International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:5, No:8, 2011

MMU Simulation in Hardware Simulator Based-on State Transition Models

Zhang Xiuping, Yang Guowu, Zheng Desheng

Abstract—Embedded hardware simulator is a valuable computer-aided tool for embedded application development. This paper focuses on the ARM926EJ-S MMU, builds state transition models and formally verifies critical properties for the models. The state transition models include loading instruction model, reading data model, and writing data model. The properties of the models are described by CTL specification language, and they are verified in VIS. The results obtained in VIS demonstrate that the critical properties of MMU are satisfied in the state transition models. The correct models can be used to implement the MMU component in our simulator. In the end of this paper, the experimental results show that the MMU can successfully accomplish memory access requests from CPU.

Keywords—MMU, State transition, Model, Simulation.

I. Introduction

THE rapid growth of demands in consumer electronics has accelerated the embedded application development. To develop embedded applications, an embedded hardware platform or a hardware simulator is indispensable. An instruction-set hardware simulator is a program that simulates a target computer by interpreting the effect of instructions on the host computer, one instruction at a time. Simulator simulates the typical embedded hardware components, such as CPU, MMU, UART Controller, LCD Controller, etc. For each hardware component, in generally, special registers and controlling processes of the hardware are simulated. Fig.1 presents the block diagram of a hardware simulator. vmlinux is the kernel file that is run on the simulator.

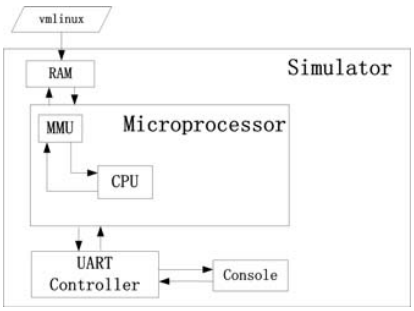

Fig. 1. Block diagram of a hardware simulator

MMU (Memory Management Unit) [1] is a coprocessor integrated in microprocessor. When the microprocessor requests to access memory, a virtual address is provided to the MMU. MMU coprocessor provides the memory access protection, virtual memory features, and memory access requests. MMU is integrated in microprocessor is the reason why SoC is available to run complex operating systems that developed with the virtual memory principle. From the ARM9 family of general-purpose microprocessors, the ARM microprocessors have integrated MMU coprocessor. In this paper, we focus on the ARM926EJ-S microprocessor and ARM926EJ-S MMU.

Currently, in industry or in college research center, MMU has been simulated for different architectures, such as ARM [2] and PowerPC [3]. ARMulator and SimSoC are ARM simulators, which include general MMU interfaces, but there are no specific MMU simulation for a microprocessor. SkyEye is also an open source hardware simulator, which includes ARM920T MMU simulation. The ARM920T MMU is tested successfully in terms of provided test cases. In SkyEye, ARM926EJ-S MMU simulation files are also included, but they are incorrect and almost just copied from ARM920T MMU files.

In literatures, the researches about MMU almost focused on the hardware architectures and performances, etc[4], [5], [6]. There were few works on how to model the MMU and simulate it correctly in a simulator. Our previous work [7] presented a component-based modeling method to model the MMU. But the component-based modeling method just gives general interfaces when the MMU is implemented. It couldn't help to design algorithms for the operational processes of the MMU. Since there is no ARM926EJ-S MMU simulation which can be applied successfully in a hardware simulator, this paper intends to build correct models for the MMU and use these models to simulate the MMU for a hardware simulator.

A state transition model [8], [9], [10] has an advantage at representing a system's states and the conditions satisfied between state transitions. In our work, we attempt to build state transition models for the operational processes of memory access, which includes loading instruction, reading and writing data. These three models are basic when an usable MMU component is designed and implemented for a simulator, so they must be correct.

