason is that most clauses contain auxiliary variables introduced during the CNF encoding process. Since these auxiliary variables are unlabeled, the correspondence between clauses from different cores cannot be established directly. Although the spatial symmetry can be partially recovered by solving a graph automorphism problem [3, 5, 30], it may require impractical time for large designs, because no polynomial time solution is found for graph automorphism problem. The underlying reason for this dilemma is that the high level information is lost after the CNF encoding. In other words, a single flattened CNF SAT instance is not suitable to exploit the spatial symmetry.

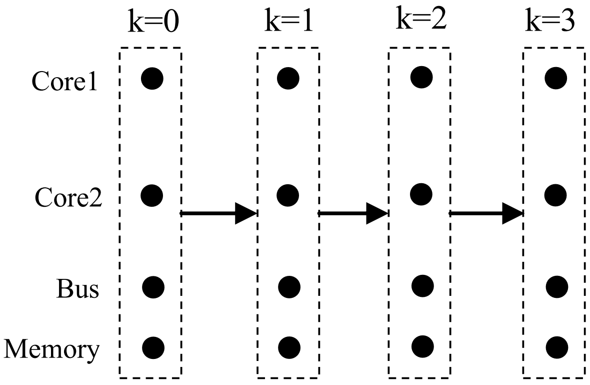

Figure 4-2. Incremental SAT solving technique [79]

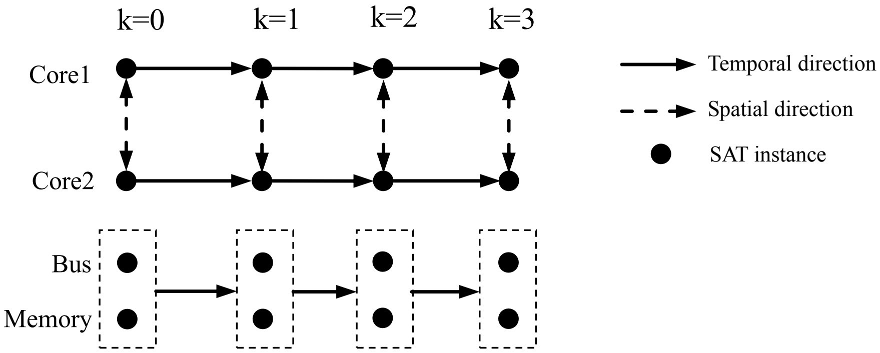

Figure 4-3. Test generation for multicore architectures

Instead of using a monolithic CNF as input, our approach solves this problem by composing the CNF description of the system using CNF formulae for one core, bus and the memory subsystem. Since the cores are identical, their CNF representations are identical as well. We just need to perform variable name substitution to obtain the CNF for all other cores. As shown in Theorem 4.1, when the state variables are substituted by the correct names, the system CNF composed by these replicated CNF for cores, bus as well as memory subsystem will have the same satisfiability behavior as the original monolithic CNF representation. Since both the state variables and auxiliary variables in replicated cores are assigned by our algorithm, it is easy to obtain the correct mapping between variables and clauses in different cores. The spatial symmetry can then be effectively exploited during the SAT solving process. Before we describe our algorithm in details, we first introduce some notations.

Definition 1. ***Symmetric Component (SC)*** *is a set of identical finite state machines (FSM). For the j^{th} FSM within a SC, we denote its initial condition and transitional constraints as $I(s^{is}_{0,j})$ and $R(s^{is}_{i,j}, s^{in}_{i,j}, s^{is}_{i+1,j}, s^{out}_{i+1,j})$ $(0 \leq i \leq k-1)$, where $s^{in}_{i,j}, s^{out}_{i+1,j}, s^{is}_{i,j}$ are its input variables, output variables, and internal state variables at the i^{th} $(i + 1^{th})$ time step. It should be noted that a symmetric component itself can also be viewed as FSM, whose input and output variables are the collection of all the input and output variables of FSMs within it.*

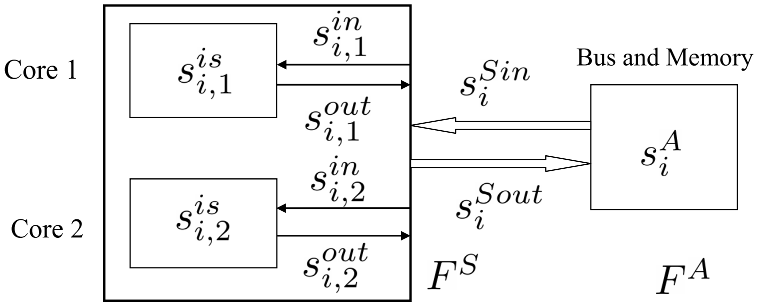

Figure 4-4. FSM representation of Figure 4-1 at time step i

In a multicore system with N_S identical cores, we model the set of all cores as a symmetric component F^S. Other asymmetric components, such as bus and memory subsystem, are modeled as a single finite state machine F^A. We also map the input and output of F^A to the output and input of F^S so that different cores can perform communication through bus and memory subsystem. Formally, we denote the initial condition and transition constraints of F^A as $I(s_0^A)$ and $R(s_i^A, s_i^{Sout}, s_{i+1}^A, s_{i+1}^{Sin})$ $(0 \leq i \leq k - 1)$, where s_i^A represent internal state variables in bus and memory subsystem at the i^{th} time step. Moreover, $s_i^{Sin} = \{s_{i,j}^{in} | 1 \leq j \leq N_S\}$ and $s_i^{Sout} = \{s_{i,j}^{out} | 1 \leq j \leq N_S\}$ are the input and output variables of the symmetric component F^S, which is the combination of the inputs and outputs of all cores. For example, Figure 4-4 shows the FSM representation of the system in Figure 4-1. The symmetric component F^S is composed of core 1 and core 2. The rest of the system is represented by F^A. In the i^{th} time step, the internal state variable of F^S are $\{s_{i,1}^{is}, s_{i,2}^{is}\}$ and s_i^A. The input and output variables of F^S (also the output and input variable of F^A) are $s_i^{Sin} = \{s_{i,1}^{in}, s_{i,2}^{in}\}$ and $s_i^{Sout} = \{s_{i,1}^{out}, s_{i,1}^{out}\}$, respectively.

The BMC formula of the multicore system can be expressed as

$$\begin{aligned} BMC(M,p,k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i) \\ = I(s_0^A) \wedge \bigwedge_{j=1}^{N_S} I(s_{0,j}^{is}) \wedge \bigwedge_{i=0}^{k-1} (R(s_i^A, s_i^{Sout}, s_{i+1}^A, s_{i+1}^{Sin}) \\ \wedge \bigwedge_{j=1}^{N_S} R(s_{i,j}^{is}, s_{i,j}^{in}, s_{i+1,j}^{is}, s_{i+1,j}^{out})) \wedge \bigvee_{i=0}^{k} \neg p(s_i) \end{aligned}$$

The basic idea of our approach is to generate CNF formula

$$\begin{aligned} BMC'(M,p,k) = CNF_I^A \wedge \bigwedge_{j=1}^{N_S} CNF_I^S(j) \\ \wedge \bigwedge_{i=0}^{k-1} (CNF_R^A(i) \wedge \bigwedge_{j=1}^{N_S} CNF_R^S(i,j)) \wedge CNF^p(k) \end{aligned}$$

Algorithm 3: Test Generation for Multicore Architectures

```
    Input: CNF formulae $CNF_I^A$, $CNF_I^S(1)$, $CNF_R^A(i)$, $CNF_R^S(i,1)$, $CNF^p(k)$,
           Number of cores $N_S$, Maximum bound $K_{max}$,
    Output: Test $test_p$
    Bound $k \longleftarrow 0$;
    Initialize variable mapping table $T$;
    Common Clause Set $CCS \longleftarrow \emptyset$;
    Generate $CNF_I^S(j)$ using $CNF_I^S(1)$ for $1 < j \le N_S$;
    Add Clauses in $CNF_I^S(j)$ to $CCS$ for $1 \le j \le N_S$;
    Update $T$;
    Add Clauses in $CNF_I^A$ to $CCS$;
    while $k \le K_{max}$ do
        Generate $CNF_R^S(k,j)$ using $CNF_R^S(k,1)$ for $1 < j \le N_S$;
        Add Clauses in $CNF_R^S(k,j)$ to $CCS$ $1 \le j \le N_S$;
        Update $T$;
        Add Clauses in $CNF_R^A(k)$ to $CCS$;
Step1:  $(ConflictC, test_p) \longleftarrow$ SAT($CCS \bigcup CNF^p(k)$,$T$);
Step2:  $CCS \longleftarrow CCS \bigcup$ Filter($ConflictC$);
        if $test_p \neq null$ then return $test_p$;
        $k \longleftarrow k + 1$;
    end
```

and perform SAT solving on $BMC'(M, p, k)$ instead of solving the CNF formula directly synthesized from $BMC(M, p, k)$, where CNF_I^A, $CNF_I^S(j)$, $CNF_R^A(i)$, $CNF_R^S(i,j)$ and $CNF^p(k)$ are the CNF representations of $I(s_0^A)$, $I(s_{0,j}^{is})$, $R(s_i^A, s_i^{Sout}, s_{i+1}^A, s_{i+1}^{Sin})$, $R(s_{i,j}^{is}, s_{i,j}^{in}, s_{i+1,j}^{is}, s_{i+1,j}^{out})$ and $\bigvee_{i=0}^{k} \neg p(s_i)$, respectively.

Algorithm 3 shows our test generation method for multicore architectures. It accepts the CNF representation of one core, bus, the memory subsystem as well as the properties at different time steps as inputs and produces corresponding directed tests. As indicated before, we first generate the CNF representations of the initial condition and transition constraints of all other FSMs in F^S based on the input CNF formulae $CNF_I^S(1)$ and $CNF_R^S(i,1)$, which are the initial condition and transition constraints of the first FSM (Core 1). It is accomplished by replacing variable in $CNF_I^S(1)$ and $CNF_R^S(i,1)$ with corresponding variables for other FSMs (cores). At the same time, we maintain a table T [2] to record the symmetric set of variables for both state variables and auxiliary variables. After that, we invoke the SAT solving process on the conjunction of clauses in CCS and $CNF^p(k)$, which is equivalent to $BMC'(M, p, k)$ defined above. Next, we perform the following 2 steps.

1. During SAT solving, analyze any conflict clause cls found by the SAT solver. If cls is purely deduced by the clauses which belong to a single FSM, replicate and forward cls to all other FSMs. This is implemented by substituting the variables in cls by their counterparts for each FSM in F^S based on table T. At the same time, we also replicate the cls in temporal direction, as discussed in [79].
2. After the solving process, only keep new conflict clauses that are deduced independent of $CNF^p(k)$, and merge them into CCS.

If the satisfied assignment, or a counterexample $test_p$ is found in step 1, the algorithm returns it as a test. Otherwise, the algorithm repeats for each bound k until the maximum bound is reached.

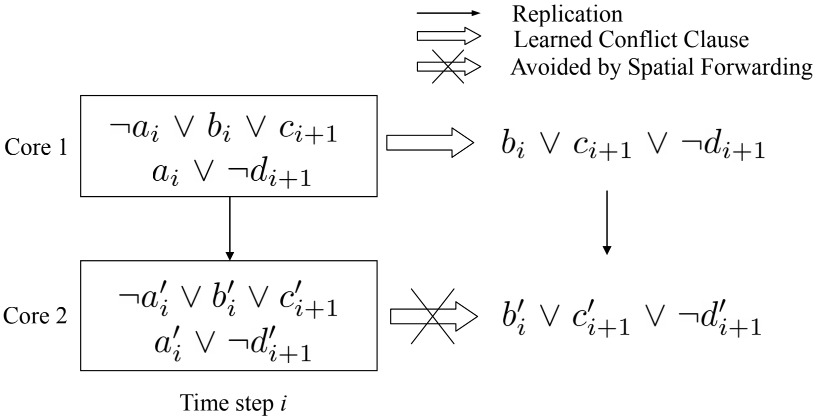
Replication
Learned Conflict Clause
Avoided by Spatial Forwarding
Core 1
$\neg a_i \vee b_i \vee c_{i+1}$
$a_i \vee \neg d_{i+1}$
$b_i \vee c_{i+1} \vee \neg d_{i+1}$
Core 2
$\neg a'_i \vee b'_i \vee c'_{i+1}$
$a'_i \vee \neg d'_{i+1}$
$b'_i \vee c'_{i+1} \vee \neg d'_{i+1}$
Time step i

Figure 4-5. Test generation for multicore architectures

We use the same example in Figure 4-1 to illustrate the flow of Algorithm 1. The two different clause forwarding paths employed in our approach are shown in Figure 4-5. Suppose $(\neg a_i \vee b_i \vee c_{i+1})$ and $(a_i \vee \neg d_{i+1})$ are two clauses within $CNF_R^S(i, 1)$ (transition

[2] As discussed in Section 4.1.2, a physical table is not required, instead a mapping function is used in our framework.

constraint of Core 1), in the first iteration for $k = 0$, two clauses $(\neg a'_i \vee b'_i \vee c'_{i+1})$ and $(a'_i \vee \neg d'_{i+1})$ will be produced during the generation of $CNF_R^S(i, 2)$ (transition constraint of Core 2). In the subsequent SAT solving process, suppose a conflict clause $(b_i \vee c_{i+1} \vee \neg d_{i+1})$ is deduced based on $(\neg a_i \vee b_i \vee c_{i+1})$ and $(a_i \vee \neg d_{i+1})$, it will be forwarded to Core 2, because its two parent clauses are all from the CNF formula for Core 1. Therefore, $(b'_i \vee c'_{i+1} \vee \neg d'_{i+1})$ can now be used by Core 2 to prevent the partial assignment $\{b'_i, c'_{i+1}, d'_{i+1}\} = \{0, 0, 1\}$, which will result in a conflict on a'_i. Such forwarding of conflict clauses is not possible using Strichman's approach [79], which only considers temporal symmetry but not spatial symmetry.

In the remainder of this section, we prove the correctness of our approach and discuss the implementation details of our directed test generation algorithm for multicore architectures.

4.1.1 Correctness of Our Proposed Approach

To prove the correctness of our test generation approach, we need to ensure that the produced CNF formula $BMC'(M, p, k)$ in Algorithm 3 has the same satisfiability as $BMC(M, p, k)$.

Theorem 4.1. *$BMC(M, p, k)$ and $BMC'(M, p, k)$ have the same satisfiability.*

Proof. Clearly, we have

$$BMC(M, p, k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i)$$

$$= I(s_0^A) \wedge \bigwedge_{j=1}^{N_S} I(s_{0,j}^{is}) \wedge \bigwedge_{i=0}^{k-1} (R(s_i^A, s_i^{Sout}, s_{i+1}^A, s_{i+1}^{Sin})$$

$$\wedge \bigwedge_{j=1}^{N_S} R(s_{i,j}^{is}, s_{i,j}^{in}, s_{i+1,j}^{is}, s_{i+1,j}^{out})) \wedge \bigvee_{i=0}^{k} \neg p(s_i)$$

By their definitions, CNF formulae CNF_I^A, $CNF_I^S(j)$, $CNF_R^A(i)$, $CNF_R^S(i, j)$ and $CNF^p(k)$ are CNF representation of propositional formulae $I(s_0^A)$, $I(s_{0,j}^{is})$, $R(s_i^A, s_i^{Sout}, s_{i+1}^A, s_{i+1}^{Sin})$, $R(s_{i,j}^{is}, s_{i,j}^{in}, s_{i+1,j}^{is}, s_{i+1,j}^{out})$ and $\bigvee_{i=0}^{k} \neg p(s_i)$, where $0 \leq i \leq k-1$ and $1 \leq j \leq N_S$.

Therefore, $BMC(M,p,k)$ has the same satisfiability as

$$BMC'(M,p,k) = CNF_I^A \wedge \bigwedge_{j=1}^{N_S} CNF_I^S(j)$$
$$\wedge \bigwedge_{i=0}^{k-1} (CNF_R^A(i) \wedge \bigwedge_{j=1}^{N_S} CNF_R^S(i,j)) \wedge CNF^p(k)$$

because the auxiliary variables introduced during CNF conversion do not change the satisfiability. In other words, $BMC(M,p,k)$ and $BMC'(M,p,k)$ have the same satisfiability. □

In fact, the value of state variables in a satisfying assignment of $BMC'(M,p,k)$ also satisfy $BMC(M,p,k)$ and therefore can be used as a counterexample of the property p. The reason is that the value of the variables in a satisfying assignment of $BMC'(M,p,k)$ will also satisfy all CNF formulae CNF_I^A, $CNF_I^S(j)$, $CNF_R^A(i)$, $CNF_R^S(i,j)$ and $CNF^p(k)$. Thus, the value of the state variables will satisfy corresponding propositional formulae $I(s_0^A)$, $I(s_0^j)$, $R(s_i^A, s_{i+1}^A)$, $R(s_i^j, s_{i+1}^j)$ and $\bigvee_{i=0}^k \neg p(s_i)$. Hence, they together will satisfy $BMC(M,p,k)$, which is a conjunction of above propositional formulae. Therefore, the correctness of our algorithm is justified.

4.1.2 Implementation Details

Our test generation algorithm for multicore architectures is built around NuSMV model checker [18] and zChaff SAT solver [38]. We first model the system using SMV language, then use NuSMV to generate the CNF formulae CNF_I^A, $CNF_I^S(1)$, $CNF_R^A(i)$, $CNF_R^S(i,1)$ and $CNF^p(k)$ in DIMACS format as the input of Algorithm 3. zChaff is employed as the internal SAT solver. In this section, we briefly explain CNF generation process and the implementation of Step 1 and Step 2 in Algorithm 3.

The generation of CNF descriptions for a single core, bus and memory subsystem using NuSMV is straight forward. The only practical consideration is that all variables are represented by their indices in CNF clauses. As a result, it is important to avoid the same index to be used by two different variables. Since NuSMV does not offer any external interface to control the index assignment, we modified the source code to make the index space suitable for our purpose. The basic idea is to make the assignment of indices satisfy the following two constraints: 1) the indices of variables from the same core at the same time step are assigned continuously; 2) the indices of variables of the same time step across cores are assigned continuously as well. For example, in a 2-core system with each core having 100 variables, in time step 1 for core 1 we can use indices from 1-100 (controlled by the first constraint) whereas the second constraint indicates that the variables for core 2 at time step 1 should be 101-200. Therefore, 201-300 can be used to represent variables of core 1 in time step 2, and so on. Based on these two constraints, the computation of the indices of symmetric variables can be efficiently implemented as increasing or decreasing by a certain offset.

During SAT solving, we also need to track the dependency of generated conflict clauses to determine whether they can be forwarded to other cores. This can be easily implemented within zChaff, which provides clause management scheme to support incremental SAT solving. For each clause in its clause database DB, zChaff uses a 32-bit group ID to track the dependency. Each bit identifies whether that clause belongs to a certain group. When a conflict clause is deduced based on clauses from multiple groups, its group ID is a “OR” product of the group ID of all its parent clauses, i.e., this clause belongs to multiple groups. zChaff also allows user to add or remove clauses by group ID between successive solving processes. If one clause belongs to multiple groups, it is removed when any of these groups are removed.

With these mechanisms, the step 1 and 2 in Algorithm 3 can be implemented efficiently as follows:

1. Add clauses in $CNF_I^S(j)$ and $CNF_R^S(i,j)$ with group ID j, $1 \leq j \leq N_S$
2. Add clauses in CNF_I^A, $CNF_R^A(i)$with group ID $N_S + 1$.
3. Add clauses in $CNF^p(k)$ with group ID $N_S + 2$.

4. When a new conflict clause is obtained during SAT solving, if it only belongs to a single group with ID smaller than $N_S + 1$, replicate this clause to all other cores with proper group ID.

5. After solving all clauses in DB with zChaff, remove clauses with group ID $N_S + 2$.

The overhead introduced by dependency identification and tracking in our algorithm is negligible compared to the improvement in SAT solving time. At the same time, since the indices of variables in symmetric cores are carefully assigned, the mapping table T is not maintained explicitly, but implemented as a simple mapping function, which is used to generate forwarding clauses for different cores. In that way, we avoid the potential caching overhead which may deteriorate the performance of the SAT solver.

4.1.3 Heterogeneous Multicore Architectures

So far, we discussed our algorithm using homogeneous cores. This section describes the application of our approach in the presence of heterogeneous cores. In a heterogeneous multicore system, if any two cores are completely different, it is not possible to reduce the test generation time by exploiting the symmetry. However, most real systems usually employ a cluster of identical cores for same computational purpose. In this case, we can first group them into symmetric components based on their types, then apply our algorithm to each symmetric component. For example, in the 5-core system shown in Figure 4-6, core 5 is used for monitoring and core 1-4 are identical cores for computation. We can define core 1-4 as the symmetric component and apply our algorithm on them. In general, we can apply our algorithm on each cluster of identical cores in a system.

However, when the heterogeneous cores are not completely different, i.e., only some functional units in them are different, our proposed algorithm can be employed in a more efficient way. Recall that the FSMs in a symmetric component are not restricted to cores. We can actually define the symmetric component in such a way that it includes only the identical functional units in different cores. For example, Figure 4-7 shows a system with heterogeneous cores. Both of the cores are pipelined with five stages:

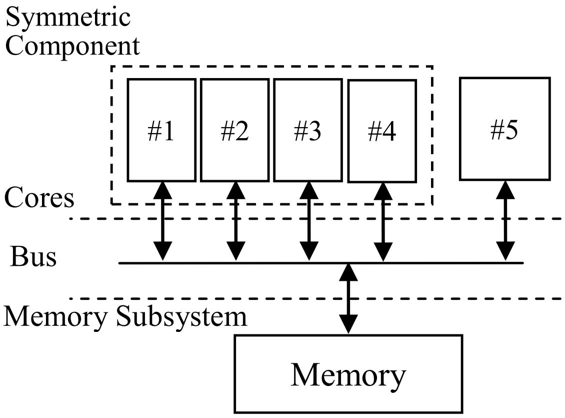

Figure 4-6. Multicore system with different types of cores

fetch, decode, execute, memory access, and writeback. The only difference is that they have different implementation in the execute stage EX. In this case, we define our symmetric component F^Sas the set of all functional units in two cores except EX. These two execution stages as well as bus and memory subsystem are modeled in the asymmetric part F^A. Of course, the input and output of F^S here will include not only the input and output variable of the cores, but also all the interface variables between EX and other stages. In this way, the information learned on all other stages of one core can still be shared by the other core. Clearly, the correctness of our approach is still guaranteed, because the selection of the symmetric component satisfies its definition.

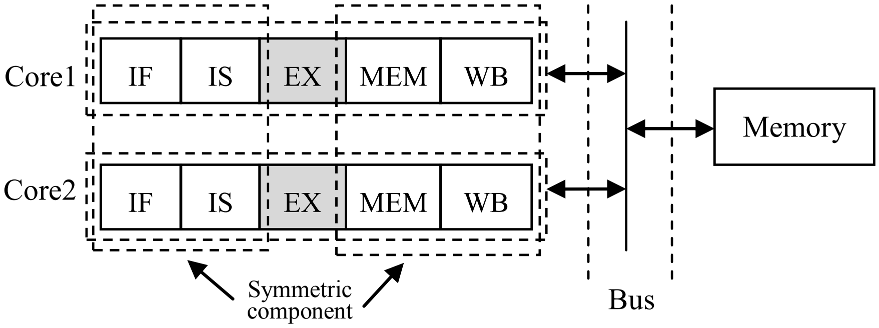

Figure 4-7. Multicore system with different types of execution units

4.2 Experiments

We have evaluated the applicability and usefulness of our test generation technique on different multicore architectures.

4.2.1 Experimental Setup

As described in Section 4.1.2, the designs and properties are described in SMV language and converted to required CNF formulae (DIMACS files) using modified NuSMV [18]. We used zChaff [38] as our SAT solver to implement our test generation algorithm. Experiments were performed on a PC with 3.0GHz AMD64 CPU and 4GB RAM.

First, we present results of our approach using a multicore design that is composed of different number of identical cores, one bus, and memory subsystem. The pipeline inside each core has five stages: fetch, decode, execute, memory access, and writeback. Besides, each core has its own cache, which is connected with the memory through the bus. Next, we will present (in Figure 4-10) the applicability of our approach on heterogeneous multicore architectures.

In order to activate the desired system behaviors, we used different number of properties on designs with different complexity. For instance, we used 375 properties in case of 16 core design that trigger two simultaneous activities between cores. We have also used several properties that involves multicore interactions. For example, one test will activate the following scenario: “if the value in a memory location which is initialized as one by core 1, is increased by one by all other cores, it should be equal to the number of cores when it is readback by core 2”. It should be noted that the corresponding property is not symmetric with respect to all cores.

4.2.2 Results

We compared our approach with Strichman’s approach [79] and original BMC [28]. Each approach was used to solve a sequence of SAT instances for the same property with varying bounds until a satisfiable instance is found. The input SAT instances for

Strichman's approach and the original BMC was directly synthesized from $BMC(M, p, k)$ to improve their performance. When our approach was applied, we performed the SAT solving on $BMC'(M, p, k)$ as indicated in Section 4.1. We also tried to compare with [3]. Unfortunately, the implementation [4] failed to produce the symmetry breaking predicates due to the large size of our input CNF (more than 600k clauses).

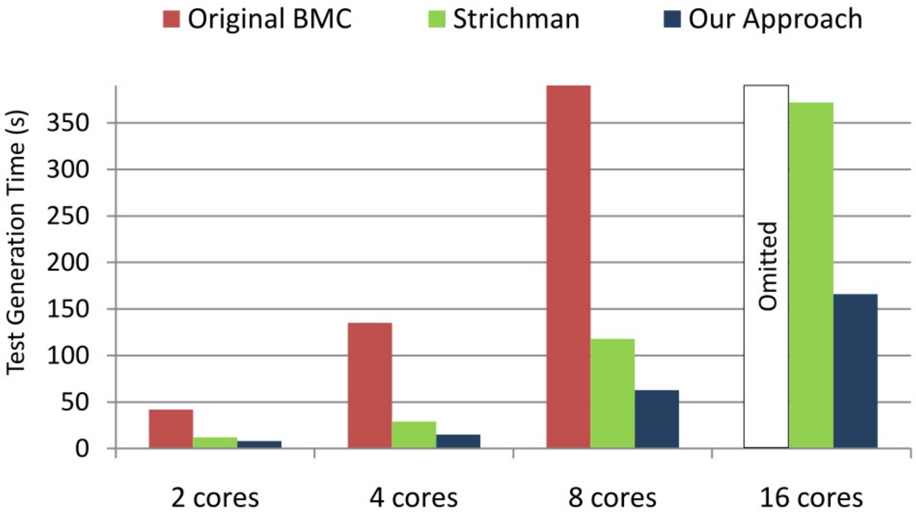

Figure 4-8. Test generation time with different number of cores

Figure 4-8 presents the average test generation time for different number of cores. The original BMC failed to produce results within 3000 seconds on several properties for the 16 core system. Therefore, its time is omitted. As expected, the time consumption increases with the number of cores. Both our approach and Strichman's approach [79] are remarkably faster than original BMC [28]. By effective utilization of both spatial and temporal symmetry, our approach outperforms [79] (which only considers temporal symmetry) by nearly 2 times.

Table 4-1 shows a more detailed comparison of different approaches on the 8 core system for 10 most time consuming properties. The first column represents the names of properties used. The second column shows the corresponding bounds or time steps to activate each property. The next three columns present the test generation time (in seconds) for each property using the original BMC [28], Strichman's approach [79],

Table 4-1. Test generation time for 8 core system

Prop.	Bound	[28] Time(s)	[79] Time(s)	Our Approach	Speedup over [28]	Speedup over [79]
1	28	79	56	25	3.16	2.24
2	22	67	44	21	3.19	2.10
3	32	93	62	30	3.10	2.07
4	28	208	94	17	12.24	5.53
5	33	*	342	148	-	2.31
6	20	413	124	47	8.79	2.64
7	20	*	125	48	-	2.60
8	**23**	**883**	**140**	**63**	**14.02**	**2.22**
9	25	2106	157	128	16.45	1.23
10	25	1991	106	101	19.71	1.05
Total	-	5840	1250	628	9.30	1.99

* represent run times exceeding 3000 sec.

Table 4-2. Detailed test generation information

k	[79]				Our approach			
	#Cls in DB	#Decision	#Fwd Cls	Time(s)	#Cls in DB	#Decision	#Fwd Cls	Time(s)
19	721427	40045	25608	2.4	756149	21231	4441	1.2
20	762855	71854	27329	3.6	857103	30049	26685	2.7
21	827272	56692	22824	3.4	900428	35687	24534	3.1
22	893382	203112	102202	15.4	965925	30873	6834	1.9
23	954998	2652411	142585	97.3	1029266	1228603	261989	52.8
Total	-	3024114	320548	122.1	-	1346443	324483	61.7

and our approach, respectively. The time is calculated as the summation of the time to solve all the SAT instances from $k = 0$ to the bound of the property. The time calculation also includes the time consumed by non-SAT-solving steps in Algorithm 3. The last two columns indicate the speedup of our approach over [28] and [79]. It can be seem that our approach outperforms [79] by two times and [28] by an order of magnitude.

To inspect the reason of our improvement over [79], we analyze the behavior of the SAT solver. Table 4-2 shows details of the last five SAT instances immediately before the bound was found during the BMC of property 8 on the 8-core system (highlighted entry in Table 4-1). The first column in Table 4-2 is the time step of each SAT instance. The next four columns contain the real size of the clause database before the solving process, the number of decisions made by zChaff, the number of forwarded conflict clauses and the time consumption in [79]. Similar information of our approach is represented in the last four columns. Compared to [79], the total number of decisions made by the SAT solver is much smaller when our approach is applied. At the same time, the number of forwarded clauses are comparable. In other words, our approach saves the time to rediscover the same knowledge for each core, without the overhead of forwarding too many conflict clauses.