In computer science, model checking [11], [12], [13] refers to the following problem: Given a model of a system, test automatically whether this model meets a given specification. Typically, the systems that have in mind are hardware or software systems. Model checking is a technique for automatically verifying correctness properties of finite-state systems. VIS (Verification Interacting with Synthesis) [14], [15], [16] is a tool that integrates the verification, simulation, and synthesis

Zhang Xiuping is with the Department of Computer Science, University of Electronic Science and Technology of China, Sichuan, China;(e-mail:knpingan@163.com)
Yang Guowu is with University of Electronic Science and Technology of China, Sichuan, China;(e-mail:guowu@uestc.edu.cn)
Zheng Desheng is with University of Electronic Science and Technology of China, Sichuan, China;(e-mail:zheng_de_sheng@163.com)

of finite-state hardware systems. It uses a Verilog front end and supports fair CTL model checking, language emptiness checking, combinational and sequential equivalence checking, cycle-based simulation, and hierarchical synthesis.

To formally verify the state transition models that are built for MMU, model checking technique is preferred. The CTL will be used to describe critical properties of our models, and the critical properties are verified in VIS. After critical properties of the MMU models are satisfied, MMU is designed and implemented. In the end, we compile a kernel for specific embedded platform and run the kernel on simulator to test the correctness of MMU component in simulator.

The rest of this paper is organized as follows. Section 2 is dedicated to build models for the operational processes of loading instruction, reading and writing data. The model verification is discussed in Section 3. Section 4 gives the experimental results and analysts, and conclusions are given in Section5.

II. Models for MMU

A. Loading Instruction Model

When CPU access to memory, the operation could be one of these three processes, loading instruction, reading data, or writing data. In any of these three processes, the operation must include fault checking, address translation, and access memory. To simulate the MMU coprocessor successfully, load instruction model, read data model, and write data model must be built correctly. These models are the base to design algorithms and implement the MMU component. In this section, we present the process of building state transition model for loading instruction in details. Because of the processes of building read and write data models are similar with the load instruction model, the read and write data models are given directly, without detailed process again.

Before state transition model diagram is presented, we define a 4-tuple to represent the elements in the state transition model.

Definition 2.1. A process is a 4-tuple (A, S, T, R) with alphabet A - the set of all possible conditions which are included in system; S - the set of states of system; $T \in S \times A \times S$ - the set of transitions; $R \subseteq S, R \neq \Phi$ (Φ is the empty set) - the set of the initial state of system.

In the followings, the structures and operations of ARM926EJ-S MMU and how to set up a 4-tuple for loading instruction are involved. For each description, we connect it with the 4-tuple, and explain how the structures and operations of MMU have a relation with the elements of the 4-tuple.

In the beginning, when hardware simulator is started, all binary instructions and data are stored in external memory. In the ARM926EJ-S microprocessor, there is a separate Cache, ICache, used to cache the instructions loaded by the CPU core. In other words, the instructions may be stored in ICache or in external memory.When loading instruction is requested, a virtual address is provided to MMU coprocessor. If there is a request to load instruction, the MMU coprocessor checks on the CP15 C1 special register to figure out whether the MMU or the ICache is enabled or not. In the case that the MMU is enabled, the memory protection and the virtual memory features operations must be executed.

At first, MMU translates virtual address into modified virtual address. After that, MMU checks address alignment and then translates the modified virtual address into physical address. The two operations reflect the attributes of memory protection and virtual memory features respectively. In the process of mapping the modified virtual address into physical address, MMU searches the TLB (Translation Lookaside Buffer) to conduct a quick translation, without accessing the external memory, which would slow down the performance of system. During the translation, a binary bit involved in descriptor indicates whether the instruction is cacheable or not.

In computer science, the principle of locality, is the phenomenon of the same value or related storage locations being frequently accessed. If the ICache is disabled, the instruction must be loaded from external memory. After the ICache is enabled, each time of loading instruction, the ICache line will be searched and instruction is loaded from it if hit. In the case that the instruction is missed in ICache line, a few consecutive instructions will be loaded from external memory to fill up a line of the ICache.