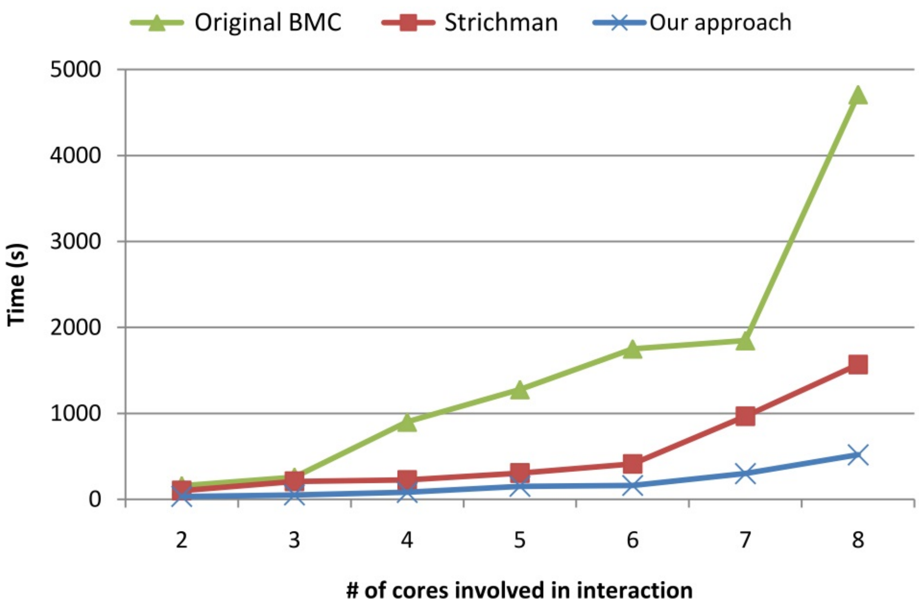

Figure 4-9. Test generation time with different interactions

We also investigated the impact of different number of cores involved in the interaction on the test generation time. In this experiment, we use a processor with eight 3-stage cores. They are connected to the memory subsystem using snoopy protocol. The desired test should trigger all cores perform read and write operation on the same shared memory variable in certain order. The results are given in Figure 4-9. When the interaction involves only a small number of cores, the difference in test generation time of [28], [79], and our approach is quite small. However, when more and more cores are involved, our approach outperforms both [28] and [79] remarkably, due to the usage of symmetry information.

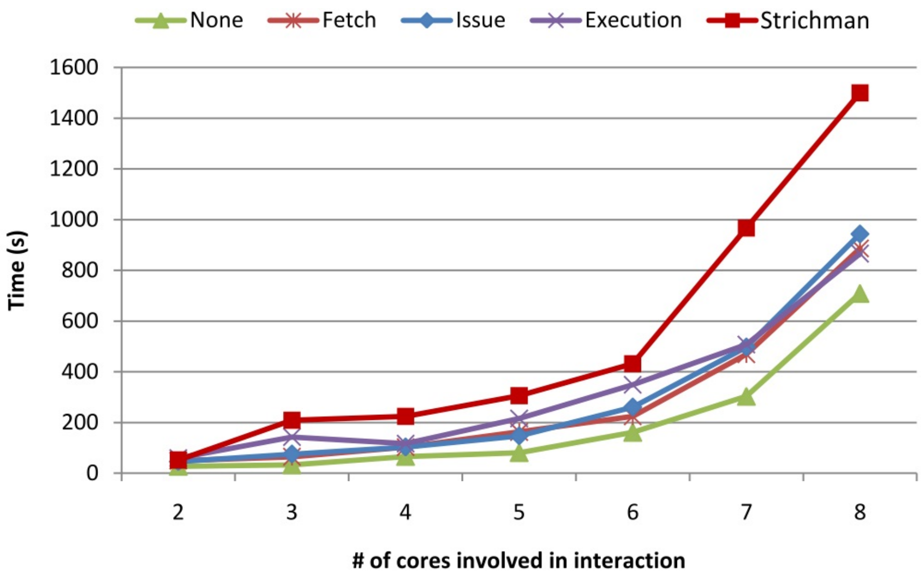

Figure 4-10. Test generation time with heterogeneous cores

Finally, to illustrate the effectiveness of our approach in a more general scenario, we measure the test generation time on a system with heterogeneous cores. We use cores with different implementations in their fetch, issue, execution stages, and repeat the previous test generation experiment. As discussed in Section 4.1.3, we only replicate learned conflict clauses within the symmetric components. Figure 4-10 shows the result. The "fetch" curve corresponds to a system where the 8 cores are identical except their fetch stages. Similarly, curves marked as "Issue" and "Execution" represent cores with different issue and execution stages, respectively. We also show the test generation time for homogeneous cores using our approach ("None") and [79] as reference. It can be observed that due to less scope of knowledge reuse, the time consumption of our approach for heterogeneous cores are generally larger than homogeneous cores. Nevertheless, our approach still outperforms [79] especially for complicated interactions involving many cores.

4.3 Summary

Functional verification of multicore architectures is challenging due to the increased design complexity and reduced time-to-market. Existing incremental SAT approaches have only exploited the symmetry in BMC across different time steps. We presented a novel approach for directed test generation of multicore architectures that exploits both spatial and temporal symmetry in SAT-based BMC. The CNF description of the design is synthesized using CNF for cores, bus and memory subsystem to preserve the mapping information between different cores. As a result, the symmetric high level structure is well preserved and the knowledge learned from a single core can be effectively shared by other cores during the SAT solving process. The experimental results using homogeneous as well as heterogeneous multicore architectures demonstrated that the test generation time using our approach is remarkably smaller (2-10 times) compared to existing methods.

CHAPTER 5
VALIDATION OF CACHE COHERENCE PROTOCOLS

Caching has been the most effective approach to reduce the memory access time for several decades. When the same data is cached by different processors, cache coherence protocols are employed to coordinate the accesses and guarantee that the most recent written data is returned. As the protocols are growing more and more complex, the verification teams are facing significant challenges to achieve the required coverage within tight time-to-market window.

Since all possible behaviors of the cache blocks in a system with n cores[1] can be defined by a global finite state machine (FSM), the entire state space is the product of n cache block level FSMs. Intuitively, full state or transition coverage can be achieved by performing a breadth first search (BFS) on this product FSM. The path that leads to each distinct state from the initial state can be used as a test case for that state. Unfortunately, since each test is used to activate only one transition, a large number of transitions may be unnecessarily repeated, if they are on the shortest path to many other transitions. Therefore, it is desirable to replace BFS with another efficient algorithm, which creates an input sequence that covers all transitions with minimum transition overhead. Since the number of directed tests can be quite large in many practical scenarios,it may be beneficial to generate the directed tests on-the-fly, so that the created tests can be directly fed to the simulator or the device under test without extra storage requirement. Clearly, the development of such algorithms requires a clear understanding of the state space of the complex global FSM. Although the FSM of each cache controller is easy to understand, the structure of the product FSM for

[1] In this chapter, we use the term “**core**” to refer to each single processing units in multicore or multiprocessor systems.

modern cache coherence protocols can have quite obscure structure that can be hard to analyze.

In work [70], we propose an on-the-fly test generation for cache coherence protocols by analyzing the state space structure of their corresponding global FSMs. Instead of using structure-independent BFS to obtain the directed tests, we show that the entire complex state space can be decomposed into several components with simple structures. Since the activation of state and transition can be viewed as a path searching problem in the state space, these decomposed components with known structures can be exploited for efficient test generation. Our contributions are:

1. We develop a graphical description of the state space structure of several commonly used cache coherence protocols that can be viewed as a composition of simple structures.
2. We present an on-the-fly directed test generation algorithm based on the Euler tour of hypercubes. Our approach only requires linear space requirement with respect to the number of cores. The generated test forms a tour in the state space of corresponding global FSM, which activates all possible transitions of the global FSM with small overhead.

The rest of this chapter is organized as follows. Section 5.1 provides related background information. Section 5.2 describes our contributions in details. Experimental results are presented in Section 5.3. Finally, Section 5.4 concludes the chapter.

5.1 Background and Motivation

In modern computer systems, since the latency to transfer data from the main memory to processing units is much larger than the time consumption for computation, each processing unit usually maintains its local copy of the main memory, or cache for fast access. One major problem of caching is that when the same data, memory block, is cached in two or more different places, any future modification to it should be propagated to all the cached copies. Otherwise, it can lead to incorrect functional behaviors. Cache coherence protocols are therefore proposed to define the correct behavior of each cache controller, when different processing units issue loads and stores to the same memory location.

One of the simplest cache coherence protocol is the MSI snoopy protocol [42]. The behavior of the cache controller in a processing unit is modeled as a finite state machine (FSM). Figure 5-1 shows the state transition diagram of MSI protocol. The state of a cache block (line) can be either “Invalid”(I), “Modified”(M), or “Shared”(S). At the beginning, all cache blocks are in the invalid state. When a load request arrives from the core side (Self LD), the cache controller will request the data from the main memory and switch to shared state. When the core issues a store request (Self ST), the cache controller will first broadcast an invalidated request on the bus and then change to modified state. Such an invalidate request will inform all other cache controllers that are in shared or modified states to change to invalid state. A cache block may also change to invalid state, when it is evicted by another cache block which is mapped to the same location in the cache, or other cores issue store requests (Other ST).

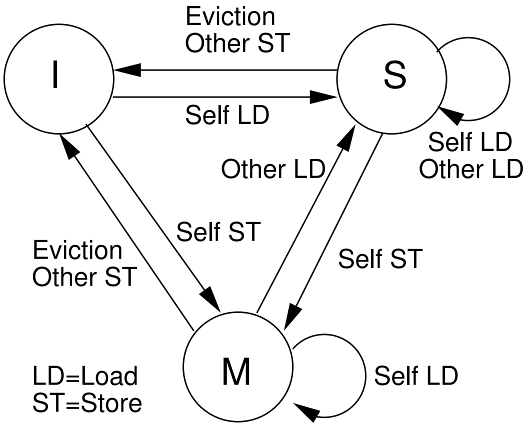

Figure 5-1. State transitions for a cache block in MSI protocol

Although MSI protocol is enough to guarantee the coherence of the cache system, it causes some unnecessary delay and traffic on the communication channels. Many variants of the MSI protocols are invented to further improve its performance. For example, “Exclusive” (E) state is introduced in MESI protocol to avoid the traffic when a cache block is only used by one core. “Owned” (O) state is used in MOSI and MOESI protocol to reduce the delay when a modified block is loaded by other cores.

As cache coherence protocols are becoming more and more complex, it is getting harder to verify their implementations. From the validation perspective, it is always desirable to activate all possible state transitions of the entire multicore cache system. In other words, it is necessary to have a high state and transition coverage (100%, if possible) in the global FSM of the entire memory (cache) subsystem.

5.2 Test generation for Transition Coverage

Our approach is motivated by the basic Breadth First Search (BFS) in the state space of a global FSM. Given the FSM description of any cache coherence protocol, it is possible to compose a test suite which can activate all states and transitions using two steps: 1) for each state, we find out the instruction sequences to reach it by performing a BFS on the global FSM; and 2) for each transition, we create the test by appending the required instructions after the instruction sequences to reach the initial state of this transition. However, such a naive approach has two problems. First, transitions close to the initial state are visited for many times. Thus, a large portion of the overall test time is wasted. Secondly, it is difficult to generate tests on-the-fly, because the memory requirement to run the BFS routine is quite large. Since we have to remember all visited states in BFS process, its runtime memory requirement also grows exponentially.

To address these challenges, our approach needs to satisfy two requirements: 1) we should reduce the number of transitions as much as possible without sacrificing the coverage goal; and 2) the space requirement for the test generation algorithm should be small. Fortunately, due to the highly symmetric and regular structure of the state space, it is possible to design a deterministic test generation algorithm, which can efficiently activate all states and transitions of popular cache coherence protocols. The basic idea is to divide the complex state space into several large hypercubes and other small components. Since hypercubes can be traversed with no extra overhead, a large number of unnecessary transitions can be avoided during activating all transitions.

In the rest of this section, we first describe how to generate tests to activate all transitions of a simplified cache coherence protocol: SI protocol. Next, we discuss our test generation techniques for a variety of popular protocols including MSI, MESI, MOSI, MOESI, and MESIF. In this work, we focus on the transition between two stable states. We assume that the transition between stable states to transient state are correct.

5.2.1 SI Protocol

SI protocol is a trimmed version of MSI protocol, in which we do not allow cores to issue store operation. For a system with n cores, a valid global state of the system allows the cache blocks in any m cores in I state and cache blocks in the other $n - m$ cores in S state. Thus, there are 2^n valid global states. Besides, since any core in I (or S) state can be converted to S (or I) state within one transition, there are n outgoing and n incoming edges. It is easy to see that the entire state space of SI protocol with n cores is a n dimensional hypercube[2] . Figure 5-2 shows such a state space with three cores. *Since all edges are bidirectional for state transitions, we do not show transition directions explicitly.* For example, state III can be transformed to IIS when the first core loads the cache block. Similarly, state IIS can also be transformed to III, when the first core evicts this cache block.

To achieve full state and transition coverage of the state space, we need to traverse each edge of the hypercube at least once in both directions. Since each global state has

[2] There are many transitions that start and end in the same states. For example, the global state will not change if a core in S state issues a load operation. Usually, these transitions are easier to cover, because they can be activated by appending one more operation at the end of existing tests, which are used to activate corresponding initial states. As a result, we omit them in the state space structure description in this section. However, all possible transitions are considered in the actual implementation of our test generation approach as well as in the experimental results.

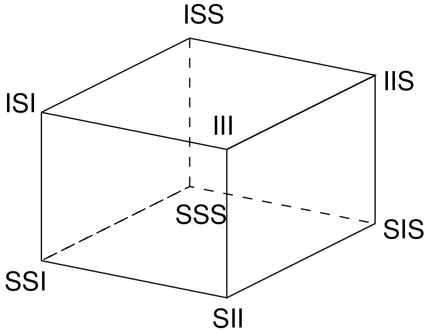

Figure 5-2. Global FSM state space of SI protocol with 3 cores

the same number of incoming and outgoing edges, it is possible to form an Euler tour [35] of the state space, which visits each edge exactly once in both directions.

Algorithm 4: Test generation for SI protocol with n cores

```
$CreateTestsSI(n)$
for $i = 0$ to $n - 1$ do
    Output “load(i)”
    $VisitHypercube(1, n - 1, i)$
    Output “evict(i)”
end for

$VisitHypercube(id, m, shift)$
for $i = 1$ to $m$ do
    $newid = id + 2^i$
    $p = (i + shift) \mod n$
    Output “load(p)”
    if $i > 1$ then
        $VisitHypercube(newid, i - 1, shift)$
    end if
    Output “evict(p)”
end for
return ;
```

Algorithm 4 shows our test generation algorithm for SI protocol, which performs an Euler tour on a n dimensional hypercube. Here, *load(p)/evict(p) means the p^{th} core performs a load/evict operation in a particular cycle, while all other cores remain idle*. We use the state space in Figure 5-2 to show the execution of Algorithm 4. The algorithm starts by calling $CreateTestsSI(n)$. All cores are in I state at the beginning.

In the first round of the *for* loop in line 2, the system first perform transition III-IIS by executing load(0). During $VisitHypercube$, we will first visit transition IIS-ISS and ISS-IIS for $i = 1$ and IIS-SIS for $i = 2$. Since $i > 1$, we invoke $VisitHypercube$ at line 6, which activates two transitions: SIS-SSS and SSS-SIS. Next, transition SIS-IIS is covered by executing evict(2) in line 7 of $VisitHypercube$. Finally, the global state goes back to III via transition IIS-III after evict(2) in line 5 of $CreateTestsSI$. In the next two rounds of the *for* loop in $CreateTestsSI$, we are essentially performing a "rotated" version of the previous traversal, which are going to cover all transitions in paths III-ISI-SSI-ISI-ISS-SSS-ISS-ISI-III and III-SII-SIS-SII-SSI-SSS-SSI-SII-III. Eventually, all transitions in the hypercube are covered by the generated test sequences.

Although the execution of Algorithm 4 seems to be complicated for larger n, the basic idea of this algorithm is quite easy: the hypercube is actually partitioned into n isomorphic trees with no overlapping edges. Once the hypercube is correctly partitioned, an Euler tour is performed on trees, because all edges are bidirectional. The correctness of our algorithm can be proved as follows.

First, we show that the total number of transitions in an SI protocol with n cores is $n * 2^n$. Notice that an n dimensional hypercube has $n * 2^{n-1}$ edges. Since each edge corresponds two transitions in the SI protocol, the total number of transitions becomes $n * 2^n$.

Next, we prove that Algorithm 4 traverses all these transitions by constructing a test sequence of length $n * 2^n$.

Lemma 2. *The length of the test sequence generated by Algorithm 4 is $n * 2^n$.*

Proof. Clearly, $VisitHypercube$ in $CreateTestsSI$ for n times. Since each different value of id is associated with 2 transitions and id grows from 1 to $2^{n-1} - 1$, we can conclude that each invocation of $VisitHypercube$ in $CreateTestsSI$ will produce $2^n - 2$ transitions. Therefore, the total number of transitions are $n * (2^n - 2 + 2) = n * 2^n$. □

Finally, we show that no transition is covered twice. Due to the “shift” operation and the structure of “id”, it can be verified that the global state will never repeat before the execution of the test sequence produced by each $VisitHypercube$ with same m. Therefore, it is guaranteed that every load or evict operation in Algorithm 4 always drives the system through a uncovered transition. In other words, the test sequence constructed by Algorithm 4 does perform an Euler tour of entire state space.

The space complexity of Algorithm 4 is linear with the number of cores n. The reason is that the function $VisitHypercube(id, m, shift)$ can be recursively called for at most $n - 1$ times. The algorithm therefore requires a stack that with at most $n - 1$ levels. As a result, the space complexity is $O(n)$.

5.2.2 MSI Protocol

The difference between MSI protocol and SI protocol is that a cache block can be changed to the modified (M) state, when it receives a store request. For the ease of discussion, we define the following terms.

Definition 2. ***Global shared state*** *is a global state within which cores are in either shared or invalid states (e.g, IIS, ISI, ISS, SII, SIS, SSI, and SSS in Figure 5-3).*

Definition 3. ***Global invalid state*** *is a global state within which all cores are in the invalid state (e.g, III in Figure 5-3).*

Definition 4. ***Global modified state*** *is a global state within which one core is in the modified state (e.g, IIM, IMI, and MII in Figure 5-3).*

Figure 5-3 shows the state space of MSI protocol with three cores. Since only one core can be in the modified state for MSI protocol, there are n global modified states in the state space of a system with n cores. Global modified states are reachable from any other global states by store requests from corresponding cores. Besides, a global modified state can also be converted to the global invalid state or global shared states. For example, global modified state IMI can be converted to global invalid state III by evict(1), or global shared states ISS and SSI by load(0) or load(2), respectively.

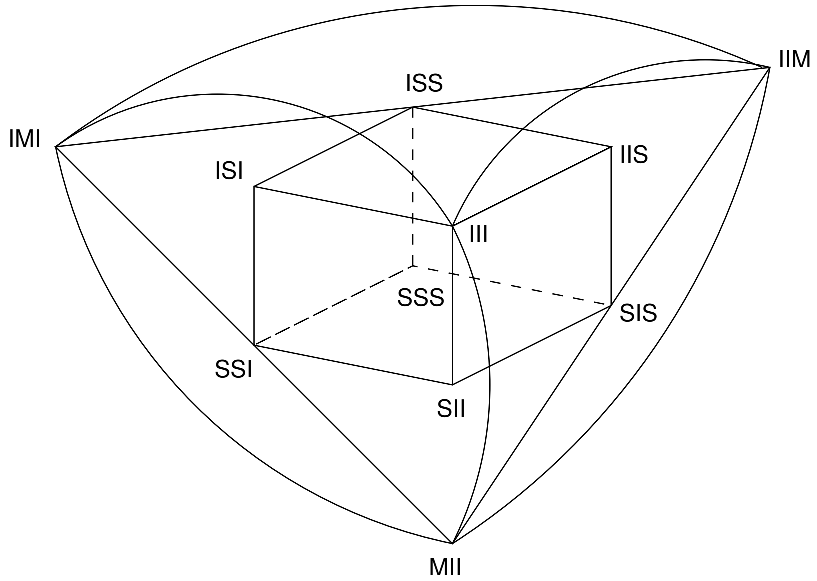

Figure 5-3. State space of MSI protocol with 3 cores. For the clarity of presentation, the transitions to global modified states (IIM, IMI, MII) are omitted, if the transition in the opposite direction does not exist.

Clearly, all n global modified states form a clique, because there are two transitions with opposite directions between each pair of them. As a result, these transitions can be covered with an Euler tour. Unfortunately, *it is not possible to cover all transitions in the state space of MSI by a single Euler tour.* The reason is that for some global shared state like IIS, there are only outgoing transitions to global modified states, but no incoming transitions from them. Therefore, outgoing transitions are twice of incoming transitions. The similar scenario can also be observed for global modified states, which have more incoming transitions than outgoing transitions. To cover all transitions, some of them must be reused. In fact, the problem to minimize the number of reused transitions is called Chinese Postman Problem (CPP) [35], which can be solved by calculating the min-cost max-flow. Since we need to perform the test generation on-the-fly, we decided not to obtain the optimal solution by solving CPP, because the state space can be too large to fit into memory when there are many cores in the system. Instead, we visit the uncovered transition to global modified state one by one and use the shortest path to link the end state of the previous transition and start state of the next transition.

Algorithm 5: Test generation for MSI protocol with n cores

```
$CreateTestsMSI(n)$
 1: $CreateTestsSI(n)$ /* Invoke Algorithm 1 */
 2: $VisitClique(0)$
 3: for each global shared state $s$ do
 4:    for $i = 0$ to $n - 1$ do
 5:       Output "store(i)"
 6:       Output the shortest path from current state to $s$
 7:    end for
 8: end for

$VisitClique(p)$
 1: Output "store(p)"
 2: Output operations to visit all bidirectionally reachable global shared states
 3: for $i = p + 1$ to $n - 1$ do
 4:    Output "store(i)"
 5:    if $i = p + 1$ then
 6:       $VisitClique(i)$
 7:    end if
 8:    Output "store(p)"
 9: end for
10: return
```

Algorithm 5 presents the test generation algorithm for MSI protocol. We first invoke $CreateTestsSI(n)$ in Algorithm 4 to cover all transitions that also exist in SI protocol. Next, $VisitClique$ will recursively perform an Euler tour in the clique of all global modified states. For example, when we execute $VisitClique$ in the state space shown in Figure 5-3, we are first going to cover transition IIM-IMI. In the recursive call of $VisitClique$ in line 6, transition IMI-MII and MII-IMI are visited. After that, transition IMI-IIM is covered by execution of line 7. In the next round of iteration, IIM-MII and MII-IIM are visited. To improve the efficiency, we also traverse all global shared states that are bidirectionally reachable from current global modified state. Finally, in line 3-6 of $CreateTestsMSI(n)$ we are visiting all uncovered transitions from global shared states to global modified states. Notice that we do not need to run Dijkstra's algorithm to find shortest path in line 6, because we must be in a global modified state after executing the store operation in line 5. The target global shared state can be reached by issuing load and evict requests based on the position of "S" in its state vector.

5.2.3 MESI Protocol

In MESI protocol, a cache block goes to exclusive (E) state when it is the first one, which loads a memory address. In a system with n cores, there are n global exclusive states[3] . Figure 5-4 shows the state space with three cores. Unlike global modified states, global exclusive states cannot be converted to each other directly. Therefore, the test generation algorithm $CreateTestsMSI$ for MSI protocol needs to be modified to create tests for MESI protocol. We can add n groups of operations to cover transitions from the global invalid state to global exclusive states as well as transitions from global exclusive states to global modified states. Notice that the $CreateTestsSI$ routine, which is used to visit all transitions between global shared states, also needs to be modified slightly. The reason is that in MESI protocol, the global invalid state will be converted to global exclusive states after any load request (III goes to IIE instead of IIS when the first core issues a load request).

5.2.4 MOSI Protocol

The MOSI protocol contains a new state "owned" (O), which can be used to avoid unnecessary writeback to memory. A cache block in the modified state is converted to the owned state, when other cores are trying to load the same cache block. The owned state can coexist with shared and invalid states. As a result, for a system with

[3] A **global exclusive state** is a global state with a cache block in exclusive state (e.g, IIE, IEI, and EII in Figure 5-4).

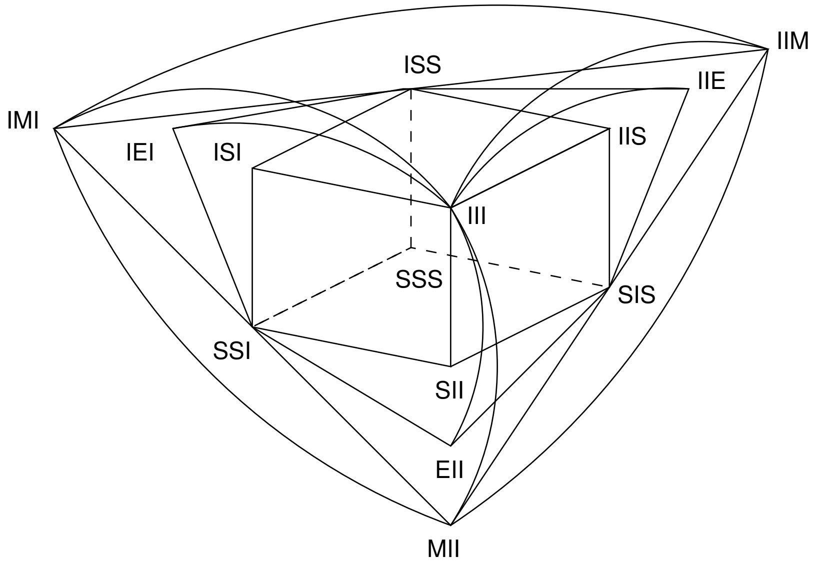

Figure 5-4. State space of MESI protocol with 3 cores

n cores, there are $n * 2^{n-1}$ global owned states[4] . Considering the fact that there are only $n + 2^n$ global states in MSI protocol with n cores, the state space of MOSI is much larger. Despite the large number of states, the state space structure of MOSI protocol is not complex. The entire space can be divided into three components. The first and second parts are the hypercube of global shared states and the clique of global modified states, respectively. They are identical to corresponding structures in MSI protocol. The third part is a set of n hypercubes with dimension $n - 1$. Each of the $n - 1$ dimensional hypercubes consists of 2^{n-1} global owned states, whose state vectors have "O" in the same position. For example, Figure 5-5 shows the state space with three cores. It is easy to see that states (IOI,IOS,SOS,SOI) (IIO,SIO,SSO,ISO) and (OII,OSI,OSS,OIS) are composed of three 2-d hypercubes (squares).

One nice property of this state space structure is that there is no transition between the n hypercubes of global owned states. Therefore, a large number of transitions

[4] A **global owned state** is a global state with a cache block in owned state (e.g, IOI, IOS, ... , OSS in Figure 5-5).

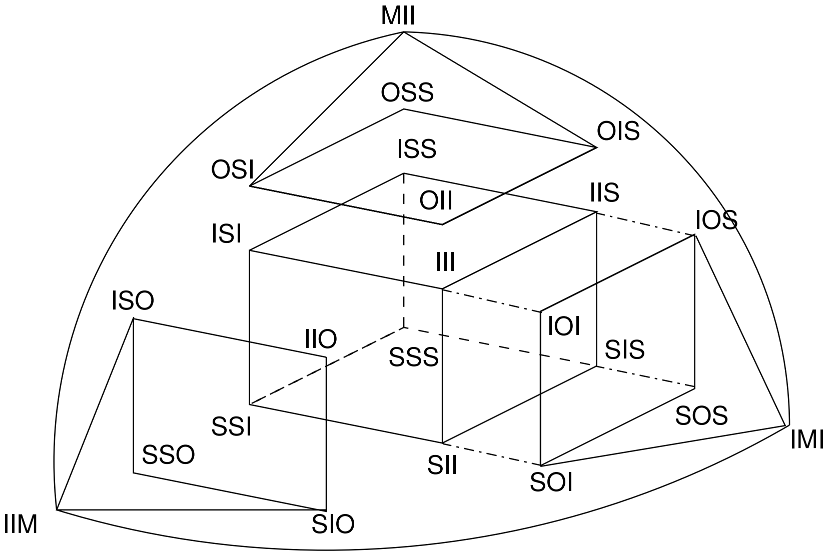

Figure 5-5. State space of MOSI protocol with 3 cores

between global owned states can be efficiently covered. We can perform an Euler tour in each $n-1$ dimensional hypercube by invoking routine $CreateTestsSI$ on global owned states like IIO, IOI and OII, where all but one core are in invalid state. In order to cover transitions from global owned states to global shared states, like IOS-IIS, we have to use a similar technique which was used in $CreateTestsMSI(n)$ to cover the store transitions.

5.3 Experiments

To analyze the performance of our proposed test generation framework, we conducted a number of experiments using M5 simulator [15]. M5 is a full system simulator, which implements a MOESI cache coherence protocol. In order to verify that our generated tests can achieve all transitions, we modify the cache subsystem in M5 slightly to allow different processes to access the same physical block. The load and store operations in the generated tests are translated into corresponding ALPHA instructions, while *evict* operation is achieved by loading a different memory address which is also mapped to the same location in the cache as the cache block under test. We use the *load-linked* and *store-conditional* instruction pairs to ensure the execution order of instructions in different cores.