Based on the above explanations, the A in the 4-tuple of loading instruction must include request of loading instruction flag, ICache enabled flag, MMU enabled flag, instruction cacheable flag, instruction hit flag. Except that, *None* is used to indicate that no condition required in state translation. The S in the 4-tuple includes waiting state, checking on C1 state, searching TLB state, searching ICache line state, updating ICache line state, loading from external memory state, and loading from ICache line state. And the T in the 4-tuple, can be defined according to the operational processes of loading instruction. The 4-tuple (A, S, T, R) defined for loading instruction model is as follows.

$$
\begin{aligned}
&A = \{R_0, R_1, I_0, I_1, M_0, M_1, C_0, C_1, H_0, H_1, None\};\\
&S = \{S0, S1, S2, S3, S4, S5, S6\};\\
&R = \{S0\};\\
&T = \{S0 \xrightarrow{R_0} S0, S0 \xrightarrow{R_1} S1, S1 \xrightarrow{M_1,I_1} S2, S1 \xrightarrow{I_0} S3,\\
&\quad S1 \xrightarrow{M_0,I_1} S4, S2vC_0 \longrightarrow S3, S2 \xrightarrow{C_1} S4, S4 \xrightarrow{H_0} S5,\\
&\quad S4 \xrightarrow{H_1} S6, S5 \xrightarrow{None} S6, S3 \xrightarrow{None} S0, S6 \xrightarrow{None} S0\}.
\end{aligned}
$$

In A, the set of conditions, R_i - there is request to load instruction or not (i=1 means true, and i=0 means false. It is same in the followings); I_i - the ICache is enable or not; M_i - the MMU is enable or not; C_i - the instruction to load is cacheable or not; H_i - the instruction is hit in ICache line or not; *None* - there is no condition required in state transition.

In S, the set of states of system, S0 is *waiting state*, also initial state; S1 is *checking register C1 state*; S2 is *searching TLB state*; S3 is *loading from external memory state*; S4 is *searching ICache line state*; S5 is *updating ICache line state*; S6 is *loading from ICache line state*.

In R, the set of the initial state of system, S0 is the unique initial state. In T, all state transitions are included. For each state transition, there exists the required condition, which

means the next state will be reached only if the condition is satisfied. Note that the condition, such as "M_1, I_1", means that the next state is reachable if M_1 and I_1 are both satisfied. With the 4-tuple (A, S, T, R) defined for loadling instruction model, Fig.2 is the state transition model that is built for loading instruction. In Fig.2, there are conditions on all arrow lines. For each condition, the top is the condition required in state transition, and the bottom is the description of next state.

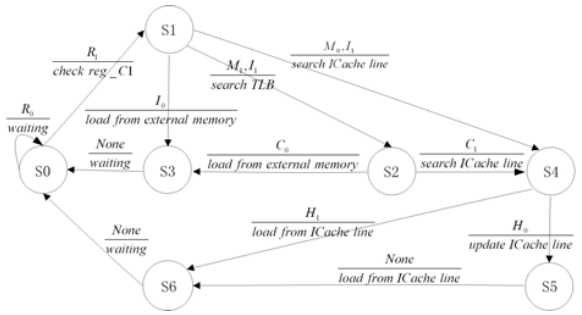

Fig. 2. State transition model for the process of loading instruction

B. Reading and Writing Data Models

In this part, we give the definitions of 4-tuple and models of reading and writing data directly. The 4-tuple (A, S, T, R) defined for reading data model is as follows.

$$
\begin{aligned}
&A = \{R_0, R_1, D_0, D_1, M_0, M_1, C_0, C_1, H_0, H_1, None\};\\
&S = \{S0, S1, S2, S3, S4, S5, S6\};\\
&R = \{S0\};\\
&T = \{S0 \xrightarrow{R_0} S0, S0 \xrightarrow{R_1} S1, S1 \xrightarrow{M_1,D_1} S2, S1 \xrightarrow{M_0 or D_0} S3,\\
&\quad S2 \xrightarrow{C_0} S3, S2 \xrightarrow{C_1} S4, S4 \xrightarrow{H_0} S5, S4 \xrightarrow{H_1} S6,\\
&\quad S5 \xrightarrow{None} S6, S3 \xrightarrow{None} S0, S6 \xrightarrow{None} S0\}.
\end{aligned}
$$

In A, the set of conditions, R_i - there is request to read data or not (i=1 means true, and i=0 means false. It is same in the followings); D_i - the DCache is enable or not; M_i - the MMU is enable or not; C_i - the data is cacheable or not; H_i - the data is hit in DCache line or not; *None* -there is no condition required in state transition.