Table 5-1. Statistics of our test generation algorithm for different protocols

			BFS		Our approach			
	# States	# Transitions	Total cost (transition)	Average cost per transition	Total cost (transition)	Average cost per transition	Improv. factor	Test generation time (sec)
MSI 8 cores	264	5256	36896	7.0	14664	2.8	60.3%	< 1
MESI 8 cores	272	5392	37712	7.0	15312	2.8	59.4%	< 1
MOSI 8 cores	1288	26248	196400	7.5	100807	3.8	48.7%	6.2
MOESI 8 cores	1296	26384	197216	7.5	101455	3.8	48.6%	6.2
MSI 16 cores	65552	2621968	29100096	11.1	11567888	4.4	60.2%	54.4
MESI 16 cores	65568	2622496	29103264	11.1	11570464	4.4	60.2%	54.5
MOSI 16 cores	589840	23855632	275254368	11.5	131122063	5.5	52.4%	586

Since M5 only supports MOESI cache coherence protocol, we also developed a protocol simulator, which can be configured to simulate the state transition of a multicore system using MSI, MESI and MOSI protocols. We used this simulator to validate the performance of our test generation approach on other protocols.

In the first experiment, we compared the efficiency of our test generation method with the tests generated by performing breadth first search (BFS) directly on the global FSM on different cache coherence protocols with various number of cores. Since tests generated by BFS are the shortest tests to drive the system from the global invalid state to the required transition, we use additional operations to reset the global state after execution of each test. Table 5-1 gives the results. Column "Total cost" presents the total number of transitions traversed to activate all transitions. Column "Average cost per transition" gives the average number of transitions we need to traverse in order to activate an uncovered transition. It can be observed that the total size of the tests generated by our approach is 50%-60% smaller than the ones generated directly by BFS. This result can be explained by the fact that the Euler tour exploited in our algorithm typically covers load and evict transitions on global shared state. The store transitions on the other hand, are covered in a similar way as the BFS approach. Since the numbers of allowed load and evict transitions for any global state are equal, we can save around half of the tests by exploiting the space structure.

We also compared the state and transition coverage of our test generation approach with a directed random test generator, MCjammer [83]. Figure 5-6 and

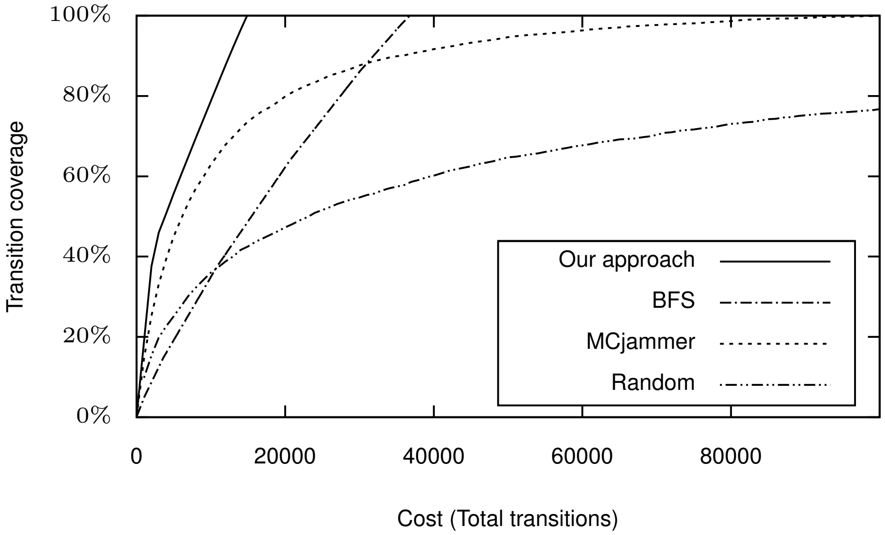

Figure 5-6. Transition coverage vs. cost for different test generation methods on MESI protocol with 8 cores

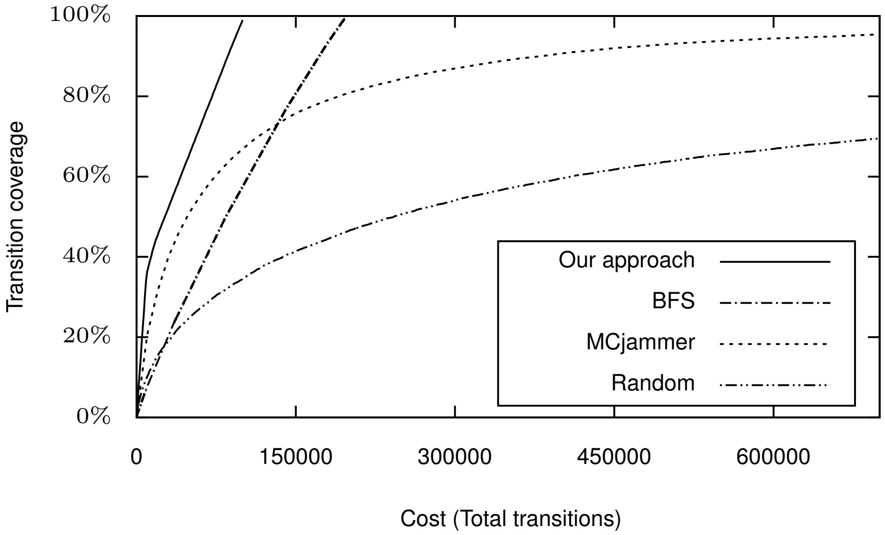

Figure 5-7. Transition coverage vs. cost for different test generation methods on MOSI protocol with 8 cores

Figure 5-7 show the relation between transition coverage and testing cost on the same system. It can be seen that MCjammer is very efficient at the beginning. Actually, it is more efficient than BFS to achieve 70% coverage. However, it becomes much slower to cover all transitions. The reason is that it is very unlikely for the algorithm with randomness to cover remaining uncovered transitions among all allowed transitions. On the other hand, our proposed test generation approach can always achieve 100% state and transition coverage with stable higher coverage speed than the BFS based tests.

Based on our experimental results, we can also estimate the overhead of our approach. Although we described our algorithm in recursive forms to simplify the presentation, they can also be implemented as iterative routines. As discussed in Section 5.2.1, our algorithms have linear space complexity with the number of cores. Since our tests can be generated on-the-fly, its overall space requirement is very small. The test generation time in Table 5-1 suggests that the runtime of our algorithms is reasonable. For MOSI protocol with 23 million transitions, we can create all the tests within 10 minutes, which indicates that our algorithm is quite light-weighted for entire simulation based verification phase.

5.4 Summary

We proposed an efficient test generation approach for a wide variety of cache coherence protocols. Based on detailed analysis of the space structure, our approach creates efficient test sequences for different parts of the global FSM state space to achieve 100% state and transition coverage for each cache coherence protocol. Compared with existing approaches based on directed random test generation, our approach significantly increases the transition coverage metric with linear memory requirement. Our experimental results on different cache coherence protocols demonstrated the effectiveness of our approach on systems with many cores, making it suitable for future multicore architectures.

CHAPTER 6
SCALABLE DIRECTED TEST GENERATION

Model checking is a promising technique to automatically generate directed tests. Model checkers usually accept models presented in special verification languages. It is not easy to apply them on real implementations. For example, while the Register Transfer Level (RTL) model of real processors are commonly designed in Verilog or VHDL, model checking tools like NuSMV only takes SMV models as input. Thus, real designs must be translated first, which itself can be an error-prone process. Since model checking is based on static analysis, the complexity of real world designs usually exceeds the capacity of model checking tools. The solving process may run out of memory before producing any useful results for real life designs. On the other hand, random or constrained-random test generation techniques are suitable for real designs, because they usually perform little reasoning on internal design logic. A large amount of random stimuli can be generated easily and simulated on real designs. However, random tests are inefficient to activate specific behaviors. It is therefore desired to have a test generation approach, which can handle real-life designs, but still able to activate any required system behavior with a small number of tests.

To bridge the gap between model checking based directed tests and random tests, various techniques (STAR [55] and HYBRO [56]) combines static and dynamic analysis. STAR generates tests to activate all control paths of an RTL design, although it suffers from the path explosion problem. HYBRO addresses this problem using branch coverage metric in RTL Control-Flow-Graph (CFG). The CFG is "sequentially unrolled" during the concrete/symbolic simulation to obtain the path constraints and measure the branch coverage. However, since the CFG and Use-Define chain is obtained using static analysis, this technique cannot be applied when dynamic array references [11] are involved. For example, a processor design may write to location "*ram[wb_addr]*" and read from "*ram[ld_addr]*" in the following cycle, where *ram* is an array corresponding to the main memory and *wb_addr* and *ld_addr* are two variables. If *ram[wb_addr]* and *ram[ld_addr]* are referring to the same element during execution and *ram[wb_addr]* is involved in the control path, all the assignments to *ram[ld_addr]* must also be considered. However, it is difficult to detect such dependency by static analysis. As a result, the applicability of [56] is limited, since dynamic array references is widely used in modern RTL designs to implement register files, buffers, caches and memory.

In this chapter, we address the dynamic array reference problem by making the array reference concrete. Instead of performing static analysis of the entire design to get the variable dependency, we compose an instrumented version of the original design and execute the instrumented design on a Hardware Description Language (HDL) simulator. During the simulation, the instrumented code will produce a trace file, which records all the logical operations performed by the design. All dynamic references to array elements are replaced by their concrete indices in the trace file during the concrete simulation. Next, the trace is analyzed using a constraint solver. In this way, our approach is able to analyze real hardware designs with dynamic array references and detect data dependency through array elements. We also propose several optimization techniques, which makes our proposed algorithm to have comparable efficiency as the state-of-art techniques [55][56]. Note that existing techniques [55] and [56] cannot handle designs with dynamic array references. Our experimental results demonstrate that our approach is capable of generating directed test efficiently on a variety of hardware designs. To the best of our knowledge, our approach is the first attempt to create directed tests for HDL designs with dynamic array references by interleaving concrete and symbolic simulation.

The rest of the chapter is organized as follows. Section 6.1 describes our test generation methodology for real HDL designs. Section 6.2 discusses the implementation details of our approach. Section 6.3 presents our experimental results. Finally, Section 6.4 concludes the chapter.

6.1 Directed Test Generation by Interleaving Concrete and Symbolic Execution

The basic idea of our work is to obtain the logic operations performed by the design on a single concrete execution path, and perform reasoning on top of it to obtain new test input. Figure 6-1 shows the key steps in our proposed approach. To explore different execution behaviors of the design, we first instrument the design with trace generation code. We also define the input variable set I, which present the input to the design under test (DUT). Next, we repeatedly simulate the instrumented design as follows:

1. Use I as input to the DUT.
2. Simulate DUT on a simulator. Collect all the operations performed by the design and activated path constraints from the trace output.
3. Invoke the constraint solver to check whether the desired behavior p is on current execution path. If this is the case, record the assignment of I as a test of p. Otherwise, negate one of the path constraints and use the constraint solver to obtain the assignment I', which forces the design to exercise a different execution path.

We first explain our test generation workflow using a simple example. Next, we describe the system model of our target design and several key steps in our workflow. Finally, we discuss some important optimization techniques to reduce the overall test generation time.

6.1.1 Illustrative Example

In this section, we use a simple example to show the basic workflow of our approach. The design is a simple counter module written in Verilog (Figure 6-2). The test input is the initial value on line 15. Our goal is to let the module execute the code on line 11 at clock cycle 2.

We first instrument the code and simulate the module for 3 cycles using a random input value, e.g., out = 0. The output trace is shown in Figure 6-3. The trace is produced by the instrumented code, which performs the same operations as the original code. In addition, the instrumented code also prints the performed operation during simulation as

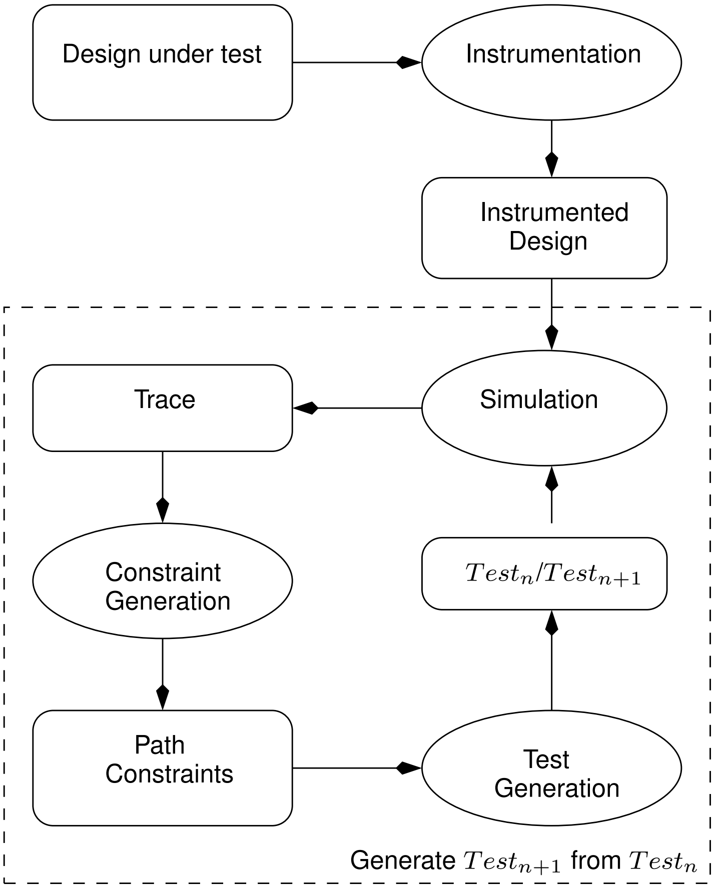

Figure 6-1. The workflow of our approach

a trace file. We use (out,0), (out,1), (out,2) and (out,3) to represent each *out* in different cycles. Notice that “IF (out,0) == 40 not taken” statement indicates that the if statement on line 10 is evaluated to be false. Clearly, line 11 is not executed when the initial value of out is 0.

Since our goal is to let line 11 to be executed at cycle 2, the variable *out* must have value 40 at cycle 2. Similarly, (out,0), (out,1), (out,2) and (out,3) must satisfy constraints in Figure 6-4. Therefore, we can use constraint solvers like Yices [34] to solve these constraints, and produce the satisfiable assignments to all variables. In this case, the solver determines (out,0), (out,1), and (out,2) should be 38, 39, and 40, respectively. In other words, the initial value of (out,0) should be 38 in order to activate line 11 at cycle 2. This is the intended directed test.

```
module counter(out, clk, reset);
  parameter WIDTH = 8;
  output [WIDTH-1 : 0] out;
  input                clk, reset;
  reg [WIDTH-1 : 0]  out;
  wire         clk, reset;
  always @(posedge clk)
  begin
    out <= out + 1;
    if(out == 40)
      $display ("Activated");
  end
  always @reset
    if (reset)
      out = 0; // initial value
endmodule
```

Figure 6-2. Counter.v

```
(out,0) = 0
(out,1) = (out,0) + 1
IF (out,0) == 40 not taken
(out,2) = (out,1) + 1
IF (out,1) == 40 not taken
(out,3) = (out,2) + 1
IF (out,2) == 40 not taken
```

Figure 6-3. Sample Trace

```
(out,1) = (out,0) + 1
(out,0) != 40
(out,2) = (out,1) + 1
(out,1) != 40
(out,3) = (out,2) + 1
(out,2) = 40
```

Figure 6-4. Sample Path Constraints

If we observe the path constraints (Figure 6-4) obtained from trace during concrete simulation (Figure 6-3), it is easy to see that we are essentially performing a chronological back tracking in the space of execution paths. By negating the topmost

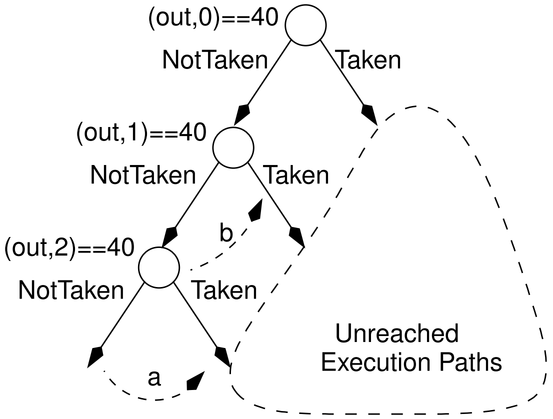

Figure 6-5. Chronological Back Tracking

constraint [1] in the trace file ((out,2) != 40), we force the design to switch to a different execution path (transition "a" in Figure 6-5). Sometimes, it is also possible that our desired path constraints (Figure 6-4) are not satisfiable, i.e., the branch (out,2)==40 in Figure 6-5 can not be taken. In this case, we can negate the next topmost constraint ((out,1) != 40) and use the constraint solver to check whether the branch at node (out,1)==40 can be taken. The process is repeated, until the desired test is found, or all branches are activated.

Although this test generation example is performed on a simple Verilog design, it illustrates the basic idea of our proposed approach. In the rest of the section, we are going to discuss how to automate every step during this process, and generate the entire directed test suite automatically.

6.1.2 System Model

Our approach takes Verilog HDL program as input. Our current implementation supports most common features of Verilog, such as always@(... sensitive list ...), continuous assignment, conditional branches (if, case), and different variable types (reg,

[1] Due to use of stack in our implementation, the last path constraint is the topmost constraint.

wire). Although our implementation is based on Verilog, the same working principle can also be applied to VHDL designs, since it also describes concurrent finite state systems.

Our current implementation supports common fault models such as path activation fault and stuck-at fault. These fault models describe possible faults that can occur during the execution of the system. The path activation fault model can be used to check whether there is any unreachable code in the design. The stuck-at fault can be used to check whether the given variable always has the same value. Based on the given fault model, our test generation technique will generate the test suite, which can activate all possible faults of the system under the fault model. It is important to note that these fault models are by no means the golden model rather it is a representative model. Various graph-based fault models (including node fault, edge fault, sequence fault and interactions fault) are explored in Section 6.3.3.

Without loss of generality, we discuss our approach in the context of single clock domain. We use tuple $(name, clk)$ to index each variable in every cycle. When multiple clock domains are used, the *clk* should be the cycle number in corresponding clock domain.

6.1.3 Instrumentation

The primary purpose of the instrumentation is to use the simulator to produce a trace file during concrete simulation of the RTL design. The resultant trace file is crucial to our test generation framework for two reasons. First, the trace file records all logic operations performed during the concrete simulation, which enables us to perform symbolic simulation and directed test generation. Besides, the trace file also provides information about different concrete execution paths. To ensure that each variable is unique, we need to flatten all module instances before instrumentation. The details are described in Section 6.2.1.

Table 6-1 shows the instrumentation rules. For ease of presentation, we use Verilog syntax for illustration. We use variable cc to denote the number of clock cycles from the

Table 6-1. Verilog instrumentation code

//Continuous assignment	always
	# clkwidth \$display($(v, cc) = e$);
assign $v = e$;	assign $v = e$;
//Blocking assignment	
$v = e$;	\$display($(v, cc) = e$);
	$v = e$;
//Assignment within	
//always@(pos/negedge ...)	\$display($(v, cc + 1) = e$);
$v <= e$;	$v <= e$;
//Assignment within	
//other always blocks	\$display($(v, cc) = e$);
$v <= e$;	$v <= e$;
//If	
if(p)	if(p)
s;	begin \$display(IF p taken); s; end
else	else
s';	begin \$display(IF p not taken); s'; end
//Case	case(e)
	x:
case(e)	begin \$display(CASE $e = x$); s; end
x: s;	y:
y: s';	begin \$display(CASE $e = y$); s'; end
default: s'';	default:
	begin \$display(CASE $e! = x, y$); s''; end
//Array index	
//$b[e]$ is an array reference	\$display(...b_eval(e)...);
// in a statement	
...$b[e]$...;	...$b[e]$...;
//Beginning of a cycle	\$display(New cycle);
	$cc = cc + 1$;

beginning of the simulation. We use the “display” statement to print the syntactic objects into the trace file during the simulation of the instrumented code. For normal arithmetic operations, the instrumented code just record the exact operation that is performed by the design. For example, for continuous assignment (first row in Table 6-1), the original code is

```
assign a=b+c;
```

The instrumented code is

```
always
# cycle $display((x,cc)= (y,cc)+(z,cc));
assign x=y+z;
```

which have the same funtionality as the original code and print $(x, cc) = (y, cc) + (z, cc)$ in every cycle with corresponding cycle number (cc). In fact, the value of cc are populated automatically during concrete simulation. The details can be found in Section 6.2.2.

For other assignment statements, the instrumented code also marks whether the assignment is made within
always@(pos/negedge ...) block. In this way, the trace file records whether the left hand side variable receives the value of right hand side expression in the same clock cycle.

Our framework enables natural analysis of arrays. To reason with dynamic array references, we replace the index expression of each array elements into its concrete value, and treat each array element as an independent variable. During the concrete simulation, the index expression e is evaluated. The corresponding array element is refereed by concatenating the concrete results $eval(e)$ to the array name in the trace file. We discuss the details in Section 6.2.3.

6.1.4 Concrete Simulation

Once the design is flattened and instrumented, we interleave concrete and symbolic simulation of the design. In each iteration, we perform the concrete simulation of the instrumented design using a simulator with desired number of cycles. Since the instrumentation process does not affect the functionality of the design, the behavior of the instrumented design is identical to the original design. At the same time, the instrumented design produces a trace file, which records every operation performed by the design in the correct order. This trace file will be used for the symbolic simulation of the concrete execution path.

6.1.5 Path Constraint Generation

In this step, we convert the trace file into a path constraint file. This step is required for two reasons. First, the continuous assignments are simulated using always blocks. As a result, the constraint corresponding to the continuous assignment may be printed after the trace is produced by the real always block in the same cycle. To simplify the solving process, we re-arrange the trace file so that all constraints produced by continuous assignments are placed before the constraints corresponding to normal always blocks.

The semantics of a register variable requires that if a variable is not updated, it should keep its value from the previous cycle. However, this property is not enforced by the constraints in the trace file. Thus, we have to examine that all assignments made during a cycle, and add additional constraints to ensure that all registers still maintains their values if they are not updated. The structure of a valid path constraint file is shown in Figure 6-6.

...	...
Cycle k	Continuous Assignments
Cycle k	Additional Constraints
Cycle k	Always blocks
Cycle $k+1$	Continuous Assignments
Cycle $k+1$	Additional Constraints
Cycle $k+1$	Always blocks
...	...

Figure 6-6. Path constraint file structure

6.1.6 Test Generation

First, we discuss the test generation for path activation fault. Since the goal is to explore unreached execution paths, we can negate a path constraint and use the constraint solver to create a new input assignment, which will guide the design to a different path. Currently, we negate the top most path constraint. As a result, we are essentially performing a depth first search.

Algorithm 6: Test Generation Algorithm

```
$test\_gen(constr[0, ..., top])$
for $i = top$ to $0$ do
  if $constr[i]$ is a branch constraint then
    $c = find\_next(constr[i])$
    while $c \neq null$ do
      $I' = satisfy(constr[0, ..., i-1] \wedge c)$
      if $I' \neq null$ then
        return $I'$
      end if
      $c = find\_next(constr[i])$
    end while
  end if
end for
return $null$

$find\_next(branch)$
Add $branch$ into $covered$
if $branch$ is an IF statement then
  if $\neg branch \notin covered$ then
    return $\neg branch$
  end if
end if
if $branch$ is a CASE statement then
  find and return next uncovered case, if any.
end if
return $null$
```

Algorithm 6 presents our test generation algorithm $test\_gen$ for path activation fault in detail. The algorithm takes the path constraint file $constr[0, ..., top]$ as input, where $constr[top]$ and $constr[top]$ are the first and last constraints in the file, respectively.

Function $test\_gen$ examines all constraints produced by branch conditions in the reverse order. For every branch constraint, we first mark it as covered, then try to find the next uncovered branch constraint. For IF statement, we just need to check the negated version of the branch constraint. For CASE statement, we have to search for the next uncovered case. After that, the new branch constraint c is added to all previous path constraints $constr[0, ..., i-1]$ to form the constraints for the next test. If it is satisfiable, the assignment I' (returned from the constraint solver) will be returned as the next test input. Otherwise, we examine next uncovered branch, until all branches are checked. In this way, it is guaranteed that I' will force the design to exercise a different execution path during the next round of simulation. Recall that the design is simulated for a fixed number of cycles. Our algorithm eventually terminates once all reachable branches within the given number of cycles are explored.

Each branch is uniquely identified by its line number, flattened instance name, and cycle number. To avoid the path explosion problem, a covered branch is marked and not explored again in the following test generation process.

For other fault models (including stuck-at, node, edge, sequence and interaction fault model), the desired behavior can be checked during the exploration of different execution paths. Once we obtain a new execution path, the constraint solver is employed to check whether the desired behavior is possible on the path.

6.1.7 Constraint Solving Optimization

In our current implementation, we employed Yices [34] as our constraint solver. Since the path constraint usually contains a very large number of constraints, it is very important to reduce time consumption in constraint solving. Currently, we use three optimization techniques.

1. *Cone-of-influence (COI) reduction*: In many designs, a large number of variables are used for data transfer and not involved in the control path.In other words, they are not in the cone-of-influence of any branch constraints in current execution path. It is therefore safe to remove the constraints involving these variables from the path constraint file without changing its satisfiability. This optimization is

similar to the CFG unrolling and UD chain slicing technique proposed in [56]. It should be noticed that since the variable indices in arrays are replaced by their concrete values in the trace file, we are able to detect the data dependency through dynamic array reference.

2. *Early unsatisfiable detection*: Some variables, like reset signal, are used widely across the entire design as switch variables. As a result, they appears in the path constraint for several times in every clock cycle. It is enough to negate the first occurrence of a recurring path constraint, because the negation of its other occurrence must be unsatisfiable.

3. *Unsatisfiable core detection*: Some constraint solver is capable to return the unsatisfiable core of a unsatisfiable model. Clearly, if all constraints in the unsatisfiable core remains in the path constraint file, the model must be still unsatisfiable.This information can be utilized to reduce the number of expensive constraint solver calls by skipping the negation of some path constraints.

6.2 Implementation Details

6.2.1 Design Flattening

Ideally, we can use an HDL parser to produce a flattened version of the original design. Unfortunately, modern HDL parsers usually perform some optimizations during the flattening process. For example, some arithimetic operations may be replaced by synthesizable components. As a result, it is not easy to map the operation performed by the flattened design back to the original design. We use a different approach to solve this problem. Instead of flattening the real design and then performing instrumentation, we solve the problem from the simulator side.

In modern Verilog simulators, the input Verilog file is usually compiled into a simulation file before real simulation execution. For some simulators like Icarus Verilog [92], the compiled simulation file is presented as an assembly code like program. Suppose the instrumented display statement is

```
$display("( assert (= need_off  0b0 ))" ,need_off);}
```

where “assert (= need_off 0b0)” means *need_off* equals to zero in Yices input language. In this simulation file, the display statement is presented as,

```
%vpi_call 2 6077 "$display","( assert ( =need_off 0b0));", v0x130e570_0;
```

Here, the argument list on the second line is a list of addresses, each of which corresponds to a register, or wire variable in the Verilog file. For example, v0x130e570_0 is the address of variable *need_off*. We postfix the variable names with their addresses to remove the ambiguity caused by instantiation of the same module. For example, the above statement is written as

```
%vpi_call 2 6077 "$display",";( assert (= need_off130e570 0b0 ));";
```

In this way, different instantiation of the same variable is disambiguated within the trace file. For example, suppose there is another instantiation of variable *need_off*, it must have a different address other than v0x130e570_0 in the compiled simulation file.

6.2.2 Clock Cycle Population

To differentiate the same variable in different cycles, we concatenate each variable name with the concrete value of the current clock cycle number. During the concrete simulation, the value is populated automatically. To accomplish this, we postfix the variable name with “%0d”. For example,

```
%vpi_call 2 6077 "$display",";( assert (= need_off130e570c%0d 0b0 ));",
v0x13233b0_0;
```

where v0x13233b0_0 is the address of cc. During the concrete simluation, the trace file automaticly receives the correct cycle count, i.e., the trace output in the second cycle becomes

```
( assert (= need_off130e570c2 0b0 ));
```

6.2.3 Dynamic Array Reference Disambiguation

As discussed in Section 6, we address variable-indexed elements in the array using the concrete value of the indices, so that we can reason about dynamic data without alias analysis. We implement this feature as follows. Suppose the following display statement is used to assign array element *ram[wb_adr_i]* the value 0b.

```
%vpi_call 2 42 "$display", " ;( assert ( = ram[wb_adr_i]  0b0 ));",
v0x1322030, v0x13221b0_0;
```

Since *wb_adr_i* is the index of the element in the array *ram*, we rewrite the above statement as

```
%vpi_call 2 42 "$display", " ;( assert ( = ram1322030_%d 0b0) );"
,v0x13221b0_0;
```

Assume *ram[wb_adr_i]* refers to the element *ram[65535]* when the corresponding assignemnt is made during the concrete simulation, i.e., *wb_adr_i* is 65535. Since we replace *wb_adr_i* with its concrete value using%d, the resultant trace output becomes

```
( assert (= ram1322030_65535 0b0 ));
```

Clearly, all references to *ram[65535]* can be easily identified by checking whether it refers to variable *ram1322030_65535*.