In S, the set of states of system, S0 is *waiting state*, also the initial state; S1 is *checking register C1 state*; S2 is *searching TLB state*; S3 is *reading from external memory state*; S4 is *searching DCache line state*; S5 is *updating DCache line state*; S6 is *reading from DCache line state*. In R, the set of the initial state of system, S0 is the unique initial state. In T, all the state transitions are included. With the 4-tuple (A, S, T, R) of reading data model, Fig.3 is the state transition model that is built for reading data. Note that the condition, such as "M_0 *or* D_0", means that the next state is reachable if M_0 or D_0 is satisfied.

The 4-tuple (A, S, T, R) defined for writing data model is as follows.

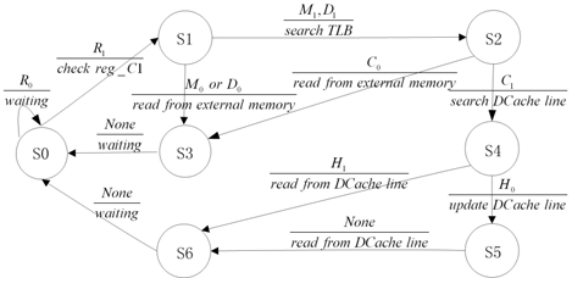

Fig. 3. State transition model for the process of reading data

$$
\begin{aligned}
&A = \{R_0, R_1, D_0, D_1, M_0, M_1, C_0, C_1, B_0, B_1, H_0, H_1, None\};\\
&S = \{S0, S1, S2, S3, S4, S5, S6, S7\};\\
&R = \{S0\};\\
&T = \{S0 \xrightarrow{R_0} S0, S0 \xrightarrow{R_1} S1, S1 \xrightarrow{M_1,D_1} S2, S1 \xrightarrow{M_0 or D_0} S4,\\
&\quad S2 \xrightarrow{C_1} S3, S2 \xrightarrow{C_0,B_0} S4, S2 \xrightarrow{C_0,B_1} S5, S3 \xrightarrow{H_0} S5,\\
&\quad S3 \xrightarrow{B_0,H_1} S6, S3 \xrightarrow{B_1,H_1} S7, S4 \xrightarrow{None} S0, S5 \xrightarrow{None} S0,\\
&\quad S6 \xrightarrow{None} S0, S7 \xrightarrow{None} S0\}.
\end{aligned}
$$

In A, the set of conditions, R_i - there is request to write data or not (i=1 means true, and i=0 means false. It is same in the followings); D_i - the DCache is enable or not; M_i - the MMU is enable or not; C_i - the data is cacheable or not; B_i - the data is bufferable or not; H_i - the data is hit in DCache line or not; *None* -there is no condition required in state transition.

In S, the set of states of system, S0 is *waiting state*, also the initial state; S1 is *checking register C1 state*; S2 is *searching TLB state*; S3 is *searching DCache line state*; S4 is *writing to external memory state*; S5 is *writing to WriteBuffer state*; S6 is *writing to DCache line state*; S7 is *writing to DCache line and WriteBuffer state*. In R, the set of initial state of system, S0 is the unique initial state. In T, all the state transitions are included. With the 4-tuple (A, S, T, R) of writing data model, Fig.4 is the state transition model that is built for writing data.

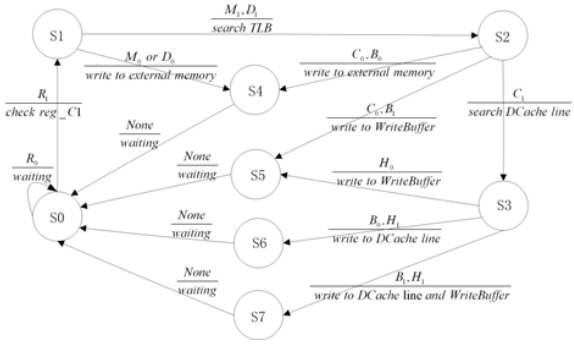

Fig. 4. State transition model for the process of writing data

III. Model Verification

The loading instruction model, reading data model and writing data model have been built in Section II. To verify whether critical properties of MMU are satisfied in these

models or not, model verification is the indispensable work. In the followings, the critical properties of load instruction model are verified formally. The verifications of reading and writing data models are not involved in this section, which can be verified in the same way.