6.3 Experiments

We developed a prototype of our directed test generation framework. Our test generation tool takes a Verilog design as input and iteratively produces new tests. We have modified Icarus Verilog [92] for instrumentation with approximately 500 lines of C++ code. We also implemented a test generation engine (approximately 2000 lines of C++ code) to perform concrete simulation on the HDL simulator, analysis the trace file, generate path constraints and invoke the SMT solver. Our framework is fully automated and there is no need to manual intervention at any stage.

In this section, we present the experimental results of our case studies. We compared our approach with existing methods including HYBRO [56], the random test technique, and model checking based approach [22]. The experiments are performed using RTL models from ITC99 and two processor designs. As discussed in Section 6.2.1, we used Icarus Verilog as Verilog parser and simulator. Yices [34] was employed for constraint solving. All experiments were performed on 3GHz AMD Opteron Processor with 10GB memory.

6.3.1 Designs without Dynamic Array References

In this section, we compare the performance of our approach with HYBRO [56]. To make fair comparison, we choose the same ITC99 RTL models as [56], with same number of unrolled cycles and the same SMT solver. We only compare the branch coverage in our experiments, because the assertions used for functional coverage in [56] is not available to public.

Table 6-2. Comparison with HYBRO [56]

Bench	Unroll	HYBRO[56]		Our approach	
mark	Cycles	Bran_Cov	Time	Bran_Cov	Time
b01	10	94.44%	0.07s	96.30%	2.24s
b06	10	94.12%	0.10s	96.30%	2.36s
b10	30	96.77%	52.14s	96.67%	24.61s
b11	10	78.26%	0.28s	82.35%	3.75s
b11	50	91.30%	326.85s	94.44%	270.28s
b14	15	83.50%	301.69s	98.95%	257.59s

Table 6-2 presents the experimental results. The first two columns indicates the design name and the number of unrolled cycles. The next four columns show the branch coverage rate and the time consumption of HYBRO [56] and our approach, respectively. The branch coverage rate is calculated using the same convention in [56], where unreachable default branches in "case" statement are also included. The results suggest that our approach has comparable performance with HYBRO [56] on these benchmarks. Comparable performance is expected because the cone-of-influence reduction employed in our approach is essentially equivalent to the CFG unrolling and UD chain slicing optimization in HYBRO [56]. Note that these are 8 ITC99 benchmarks that have arrays. Since [56] is not applicable on dynamic array references, we do not present those results.

6.3.2 Designs with Dynamic Array References

This experiment was performed on Zet processor, which is an open source implementation of the 16-bits x86 instruction set architecture. When synthesized in configurable devices like FPGA, Zet processor can boot MS-DOS 6.22 and run Microsoft Windows 3.0. The processor is implemented using 5K+ lines of Verilog code, 289 continues assignments, 53 always blocks, 324 register variables and 666 wire variables. Both the main memory and the registers file are modeled as arrays and addressed with variables.

Our goal in this experiment is to achieve high branch coverage in source code level. This is important because there are a large number of conditional branches in the opcode decode stage. Besides, since x86 instruction set has variable length binary encoding, it is not easy to invoke all branches in the design. The primary input of the design is the lowest 4 bytes of the memory space (0x00000-0x00003). Before executing the test, the processor only executes a jump instruction (to 0x00000) after reset. The design is simulated for 10 cycles. We compared with random tests since it is the only test generation technique that supports HDL designs with dynamic array reference.

Table 6-3. Comparison with random testing

Method	#Tests	Explored	Branch	Time
		Branches	Coverage	
Random	1000	197	89.95%	366.45s
Random	5000	204	93.15%	1981.73s
Random	10000	208	94.98%	3785.49s
Random	20000	212	96.80%	7386.92s
Random	40000	213	97.26%	14585.83s
Our approach	140	218	99.54%	1320.58s

Table 6-3 shows the experiment result. The first five rows depict the results by using 1000, 5000, 10000, 20000, and 400000 random tests, respectively. The performance of our approach is shown in the last row. It can be seen that due to the random nature, it is very time consuming to reach 100% branch coverage even using thousands of random tests. On the other hand, our directed test generation scheme effectively explored execution paths by avoiding covered branches. With less than 200 tests, our framework achieves higher coverage than 40000 random tests.

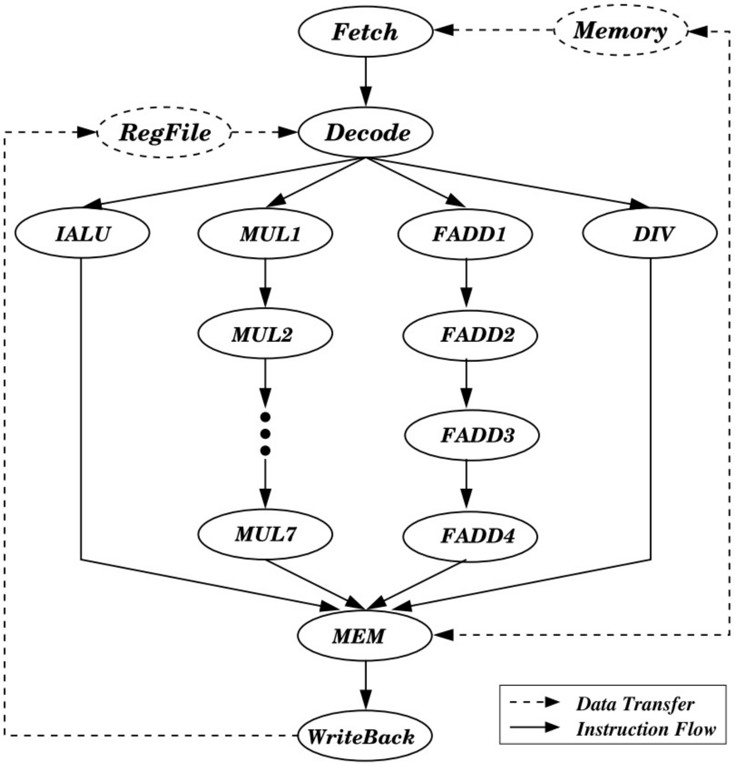

Figure 6-7. The MIPS architecture [22]

6.3.3 SAT-based BMC versus Our Approach

To evaluate the effectiveness of our proposed approach in other fault models, we compare our approach with the BMC-based test generation technique [22]. The experiment is performed on the model of a single-issue MIPS processor. Figure 6-7 shows its brief structure. It has five pipeline stages: fetch, decode, execute, memory (MEM), and writeback. The execute stage has four parallel execution paths: integer ALU, 7 stage multiplier (MUL1 - MUL7), four stage floating-point adder (FADD1 - FADD4), and multi-cycle divider (DIV). The SAT-based BMC approach accepts the SMV description of MIPS as input, where as our approach accepts an equivalent Verilog description. We use four different fault models from :

1. Node Fault, i.e., a node cannot be activated.
2. Edge Fault, i.e., successive state of a node cannot be activated in certain order.

3. Sequence Fault, i.e., the associate nodes and edges cannot be activated in correct order.

4. Interaction Fault, i.e., a set of nodes cannot be activated at the same time.

Since the faults in are described using LTL properties, we manually translated all properties into Verilog assertions regarding corresponding registers in different cycles. The design is simulated for 10 cycles.

Table 6-4. Comparison with BMC [22]

Types	#Faults	BMC [22]	Our approach
		Time	Time
Node Fault	20	17.53s	22.56s
Edge Fault	25	37.51s	24.27s
Sequence Fault	16	21.10s	20.29s
Interaction Fault	110	375.22s	340.56s

Table 6-4 shows the experimental results. The first two columns present the fault types and the number of properties to be activated for each fault type. The next two columns depicts the time consumption of BMC [22] and our proposed technique, respectively. Although the test generation processes are quite different, the total time consumption of BMC is close to the time consumption of our approach. Actually, the trace file generated by our approach can be viewed as a unrolled version of the design on current execution path, while the BMC-based approach unrolls the SAT presentation of the transition relation of the design. Therefore, our approach has comparable performance with SAT-based BMC when a simple design is involved. However, it should be noticed that it is not always possible to use model checking on real designs directly. When the design contains many branches and dynamic array references, it becomes difficult to apply BMC-based approach without translation or abstraction, while our proposed approach can still be applied as illustrated in Section 6.3.2.

6.4 Summary

Functional verification of modern SOC designs is challenging due to increased design complexity and reduced time-to-market. Directed tests are promising because it requires significantly less number of tests to achieve the same coverage goal compared to random tests. Unfortunately, model checkers usually do not accept real hardware designs or support features such as arrays. Moreover, the real designs usually exceeds the capacity of model checkers due to the complexity of static analysis. In this chapter, we presented a novel test generation approach that addresses both of these problems using interleaved concrete and symbolic execution. The design is first simulated to generate an execution trace. The constraint solver is then applied to find the test inputs which can force the real design to exercise the desired behavior. Compared with existing approaches based on combined concrete and symbolic execution, our approach is capable of analyzing real processor designs with dynamic array references. The experimental results demonstrate that our proposed technique is scalable, and enables directed test generation for real designs.

CHAPTER 7
TEMPERATURE- AND ENERGY-CONSTRAINED SCHEDULING IN REAL-TIME SYSTEMS

Since high on-chip thermal dissipation has severe detrimental impact, we have to control the instantaneous temperature so that it does not go beyond a certain threshold. Dynamic voltage scaling (DVS) is acknowledged as one of the most efficient techniques used in both energy optimization [21] and temperature management [101]. In existing literatures, ***temperature (energy)-constrained*** means that there is a temperature threshold (energy budget) which cannot be exceeded, while ***temperature (energy)-aware*** means that there is no constraint but maximum instantaneous temperature (total energy consumption) needs to be minimized. In this chapter, we propose a formal method based on model checking for temperature- and energy-constrained (**TCEC**) scheduling problems in multitasking systems. In this work [71], we present an approximation algorithm, which effectively addresses the state space explosion problem caused by model checkers [88]. The approximation scheme will give no false positive answer, while its possibility to report false negative answer can be small enough for practical usage.

The rest of the chapter is organized as follows. Section 7.1 provides related background information. Section 7.2 provides an overview of our framework. Section 7.3 describes our contribution in details. Experimental results are presented in Section 7.5. Finally, Section 7.6 summarizes the chapter.

7.1 Background and Problem Formulation

This section provides the formal description of the TCEC scheduling problem. Since many aspects of real-time systems are involved, we first provide some background information.

7.1.1 Thermal Model

A thermal RC circuit is normally utilized to model the temperature variation behavior of a microprocessor [101]. We adopt the RC circuit model proposed in [75], which is widely used in recent research [101] [46], to capture the heat transfer phenomena in the processor. If P denotes the power consumption during a time interval, R denotes the thermal resistance, C represents the thermal capacitance, T_{amb} and T_{init} are the ambient and initial temperature, respectively, the temperature at the end of the time interval t can be calculated as:

$$\begin{aligned} T &= P \cdot R + T_{amb} - (P \cdot R + T_{amb} - T_{init}) \cdot e^{\frac{-t}{RC}} \\ &= (1 - e^{\frac{-t}{RC}})T_s + e^{\frac{-t}{RC}} T_{init} \end{aligned} \tag{7–1}$$

where t is the length of the time interval, $T_s = P \cdot R + T_{amb}$ is the steady-state temperature.

7.1.2 Energy Model

We adapt the energy model proposed in [60]. Processor's dynamic power can be represented as

$$P_{dyn} = \alpha \cdot C \cdot V_{dd}^2 \cdot f \tag{7–2}$$

Here V_{dd} is the supply voltage and f is the operation frequency. C is the total capacitance and α is the actual switching activity which varies for different applications [8]. In other words, task's power profile can be different from each other. Static power is given by $P_{sta} = V_{dd} \cdot I_{subth} + |V_{bs}| \cdot I_j$ where V_{bs}, I_{subth} and I_j denote the body bias voltage, subthreshold current and reverse bias junction current, respectively. Hence, we have $P = P_{dyn} + P_{sta}$. Our technique is, however, independent of the power model and thermal model.

7.1.3 System Model

The system we consider can be modeled as:

1. A voltage scalable processor which supports *l* discrete voltage levels $\{v_1, v_2, ..., v_l\}$
2. A set of *m* independent tasks $\{\tau_1, \tau_2, ..., \tau_m\}$.
3. Each task $\tau_i \in \{\tau_1, \tau_2, ..., \tau_m\}$ has known attributes including worst-case workload, arrival time, deadline, period (if it is periodic) or inter-arrival time (if it is aperiodic/sporadic).

The runtime overhead of voltage scaling is variable and depends on the original and new voltage levels. The context switching overhead is assumed to be constant. For ease of discussion, the terms *task*, *job* and *execution block* refer to the same entity in the rest of this chapter.

7.1.4 TCEC problem

The proposed methodology can be applied to both scenarios in which task set has a common deadline and each task has its own deadline. For ease of discussion, the following definition of TCEC problem is constructed for task sets with a common deadline. The second case will be discussed in Section 7.4.

Given a trace of m jobs $\{\tau_1, \tau_2, \cdots, \tau_m\}$, where task τ_{i+1} is executed after τ_i $(1 \leq i < m)$. If tasks are assumed to have the same power profile (i.e., α is constant), the energy consumption and execution time for τ_i under voltage level v_j, denoted by $w_{i,j}$ and $t_{i,j}$ respectively, can be calculated based on the given processor model. Otherwise, they can be collected through static profiling by executing each task under every voltage level. Let $\psi_{i,j}$ and $\omega_{i,j}$ denote runtime energy and time overhead, respectively, for scaling from voltage v_i to v_j. Since power is constant during an execution block, temperature is monotonically either increasing or decreasing [46]. We denote $T(i)$ as the final temperature of τ_i. If the task set has a common deadline D, the safe temperature threshold is T_{max} and the energy budget is W, TCEC scheduling problem can be defined as follows.

Definition 5. ***TCEC instance**: Is there a voltage assignment $\{l_1, l_2, ..., l_m\}$[1] such that:*

$$\sum_{i=1}^{m}(t_{i,l_i} + \omega_{l_{i-1},l_i}) \leq D \tag{7–3}$$

$$\sum_{i=1}^{m}(w_{i,l_i} + \psi_{l_{i-1},l_i}) \leq W \tag{7–4}$$

$$T(i) \leq T_{max}, \forall i \in 1, ..., m \tag{7–5}$$

$T(i)$ is calculated based on Equation (7–1) for each i, i.e.,

$$T(i) = (1 - \beta_i)T_s^{l_i} + \beta_i T(i-1) \tag{7–6}$$

where $\beta_i = e^{-t_{i,l_i}/RC}$ (Recall that t_{i,l_i} is the worst case execution time of task τ_i under voltage level l_i), $T(0) = T_{init}$, and $T_s^{l_i}$ is the steady-state temperature of the system, when l_i is applied. Equation (7–3), (7–4) and (7–5) denote the common deadline, energy and temperature constraints, respectively.

When the workload is periodic, we also require the temperature at the end of the hyperperiod to be less than or equal to the initial temperature. In this way, the resultant schedule is guaranteed not to exceed the temperature constraint irrespective of the length of the execution time (i.e., the hyper-period may repeat many times).. Formally, suppose there are m tasks within the hyperperiod, and the last task is τ_m, in addition to the temperature constraints in (7–5), we also require

$$T(m) \leq T_{init} \tag{7–7}$$

More detailed discussion about periodic workload can be found in Section 7.4.

[1] l_i denote the index of the processor voltage level assigned to τ_i.

7.2 Overview

Figure 7-1 illustrates the workflow of our approach, which accepts a task execution trace as input. The task execution trace can be produced by a scheduler with certain scheduling policy. The scheduler executes the task set under the highest voltage level and produces a trace of *execution blocks*. An execution block is defined as a piece of task execution in a continuous period of time under a single processor voltage/frequency level. Each execution block is essentially a whole task instance in non-preemptive systems. However, in preemptive scheduling, tasks could be preempted during execution hence one block can be a segment of one task. The scheduler records runtime information for each block including its corresponding task, required workload, arrival time and deadline, if applicable.

The task execution trace, along with system specification (processor voltage, frequency levels, temperature constraints or/and energy budget) and thermal/power models are fed into the timed automata generator (TAG) that we have developed. TAG generates two important outputs. One is the corresponding timed automata model [88], and the other one is properties reflecting the temperature/energy/deadline constraints defined in system specification. After that, a suitable solver (e.g., a model checker) is applied to find a feasible schedule of the tasks, or confirm that the required constraints cannot be met. This methodology is flexible and completely automatic. It is based on formal technique and suitable in early design stages.

As discussed in [88], model checkers like UPPAAL can be used to verify the generated model directly. However, when the number of jobs is large, it can be time consuming to check the properties on the timed automata directly. The reason is that the underlying symbolic model checker sometimes cannot handle large problems due to the state space explosion problem. To address the state space explosion problem in model checking, we propose an approximation algorithm for TCEC scheduling in Section 7.3.

We also demonstrate the applicability of our approach to solve other problem variants including TC, TA, TAEC and TCEA in Section 7.4.

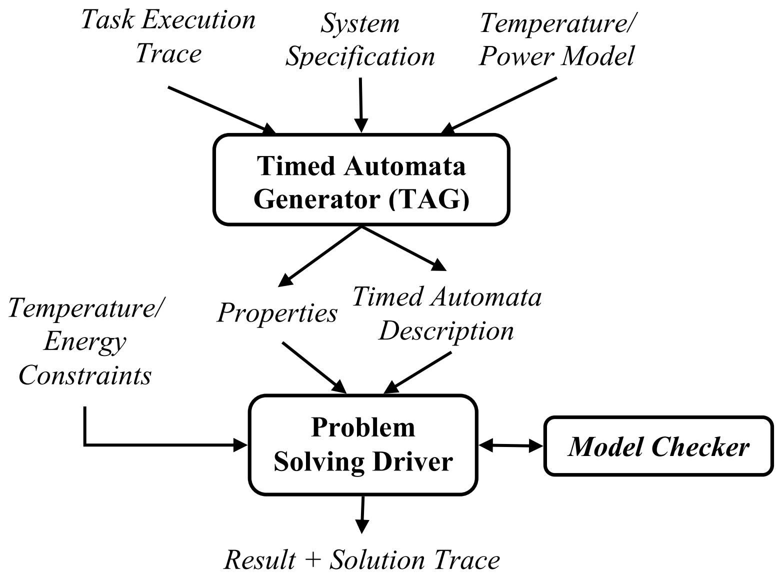

Figure 7-1. Overview of our TCEC schedulability framework.

7.3 Approximation Algorithm for TCEC Scheduling

To alleviate the state explosion problem in TCEC scheduling, we can formulate our model checking problem as a Multi-Constrained Path problem (MCP). Although MCP is NP-Complete for more than one constraints, we are able to design polynomial time approximation scheme which can be tuned with enough accuracy for practical design usage. In this section, we first explain how to model TCEC problem as MCP. Next, we discuss the pseudo-polynomial time model checking algorithm based on Bellman-Ford algorithm. Finally, we present our polynomial time approximation algorithm for TCEC.

7.3.1 Notations

Given a directed graph $G = (V, E)$, a path $p = s \rightarrow n_1 \rightarrow \cdots \rightarrow n_i$ and an edge $e_i = (n_i, n_{i+1}) \in E$, where $s, n_1, \cdots, n_i \in V$, the notation $p||e_i$ denotes the path $s \rightarrow n_1 \rightarrow \cdots \rightarrow n_i \rightarrow n_{i+1}$. In other words, p can also be expressed as $e_0||e_1||\cdots||e_i$, where $e_0 = (s, n_1)$, $e_1 = (n_1, n_2)$,$\cdots$,$e_i = (n_i, n_{i+1})$.

Given vectors $\boldsymbol{a}, \boldsymbol{b} \in R^N$, we say that $\boldsymbol{a}$ is dominated by $\boldsymbol{b}$, or $\boldsymbol{a} \leq \boldsymbol{b}$, iff each component of $\boldsymbol{a}$ is smaller or equal to the corresponding component in $\boldsymbol{b}$. For a vector $\boldsymbol{a}$, we use a_1, a_2, a_3 to denote the first, second and third component of $\boldsymbol{a}$.

7.3.2 TCEC as MCP

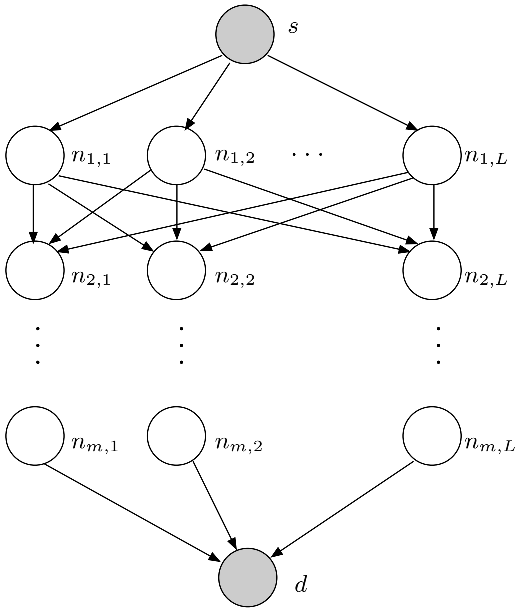

Figure 7-2. Job execution graph

An instance TCEC can be reduced to an instance of MCP, if we view the execution jobs at different voltage levels as a path in job execution graph (JEG). As shown in Figure 7-2, a JEG contains a source node s, a destination node d, and m layers of job (task) nodes. In each layer, there are l nodes for each voltage level. Edges only exist between different layers of job nodes, or job nodes and source/destination nodes. Formally, we define JEG as follows.

Definition 6. ***Job execution graph (JEG)*** *is an acyclic directed graph* $G = (V, E)$ *with following properties:* $V = \{s, d\} \bigcup \{n_{i,j} | 1 \leq i \leq m, 1 \leq j \leq l\}$*;* $E = \{(s, n_{1,j}) | 1 \leq j \leq l\} \bigcup \{(n_{m,j}, d) | 1 \leq j \leq l\} \bigcup \{(n_{i,j}, n_{i+1,j'}) | 1 \leq i < m, 1 \leq j, j' \leq l\}$.

In order to calculate the values of time, energy and temperature on JEG, we recursively define path transfer functions for path $p = e_0 || e_1 || \cdots || e_{i-1} || e_i$ $(1 \leq i \leq m)$ from s to $n_{i,j}$ as:

$$f_t^p(t_0) = f_t^q(t_0) + t_{i,j} + \omega(j', j) \tag{7–8}$$

$$f_w^p(w_0) = f_w^q(w_0) + w_{i,j} + \psi(j', j) \tag{7–9}$$

$$\begin{aligned} f_T^p(T_0) =& \beta \cdot f_T^q(T_0) + (1 - \beta) \cdot T_s, \\ \beta =& e^{-t_{i,j}/RC} \end{aligned} \tag{7–10}$$

where $q = e_0||e_1|| \cdots ||e_{i-1}$ is a prefix of p, which starts from s and ends at $n_{i-1,j'}$. For $p = e_0 = (s, n_{1,j})$,

$$f_t^p(t_0) = t_0 \tag{7–11}$$

$$f_w^p(w_0) = w_0 \tag{7–12}$$

$$f_T^p(T_0) = T_0 \tag{7–13}$$

where t_0, w_0 and T_0 are the time, energy consumption and temperature before the execution of the task set. Normally, we have $t_0 = w_0 = 0$ and $T_0 = T_{init}$. We can also write the path transfer functions in vector form

$$\boldsymbol{f}^p(\boldsymbol{I}) \triangleq \begin{bmatrix} f_t^p(t_0) \\ f_w^p(w_0) \\ f_T^p(T_0) \end{bmatrix} \tag{7–14}$$

where $\boldsymbol{I} = [t_0 \quad w_0 \quad T_0]^T$.

Using the above definition, the value of time, energy consumption and temperature of first i jobs with voltage assignment $\{j_1, j_2, \cdots, j_i\}$ can be expressed as $\boldsymbol{f}^p(\boldsymbol{I})$, where $p = s \rightarrow n_{1,j_1} \rightarrow \cdots \rightarrow n_{i,j_i}$. We use the example in Figure 7-3 to illustrate such computation in practice. In this case, we have $m = 2$ jobs and $l = 2$ voltage levels. Suppose that the initial temperature $T_0 = 65°C$ and constant $RC = 30us$. The design constraints are deadline $D = 32us$, energy budget $W = 55mJ$ and

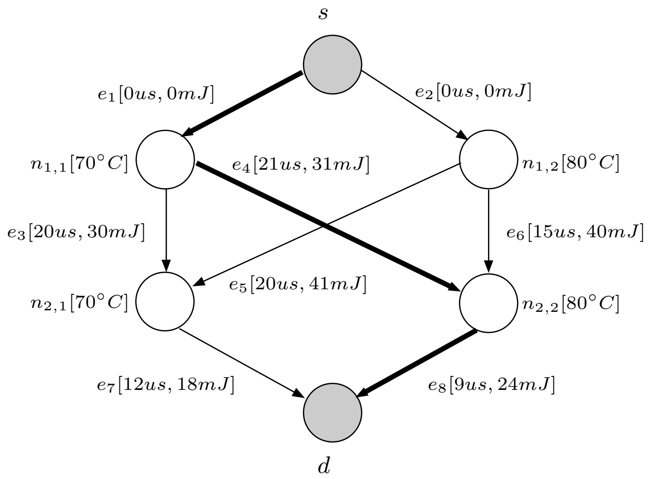

Figure 7-3. JEG of TCEC. The values next to each edge are corresponding time and energy consumption.

maximum temperature $T_{max} = 75^\circ C$. Assume that we decide to use voltage level 1 and 2 to execute job 1 and 2 respectively. Based on the definition of JEG, this voltage assignment corresponds to $s - d$ path $p = e_1||e_4||e_8$ (highlighted). The time consumption after the execution of all jobs can therefore be computed as

$$\begin{aligned} f_t^{e_1||e_4||e_8}(0) &= f_t^{e_1||e_4}(0) + 9us \\ &= f_t^{e_1}(0) + 21us + 9us \\ &= 0us + 21us + 9us \\ &= 30us \end{aligned}$$

Similarly, we can compute the energy consumption of p as

$$f_w^{e_1||e_4||e_8}(0) = 0mW + 31mJ + 24mJ = 55mJ$$

and the final temperature of p as

$$\begin{aligned} f_T^{e_1||e_4||e_8}(0) = (e^{-\frac{9}{30}}(e^{-\frac{21}{30}} \cdot 65^\circ C + (1 - e^{-\frac{21}{30}}) \cdot 70^\circ C) \\ + (1 - e^{-\frac{9}{30}}) \cdot 80^\circ C) = 70.8^\circ C \end{aligned}$$

In other words, our schedule or path p satisfies the constraints $D = 32us$, $W = 55mJ$ and $T_{max} = 75°C$.

Clearly, the model checking problem discussed in [88] can be answered by checking whether there exists a path p, such that $\boldsymbol{f}^p(\boldsymbol{I}) \leq \boldsymbol{C}$, where $\boldsymbol{C} = [D \quad W \quad T_{max}]^T$. The formal definition of our MCP problem is as follows.

Definition 7. $MCP(G, \boldsymbol{I}, \boldsymbol{C})$ ***instance****: Given a job execution graph G, an initial state vector $\boldsymbol{I} = [t_0, w_0, T_0]^T$, a constraint vector $\boldsymbol{C} = [D, W, T_{max}]^T$, is there an $s - d$ path $p = e_0||...||e_m$ such that for all $0 \leq i \leq m$*

$$\boldsymbol{f}^{e_0||\cdots||e_i}(I) \leq \boldsymbol{C}$$

The definition above seems to be tighter than the definition of TCEC given in Section 7.1.4, because all constraints are enforced after each job, while the deadline and energy constraint are enforced only after the last job in TCEC. However, they are essentially equivalent due to monotonic nature of execution time and energy consumption.

In the rest of the chapter, we will use MCP to present $MCP(G, \boldsymbol{I}, \boldsymbol{C})$ for ease of illustration. Our definition of MCP differs from Quality of Service (QoS) MCP problems [26, 76, 96, 98] in networking, because the computation of the temperature is not additive. As a result, the existing techniques can not be applied directly to solve our problem.

7.3.3 An Exact Algorithm for MCP

We have developed Algorithm 7, which is an extended Bellman-Ford (EBF) algorithm used for computing the exact answer for MCP problem. It is developed based on the EBF algorithms in [26, 76, 98], which were used to solve MCP with constant additive constraints. This algorithm accepts an MCP instance, including JEG G, initial state vector $\boldsymbol{I}$, constraint vector $\boldsymbol{C}$, and returns the answer to the MCP problem. The basic idea of this algorithm is to keep updating a path set $Path(v)$ for each vertex v,

Algorithm 7: Extended Bellman-Ford (EBF) Algorithm

```
$EBF(G, \boldsymbol{I}, \boldsymbol{C})$
for each $v \in V$ do
    $Path(v) = \emptyset$
end for
for $j = 1$ to $|l|$ do
    $Path(n_{1,j}) = \{(s, n_{1,j})\}$
end for
for $i = 2$ to $|m|$ do
    for $j = 1$ to $|l|$ do
        for each edge $(u, n_{i,j}) \in E$ do
            Relax$(u, n_{i,j})$
        end for
    end for
end for
for each edge $(u, d) \in E$ do
    if Relax$(u, d)$ then
        return $TRUE$
    end if
end for
return $FALSE$

$Relax(u, v)$
for each $p \in Path(u)$ such that $\boldsymbol{f}^{p||(u,v)}(\boldsymbol{I}) \leq \boldsymbol{C}$ do
    $Skip = FALSE$
    for each $q \in Path(v)$ do
        if $\boldsymbol{f}^{q}(\boldsymbol{I}) \leq \boldsymbol{f}^{p||(u,v)}(\boldsymbol{I})$ then
            $Skip = TRUE$
            Break
        end if
    end for
    if $Skip = FALSE$ then
        Insert $p||(u,v)$ into $Path(v)$
    end if
    if $v = d$ then
        return $TRUE$
    end if
end for
return $FALSE$;
```

which is a subset of all possible $s - v$ paths. By implicitly enumerating all possible paths between s and t, we just need to check whether there is any path $p \in Path(d)$ that satisfies the constraint vector $\boldsymbol{C}$. This enumeration is accomplished by calling function Relax on all edges for $|V|$ times (line 4-7 in EBF). All paths are examined implicitly because the longest path in acyclic graph G contains $|V|$ edges. To improve the efficiency, $Relax$ will add a new path $p||(u, v)$ only when it does not dominate any existing paths in $Path(v)$.