Computation Tree Logic (CTL) [17], [18] is a branching-time logic, meaning that its model of time is a tree-like structure in which the future is not determined; there are different paths in the future, any one of which might be an actual path that is realized. In the followings, the CTL is used to describe critical properties. "$\rightarrow$", "$\wedge$" and "$|$" mean logical "implication", "and" and "or", respectively. "AG", "EF" and "AX" mean "all states in all paths", "exist a state" and "next state in all the path", respectively.

Note that the following symbols S0 through S6 are the states we defined for the 4-tuple of loading instruction.

Property 1: From the initial state S0, there exists two states, the *loading from external memory state* S3 and the *loading from ICache line state* S6, that are reachable.

$$EF(S0 \rightarrow S3) \wedge EF(S0 \rightarrow S6);$$

Property 2: At anytime, if a loading instruction request is responded, future state would be the *loading from external memory state* S3 or the *loading from ICache line state* S6. In the end, it will go back to the *waiting state* S0.

$$AG((S3 \mid S6) \rightarrow AX(S0));$$

Property 3: The next state of the *searching TLB state* S2 must be the *loading from external memory state* S3 or the *searching ICache line state* S4.

$$AG(S2 \rightarrow AX(S3 \mid S4));$$

Property 4: The next state of the *searching ICache line state* S4 must be the *loading from ICache line state* S5 or the *updating ICache line state* S6.

$$AG(S4 \rightarrow AX(S5 \mid S6));$$

The above four properties are the critical properties must be satisfied in loading instruction model. In the followings, we define two properties which are incorrect in the MMU coprocessor, to verify whether they are satisfied in the state transition model of loading instruction.

Property 5: Given the condition M_0 (MMU is disabled) is satisfied, the *searching TLB state* S2 is reachable.

$$AG((M_0 \wedge S0) \rightarrow EF(S2));$$

Property 6: Given the condition I_0 (ICache is disabled) is satisfied, the *searching TLB state* S2 or the *searching ICache line state* S4 is reachable.

$$AG((I_0 \wedge S0) \rightarrow EF(S2 \mid S4));$$

After describing the properties with CTL and implementing the state transition model of loading instruction with Verilog (we don't present in paper), the above properties can be verified in VIS. The experimental results of model checking obtained in VIS are reported in Table I.

TABLE I
RESULTS OF PROPERTIES VERIFIED IN VIS

Properties	Results	Counterexamples
Property 1	Satisfied	No
Property 2	Satisfied	No
Property 3	Satisfied	No
Property 4	Satisfied	No
Property 5	Non-satisfied	$S0 \rightarrow S0 \rightarrow S0 \rightarrow \ldots$ $S0 \rightarrow S1 \rightarrow S3 \rightarrow \ldots$ $S0 \rightarrow S1 \rightarrow S4 \rightarrow \ldots$
Property 6	Non-satisfied	$S0 \rightarrow S0 \rightarrow S0 \rightarrow \ldots$ $S0 \rightarrow S1 \rightarrow S3 \rightarrow \ldots$

As we see from Table I, property 1 through property 4 are all satisfied in our state translation model, property 5 and property 6 are non-satisfied. The counterexample for property 5 is that, from the initial state S0, if the M_0 is satisfied, all probable paths include $S0 \rightarrow S0 \rightarrow S0 \rightarrow \ldots$, $S0 \rightarrow S1 \rightarrow S3 \rightarrow \ldots$, and $S0 \rightarrow S1 \rightarrow S4 \rightarrow \ldots$. We can see that S2 is unreachable. The counterexample for property 6 is that, from the initial state S0, if the I_0 is satisfied, all probable paths include $S0 \rightarrow S0 \rightarrow S0 \rightarrow \ldots$ and $S0 \rightarrow S1 \rightarrow S3 \rightarrow \ldots$. We can see that S2 and S4 are unreachable.