Example 2: We use the same example in Figure 7-3 to demonstrate the execution of EBF. First, we initialize $Path(n_{1,1}) = \{e_1\}$ and $Path(n_{1,2}) = \{e_2\}$. Then, we perform $Relax$ on edges e_3 and e_4, which start from $n_{1,1}$ (line 7-8 in EBF). In $Relax(e_3)$, we attempt to create new paths from s to $n_{2,1}$ by appending e_3 to known path from s to $n_{1,1}$ and $n_{1,2}$, which are stored in $Path(n_{1,1})$ and $Path(n_{1,2})$. Since $Path(n_{1,1}) = \{e_1\}$, we just need to check path $e_1||e_3$. It is easy to see that $\boldsymbol{f}^{e_1||e_3}(I) = [20\ 30\ 67.4]^T$ is dominated by constraints $\boldsymbol{C} = [D\ W]^T = [32\ 55\ 75]^T$, i.e., constraints are not violated. Therefore, path $e_1||e_3$ is inserted into $Path(n_{2,1})$, which was empty. On the other hand, path $e_2||e_5$ will not be added into $Path(n_{2,1})$ during $Relax(e_5)$, because $\boldsymbol{f}^{e_2||e_5}(I) = [20\ 41\ 72.3]^T$ dominates $\boldsymbol{f}^{e_1||e_3}(I) = [20\ 30\ 67.4]^T$. The reason is that if there exists a path in $Path(n_{2,1})$ like $e_1||e_3$, which has less time/energy consumption than the new path $e_2||e_5$, the new path cannot be a prefix of the optimal path. We repeat the above process until we reach node d. If $Path(d)$ contains a path , which satisfies all the constrains like $e_1||e_4||e_8$, EBF finds the required schedule and returns true. Otherwise, we conclude that such schedule does not exist.

Although EBF is guaranteed to find the exact answer to MCP, its time complexity is quite high. As shown in Algorithm 7, $Relax$ is executed for $m \cdot l$ times, while the time complexity of $Relax$ can be $O(|Path_{max}|^2)$, where $|Path_{max}| = \max_{v \in V} Path(v)$. Therefore, the overall complexity of EBF is $O(m \cdot l \cdot |Path_{max}|^2)$, which is only pseudo-polynomial, because in the worst case $|Path_{max}|$ can be $O(l^m)$. Unfortunately, we may not be able to find a solution to MCP in polynomial time. As indicated in [89], we can reduce the Partition problem to an MCP instance by properly setting $t_{i,j}$ and $w_{i,j}$. In other words, MCP is NP-Complete.

7.3.4 Approximation Algorithm

Before we introduce our approximation scheme for MCP, we first present another problem MCP_ϵ, which is closely related to MCP.

Definition 8. $MCP_\epsilon(G, \boldsymbol{I}, \boldsymbol{C})$ ***instance***: *Given a positive constant* $\epsilon > 0$*, a job execution graph* G*; an initial state vector* $\boldsymbol{I} = [t_0, w_0, T_0]^T$*; a constraint vector* $\boldsymbol{C} = [D, W, T_{max}]^T$*, there exists an* $s - d$ *path* $p = e_m||...||e_0$ *such that for all* $0 \leq i \leq m$

$$f_t^{e_0||\cdots||e_i}(t_0) \leq D$$

$$f_w^{e_0||\cdots||e_i}(w_0) \leq (1-\epsilon)W$$

$$f_T^{e_0||\cdots||e_i}(T_0) \leq (1-\epsilon)T_{max}$$

MCP_ϵ is tighter than MCP. Any $s - d$ path that satisfies the constraints in MCP_ϵ also satisfies the constraints in MCP, but not vice versa. In this section, we are going to develop an approximation algorithm EBF_ϵ to MCP, such that 1) EBF_ϵ is true implies MCP is true, and 2) EBF_ϵ is false implies MCP_ϵ is false. In other words, EBF_ϵ gives no false positive answer to MCP. It may give false negative answer when the exact answers to MCP and MCP_ϵ are true and false respectively (i.e., there are feasible paths for MCP, but no feasible path for MCP_ϵ). Since MCP_ϵ becomes MCP when $\epsilon = 0$, EBF_ϵ will be more and more accurate when $\epsilon \to 0$.

In a JEG $G = (V, E)$, we define functions h_1, h_2, h_3 on each edge to simplify the description of our approximation scheme. Here, h_1, h_2, and h_3 corresponds to the functions related to the functions of time, energy and temperature, respectively. For

$e = (s, n_{1,j}) \in E$ $(1 \leq j \leq l)$, we define

$$h_1^e(x_1) = x_1 \tag{7–15}$$

$$h_2^e(x_2) = x_2 \tag{7–16}$$

$$h_3^e(x_3) = x_3 \tag{7–17}$$

For other $e \in E$,

$$h_1^e(x_1) = x_1 + t_{i,j} + \omega(j', j) \tag{7–18}$$

$$h_2^e(x_2) = x_2 + w_{i,j} + \psi(j', j) \tag{7–19}$$

$$h_3^e(x_3) = \beta \cdot x_3 + (1 - \beta) \cdot T_s, \tag{7–20}$$

$$\beta = e^{-t_{i,j}/RC} \tag{7–21}$$

Based on the definition of path transfer functions, it is easy to see that for path $p = e_0 || \cdots || e_i$,

$$f_t^p(t_0) = h_1^{e_i} \circ \cdots \circ h_1^{e_0}(t_0)$$

$$f_w^p(w_0) = h_2^{e_i} \circ \cdots \circ h_2^{e_0}(w_0)$$

$$f_T^p(T_0) = h_3^{e_i} \circ \cdots \circ h_3^{e_0}(T_0)$$

where $\circ$ is the composition operation for successive invocation functions.

The basic idea of our approximation scheme is to build a table Z_n for each node n. Each cell in this table holds the least value of time consumption among all execution paths, which have the same energy and temperature value after scaling. In other words, each cell represents an optimal execution path. Dynamic programming is then applied to fill each Z_n. The approximated solution can be obtained by checking Z_d, which holds the approximated least time consumption of all possible execution paths.

Algorithm 8 shows the details of our approximation algorithm EBF_ϵ. Initially, we compute the table size M and the "step size" Δ_k for each constraint based on the value of ϵ (line 1 and 2 of EBF_ϵ), and then initialize $M * M$ tables Z_n for each node in G. Here,

Algorithm 8: .

```
$EBF_\epsilon(G, \boldsymbol{I}, \boldsymbol{C})$
$M = \lceil (m+1)/\epsilon \rceil$
$\Delta_k = \epsilon * C_k/(m+1), k = 2, 3$
for each $v \in G$ do
    for each $(c_2, c_3) \in \{0, 1, .., M\}^2$ do
        $Z_v(c_2, c_3) = \infty$
        $\pi_v(c_2, c_3) = null$
    end for
end for
$Z_s(\lceil I_2/\Delta_2 \rceil, \lceil I_3/\Delta_3 \rceil) = 0$
for $i = 1$ to $|m|$ do
    for $j = 0$ to $|l|$ do
        for each edge $(u, n_{i,j}) \in E$ do
            $Relax_\epsilon(u, n_{i,j})$
        end for
    end for
end for
for each edge $(u, d) \in E$ do
    if $Relax_\epsilon(u, d)$ then
        return $TRUE$
    end if
end for
return $FALSE$

$Relax_\epsilon(u, v)$
for each $(c_2, c_3) \in \{0, 1, .., M\} \times \{0, 1, .., M\}$ do
    if $h_k^{(u,v)}(c_k * \Delta_k) \leq C_k$ for $k = 2, 3$ then
        $b_k = \lceil h_k^{(u,v)}(c_k * \Delta_k)/\Delta_k \rceil, k = 2, 3$
        $Z_{new} = h_1^{(u,v)}(Z_u(c_2, c_3))$
        if $Z_{new} < Z_v(b_2, b_3)$ and $Z_{new} \leq C_1$ then
            $Z_v(b_2, b_3) = Z_{new}$
            $\pi_v(b_2, b_3) = (u, c_2, c_3)$
            if $v = d$ then
                return $TRUE$
            end if
        end if
    end if
end for
return $FALSE$;
```

the “step size” Δ_k is used to scale the energy and temperature values as indices in the table. For example, cell $(\lceil I_2/\Delta_2 \rceil, \lceil I_3/\Delta_3 \rceil)$ in Z_s holds the time consumption before we execute any jobs, which is initialized as 0 in line 7. The rest of EBF_ϵ is similar to EBF. We use dynamic programming to fill each Z_n by calling $Relax_\epsilon$, which can be viewed as a scaled version of $Relax$. In $Relax_\epsilon(u, v)$, we traverse Z_u to fill Z_v by extending paths in Z_u. Since Z_u is an M by M table, we use c_2 and $c_3 \in \{0, 1, .., M\}$ as index variables (line 1). As we have discussed previously, each cell $Z_u(c_2, c_3)$ represents an execution path from s to u with time consumption $Z_u(c_2, c_3)$, energy consumption $c_2 * \Delta_2$ and temperature $c_3 * \Delta_3$ [2] . In line 2 of $Relax_\epsilon$, we first check whether the energy and temperature constraints are violated if the job is executed based on edge (u, v). If no violation occurs, we calculate the scaled version of the new energy and temperature values (b_2, b_3). After that, we compare the new time consumption $Z_{new} = h_1^{(u,v)}(Z_u(c_2, c_3))$ with the current value in $Z_v(b_2, b_3)$ and update Z_v when necessary. If we already reach destination d and the time consumption Z_{new} is still less than the required value C_1[3] , we have found the required schedule. Compared with $Relax$, $Relax_\epsilon$ does not store the paths explicitly as $Path(v)$ in $Relax$, but implicitly in different cells within each table.

EBF_ϵ is a polynomial time algorithm for a given ϵ, because the complexity of $Relax_\epsilon$ is M^2 or $(m/\epsilon)^2$. $Relax_\epsilon$ is executed for $m \cdot l$ times. Therefore, the overall time complexity is $O(m \cdot l \cdot (m/\epsilon)^2)$. Now, we show that EBF_ϵ is a polynomial time algorithm with the approximation properties as claimed by the following two theorems.

Theorem 7.1. *Given an instance of $MCP(G, \boldsymbol{I}, \boldsymbol{C})$, if $EBF_\epsilon(G, \boldsymbol{I}, \boldsymbol{C})$ returns $TRUE$, $MCP(G, \boldsymbol{I}, \boldsymbol{C})$ is true.*

[2] Recall that indices in table are scaled version of energy and temperature values. We can obtain the actual energy and temperature values by multiplying table indices with Δ_2 and Δ_3.

[3] C_1, C_2, and C_3 are the constraints for time, energy, and temperature, respectively.

Proof. When EBF_ϵ returns $TRUE$, let the path $p = e_0||...||e_m$ be the path constructed by tracing back using table π. Clearly, p is a $s - d$ path. We need to show that for all $0 \leq i \leq m$

$$h_k^{e_i} \circ ... \circ h_k^{e_0}(I_k) \leq C_k, k = 1, 2, 3 \tag{7–22}$$

Clearly, p satisfies Equation (7–22) for $k = 1$, because the condition on line 5 of $Relax_\epsilon$ guarantees that

$$h_1^{e_i} \circ ... \circ h_1^{e_0}(I_1) \leq C_1, 0 \leq i \leq m$$

Since p is constructed by π, it is easy to see that

$$C_k \geq h_k^{e_0}(\lceil I_k/\Delta_k \rceil * \Delta_k), k = 2, 3$$

Otherwise, condition on line 2 of $Relax_\epsilon$ would not be satisfied during $Relax_\epsilon(e_0)$, and line 7 of $Relax_\epsilon$ would not be executed. This contradicts the fact that e_0 is recorded in π.

Similarly, for $0 \leq i \leq m$ and $k = 2, 3$, we have

$$C_k \geq h_k^{e_1}(\lceil h_k^{e_0}(\lceil I_k/\Delta_k \rceil * \Delta_k)/\Delta_k \rceil * \Delta_k)$$

$$...$$

$$C_k \geq h_k^{e_i}(\lceil ... \lceil h_k^{e_0}(\lceil I_k/\Delta_k \rceil * \Delta_k)/\Delta_k \rceil * \Delta_k .../\Delta_k \rceil * \Delta_k)$$

Or

$$C_k \geq h_k^{e_i} \circ g_k \circ ... \circ h^{e_1} \circ g_k \circ h_k^{e_0} \circ g_k(I_k)$$

where g_k is a “ceiling” function

$$g_k(x) = \lceil x/\Delta_k \rceil * \Delta_k$$

Since $h_k^{e_i}$ and g_k are monotonically increasing functions and $g_k(x) \geq x$, we have following relations

$$\begin{aligned} \lceil I_k/\Delta_k \rceil * \Delta_k = g_k(I_k) &\geq I_k \\ h_k^{e_0} \circ g_k(I_k) &\geq h_k^{e_0}(I_k) \\ h_k^{e_1} \circ g_k \circ h_k^{e_0} \circ g_k(I_k) &\geq h_k^{e_1} \circ h_k^{e_0}(I_k) \\ &... \\ h_k^{e_m} \circ g_k \circ ... \circ h_k^{e_0} \circ g_k(I_k) &\geq h_k^{e_m} \circ ... \circ h_k^{e_0}(I_k) \end{aligned}$$

Thus, for $0 \leq i \leq m$

$$\begin{aligned} C_k &\geq h_k^{e_i} \circ g_k \circ ... \circ h_k^{e_1} \circ g_k \circ h_k^{e_0} \circ g_k(I_k) \\ &\geq h_k^{e_i} \circ ... \circ h_k^{e_0}(I_k) \end{aligned}$$

Therefore, Equation (7–22) also holds on p for $k = 2, 3$. By the definition of MCP, $MCP(G, \boldsymbol{I}, \boldsymbol{C})$ is true. □

Lemma 3. *Given an instance of $MCP_\epsilon(G, \boldsymbol{I}, \boldsymbol{C})$, if there is an $s - d$ path $p = e_0||...||e_{m-1}||e_m$ such that*

$$h_1^{e_i} \circ ... \circ h_1^{e_0}(I_1) \leq C_1 \tag{7–23}$$

$$h_k^{e_i} \circ g_k \circ ... \circ h_k^{e_0} \circ g_k(I_k) \leq C_k, k = 2, 3 \tag{7–24}$$

holds for $0 \leq i \leq m$, EBF_ϵ will return TRUE.

Lemma 3 can be proven by considering the following fact that if we only perform $Relax_\epsilon$ on edges that are in p, Equation (7–24) and Equation (7–23) guarantees that the conditions on line 2 and 5 in $Relax_\epsilon$ are satisfied and line 6 will be executed in each round. Eventually, EBF_ϵ will return true. If we perform $Relax_\epsilon$ on more edges, the minimal value in Z_d will not increase. As a result, EBF_ϵ still returns true.

Theorem 7.2. *Given an instance of $MCP_\epsilon(G, \boldsymbol{I}, \boldsymbol{C})$, $MCP_\epsilon(G, \boldsymbol{I}, \boldsymbol{C})$ is true implies $EBF_\epsilon(G, \boldsymbol{I}, \boldsymbol{C})$ returns TRUE.*

Proof. We just need to show that if there is an $s - d$ path

$$p = s \rightarrow n_{1,j_1} \rightarrow \cdots \rightarrow n_{m,j_m} \rightarrow d$$

$$= e_0 || ... || e_{m-1} || e_m$$

such that for all $0 \leq i \leq m$

$$h_1^{e_i} \circ ... \circ h_1^{e_0}(I_1) \leq C_1 \tag{7–25}$$

$$h_2^{e_i} \circ ... \circ h_2^{e_0}(I_2) \leq (1 - \epsilon) C_2, k = 2, 3 \tag{7–26}$$

it also satisfies Equation (7–23) and (7–24).

Clearly, for any edge $e \in E$

$$h_2^e(c + \Delta) \leq h_2^e(c) + \Delta$$

For h_3^e, which represents the temperature constraints, we have

$$h_3^e(c + \Delta) = h_3^e(c) + \Delta * \beta \leq h_3^e(c) + \Delta,$$

$$\beta = e^{\frac{-t}{RC}}$$

because $\beta \leq 1$.

Using ceiling functions $g_k(x) = \lceil x / \Delta_k \rceil * \Delta_k, k = 2, 3$, it is easy to verify

$$g_k(I_k) \leq I_k + \Delta_k$$

By applying $h_k^{e_i}$ on its both sides, we have

$$h_k^{e_0} \circ g_k(I_k) \leq h_k^{e_0}(I_k + \Delta_k) \leq h_k^{e_0}(I_k) + \Delta_k, k = 2, 3$$

because $h_k^{e_0}$ is a monotonic function. Therefore,

$$h_k^{e_0} \circ g_k(I_k) \leq h_k^{e_0}(I_k) + \Delta_k,$$

$$g_k \circ h_k^{e_0} \circ g_k(I_k) \leq h_k^{e_0}(I_k) + 2\Delta_k,$$

$$h_k^{e_1} \circ g_k \circ h_k^{e_0} \circ g_k(I_k) \leq h^{e_1} \circ h_k^{e_0}(I_k) + 2\Delta_k$$

$$...$$

$$h_k^{e_m} \circ g_k \circ ... \circ h_k^{e_0} \circ g_k(I_k) \leq h_k^{e_m} \circ ... \circ h_k^{e_0}(I_k) + (m+1) * \Delta_k$$

From Equation (7–26), we know that

$$h_k^{e_i} \circ ... \circ h_k^{e_0}(I_k) \leq (1-\epsilon) * C_k, k = 2, 3$$

Thus, for $0 \leq i \leq m$ $k = 2, 3$, we have

$$\begin{aligned} h_k^{e_i} \circ g_k \circ ... \circ h_k^{e_0} \circ g_k(I_k) &\leq (1-\epsilon) * C_k + (m+1) * \Delta_k \\ &\leq (1-\epsilon) * C_k + \epsilon * C_k = C_k \end{aligned}$$

Therefore, p satisfies Equation (7–23) and Equation (7–24). Using Lemma 3, EBF_ϵ will return true. □

Now we use Theorem 7.2 to investigate under what constraints EBF_ϵ yields false negative answers. Considering the fact that $MCP(G, \boldsymbol{I}, \boldsymbol{C}_0)$ with $\boldsymbol{C}_0 = [D, W, T_{max}]^T$ and $MCP_\epsilon(G, \boldsymbol{I}, \boldsymbol{C}_0')$ with $\boldsymbol{C}_0' = [D, W/(1-\epsilon), T_{max}/(1-\epsilon)]^T \approx [D, (1+\epsilon)W, (1+\epsilon)T_{max}]^T$ (when ϵ is small) are identical, Theorem 7.2 can also be interpreted as follows.

Corollary 1. *For any small $\epsilon < 1$, $MCP(G, \boldsymbol{I}, \boldsymbol{C}_0)$ with $\boldsymbol{C}_0 = [D, W, T_{max}]^T$ is true implies $EBF_\epsilon(G, \boldsymbol{I}, \boldsymbol{C}_0')$ with $\boldsymbol{C}_0' = [D, (1+\epsilon)W, (1+\epsilon)T_{max}]^T$ returns TRUE.*

In other words, $EBF_\epsilon(G, \boldsymbol{I}, \boldsymbol{C})$ will not produce a false negative answer, when $\boldsymbol{C}$ dominates $\boldsymbol{C}_0'$. For example, Figure 7-4 shows the region where EBF_ϵ may produce false negative answers for different (W, T_{max}) pair. In this example, there are two feasible constraints $[D, W_1, T_{max1}]^T$ and $[D, W_2, T_{max2}]^T$, which dominates no other feasible constraints except themselves. It is easy to see that EBF_ϵ will generate false negative answers only in the cross-marked region based on Corollary 1. Clearly, the area of the false negative region is linearly depends on ϵ. Therefore, when ϵ is small enough, EBF_ϵ produces false negative answers in rare cases.

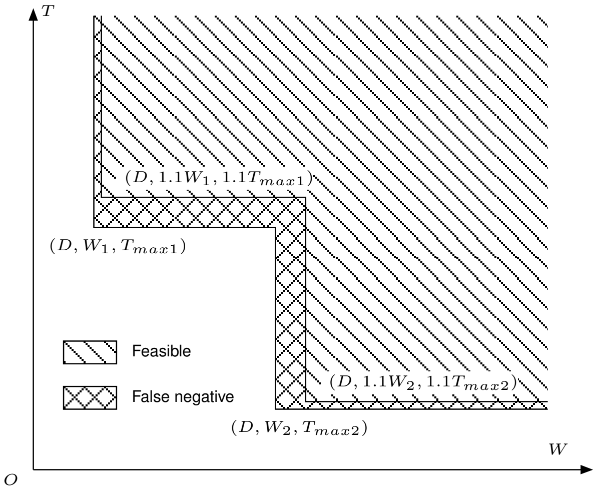

Figure 7-4. Possible false negative region. $\epsilon = 0.1$

7.4 Problem Variants

Our approach is also applicable to other problem variants by modifying the property and making suitable changes to invocation of the problem solving driver in Figure 7-1 (model checker of approximation algorithm).

Task set with individual deadlines: In the scenario where each task has its own deadline, we have to make sure that the execution blocks finish no later than their corresponding task's deadline. Suppose that the deadline of the i^{th} execution block is $D[i]$. Equation (7–3) is replaced by following constraints, for all $1 \leq i \leq m$:

$$\sum_{i=1}^{m} t_{i,l_i} + \omega_{l_{i-1},l_i} \leq D[i], \forall D[i] > 0 \tag{7–27}$$

The approximation algorithm EBF_ϵ can also be modified sightly to the individual deadline case. We only need to replace C_1 (line 5 in $Relax_\epsilon$) with $D[i]$, when node u represents job i. Since the approximation is applied on the energy and temperature constraints, all the properties and related proofs of EBF_ϵ still hold.

Periodic Tasks: We solve the TCEC scheduling of periodic tasks by considering the scheduling of tasks within a hyperperiod. We have to make sure that a) all tasks meets their corresponding deadlines in every hyper-period, b) the temperature constraints are not violated after execution of any hyperperiod. Clearly, the first requirement can be achieved by adding the deadline constraints as we discussed in task set with individual deadlines.

The second requirement is satisfied by only choosing the schedules, whose temperature at the end of the hyper-period is less than or equal to the initial temperature. As discussed in Section 7.1.4, we enforce this requirement by adding constraint (7–7).

The approximation algorithm EBF_ϵ needs to be modified as follows. When $Relax_\epsilon$ is applied to the node corresponding to the last task, we need to ensure that $h_3^{(u,v)}(c_3 * \Delta_3) \leq T_{init}$ (line 2 in $Relax_\epsilon$). In addition, the step size of temperature should be calculated based on T_{init}, i.e., $\Delta_3 = \epsilon * T_{init}/(m+1)$ (line 2 of EBF_ϵ). we verified that all the properties and related proofs of EBF_ϵ still hold.

TC: Temperature-constrained scheduling problem is a simplified version of TCEC. It only needs to ensure that the maximum instantaneous temperature is always below the threshold $temp_{max}$.

TA: To find a schedule so that the maximum temperature is minimized, we can employ a binary search over the temperature value range. Each iteration invokes the problem solving driver to test the current temperature constraint T_{max}. Initially, T_{max} is set to the mid-value of the range. If the property is unsatisfied, we search in the range of values larger than T_{max} in the next iteration. If the property is satisfied, we continue to search in the range of values lower than T_{max} to further explore better results. This process continues until the lower bound is larger than the upper bound. The minimum T_{max} and associated schedule, which makes the property satisfiable during the search, is the result. Note that the temperature value range for microprocessors is small in practice, e.g., $[30°C, 120°C]$. Hence, the number of iterations is typically no more than 7.

To adopt our approximation scheme in the above cases, we can ignore the energy constraint.

7.5 Experiments

7.5.1 Experimental Setup

In this section, we describe the experimental setup for evaluation of our approach. A DVS-capable processor StrongARM [61] is modeled with four voltage/frequency levels (1.5V-206MHz, 1.4V-192Mhz, 1.2V-162MHz and 1.1V-133MHz). We use synthetic task sets which are randomly generated with each of them having execution time in the range of 100 - 500 milliseconds. These are suitable and practical sizes to reflect variations in temperature, and millisecond is a reasonable time unit granularity [101]. We adopt the thermal resistance (R) and thermal capacitance (C) values from [46], which are 1.83$^\circ C/Watt$ and 112.2$mJoules/^\circ C$, respectively. The ambient temperature of the processor is 32$^\circ C$. The scheduler and TAG shown in Figure 7-1 are implemented in C++. The exact algorithm EBF and the approximation algorithm EBF_ϵ are also implemented in C++. All experiments are performed on a computer with AMD64 2GHz CPU and 16G RAM.

7.5.2 TCEC versus TC or EC

This section demonstrates that existing solutions based on TC or EC are not sufficient to find TCEC schedules. We compared the schedule generated by energy constrained scheduling algorithm [95] and our TCEC scheduling for the same set of jobs under the same energy constraint. We also require that the system temperature after the execution of task set does not exceed the initial temperature T_{init}, so that the temperature constraint is not violated even if the task set is executed repeatedly. The results are shown in Figure 7-5.

It can be seen that the schedule generated by [95], which considers only energy constraint takes less execution time. However, it violates temperature constraint. On the other hand, the schedules generated by our TCEC approach will not exceed the

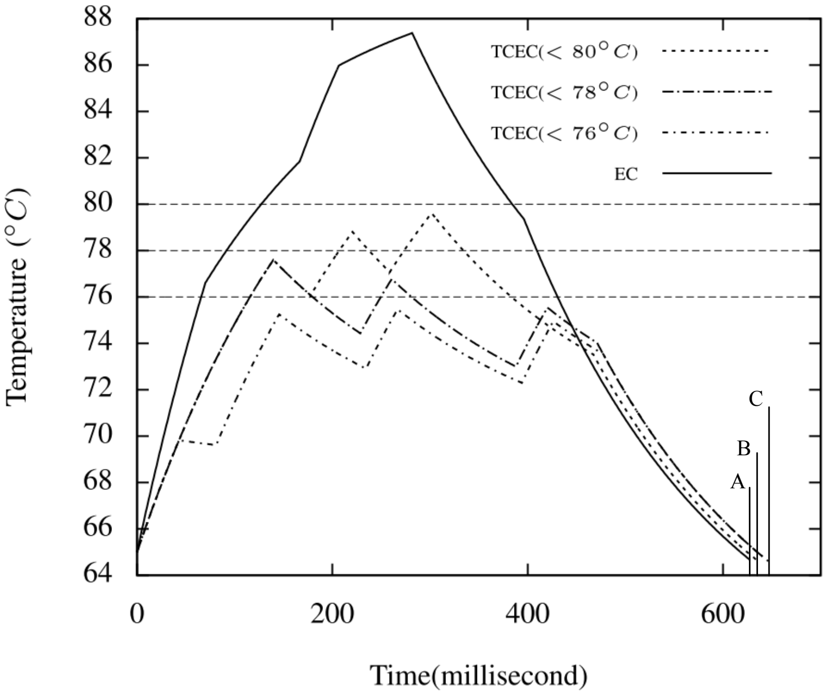

Figure 7-5. EC vs TCEC. EC finishes at A. TCEC($< 80°C$) finishes at B. Both TCEC($< 78°C$) and TCEC($< 76°C$) finish at C.

respective temperature constraints ($80°C$, $78°C$ and $76°C$, respectively), although it takes a little longer execution time. Therefore, scheduling algorithms that consider only energy constraint are not suitable, when we want to control the maximum temperature of the processor during job execution.

We also compared our TCEC scheduling with temperature constrained scheduling algorithm [101]. The experiments were performed on the same job set with the same temperature constraints. For TCEC, we applied three different energy constraints. We also require that the system temperature after the execution of task set does not exceed the initial temperature T_{init}. Figure 7-6 presents the results. Since TC has no constraint on energy consumption, it always tries to execute jobs with high voltage, which may lead to peak temperature several times. As a result, TC has the shortest execution time. However, once we consider energy constraint, it may not be possible to execute some jobs at high voltage. When the energy budget is very tight, we may not be able to reach

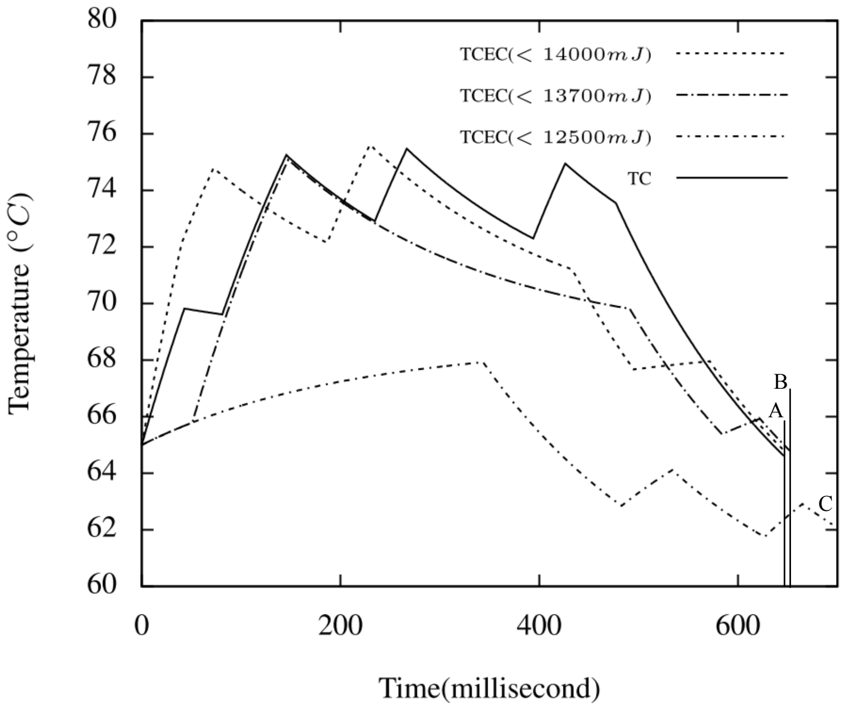

Figure 7-6. EC vs TCEC. Both TC and TCEC($< 14000mJ$) finish at A. TCEC($< 13700mJ$) finishes at B. TCEC($< 12500mJ$) finishes at C.

the maximum temperature during the entire execution, like curve “TCEC(<12500mJ)” in Figure 7-6. In this case, TC will clearly violate the energy constraint, while our TCEC obtains a schedule within the energy budget.

7.5.3 TCEC using Approximation Algorithm

We compared the efficiency of conventional symbolic model checker (UPPAAL) with our approximation algorithm EBF_ϵ on task sets with different number of blocks. Since the TCEC problem can also be modeled using ILP, we include the corresponding results of ILP formulation. The first and the second column are the index and number of blocks in each task set, respectively. The next three columns present the temperature constraint (TC, in $^\circ C$), energy constraint (EC, in mJ), and deadlines (DL, in ms) to be checked on the model. The sixth column indicates whether there exists a schedule which satisfies all the constraints. The last three columns of Table 7-1 shows the results (running time in seconds) of UPPAAL, ILP formulation solved with lpsolve [10], and our approach EBF_ϵ. Since UPPAAL failed to produce result for task set 4 and 5, we only report the running time of EBF_ϵ. It can be seen that EBF_ϵ outperforms UPPAAL by more than 10 times on average. Moreover, EBF_ϵ can solve much larger problems in reasonable running time.

Table 7-1. Running time comparison on different task sets

TS	#Blk	TC	EC	DL	Found?	UPPAAL	lpsolve	EBF_ϵ
1	10	85	180000	7000	Y	9.6	30.5	0.2
		85	150000	8000	Y	9.9	32.1	0.2
		80	140000	8000	N	9.4	29.4	0.2
2	12	85	70000	2500	Y	18.5	190.3	0.3
		85	60000	2700	Y	106.6	621.2	0.3
		80	60000	2500	N	17.5	282.4	0.3
3	14	90	90000	2600	Y	65.1	-	0.3
		85	80000	2800	Y	648.3	-	0.3
		90	80000	2700	N	208.6	-	0.3
4	50	85	380000	39500	Y	-	-	20.2
5	100	85	720000	83800	Y	-	-	102.5

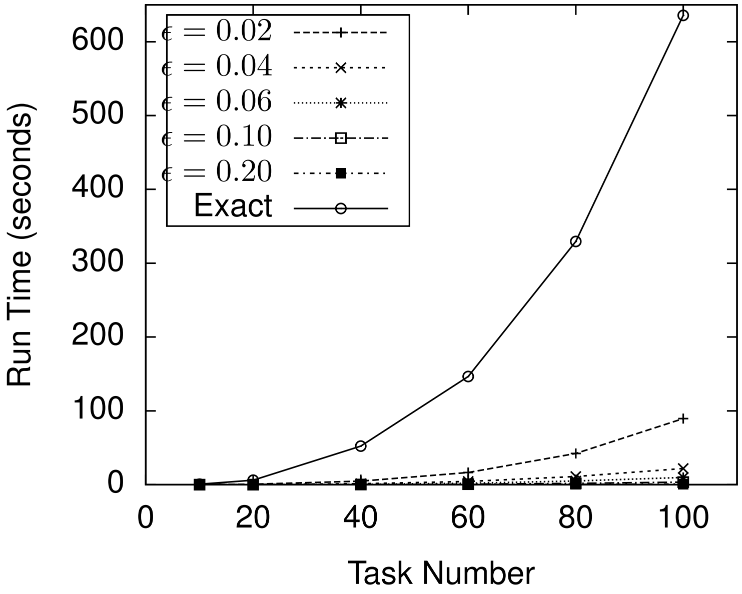

Figure 7-7. Running time with different job set size and ϵ.

We also evaluated the running time of our approximation scheme with different ϵ. The results are shown in Figure 7-7. Curve "Exact" represents the execution time of the

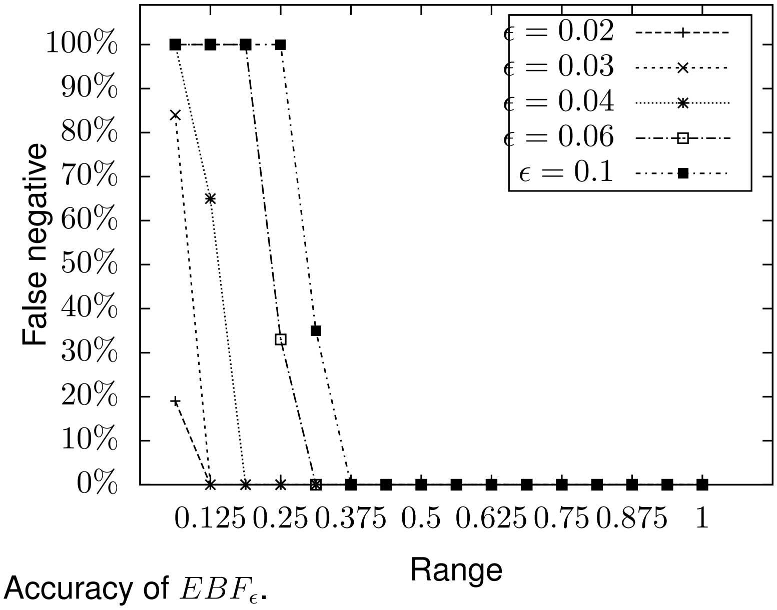

Figure 7-8. Accuracy of EBF_ϵ.

exact algorithm EBF. Other curves present the running time of EBF_ϵ with different ϵ. As expected EBF_ϵ requires more time for smaller ϵ or larger job set size. But its time consumption is sill much smaller than the exact algorithm EBF.

To investigate the accuracy of our proposed approximation scheme, we evaluated the distribution of false negative ratio along different constraint values. In this experiment, we generated 1500 instance of TCEC problem as the test set. They share the same deadline and energy budget, while the temperature constraints are uniformly distributed within $1°C$ above the lowest feasible temperature. For each instance, the exact algorithm EBF is applied first to determine whether the feasible schedule exists. Then we run EBF_ϵ on each instance and check the correctness of the return value. The experimental results are presented in Figure 7-8. Each point represent the false negative ratio of TCEC instances in each $0.0625°C$ interval. For example, the false negative ratio is 30% for instances with in interval $[0.1875\ 0.25]$ when $\epsilon = 0.06$. As we discussed in Section 7.3.4, the false negative ratio curves behaves as step functions, which fall to zero when the temperature constraint is slightly larger ($0.125°C$ for $\epsilon = 0.02$) than the lowest feasible temperature. In other words, EBF_ϵ produces false negative answers in rare cases.

7.6 Summary

In this chapter, we proposed a flexible and automatic framework to solve the temperature- and energy-constrained scheduling problem in multitasking systems with different voltage levels. We modeled the problem using extended timed automata and translated the energy/temperature constraints into CTL specifications. The user can employ a suitable model checker to determine whether there exists a schedule that satisfies the constraints. Due to the capacity limitations of symbolic model checker like UPPAAL, we also proposed a polynomial time approximation scheme that is guaranteed to generate results close to optimal value with reasonable running time. We proved mathematically that our approximation algorithm will give no false positive answer, while the false negative ratio can be negligibly small in practical scenarios. Our framework is also applicable to other scheduling problems with different energy/temperature requirement. Extensive experimental results demonstrated the effectiveness of our approach. In our future work, we plan to develop approximation algorithms to efficiently solve both task sequencing and voltage assignment together.

CHAPTER 8
SCHEDULABILITY VALIDATION FOR MULTICORE ARCHITECTURES

Chapter 7 described our energy- and temperature-aware scheduling framework for a single-core processor. In this chapter, we study the DVS scheduling problem on multicore processors under energy and temperature constraints. Since the task mapping and sequencing are already discussed in many existing works, we focus on how to assign clock rate/voltage levels to tasks that are already mapped and sequenced on different cores, so that the total time consumption is minimized under both temperature and energy constraints. Our goal is to develop a Temperature and Energy Constrained Scheduling (TECS) for multicore systems. Due to the NP-hard nature of TECS problem, it has no polynomial time solution unless P=NP. To avoid the state explosion problem, we propose an approximation scheme with polynomial time/space complexity based on the detailed analysis of the problem. To the best of our knowledge, there are no prior works that consider both energy and temperature constraints in multicore systems and are guaranteed to produce schedules close to the optimal solution with reasonable execution time.

The rest of the chapter is organized as follows. Section 8.1 describes related background information and the MCTCEC problem. Section 8.2 and Section 8.3 discuss the optimal algorithm and our approximation scheme of the MCTCEC problem in detail. Experimental results are presented in Section 8.5. Finally, Section 8.6 concludes the chapter.

8.1 Background and Problem Formulation

8.1.1 Processor Thermal Model

When the execution time of each task is long enough for the processor to reach the steady state temperature, we can use the matrix model [90] to calculate the steady state temperature on each core as

$$\boldsymbol{T}(t) = T_{amb} * \boldsymbol{I}(t) + \boldsymbol{C} * \boldsymbol{P}(t) \tag{8–1}$$

Here, T_{amb} is the ambient temperature, $\boldsymbol{C}$ is a $n \times n$ constant coefficient matrix, and $\boldsymbol{P}(t)$ is the power dissipation by each core under the clock rate assignment at time t. Since we assume that each time slice is large enough for the system to reach steady-state temperature, $\boldsymbol{T}(t)$ is only determined by T_{amb}, $\boldsymbol{C}$ and $\boldsymbol{P}(t)$. For a given system, we derive $\boldsymbol{C}$ using HotSpot [44] as proposed in [90].

8.1.2 Energy Model

We adopt the energy model proposed in [60]. Processor's dynamic power can be represented as $P_{dyn} = \alpha \cdot C \cdot V_{dd}^2 \cdot f$. Here V_{dd} is the supply voltage and f is the operation frequency. C is the total capacitance and α is the actual switching activity which varies for different applications [8]. In other words, the power profile of a task can be different from each other. Static power is given by $P_{sta} = V_{dd} \cdot I_{subth} + |V_{bs}| \cdot I_j$ where V_{bs}, I_{subth} and I_j denote the body bias voltage, subthreshold current and reverse bias junction current, respectively. Hence, the overall power $P = P_{dyn} + P_{sta}$.

8.1.3 System Model

The system we consider can be modeled as:

1. A multicore processor with M cores. Each core supports L discrete clock rate/voltage levels $\{f_1/v_1,\ f_2/v_2,\ ...\ ,\ f_L/v_L\}$, where f_{min} is the lowest clock rate, and f_{max} is the highest.

2. A set of n tasks, which has already been mapped and sequenced on different cores. We use τ_{ij} to denote the j^{th} task on core i. Let c_{ij} be the worst-case workload of τ_{ij}, and k_i be the total number of tasks mapped on core i. We also denote the total workload on core i by $w_i = \sum_{j=1}^{k_i} c_{ij}$.

For ease of discussion, the terms *task* and *job* refer to the same entity in the rest of this chapter.

8.1.4 Multicore DVS Schedule

Since all tasks are already mapped and sequenced, a DVS schedule on a multicore system with task set $\{\tau_{ij} | 1 \leq i \leq M, 1 \leq j \leq k_i\}$ can be represented as a set of tuples $\{(r_{ij}, [t_{ij}, t'_{ij}]) | 1 \leq i \leq M, 1 \leq j \leq k_i\}\}$, where $(r_{ij}, [t_{ij}, t'_{ij}])$ means we execute τ_{ij} using clock rate r_{ij} during time interval $[t_{ij}, t'_{ij}]$. It is easy to see that clock rate switches always happen when some task finishes. When all tasks mapped to a core are finished, a core is turned off.

8.1.5 Problem Formulation

Given a set of n independent tasks $\{\tau_{ij}|1 \leq i \leq M, 1 \leq j \leq k_i\}$, if the safe temperature threshold is C_T and the energy budget is C_E, TECS scheduling problem can be defined as follows.

Definition 9. ***TECS*** *is formally defined as finding a multicore DVS schedule, R_{opt}, which minimize the total execution time, i.e.,*

$$\min \max_{1 \leq i \leq M} t'_{ik_i}$$

subject to

$$c_{ij}/r_{ij} \leq t'_{ij} - t_{ij}$$

$$\sum_{0 \leq i \leq n} P(r_{ij}) * (t'_{ij} - t_{ij}) \leq C_E$$

$$T(t) \leq C_T, \forall t \geq 0$$

$$t'_{ij} \leq t_{ij+1}, \forall j < k_i$$

where $P(r_{ij})$ is power dissipation of a single core when task τ_{ij} is executing using clock rate r_{ij}. It can be proved that TECS problem is NP-hard.

8.2 Optimal Algorithm for TECS

The optimal solution of the TECS problem can be calculated using dynamic programming. The basic idea is to generate all possible execution paths of the system from the initial state. Notice that we consider inter-task DVS, i.e., the voltage switches are only allowed before the beginning of a new task. Any execution path of the system is uniquely determined by the system states at each possible switching point. Furthermore, since the number of cycles between two successive possible switching points can be estimated using remaining task workloads and clock rates on different cores, the state transition between different switching points can be performed as follows. Given a system state, we first identify the next task that is ready to execute. Next, we compute the system states at next switching point by executing this task with all possible clock rates. After that, we mark the estimated state as a valid new state, if it does not violate the temperature or energy constraints.

Formally, given a task sequence on core i, at any time instant t, we define the progress of this task sequence as $p_i = w/w_i$, where $w_i = \sum_j c_{ij}$ is the total workload mapped on core i and w $(w \leq w_i)$ is the completed workload on this core. The system status can be represented as a tuple $\boldsymbol{s} = (<p_1, r_1>, ..., <p_M, r_M>, E, t)$, where p_i and r_i are the current progress and clock rate of core i. E and t are the total energy and time consumption, respectively. The temperature of each core is not explicitly included in the system state tuple, because they can be calculated using the power of each core P_m and ambient temperature T_{amb} using Equation (8–1).

When some cores in the system are about to begin the execution of the next job in their task sequences, we encounter a potential clock rate switching point, or switching point for short. Since multiple cores can change clock rate at the same time, e.g., at $t = 0$, all possible clock rate assignments for M cores can be represented by a set of $M-$dimenional vectors. Formally, we define the set of possible clock **R**ate **A**ssignment $RA(\boldsymbol{s})$ for system state s as the direct product

$$RA(\boldsymbol{s}) = \bigotimes_{i=1}^{M} \begin{cases} \{0\} & \text{if } \boldsymbol{s}.p_i = 1 \\ \{f_1, ..., f_L\} & \text{else if } R_i(\boldsymbol{s}.p_i) = 0 \\ \{\boldsymbol{s}.r_i\} & \text{otherwise} \end{cases} \tag{8–2}$$

where

$$R_i(p_i) = \begin{cases} 0 & \text{if } p_i = 0 \\ \sum_{j=1}^{1} c_{ij}/w_i - p_i & \text{else if } \sum_{j=1}^{1} c_{ij}/w_i \geq p_i \\ \sum_{j=1}^{2} c_{ij}/w_i - p_i & \text{else if } \sum_{j=1}^{2} c_{ij}/w_i \geq p_i \\ \ldots & \\ \sum_{j=1}^{k_i} c_{ij}/w_i - p_i & \text{else if } \sum_{j=1}^{k_i} c_{ij}/w_i \geq p_i \end{cases} \tag{8–3}$$

$R_i(p_i)$ is the remaining progress until the beginning of next task on core i. $RA(\boldsymbol{s})$ returns a set of possible clock rate choices, which allows the core to choose from L voltage levels if it is about to start the next task, i.e., $R_i(p_i) = 0$. If all tasks on the same core are finished, i.e., $p_i = 1$, we shut down the core, by assigning clock rate 0. A core does not consume any more energy at clock rate 0.

In order to calculate the system state at next switching point, we define the state transition function $\boldsymbol{s'} = \boldsymbol{F}(\boldsymbol{s}, \boldsymbol{r})$ as

$$\begin{gathered} \boldsymbol{s'}.p_i = \boldsymbol{s}.p_i + r_i * \delta/w_i \quad \boldsymbol{s'}.r_i' = r_i,\ 1 \leq i \leq M \\ \boldsymbol{s'}.E = \boldsymbol{s}.E + \sum_{i=1}^{M} P(r_i) * \delta \quad \boldsymbol{s'}.t = \boldsymbol{s}.t + \delta \\ \text{where } \delta = \min_{1 \leq i \leq M, p_i < 1} (R_i(\boldsymbol{s}.p_i + \sigma)) w_i / r_i \\ \sigma \text{ is a very small positive number close to 0.} \end{gathered} \tag{8–4}$$

The state transition function $\boldsymbol{F}$ takes the system state at a switching point $\boldsymbol{s}$, and one clock rate assignment vector $\boldsymbol{r}$ as inputs and produces the system state at the next switching point.

Algorithm 9 shows the Dynamic Programming (DP) algorithm for clock Rate Assignment (DPRA) to obtain the optimal solution to the TECS problem. Initially, the set of system states $\mathcal{S}$ only contains a single state $\boldsymbol{s_1} = (<0,0>, ..., <0,0>, 0, 0)$. During the DP process, we first pick $\boldsymbol{s} \in \mathcal{S}$, which contains at least one incomplete task sequence with the least progress among all states in $\mathcal{S}$. Suppose that there are m task

Algorithm 9: Exact solution to TECS

```
$DPRA$
$\mathcal{S} = \{\boldsymbol{s_1}\} = \{(<0,0>, ..., <0,0>, 0, 0)\}$
while not all states in $\mathcal{S}$ are explored do
    Pick an unexplored state $\boldsymbol{s}$ from $\mathcal{S}$ such that $\boldsymbol{s}$ contains at least one incomplete
    task sequence with the least progress among all states in $\mathcal{S}$
    for each $\boldsymbol{r} \in RA(\boldsymbol{s})$ do
        $\boldsymbol{s'} = \boldsymbol{F(s, r)}$
        if $\boldsymbol{r}$ violates temperature constraint $C_T$ or $\boldsymbol{s'}.E > C_E$ then
            continue
        end if
        if $\exists \boldsymbol{s_0} \in \mathcal{S}$ s.t. $\boldsymbol{s_0}$ and $\boldsymbol{s'}$ agree on all values but time then
            if $\boldsymbol{s_0}.t \leq \boldsymbol{s'}.t$ then
                continue
            else
                $\mathcal{S} = \mathcal{S} - \{\boldsymbol{s_0}\}$ /*Remove $\boldsymbol{s_0}$*/
            end if
        end if
        $\mathcal{S} = \mathcal{S} \bigcup \{\boldsymbol{s'}\}$ /*Add $\boldsymbol{s'}$*/
    end for
end while
Find the state $\boldsymbol{s}_{opt}$ in $\mathcal{S}$ with the least time consumption, such that all tasks are
finished. Construct the corresponding schedlue $R_{opt}$ by backtracking from $\boldsymbol{s}_{opt}$ to $\boldsymbol{s_1}$.
```

sequences that are about to start new tasks. We try all possible combinations of clock rate assignments on these m cores, while keeping the clock rate unchanged on the rest $M - m$ cores. This will yield a set of assignments $RA(\boldsymbol{s})$, which contains L^m elements. Next, we calculate a system state $\boldsymbol{s'}$ based on $\boldsymbol{s}$ and clock rate assignment $\boldsymbol{r} \in RA(s)\}$. If $\boldsymbol{s'}$ does not violate any constraints, we add it to $\mathcal{S}$. The above process repeats until all states in $\mathcal{S}$ containing incomplete tasks are explored. Now, we need to find the state which has the least time consumption in $\mathcal{S}$.

EXAMPLE 1: This example illustrates the flow of Algorithm 9 using a processor with $M = 2$ cores. Each of them have L=2 different clock rate levels $f_1 = 100MHz$ and $f_2 = 200MHz$. Their power consumption are $P(f_1) = 1W$ and $P(f_2) = 4W$. There are three tasks $\tau_{1,1}$, $\tau_{1,2}$ and $\tau_{2,1}$ with workloads of 10^6, 10^6, and $2 * 10^6$ cycles, respectively. $\tau_{1,1}$ and $\tau_{1,2}$ are mapped to core 1, while $\tau_{2,1}$ is mapped to core 2. Therefore, we have

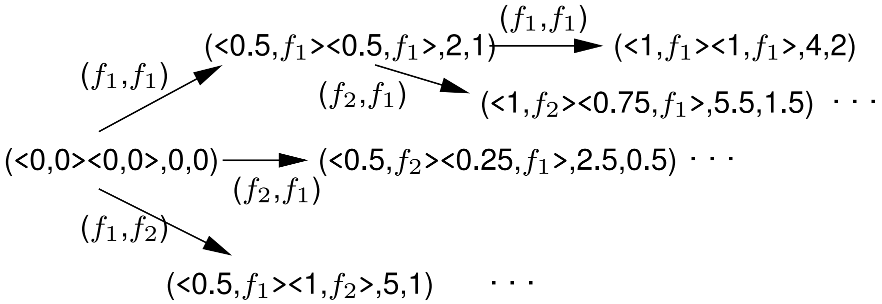
(f_1,f_1)
(<0.5,f_1><0.5,f_1>,2,1)
(f_1,f_1)
(<1,f_1><1,f_1>,4,2)
(f_2,f_1)
(<1,f_2><0.75,f_1>,5.5,1.5) · · ·
(<0,0><0,0>,0,0)
(<0.5,f_2><0.25,f_1>,2.5,0.5) · · ·
(f_2,f_1)
(f_1,f_2)
(<0.5,f_1><1,f_2>,5,1) · · ·

Figure 8-1. State exploration in Algorithm 9

$c_{1,1} = c_{1,2} = 10^6$, $c_{2,1} = 2 * 10^6$, $w_1 = c_{1,1} + c_{1,2} = 2 * 10^6$ and $w_2 = c_{2,1} = 2 * 10^6$. We choose the temperature constraint such that only one core can run at $200MHz$. We also choose $C_E = 10$J. When we apply Algorithm 9 to such a TECS instance, $\mathcal{S}$ contains only one element $\boldsymbol{s_1}$=(<0,0>,<0,0>,0,0) at the beginning. Thus, $\boldsymbol{s_1}$ is picked by line 3. Since we have $R_1(\boldsymbol{s_1}.p_1) = R_1(0) = 0$ and $R_2(\boldsymbol{s_1}.p_2) = 0$, the clock rates for both cores can be changed, i.e., $RA(\boldsymbol{s_1}) = \{f_1, f_2\} \otimes \{f_1, f_2\} = \{(f_1, f_1), (f_1, f_2), (f_2, f_1), (f_2, f_2)\}$ contains $L^M = 4$ elements, which represents four possible clock rate assignments. Next, we compute new states $\boldsymbol{s'}$ based on these assignments except (f_2, f_2), which violate the temperature constraint. If we pick $\boldsymbol{r} = (f_1, f_1)$, the new state $\boldsymbol{s_2} = \boldsymbol{F}(\boldsymbol{s_1}, \boldsymbol{r})$ can be computed as follows. First, we have $R_1(\boldsymbol{s_1}.p_1 + \sigma) = 0.5$, which means core 1 is $0.5w_1$ cycles far from the beginning of the next task $\tau_{1,2}$. Similarly, $R_2(\boldsymbol{s_1}.p_2 + \sigma) = 1$. Therefore, if we use clock rate $\boldsymbol{r} = (f_1, f_1)$, which makes both cores to run at $f_1 = 100MHz$, $\delta = \min(0.5w_1/f_1, w_2/f_1) = 1$sec. In other words, the next switching point will happen after 1sec. At that time, the progress values of core 1 and core 2 will be $\boldsymbol{s_2}.p_1 = 0 + f_1 * 1/w_1 = 0.5$ and $\boldsymbol{s_2}.p_2 = 0 + f_1 * 1/w_2 = 0.5$, respectively. We also compute the energy consumption $\boldsymbol{s_2}.E = 0 + P(f_1) * 1 + P(f_1) * 1 = 2$J and time consumption $\boldsymbol{s_2}.t = 1$sec. Therefore, the new state is $\boldsymbol{s_2}$ =(<0.5,f_1>,<0.5,f_1>,2,1). Since $\boldsymbol{s_2}$ and $\boldsymbol{r} = (f_1, f_1)$ do not violate any constraint, we add $\boldsymbol{s_2}$ into $\mathcal{S}$.

We repeat above procedure for the other two clock rate assignments and mark $\boldsymbol{s_1}$ as explored. In the next round, we pick $\boldsymbol{s_2}$ from $\mathcal{S}$ on line 3 of Algorithm 9. We have

$R_1(\boldsymbol{s_2}.p_1) = R_1(0.5) = 0$, which indicates that we can change the clock rate of core 1, because the previous task just finished. However, $R_2(\boldsymbol{s_2}.p_2) = R_2(0.5) = 0.5$, which means the current task on core 2 has not finished yet. Therefore, $RA(\boldsymbol{s_2}) = \{f_1, f_2\} \otimes \{\boldsymbol{s_2}.r_2\} = \{(f_1, f_1), (f_2, f_1)\}$ only contains two possible clock rate assignments, because the clock rate of core 2 cannot be changed. These assignments are used to produce new states and update $\mathcal{S}$. We repeat above procedure until we find a state in $\mathcal{S}$, within which all tasks are finished with minimum total time consumption. Through backtracing, we can find the path that generates it: (<0,0>,<0,0>,0,0)→ (<0.5,f_1>,<1,f_2>,5,1)→ (<1,f_2>,<1,f_1>,7,1.5). The corresponding scheduling R_{opt} is (<f_1,0,1>,<f_2,1,1.5>,<f_2,0,1>), which means $\tau_{1,1}$, $\tau_{1,2}$ and $\tau_{2,1}$ should be executed using f_1, f_2, and f_2, respectively. □

Clearly, if two system states agree on all values except the time consumption, we only need to record the one with smaller time consumption, because the one with larger time consumption will not be a part of the optimal solution. This fact is exploited by line 9 in Algorithm 9 to accelerate the computation. However, it should be noticed that we must explore the state in certain order, so that the explored state will not be updated in the future. Our algorithm satisfies such a requirement, because the state that we pick contains at least one incomplete task sequence, say i^{th}, which has the least progress among all states in $\mathcal{S}$. There is no way it can be dominated by any new states, because any new states will have a larger progress on the i^{th} task sequence. Therefore, when we pick an unexplored state $\boldsymbol{s}$ on line 3, it is guaranteed that $\boldsymbol{s}.t$ will not be updated in the future.

The time and space complexity of the exact algorithm is $O(L^n)$, because each of the n tasks can be executed at L different voltage levels, the system has L^n different execution path in the worst case. Therefore, $\mathcal{S}$ will contain up to $O(L^n)$ states, because we are performing a breadth-first search in the state space. As a result, the overall time and space complexity becomes $O(L^n)$, which is natural considering the NP-hard nature of TECS problem.

8.3 Approximation Algorithm

Like many previous works, the basic idea of our approximation algorithm is built on discretization of the state space. The space size is reduced by rounding up all values in the state vector, and by merging states that agree on all values after rounding. Unfortunately, in TECS problem, this method cannot be applied directly to progress values. Recall that we define the progress of a task sequence on each core to represent how many instruction or workload has already been completed. Rounding up progress values introduces two problems. First, the switching points, which are defined based on progress values may be skipped, because they usually do not coincide with the discretized progress values. Second, the rounding operation essentially means we skip some instructions without executing them. Therefore, if we apply the obtained scheduling in reality, the actual progress will not match with the ones we calculated in dynamic programming. As a result, the temperature or energy constraints may be violated.

We solve both problems as follows. First, we view a state $s \in \mathcal{S}$ not as a real system state, but a pessimistic approximation of a real system state. Second, we insert a suitable "idle time" at each switching point, so that the difference between the real execution and estimated value in dynamic programming can be bounded. In this way, we can obtain an approximated estimation of the actual execution under any clock rate selections. Before we introduce our approximation scheme, we first introduce the modified version of the state transition function and clock rate assignment function, which are used to build the approximation algorithm. The modified state transition function $\boldsymbol{s'} = \boldsymbol{F}_{\Delta_t}(\boldsymbol{s}, \boldsymbol{r})$ is defined as

$$\begin{aligned}
\boldsymbol{s'}.p_i' &= \boldsymbol{s}.p_i \text{ if } r_i = f_I \ ; \quad \boldsymbol{s}.p_i + r_i * \delta / w_i, \text{otherwise} \\
\boldsymbol{s'}.r_i' &= \boldsymbol{s}.r_i \text{ if } r_i = f_I \ ; \quad r_i, \text{otherwise} \\
\boldsymbol{s'}.E &= \boldsymbol{s}.E + \sum_{i=1}^{M} P(r_i) * (\delta + 2\Delta_t) \\
\boldsymbol{s'}.t &= \boldsymbol{s}.t + \delta + 2\Delta_t \\
\text{where } \delta &= \min_{1 \le i \le M, p_i < 1} R_i(\boldsymbol{s}.p_i + \sigma) * w_i / r_i \\
&\sigma \text{ is a very small positive number close to 0}
\end{aligned} \tag{8–5}$$

An extra increment ($2\Delta_t$) is added, which represents the “idle time”. $RA_\epsilon(\boldsymbol{s})$ is the modified version of $RA(\boldsymbol{s})$, which is defined as

$$RA(\boldsymbol{s}) = \bigotimes_{i=1}^{M} \begin{cases} \{0\} & \text{if } \boldsymbol{s}.p_i = 1 \\ \{f_1, ..., f_L\} & \text{else if } R_i(\boldsymbol{s}.p_i) \le \Delta_P \\ \{\boldsymbol{s}.r_i\} & \text{otherwise} \end{cases} \tag{8–6}$$

In line 9, $\boldsymbol{h}$ is a partial rounding up function $\boldsymbol{s'} = \boldsymbol{h}(\boldsymbol{s})$, which is defined as

$$\begin{aligned}
\boldsymbol{s'}.p_i &= \lceil \boldsymbol{s}.p_i / \Delta_p \rceil * \Delta_p \quad \boldsymbol{s'}.r_i = \boldsymbol{s}.r_i, \ i = 1, ..., M \\
\boldsymbol{s'}.E' &= \lceil \boldsymbol{s}.E / \Delta_E \rceil * \Delta_E \quad \boldsymbol{s'}.t' = \boldsymbol{s}.t
\end{aligned} \tag{8–7}$$

Algorithm 10 shows the details of our approximation algorithm $DPRA_\epsilon$. We first compute the “step size” Δ_E, Δ_P and Δ_t for each constraint based on the value of ϵ. After that, $DPRA_\epsilon$ parallels the exact algorithm $DPRA$ except that the progress and energy values in each new system state $\boldsymbol{s}$ is rounded up to the next available discretized value. This is achieved by applying function $\boldsymbol{h}$, which forces the progress and energy value of the resultant state to be an integer multiple of Δ_P or Δ_E. For example, suppose we have $\Delta_P = 0.1$ and $\Delta_E = 0.2$, a new state $\boldsymbol{F}_{\Delta_t}(\boldsymbol{s}, \boldsymbol{r}))$ =(<0.5,f_2>, <0.25,f_1>,1,2.5,0.5) will be

Algorithm 10: Approximation algorithm of TECS

```
$DPRA_\epsilon$
$\Delta_E = \epsilon * C_E/4n$
$\mu = \max_{1\leq i\leq M} w_i/f_{min}$
$\Delta_P = \min(\Delta_E/\mu P_{max}, \epsilon * f_{min}/(f_{max} * 2n))$. $P_{max}$ is the maximum power dissipation of the entire processor.
$\Delta_t = \Delta_P * \mu$
$\mathcal{S} = \{\boldsymbol{s_1}\} = \{(<0,0>, ..., <0,0>, 0, 0)\}$
while not all states in $\mathcal{S}$ are explored do
  Pick an unexplored state $\boldsymbol{s}$ from $\mathcal{S}$ such that $\boldsymbol{s}$ contains at least one incomplete task sequence with the least progress among all states in $\mathcal{S}$
  for each $\boldsymbol{r} \in RA(\boldsymbol{s})$ do
    $\boldsymbol{s'} = \boldsymbol{h}(\boldsymbol{F}_{\Delta_t}(\boldsymbol{s}, \boldsymbol{r}))$
    if $\boldsymbol{r}$ violates temperature constraint $C_T$ or
    $\boldsymbol{s'}.E > (1+\epsilon)C_E$ then
      continue
    end if
    if $\exists \boldsymbol{s_0} \in \mathcal{S}$ s.t. $\boldsymbol{s'}$ and $\boldsymbol{s_0}$ agree on all values but time then
      if $\boldsymbol{s_0}.t \leq \boldsymbol{s'}.t$ then
        continue
      else
        $\mathcal{S} = \mathcal{S} - \{\boldsymbol{s_0}\}$
      end if
    end if
    $\mathcal{S} = \mathcal{S} \bigcup \{\boldsymbol{s}\}$
  end for
end while
Find the state $\boldsymbol{s_{apx}}$ in $\mathcal{S}$ with the least time consumption $OPT_{apx}$, such that all tasks are finished. Construct the corresponding schedule $R_{apx}$ by backtracking from $\boldsymbol{s_{apx}}$ to $\boldsymbol{s_1}$. If a task is skipped due to rounding, it is scheduled as a part of the previous task on the same core.
```

recorded as $\boldsymbol{h}(\boldsymbol{F}_{\Delta_t}(\boldsymbol{s}_0, \boldsymbol{r}))) =$

(<⌈ 0.5/0.1⌉*0.1,f_2>,<⌈ 0.25/0.1 ⌉*0.1 ,f_1>,⌈ 2.5/0.2 ⌉*0.2, 0.5)

=(<0.5,f_2>,<0.3,f_1>,2.6,0.5)

If the optimal scheduling of $MCTCEC$ exists with time consumption OPT, our approximation algorithm $DPRA_\epsilon$ will find a schedule such that it will not violate the temperature constraint and has at most $(1+\epsilon)C_E$ energy and $(1+\epsilon)OPT$ time consumption respectively. In the rest of this section, we will show that $DPRA_\epsilon$ is an approximation algorithm with the claimed properties.

Lemma 4. *Given an $MCTCEC$ instance I, if $DPRA_\epsilon(I)$ finds a schedule of I R_{apx} with estimated time consumption OPT_{apx}, R_{apx} is a feasible of I, whose actual time consumption does not exceed OPT_{apx}.*

Proof. Since R_{apx} is found by $DPRA_\epsilon(I)$, $\mathcal{S}$ must be a state $\boldsymbol{s}_{apx}$ and a path with K states $\boldsymbol{s}_1- > \boldsymbol{s}_2- > ...- > \boldsymbol{s}_{K-1}- > \boldsymbol{s}_{apx}$. When R_{apx} is applied in reality, we apply the clock rates assignment r_i at time t_i for $1 \leq i \leq K$. When the current job on a core is finished, we keep the core running idle job until next switch point. To prove this lemma, we need to show that 1) all job are indeed finished and 2) all constraints are met.

The first statement can be proved by showing that each job has enough time to run. Suppose a task τ on core j starts from the ith switch point. If the next task on the same core starts from the i'th switch point, the time allocated for this task is $\boldsymbol{s}'_i.t - \boldsymbol{s}_i.t$. Since we perform $i' - i$ rounds of computation to obtain $\boldsymbol{s}'_i$ from $\boldsymbol{s}_i$, there can be at most $(i' - i)$ rounding up during the calculation from $\boldsymbol{s}_i.p_j$ to $\boldsymbol{s}'_i.p_j$. Therefore,

$$\boldsymbol{s}'_i.t \geq \boldsymbol{s}_i.t + (\boldsymbol{s}'_i.p_j - \boldsymbol{s}_i.p_j - (i' - i)\Delta_p)/r_i + 2(i' - i)\Delta t$$
$$\boldsymbol{s}'_i.t - \boldsymbol{s}_i.t \geq (\boldsymbol{s}'_i.p_j - \boldsymbol{s}_i.p_j + \Delta_p)/r_i$$

On the other hand, the total progress of τ can be at most $\boldsymbol{s}'_i.p_j - \boldsymbol{s}_i.p_j + \Delta_p$. Therefore, all tasks will have enough time for execution when R_{apx} is applied.

Now we prove the second statement by considering following relations among different succussive states on path $\boldsymbol{s}_1- > \boldsymbol{s}_2- > ...- > \boldsymbol{s}_{K-1}- > \boldsymbol{s}_{apx}$.

$$\begin{aligned} \boldsymbol{s}_2 &= \boldsymbol{h}(\boldsymbol{F}_{\Delta_t}(\boldsymbol{s}_1, \boldsymbol{r}_1)) \\ \boldsymbol{s}_3 &= \boldsymbol{h}(\boldsymbol{F}_{\Delta_t}(\boldsymbol{s}_2, \boldsymbol{r}_2)) \\ &\cdots \\ \boldsymbol{s}_{apx} &= \boldsymbol{h}(\boldsymbol{F}_{\Delta_t}(\boldsymbol{s}_{K-1}, \boldsymbol{r}_K - 1)) \end{aligned} \tag{8–8}$$

Based on the logic of $DPRA_{\epsilon}(I)$, the following relations hold for $1 \leq i \leq M$, $1 \leq k \leq K$

$$(1+\epsilon)\boldsymbol{C_E} \geq \boldsymbol{s}_k.E$$

Let the real state transition path produced by R_{apx} be $\boldsymbol{s}_1 - > \boldsymbol{s}'_2 - > ... - > \boldsymbol{s}'_{K-1} - > \boldsymbol{s}'_K$. Clearly, we have

$$\begin{aligned} \boldsymbol{s}'_2 &= \boldsymbol{F}_{\Delta_t}(\boldsymbol{s}'_1, \boldsymbol{r}_1) \\ \boldsymbol{s}'_3 &= \boldsymbol{F}_{\Delta_t}(\boldsymbol{s}'_2, \boldsymbol{r}_2) \\ &\ldots \\ \boldsymbol{s}'_K &= \boldsymbol{F}_{\Delta_t}(\boldsymbol{s}'_{K-1}, \boldsymbol{r}_{K-1}) \end{aligned} \tag{8–9}$$

Since components of vector functions $\boldsymbol{h}$ and $\boldsymbol{f}$ are all increasing functions, i.e., $\boldsymbol{s}_1 \geq \boldsymbol{s}_2 \Rightarrow \boldsymbol{h}(\boldsymbol{s}_1) \geq \boldsymbol{h}(\boldsymbol{s}_2)$ and $\boldsymbol{f}(\boldsymbol{s}_1) \geq \boldsymbol{f}(\boldsymbol{s}_2)$, it is easy to see that $\boldsymbol{s}_2 \geq \boldsymbol{s}'_2$,...,$\boldsymbol{s}_{K-1} \geq \boldsymbol{s}'_{K-1}$ and $\boldsymbol{s}_{apx} \geq \boldsymbol{s}'_K$. Therefore,

$$\begin{aligned} (1+\epsilon)C_E &\geq \boldsymbol{s}_k.E \\ OPT_{apx} = \boldsymbol{s}_{apx}.t &\geq \boldsymbol{s}_K.t \end{aligned} \tag{8–10}$$

Notice that temperature and energy values changes monotonically between $\boldsymbol{s}'_k$ and $\boldsymbol{s}'_{k+1}$ during real execution. Equation (8–10) ensures that all constraints are met and therefore concludes the proof. □

In order to show the second property of $DPRA_{\epsilon}(I)$, we first define precedence relation between different tasks.

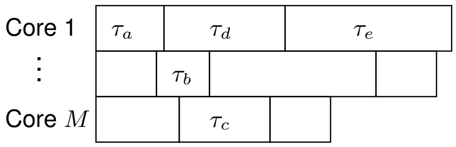

Figure 8-2. Precedence relations among tasks.

Definition 10. *Given feasible schedule of a $MCTCEC$ instance I, we define the precedence relation $\tau_i \prec \tau_j$ on tasks, which holds if and only if the finishing time of τ_i*

is less than the start time of τ_j under the given schedule. For example, in the schedule shown in Figure 8-2, we have $\tau_a \prec \tau_d$, $\tau_b \prec \tau_c$, and $\tau_c \prec \tau_d$ for τ_d, because τ_a, τ_b and τ_c finish before τ_d starts.

A precedence relation PR can also be represented as a graph G_{PR}, within which each task is denoted by a vertex, and each precedence relation is shown as a directed edge. Since all edges are pointing from tasks with larger start time to tasks with smaller finishing time, there is no cycle in the graph, i.e., G_{PR} is a DAG.

Clearly, for any tasks $\tau_1, ..., \tau_M$ that execute at the same time on M different cores under some feasible schedule, there is no precedence relation between any two of them. Besides, it is also easy to see that if there is no precedence relation between any two tasks in $\{\tau_1, ..., \tau_M\}$, they must execute at some time t_0 simultaneously.
Since the schedule is feasible, the clock rate assignment at t_0 does not violate the temperature constraints on any core. In other words, under steady temperature model, the combination of the clock rate assignment to $\tau_1, ..., \tau_M$ will not violate the temperature constraints on any core.

Now we show that our approximation scheme $DPRA_\epsilon$ does find a schedule, when the $MCTCEC$ instance if schedulable.

Lemma 5. *Given an $MCTCEC$ instance I, if I is schedulable with optimal time consumption OPT, $DPRA_\epsilon(I)$ return a schedule R_{apx} with estimated time consumption $OPT_{apx} \leq (1+\epsilon)OPT$.*

Proof. Let $\mathcal{S}$ be the state set constructed by $DPRA_\epsilon(I)$. We are going to show that there exists a path $p = \boldsymbol{s}_1 - > \boldsymbol{s}_2 - > ... - > \boldsymbol{s}_{K-1} - > \boldsymbol{s}_K$ such that

$$\begin{aligned} \boldsymbol{s}_K.t &\leq (1+\epsilon)OPT \\ \boldsymbol{s}_K.E &\leq (1+\epsilon)C_E \\ \boldsymbol{s}_K.p_j &= 1,\ 1 \leq j \leq M \end{aligned} \tag{8–11}$$

First, we construct a path p with desired properties. Since I is schedulable, let the optimal schedule of I be R_{opt}. We use R_{opt} to define the precedence relation PR on all tasks and construct the corresponding graph G_{PR}. We construct path $p = \boldsymbol{s}_1 - > \boldsymbol{s}_2 - > ... - > \boldsymbol{s}_{K-1} - > \boldsymbol{s}_K$ as follows,

1. For task τ, its corresponding node in G_{PR} has no precedence node, use its clock rate in R_{opt}, otherwise, use clock rate f_0.
2. Compute the next state $\boldsymbol{s}_{i+1} = \boldsymbol{h}(\boldsymbol{F}_{\Delta_t}(\boldsymbol{s}_i, \boldsymbol{r}))$, where $\boldsymbol{r}$ is the clock rate assignment of corresponding tasks in R_{opt}. If a task is finished based on the progress in $\boldsymbol{s}_{i+1}$, remove its corresponding node from G_{PR}.
3. Repeat above steps until all tasks are finished.

Since $\boldsymbol{s}_0$ is the initial state, and the clock rate assignment used to produce $\boldsymbol{s}_1$ is identical to the ones in R_{opt} at beginning, $\boldsymbol{s}_1$ will not violate the temperature constraints. Thus, $\boldsymbol{s}_1$ is in $\mathcal{S}$. Suppose we know $\boldsymbol{s}_i \in \mathcal{S}$ as induction hypothesis. Let the clock rate assignment used to produce $\boldsymbol{s}_{i+1}$ be $\boldsymbol{r}$. By our construction rules, $\boldsymbol{r}$ is applied on a set of tasks, which do not have any precedence relation between any two of them in R_{opt}. In other words, $\boldsymbol{r}$ will not violate the temperature constraint on any core. Such reasoning holds for any $\boldsymbol{s}_i$, $1 \le i \le K$. Therefore, all states in p do not violate the temperature constraint.

Next, we show that p also meets the time requirement, i.e.,

$$\boldsymbol{s}_K.t \le (1+\epsilon)OPT$$

Let task τ_{i_k} be the last task completed in p. Suppose τ_{i_k} start execution at some state $\boldsymbol{s}_{i_k}$ and finished at state $\boldsymbol{s}_K$. Since τ_{i_k} is assigned a non-idle clock rate in $\boldsymbol{s}_{i_k}$, there must exist a task $\tau_{i_{k-1}} \prec \tau_{i_k}$, which just finished in $\boldsymbol{s}_{i_k}$. Otherwise, if all τ_k's precedence tasks are finished in $\boldsymbol{s}_{i_{k-1}}$, τ_k should start in $\boldsymbol{s}_{i_{k-1}}$ instead of $\boldsymbol{s}_{i_k}$ based on our construction.

For the same reason, we can find $\tau_{i_{k-2}}$, such that $\tau_{i_{k-2}} \prec \tau_{i_{k-1}}$ and $\tau_{i_{k-2}}$ finishes in the same state from which $\tau_{i_{k-1}}$ starts execution. Eventually, we can determine a chain of tasks $\tau_{i_1} \prec \tau_{i_2} \prec ... \prec \tau_{i_k}$, where τ_{i_1} is the first task on some core. Let time consumption of each task in the chain under R_{opt} be $t_{i_1}, ..., t_{i_k}$ respectively. It is easy to see that

$$\boldsymbol{s}_K.t \leq \boldsymbol{s}_{i_k}.t + t_{i_k} + 2(K - i_k)\Delta_t$$

because $DPRA_\epsilon(I)$ adds $2\Delta_t$ to time consumption for each intermediate state, and we require at most t_{i_k} time to finish its progress, because the time consumption of τ_{i_k} (summation of δ) is estimated using the same clock rate as in R_{opt}. Similarly, we have

$$\begin{aligned}
\boldsymbol{s}_{i_k}.t &\leq \boldsymbol{s}_{i_{k-1}}.t + t_{i_{k-1}} + 2(i_k - i_{k-1})\Delta_t \\
&\ldots \\
\boldsymbol{s}_{i_2}.t &\leq \boldsymbol{s}_{i_1}.t + t_{i_1} + 2(i_2 - i_1)\Delta_t = t_{i_1} + 2i_2\Delta_t
\end{aligned}$$

By taking the sum of both sides, we have

$$\boldsymbol{s}_K.t \leq t_{i_1} + ... + t_{i_k} + 2K\Delta_t$$

On the other hand, since $\tau_{i_1} \prec \tau_{i_2} \prec ... \prec \tau_{i_k}$, their execution have no overlap under R_{opt}. Thus

$$t_{i_1} + ... + t_{i_k} \leq OPT$$

Therefore,

$$\begin{aligned}
\boldsymbol{s}_K.t &\leq OPT + 2K\Delta_t \leq OPT + 2n\Delta_t \\
&\leq OPT + 2n\Delta_P\mu \leq OPT + 2n\frac{\epsilon f_{min}}{f_{max} * 2n}\max_{1\leq i\leq M} w_i/f_{min} \\
&\leq OPT + \epsilon \max_{1\leq i\leq M} w_i/f_{max} \leq (1+\epsilon)OPT
\end{aligned}$$

Finally, we prove that p meets the energy requirement, i.e.,

$$\boldsymbol{s}_K.E \leq (1+\epsilon)C_E$$

Since $\boldsymbol{s}_K = \boldsymbol{h}(\boldsymbol{F}_{\Delta_t}(\boldsymbol{s}_{K-1}, \boldsymbol{r}))$, we have

$$\begin{aligned} \boldsymbol{s}_K.E &= \lceil (\boldsymbol{s}_{K-1}.E + \sum_{i=1}^{M} P(\boldsymbol{s}_K.r_i) * (\delta_{K-1} + 2\Delta_t))/\Delta_E \rceil \Delta_E \\ &\leq \boldsymbol{s}_{K-1}.E + \sum_{i=1}^{M} P(\boldsymbol{s}_K.r_i) * (\delta_{K-1} + 2\Delta_t) + \Delta_E \end{aligned} \tag{8–12}$$

We can derive similar relations for $\boldsymbol{s}_{K-1}.E$, ... ,$\boldsymbol{s}_1.E$ and plug all of them into (8–12). Notice that $\boldsymbol{s}_1.E = 0$, we have

$$\begin{aligned} \boldsymbol{s}_K.E &\leq \sum_{i=1}^{M}\sum_{j=2}^{K} P(\boldsymbol{s}_j.r_i) * (\delta_{j-1} + 2\Delta_t) + (K-1)\Delta_E \\ &\leq \sum_{i=1}^{M}\sum_{j=2}^{K} P(\boldsymbol{s}_j.r_i)\delta_{j-1} + (K-1)(2P_{max}\Delta_t + \Delta_E) \\ &\leq \sum_{i=1}^{M}\sum_{j=2}^{K} P(\boldsymbol{s}_{j-1}.r_i)\delta_{j-1}(1 - Idle(i,j)) \\ &+ \sum_{i=1}^{M}\sum_{j=2}^{K} P(\boldsymbol{s}_{j-1}.r_i)\delta_{j-1} Idle(i,j) + (K-1)(2P_{max}\Delta_t + \Delta_E) \end{aligned}$$

where

$$Idle(i,j) = \begin{cases} 1 & \text{if core i receives } f_0 \text{ in } \boldsymbol{s}_j \\ 0 & \text{otherwise} \end{cases}$$

Intuitively,

$$E_A = \sum_{i=1}^{M}\sum_{j=2}^{K} P(\boldsymbol{s}_j.r_i)\delta_{j-1}(1 - Idle(i,j))$$

is the total energy consumption when cores receive clock rates other than f_0 and make progress, while

$$E_I = \sum_{i=1}^{M}\sum_{j=2}^{K} P(\boldsymbol{s}_j.r_i)\delta_{j-1} Idle(i,j)$$

is the total energy consumption when cores receive f_0.

Due to the rounding up of progresses, E_A should be no more than the real energy consumption in R_{opt}, because we apply the same clock rate to execute at most the same amount of progress as R_{opt}. Since R_{opt} is a feasible schedule, its energy consumption is bounded by C_E, i.e.,$E_A \leq C_E$

For the second term E_I, we claim that for any core i, the total time that tasks receive clock rate f_0

$$t_i^{idle} = \sum_{j=2}^{K} \delta_{j-1} Idle(i,j)$$

is not more than $\mu(K-1)\Delta_P$. To see this, let the time consumption of all tasks on core i be OPT_i. Using the same technique we used in the time consumption analysis, we have $\sum_{j=2}^{K} \delta_j \leq OPT_i$. On the other hand, the total progress skipped due to rounding up is at most $(K-1)\Delta_P$. The time consumption for execution using clock rate other than f_0, i.e., $\sum_{j=2}^{K} \delta_{j-1}(1 - Idle(i,j))$, should be at least $OPT_i - \mu k \Delta_P$. Therefore,

$$t_i^{idle} \leq OPT_i - \sum_{j=2}^{K} \delta_{j-1}(1 - Idle(i,j)) \leq \mu(K-1)\Delta_P$$

Thus,

$$\begin{aligned} E_I &= \sum_{i=1}^{M}\sum_{j=2}^{K} P(\boldsymbol{s}_j.r_i)\delta_{j-1} Idle(i,j) \\ &\leq \frac{P_{max}}{M}\sum_{i=1}^{M} t_i^{idle} \leq P_{max}\mu(K-1)\Delta_P \end{aligned}$$

Notice that $K \leq n+1$, we have

$$\begin{aligned} \boldsymbol{s}_K.E &\leq E_A + E_I + (K-1)(2P_{max}\Delta_t + \Delta_E) \\ &\leq C_E + P_{max}\mu n \Delta_P + 2nP_{max}\Delta_t + n\Delta_E \\ &\leq C_E + \epsilon C_E/4 + \epsilon C_E/2 + \epsilon C_E/4 \leq (1+\epsilon)C_E \end{aligned}$$

Now we have finally proved that path p satisfies all constraints. Let $\mathcal{S}$ be the state set produced during the execution of $DPRA_\epsilon(I)$. Since p is constructed using the same transition function $\boldsymbol{h}$ and $\boldsymbol{F}_{\Delta_t}$, it is also a valid path in $\mathcal{S}$, unless some states in it have same progress and energy value as other states in $\mathcal{S}$ but more time consumption, and therefore be replaced. In either case, $DPRA_\epsilon(I)$ will yield a schedule with time consumption at most $(1+\epsilon)OPT$. □

Lemma 6. *$DPRA_\epsilon$ is a polynomial time algorithm in n, i.e., the number of tasks.*

Proof. We first show that the number of states in $\mathcal{S}$ is $O((n/\epsilon)^{M+1})$. It is easy to see that the energy value is discretized into $4n/\epsilon$ different values. For progress values, there are $1/\Delta_P$ different values allowed for each core. If $\Delta_E/\mu P_{max} < \epsilon * f_{min}/(f_{max} * 2n)$,

$$\begin{aligned}\frac{1}{\Delta_P} &= \mu\frac{P_{max}}{\Delta_E} \\ &= \max_{1\le i\le M}\sum_{j=1}^{k} c_{ij}\frac{P_{max}4n}{\epsilon C_E f_{min}} \le \frac{4P_{max}n}{P_{min}\epsilon}\end{aligned} \tag{8–13}$$

Otherwise, if $\Delta_E/\mu P_{max} \ge \epsilon * f_{min}/(f_{max} * 2n)$,

$$\frac{1}{\Delta_P} = \frac{f_{max}}{f_{min}\epsilon} \tag{8–14}$$

In either case, $1/\Delta_P$ is no more than n/ϵ times a constant, because both P_{max}/P_{min} and f_{max}/f_{min} are normally less than 10. Therefore, there are at most $O((n/\epsilon)^{M+1})$ states in $\mathcal{S}$. At the same time, the number of different voltage assignments we can choose, i.e., the size of $RA_\epsilon(\boldsymbol{s})$, is also no more than $(L+1)^M$, which is a constant. Therefore, the overall complexity of $DPRA_\epsilon$ is $O((n/\epsilon)^{M+1})$. □

As a direct result of Lemma 4, Lemma 5 and Lemma 6, we have

Theorem 8.1. *Given an $MCTCEC$ instance I, if I is schedulable with optimal time consumption OPT, $DPRA_\epsilon(I)$ will return a schedule in polynomial time, which does not violate the temperature constraint, while its energy and time consumption are at most $(1+\epsilon)C_E$ and $(1+\epsilon)OPT$, respectively.*

8.4 Problem Variants

8.4.1 Task Set with Dependence

So far, we only discussed the application of our approach using independent task set. When some task depends on other tasks and cannot start before the completion of other tasks, we can naturally add such constraints to the constraint checking statement (line 6 of Algorithm 9 and line 10 of Algorithm 10), because we allow the dummy clock rate f_I. If the dependence constraint is not met based on the progresses on different cores, we just drop the new state.

8.4.2 Hard Energy Constraint

When the energy constraint is tight, $\boldsymbol{s'}.E > (1+\epsilon)C_E$ on line 10 of Algorithm 10 should be replaced by $\boldsymbol{s'}.E > C_E$. In this case, our approximation algorithm will find a schedule such that it will not violate the energy and temperature constraints and has at most $(1+\epsilon)OPT_\epsilon$ time consumption. However, the approximation algorithm is only guaranteed to find a schedule, when the original TECS problem is schedulable with energy consumption of $(1-\epsilon)C_E$ with optimal time consumption OPT_ϵ.

8.5 Experiments

The experiments were conducted on 2 core, 4 core, and 6 core processors. Each core is abstracted as a $8mm \times 8mm$ square. The cores are arranged in 2×1, 2×2 and 3×2 meshes, respectively. We model each core as a DVS-capable processing unit with three voltage/frequency levels (1.5V-206MHz, 1.1V-133MHz, and 0.8V-103MHz) like StrongARM [61]. We choose some tasks from the Mibench and obtain the workload (worst case cycle numbers) from M5 simulator. We also use synthetic task sets which are randomly generated with each of them having execution time in the range of 500 - 5000 milliseconds. We adopt the approach in [90] to compute the steady state temperature. The ambient temperature and initial temperature of the processor are set to $32°C$ and $40°C$, respectively. The exact and approximation algorithms are implemented in C++. All experiments were performed on 3GHz workstation with 20GB RAM.

We choose 6 jobs from MiBench [41], including algorithms from communication (*FFT, CRC32*), security (*sha*), sound compression (*untoast*), and automotive (*basicmath, qsort*). The workload of these jobs were in range of $5 * 10^7 - 3 * 10^8$ cycles. We use the exact algorithm DPRA to schedule these tasks on 2 core processor. *CRC32* ($\tau_{1,1}$), *qsort* ($\tau_{1,2}$), and *untoast* ($\tau_{1,3}$) are mapped to core 1. *sha* ($\tau_{2,1}$), *FFT* ($\tau_{2,2}$), and *basicmath* ($\tau_{2,3}$) are mapped to core 2. We depict the temperature curves of each core in Figure 8-3, when different temperature and energy constraints are applied.

In Figure 8-3a, the temperature constraint is not violated when both cores run at 1.5V. DPRA schedules jobs on different cores to execute using the maximum voltage level at the same time, i.e., task $\tau_{1,1}$ and $\tau_{2,2}$, to minimize the time consumption. When the energy budget reduces, tasks with large workload will be executed using lower voltage level to save energy as shown in Figure 8-3b. As we can see, $\tau_{2,2}$ is executed using 1.1V instead of 1.5V, when the energy budget reduces to 22000mJ. Similarly, when the temperature constraint becomes tighter, less number of tasks are executed using the maximum voltage level to decrease the peak temperature. As shown in Figure 8-3c, two cores no longer run using 1.5V at the same time. Although the energy budget is still sufficient, the time consumption increases slightly compared to Figure 8-3a.

We evaluated the performance of our approximation scheme using task sets with different sizes. Figure 8-4 and Figure 8-5 show the actual ratio between the approximation results and the optimal solution for time and energy consumption, respectively. It can be seen that the actual ratio is usually smaller than the expected ratio $1 + \epsilon$. For example, for $\epsilon = 0.02$, it is expected to produce results within 2% of the optimal values. The actual gap between the optimal solution and the approximation scheduling is significantly less than 2%.

We also evaluated the running time of our approximation scheme with different ϵ and number of tasks. The results on 2 core and 4 core systems are shown in Figure 8-6. Curve $DPRA$ represents the execution time of the exact algorithm $DPRA$. As expected, $DPRA_\epsilon$ requires more time for smaller ϵ or larger job set size. Its time consumption is sill

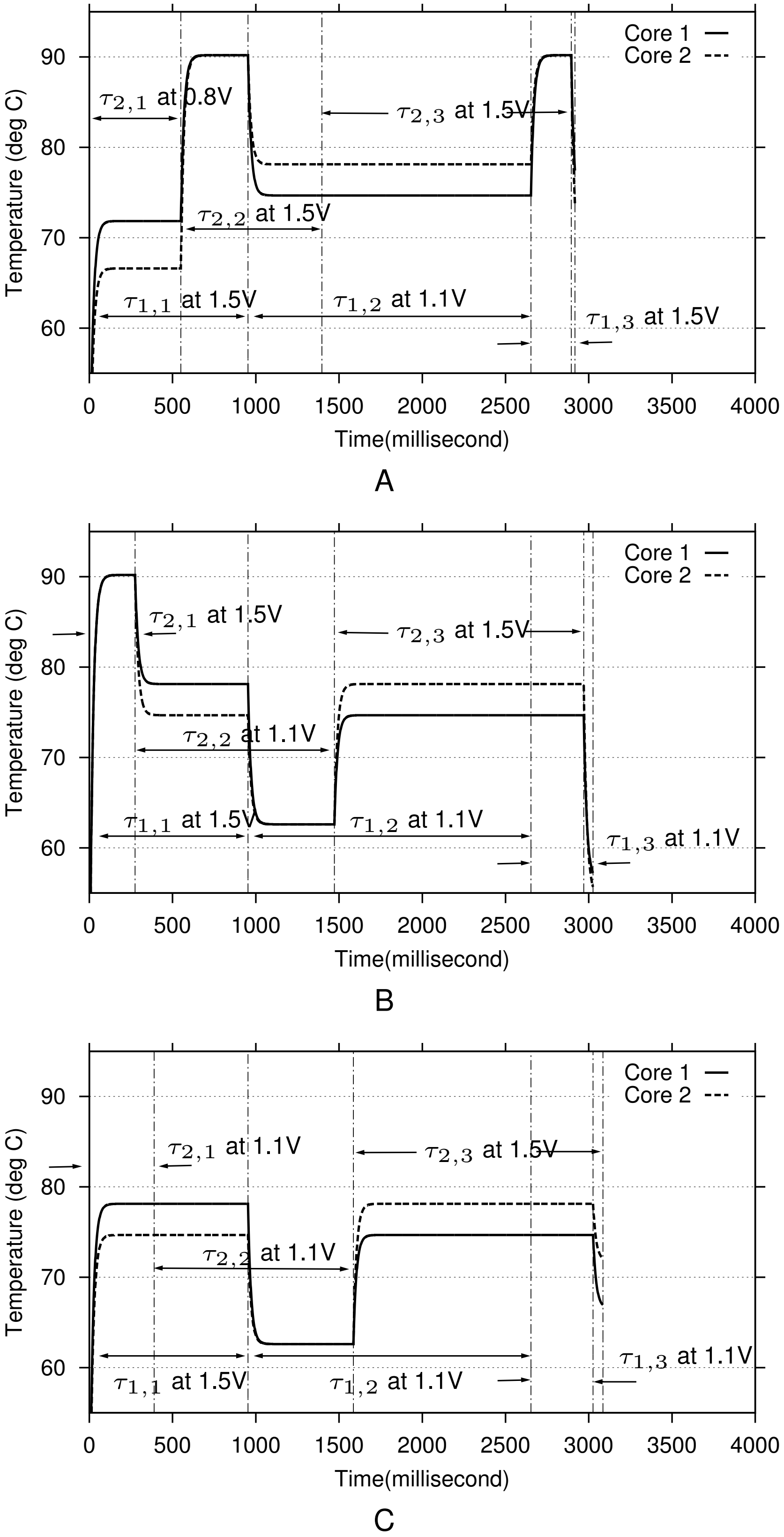

Figure 8-3. Temperature and energy constrained scheduling. A) $C_T = 95°C$ and $C_E = 23000mJ$. B) $C_T = 95°C$ and $C_E = 22000mJ$. C) $C_T = 85°C$ and $C_E = 23000mJ$.

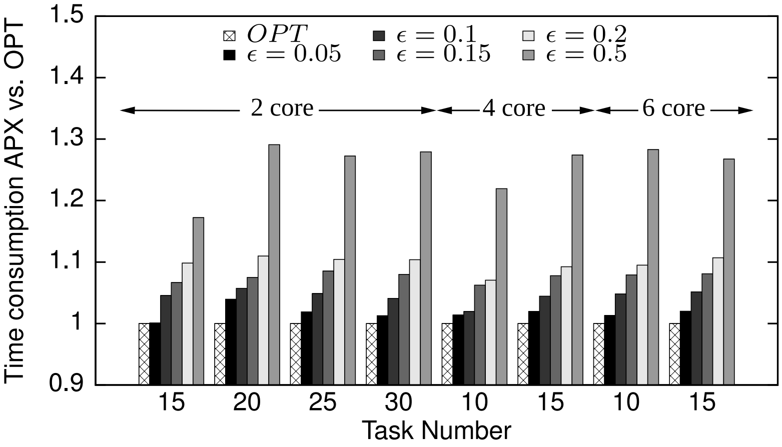

Figure 8-4. Actual time consumption of $DPRA_\epsilon$.

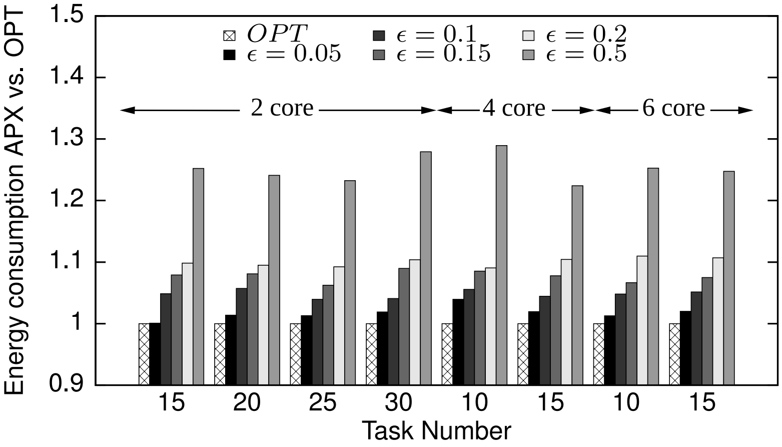

Figure 8-5. Actual energy consumption of $DPRA_\epsilon$.

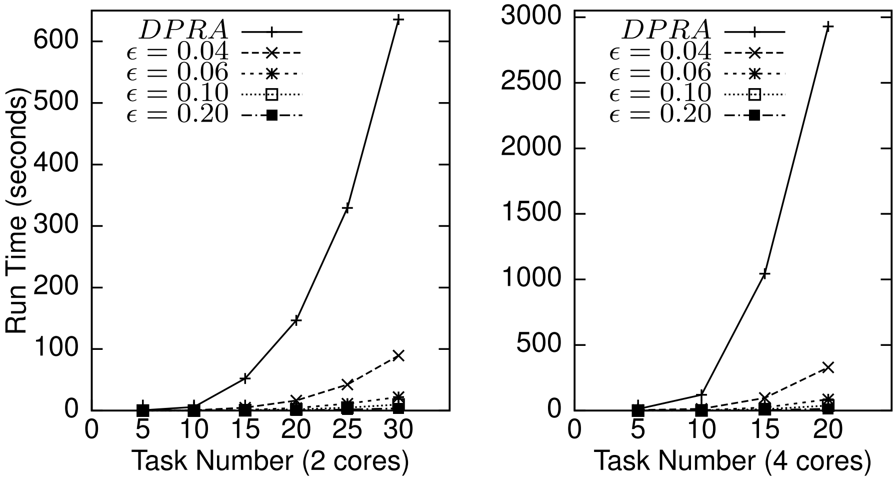

Figure 8-6. Running time with different job set size and ϵ.

significantly smaller than the exact algorithm $DPRA$, which grows exponentially with the number of tasks.

8.6 Summary

In this chapter, we studied task scheduling problem on a multicore processor with DVS capability under both temperature and energy constraints. We present a polynomial time approximation scheme. When the original problem is schedulable, our approximation algorithm is guaranteed to generate a solution, which will not violate the temperature constraint, and consume no more than a designer specified bound time and energy compared with the optimal solution. We evaluated our approach using both real and synthetic benchmarks mapped on real multicore processors. The experimental results demonstrate that our technique is able to produce schedules close to optimal solution with reasonable execution time.

CHAPTER 9
CONCLUSIONS AND FUTURE WORK

Multicore architectures are widely used in today's desktop, server and embedded systems. Increasing complexity of modern multicore architectures introduces unique validation challenges. This dissertation described a set of novel techniques and methodologies for system level validation of multicore architectures. This chapter concludes this dissertation and outlines possible future research directions.

9.1 Conclusions

To design reliable multicore systems, it is crucial to satisfy both functional and non-functional requirements. The functional requirements ensure that the design performs all the logical operations as specified. The non-functional requirements guarantee that the system does not violate various design constraints such as area, power, energy, and temperature. Increasing complexity of multicore architectures introduces significant challenges during validation of both functional behavior and non-functional requirements. This dissertation developed novel techniques to address these validation challenges. This dissertation's contributions are summarized as follows.

In Chapter 3 and Chapter 4, we presented directed test generation techniques for the functional validation of multicore architectures. Although simulation using directed tests requires significantly less number of tests to achieve the same coverage goal compared to random tests, it is very time consuming to generate the directed tests automatically due to the limitation of current model checking tools. While existing works have exploited the temporal symmetry in bounded model checking (BMC) across different time steps, we presented a novel approach for directed test generation of multicore architectures that exploits temporal, structural, as well as spatial symmetry in SAT-based BMC. The CNF description of the design is synthesized using CNF for cores, bus and memory subsystem to preserve the mapping information between different cores. As a result, the symmetric high level structure, i.e., structural symmetry, is well preserved and the knowledge learned from a single core was effectively shared by other cores during the SAT solving process. The experimental results using homogeneous as well as heterogeneous multicore architectures demonstrated that our approach is remarkably faster (3-10 times) compared to existing methods.

Chapter 5 described an efficient test generation approach for a wide variety of cache coherence protocols. We have performed detailed analysis of the space structure of several popular protocols, and developed novel techniques to generate efficient test sequences to achieve 100% state and transition coverage for each cache coherence protocol. Our approach outperformed existing approaches based on constrained random tests providing higher transition coverage with linear memory requirement. We also conducted experiments using a wide variety of cache coherence protocols to demonstrate the effectiveness of our approach on systems with different number of cores, making it suitable for future multicore architectures.

In Chapter 6, we addressed a major obstacle in applying directed test generation technique on real world designs. Since model checkers do not directly accept designs written in hardware description language or do not support all the features, real designs must be translated or abstracted before test generation. We presented a novel approach for directed test generation using interleaved concrete and symbolic execution that accepts Verilog designs. The design is first simulated to generate an execution trace. The constraint solver is then applied to find a suitable test pattern which can force the real design to exercise the desired behavior. Our approach alleviates the design translation problem by directly recording the logical operations performed during the concrete simulation. The constraint solving complexity is also reduced, because we only apply the solver to one execution trace at a time.

Chapter 7 presented a flexible and automatic framework to address the temperature- and energy-constrained schedulability validation problem in multitasking systems with different voltage levels. The problem is modeled by extended timed automata, while the energy/temperature constraints are translated into CTL specifications. A model checker was employed to determine whether the given task set is schedulable. In addition, we also proposed a polynomial time approximation scheme to circumvent the capacity limitations of symbolic model checkers. Our approximation algorithm is guaranteed to generate results close to optimal value with reasonable running time. We proved mathematically that our approximation algorithm will give no false positive answer, while the false negative ratio can be negligibly small in practical scenarios. Extensive experimental results demonstrated the effectiveness of our approach.

Chapter 8 studied task schedulability validation of DVS-enabled multicore processors. We have designed a polynomial time approximation scheme, such that when the original problem is schedulable, our approximation algorithm is guaranteed to generate a solution, which will not violate the temperature constraint, and consume no more than a specified amount of time and energy compared with the optimal solution. Both real and synthetic benchmarks are used to evaluate our approach. The experimental results suggest that our technique is able to produce results close to optimal solution with reasonable execution time.

In conclusion, this dissertation presented a comprehensive study of the system-level validation of multicore architectures. We developed a set of efficient validation techniques and evaluated them on a variety of multicore systems. Our research will enable designers and validation engineers to significantly improve the quality of future multicore designs.

9.2 Future Research Directions

The validation of multicore architectures will continue to be one of the most important challenges in the development of future desktop, server, and embedded systems. The research described in this dissertation can be extended in the following directions:

The capacity of underlying SAT solvers is an important bottleneck for directed test generation of multicore architectures.. Although our proposed techniques have shown significant reduction of the overall solving time, the time consumption needs to be further reduced to make it applicable on complex industrial designs. Further studies are required to analyze the SAT solving process and make more efficient learning techniques to increase the capacity of existing approaches.

We have shown that the state space of many cache coherence protocols in modern multicore architectures have quite regular structure. We believe that our proposed techniques can be further extended to effectively analyze the protocol implementation with large number of cores. Although the full transition coverage may become infeasible for too many cores, the knowledge about the space structure can be used to effectively distribute the test vectors within the state space, so that complex bugs can be detected.

Our work in the direction of scalable directed test generation has demonstrated the effectiveness of the integration of concrete simulation and static analysis. Further studies can investigate how to support more HDL features and employ more constraint solving optimizations. The proposed technique can also be incorporated with random test generation to reduce the overall validation time.

Our validation techniques for task schedulability analysis can be further extended to support more complex thermal models. Different validation techniques can be developed for task sets with relatively small execution time. Although polynomial-time approximation algorithms can provide results with bounded errors, their computational complexity can be quite high when the system contains many cores. It is therefore desirable to have fast and efficient heuristic algorithms for schedulability validation in future multicore and manycore systems.

REFERENCES

[1] D. Abts, S. Scott, and D. Lilja. So many states, so little time: verifying memory coherence in the Cray X1. In *Proceedings of International Parallel and Distributed Processing Symposium,*, 2003.

[2] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv. Genesys-pro: innovations in test program generation for functional processor verification. *IEEE Design Test of Computers*, 21(2):84 – 93, 2004.

[3] F. A. Aloul, I. L. Markov, and K. Sakallah. Shatter: efficient symmetry-breaking for boolean satisfiability. In *Proceedings of Design Automation Conference*, pages 836–839, 2003.

[4] F. A. Aloul, I. L. Markov, and K. A. Sakallah. *Shatter*. University of Michigan, 2003. Available from: http://www.aloul.net/Tools/shatter/.

[5] F. A. Aloul, A. Ramani, I. L. Markov, and K. Sakallah. Solving difficult SAT instances in the presence of symmetry. In *Proceedings of Design Automation Conference*, pages 731–736, 2002.

[6] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[7] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Transactions on Software Engineering*, 36(4):474 –494, 2010.

[8] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In *Proceedings of Euromicro Conference on Real-Time Systems*, pages 225–232, 2001.

[9] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Transactions on Computers*, 53(5):584–600, 2004.

[10] M. Berkelaar, K. Eikland, and P. Notebaert. *lpsolve*. Eindhoven Univeristy of Technology, 2010. Available from: http://lpsolve.sourceforge.net/.

[11] D. Bernstein, D. Cohen, and D. Maydan. Dynamic memory disambiguation for array references. In *Proceedings of International Symposium on Microarchitecture*, pages 105 – 111, 1994.

[12] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[13] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, 1999.

[14] A. Biere and C. Sinz. Decomposing SAT problems into connected components. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:191–198, 2006.

[15] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52 –60, 2006.

[16] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter variations and impact on circuits and microarchitecture. In *Proceedings of Design Automation Conference*, pages 338–342, 2003.

[17] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[18] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltse. *NuSMV*. ITC-Irst, 2010. Available from: http://nusmv.irst.itc.it/.

[19] T. Chantem, R. P. Dick, and X. S. Hu. Temperature-aware scheduling and assignment for hard real-time applications on mpsocs. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 288–293, 2008.

[20] J.-J. Chen, C.-M. Hung, and T.-W. Kuo. On the minimization fo the instantaneous temperature for periodic real-time tasks. In *Proceedings of IEEE Real Time and Embedded Technology and Applications Symposium*, pages 236–248, 2007.

[21] J.-J. Chen and C.-F. Kuo. Energy-efficient scheduling for real-time systems on dynamic voltage scaling (dvs) platforms. In *Proceedings of IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 28–38, 2007.

[22] M. Chen and P. Mishra. Functional test generation using efficient property clustering and learning techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(3):396 –404, 2010.

[23] M. Chen and P. Mishra. Decision ordering based property decomposition for functional test generation. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 167–172, 2011.

[24] M. Chen and P. Mishra. Property learning techniques for efficient generation of directed tests. *IEEE Transactions on Computers*, 60(6):852–864, 2011.

[25] M. Chen, X. Qin, and P. Mishra. Efficient decision ordering techniques for sat-based test generation. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 490–495, 2010.

[26] S. Chen and K. Nahrstedt. On finding multi-constrained paths. In *Proceedings of IEEE International Conference on Communications*, volume 2, pages 874 –879, 1998.

[27] X. Chen, Y. Yang, M. Delisi, G. Gopalakrishnan, and C.-T. Chou. Hierarchical cache coherence protocol verification one level at a time through assume guarantee. In *Proceedings of IEEE International High Level Design Validation and Test Workshop*, pages 107 –114, 2007.

[28] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[29] A. Coskun, T. Rosing, K. Whisnant, and K. Gross. Static and dynamic temperature-aware scheduling for multiprocessor socs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(9):1127–1140, 2008.

[30] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry detection for cnf. In *Proceedings of Design Automation Conference*, pages 530–534, 2004.

[31] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communication of ACM*, 5(7):394–397, 1962.

[32] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of ACM*, 7(3):201–215, 1960.

[33] D. Dill, A. Drexler, A. Hu, and C. Yang. Protocol verification as a hardware design aid. In *Proceedings of International Conference on Computer Design*, pages 522–525, 1992.

[34] B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proceedings of International Conference on Computer Aided Verfication*, pages 81–94, 2006.

[35] J. Edmonds and E. L. Johnson. Matching, Euler Tours, and the Chinese Postman. *Mathematical Programming*, 5:88–124, 1973.

[36] E. Emerson and V. Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In *Proceedings of IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 2860, pages 247–262, 2003.

[37] E. Fersman, P. Pettersson, and W. Yi. Timed automata with asynchronous processes: Schedulability and decidability. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 67–82, 2002.

[38] Z. Fu, Y. Mahajan, and S. Malik. *zChaff*. Princeton University, 2001. Available from: http://www.princeton.edu/~chaff/zchaff.html.

[39] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International*

Symposium on Foundations of Software Engineering, volume 24, pages 146–162, 1999.

[40] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[41] M. Guthaus, J. Ringenberg, D.Ernest, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of IEEE International Workshop on Workload Characterization*, pages 3–14, 2001.

[42] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 2003.

[43] J. N. Hooker. Solving the incremental satisfiability problem. *Journal of Logic Programming*, 15(1-2):177–186, 1993.

[44] W. Huang, K. Sankaranarayanan, K. Skadron, R. J. Ribando, and M. R. Stan. Accurate, pre-rtl temperature-aware design using a parameterized, geometric thermal model. *IEEE Transactions on Computers*, 57:1277–1288, 2008.

[45] M. A. J. Bhadra, E. Trofimova. Validating power architecture technology-based mpsocs through executable specifications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(4):388–396, 2008.

[46] R. Jayaseelan and T. Mitra. Temperature aware task sequencing and voltage scaling. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 618–623, 2008.

[47] R. Jejurikar and R. Gupta. Energy aware non-preemptive scheduling for hard real-time systems. In *Proceedings of Euromicro Conference on Real-Time Systems*, pages 21–30, 2005.

[48] R. Jejurikar, C. Pereira, and R. K. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proceedings of Design Automation Conference*, pages 275–280, 2004.

[49] Z. Khasidashvili, A. Nadel, A. Palti, and Z. Hanna. Simultaneous SAT-based model checking of safety properties. In *Proceedings of Haifa Verification Conference*, pages 56–75, 2005.

[50] H.-M. Koo and P. Mishra. Functional test generation using property decompositions for validation of pipelined processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1240–1245, 2006.

[51] H.-M. Koo and P. Mishra. Test generation using SAT-based bounded model checking for validation of pipelined processor. In *Proceedings of ACM Great Lakes Symposium on VLSI*, pages 362–365, 2006.

[52] H.-M. Koo and P. Mishra. Functional test generation using design and property decomposition techniques. *ACM Transactions on Embedded Computing Systems*, 8(4):32:1–32:33, 2009.

[53] A. Kuehlmann. Dynamic transition relation simplification for bounded property checking. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 50–57, 2004.

[54] L. Liu, D. Sheridan, W. Tuohy, and S. Vasudevan. Towards coverage closure: Using goldmine assertions for generating design validation stimulus. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 173–178, 2011.

[55] L. Liu and S. Vasudevan. Star: Generating input vectors for design validation by static analysis of RTL. In *Proceedings of IEEE HLDVT Workshop*, 2009.

[56] L. Liu and S. Vasudevan. Efficient validation input generation in rtl by hybridized source code analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1–6, 2011.

[57] Y. Liu, H. Yang, R. P. Dick, H. Wang, and L. Shang. Thermal vs energy optimization for dvfs-enabled processors in embedded systems. In *Proceedings of International Symposium on Quality Electronic Design*, pages 204–209, 2007.

[58] A. Lungu, P. Bose, D. J. Sorin, S. German, and G. Janssen. Multicore power management: Ensuring robustness via early-stage formal verification. In *Proceedings of IEEE/ACM International Conference on Formal Methods and Models for Co-Design*, pages 78–87, 2009.

[59] J. P. Marques-Silva and K. A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.

[60] S. M. Martin, K. Flautner, T. Mudge, and D. Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 721–725, 2002.

[61] Marvell. *Marvell StrongARM 1100 processor*. Marvell Technology Group Ltd., 2004.

[62] A. Miller, A. Donaldson, and M. Calder. Symmetry in temporal logic model checking. *ACM Computer Survey*, 38(3):8, 2006.

[63] P. Mishra and M. Chen. Efficient techniques for directed test generation using incremental satisfiability. In *Proceedings of International Conference on VLSI Design*, pages 65–70, 2009.

[64] P. Mishra and N. Dutt. Graph-based functional test program generation for pipelined processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 182–187, 2004.

[65] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of Design Automation Conference*, pages 530–535, 2001.

[66] C. Norström, A. Wall, and W. Yi. Timed automata as task models for event-driven systems. In *Proceedings of International Conference on Real-Time Computing Systems and Applications*, page 182, 1999.

[67] M. Prasad, A. Biere, and A. Gupta. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(2):156–173, 2005.

[68] X. Qin, M. Chen, and P. Mishra. Synchronized generation of directed tests using satisfiability solving. In *Proceedings of International Conference on VLSI Design*, pages 351–356, 2010.

[69] X. Qin and P. Mishra. Efficient directed test generation for validation of multicore architectures. In *Proceedings of International Symposium on Quality Electronic Design*, pages 276–283, 2011.

[70] X. Qin and P. Mishra. Automated generation of directed tests for transition coverage in cache coherence protocols. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 3–8, 2012.

[71] X. Qin, W. Wang, and P. Mishra. Tcec: Temperature- and energy-constrained scheduling in real-time multitasking systems. *To appear in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2012.

[72] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Proceedings of International Conference on Computer Aided Verfication*, pages 419–423, 2006.

[73] D. Shin and J. Kim. Dynamic voltage scaling of periodic and aperiodic tasks in priority-driven systems. In *Proceedings of Asia and South Pacific Design Automation Conference*, pages 653–658, 2004.

[74] S. Shukla and R. Gupta. A model checking approach to evaluating system level dynamic power management policies for embedded systems. In *Proceedings of IEEE International High-Level Design Validation and Test Workshop*, pages 53–57, 2001.

[75] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *ACM Transactions on Architecture and Code Optimization*, 1(1):94–125, 2004.

[76] M. Song and S. Sahni. Approximation algorithms for multiconstrained quality-of-service routing. *IEEE Transactions on Computers*, 55(5):603 – 617, 2006.

[77] G. S. Spirakis. Designing for 65nm and beyond. In *Keynote Address at the Conference on Design, Automation and Test in Europe*, 2004.

[78] O. Strichman. Pruning techniques for the SAT-based bounded model checking problem. In *Proceedings of IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 58–70, 2001.

[79] O. Strichman. Accelerating bounded model checking of safety properties. *Formal Methods in System Design*, 24(1):5–24, 2004.

[80] D. Tang, S. Malik, A. Gupta, and C. N. Ip. Symmetry reduction in SAT-based model checking. In *Proceedings of International Conference on Computer Aided Verfication*, pages 125–138, 2005.

[81] S. Vasudevan, D. Sheridan, D. Tcheng, S. Patel, W. Tuohy, and D. Johnson. Goldmine: Automatic assertion generation using data mining and static analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 626–629, 2010.

[82] R. Viswanath, V. Wakharkar, A. Watwe, and V. Lebonheur. Thermal performance challenges from silicon to systems. *Intel Technology Journal*, 4(3):1–16, 2000.

[83] I. Wagner and V. Bertacco. Mcjammer: adaptive verification for multi-core designs. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 670–675, 2008.

[84] S. Wang and R. Bettati. Reactive speed control in temperature-constrained real-time systems. In *Proceedings of Euromicro Conference on Real-Time Systems*, pages 10pp.–170, 2006.

[85] W. Wang and P. Mishra. Leakage-aware energy minimization using dynamic voltage scaling and cache reconfiguration in real-time systems. In *Proceedings of International Conference on VLSI Design*, pages 357–362, 2010.

[86] W. Wang and P. Mishra. Predvs: Preemptive dynamic voltage scaling for real-time systems using approximation scheme. In *Proceedings of Design Automation Conference*, pages 705–710, 2010.

[87] W. Wang, P. Mishra, and A. Gordon-Ross. Sacr: Scheduling-aware cache reconfiguration for real-time embedded systems. In *Proceedings of International Conference on VLSI Design*, pages 547–552, 2009.

[88] W. Wang, X. Qin, and P. Mishra. Temperature- and energy-constrained scheduling in multitasking systems: a model checking approach. In *Proceedings of*

ACM/IEEE International Symposium on Low Power Electronics and Design, pages 85–90, 2010.

[89] Z. Wang and J. Crowcroft. Quality-of-service routing for supporting multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1228 –1234, 1996.

[90] Z. Wang and S. Ranka. A simple thermal model for multi-core processors and its application to slack allocation. In *Proceedings of IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11, 2010.

[91] J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. In *Proceedings of Design Automation Conference*, pages 542–545, 2001.

[92] S. Williams. *Icarus Verilog*. Icarus Verilog, 2012. Available from: http://iverilog.icarus.com/.

[93] D. Wood, G. Gibson, and R. Katz. Verifying a multiprocessor cache controller using random test generation. *IEEE Design Test of Computers*, 7(4):13 –25, 1990.

[94] F. Xie, M. Martonosi, and S. Malik. Compile-time dynamic voltage scaling settings: opportunities and limits. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 49–62, 2003.

[95] F. Xie, M. Martonosi, and S. Malik. Bounds on power savings using runtime dynamic voltage scaling: an exact algorithm and a linear-time heuristic approximation. In *Proceedings of International Symposium on Low Power Electronics and Design*, pages 287–292, 2005.

[96] G. Xue, W. Zhang, J. Tang, and K. Thulasiraman. Polynomial time approximation algorithms for multi-constrained qos routing. *IEEE/ACM Transactions on Networking*, 16(3):656 –669, 2008.

[97] L. Yuan and G. Qu. Alt-dvs: Dynamic voltage scaling with awareness of leakage and temperature for real-time systems. In *Proceedings of NASA/ESA Conference on Adaptive Hardware and Systems*, pages 660–670, 2007.

[98] X. Yuan. Heuristic algorithms for multiconstrained quality-of-service routing. *IEEE/ACM Transactions on Networking*, 10(2):244–256, 2002.

[99] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 279–285, 2001.

[100] S. Zhang, K. Chatha, and G. Konjevod. Approximation algorithms for power minimization of earliest deadline first and rate monotonic schedules. In *Proceedings of International Symposium on Low Power Electronics and Design*, pages 225–230, 2007.

[101] S. Zhang and K. S. Chatha. Approximation algorithm for the temperature aware scheduling problem. In *Proceedings of International Conference on Computer-Aided Design*, pages 281–288, 2007.

[102] Y. Zhang, X. Hu, and D. Z. Chen. Task scheduling and voltage selection for energy minimization. In *Proceedings of Design Automation Conference*, pages 183–188, 2002.

[103] X. Zhong and C. Xu. System-wide energy minimization for real-time tasks: Lower bound and approximation. In *Proceedings of International Conference on Computer-Aided Design*, pages 516–521, 2006.

BIOGRAPHICAL SKETCH

Xiaoke Qin received the B.S. and M.S. degrees from the Department of Automation, Tsinghua University, Beijing, China, in 2004 and 2007 respectively. He received his Ph.D. from the Department of Computer and Information Science and Engineering, University of Florida, USA, in 2012. His research interests are in the area of code compression, model checking and system verification.