If the MMU is disabled, then no virtual address translation occurs. The virtual address is mapped into physical address directly. So there are no fault check operation, no memory protection. At this case, instruction is loaded from external memory or from the ICache line with the input virtual address, the *searching TLB state* is unreachable. For that reason, the property 5 must be non-satisfied.

In the case that the ICache is disabled, no searching ICache line operation occurs. So, searching the TLB operation isn't needed even though the MMU is enabled, the instruction is loaded directly from external memory. The property 6 must be non-satisfied.

IV. EXPERIMENTATIONS

The state transition models and model verification are included in Section II and Section III. As we see from the model checking results obtained in VIS, the correct critical properties are satisfied and the incorrect properties are non-satisfied in the loading instruction model. With these three models, the MMU component can be designed and implemented for a hardware simulator.

In this section, to test the correctness of the implemented MMU component, we compile a Linux kernel for specific platform and run it on simulator. The S3C2413X SoC uses ARM926EJ-S microprocessor, and the Linux kernel version 2.6.18 published has supported the S3C2413X platform. We decide to compile kernel for platform S3C2413X, and run it on our simulator Apsim.

The entry address of the kernel file is 0xC0008000. In the beginning, the MMU, ICache and DCache is disabled, all memory access requests are responded from external memory. In our experimentation, we track on the key addresses, which include the first addresses for loading instruction, reading data, and writing data. The address 0xC0008000 is the first

instruction's address that MMU loads from external memory; 0xC0008344 is the first data's address that MMU reads from external memory; 0xC0004000 is the first data's address that MMU writes to external memory. After the MMU, ICache, and DCache are all enabled, 0xC0008070 is the first instruction's address that MMU load from ICache; 0xC0008124 is the first data's address that MMU reads from DCache; 0xC03D0AE0 is the first data's address that MMU writes to DCache.

TABLE II
THE STATISTICAL RESULTS OF MEMORY ACCESS OPERATIONS

Operations	Items	Correct Rates
Load Instruction	Instruction	100%
	Location	100%
Read Data	Data	100%
	Location	100%
Write Data	Data	100%
	Location	100%

Except for the key addresses, we track on the operations of loading instruction, reading data and writing data for 50 times, respectively. For each operation, we focus on the instruction or data, and the location of accomplishing this operation. Table II summarizes results of memory access operations for 50 times.

We can see from Table II, all the instructions and data to access and the locations are correct. During the time that the kernel is run on the simulator, numerous operations of accessing memory are executed. If the MMU could successfully accomplish all the memory access requests from CPU, the results of running the kernel on the simulator must be same with that run on hardware. Fig. 5 presents the output of running the kernel on the simulator. In Fig. 5, we can see that the kernel is run successfully on the simulator, and the MMU component of simulator can correctly respond all the requests of memory access from CPU.

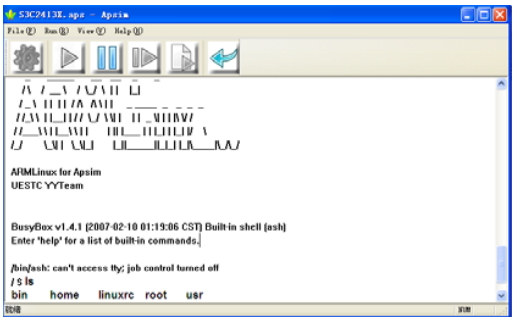

Fig. 5. The output of running kernel file on simulator Apsim

V. CONCLUSIONS

In this paper, we focused on the ARM926EJ-S MMU, defined 4-tuple to describe the elements of models and built state transition models for MMU, which include loading instruction model, reading data model, and writing data model. To formally verify the critical properties of our model, model checking was used and results obtained in VIS demonstrated the correctness of the load instruction model. In the end, the MMU component was designed and implemented for hardware simulator, and our experimental results showed that the MMU component can successfully accomplish the memory access requested by CPU.

Our works use state transition technique to model the ARM926EJ-S MMU coprocessor and critical properties of models are verified formally. Since the ARM926EJ-S MMU hasn't been simulated successfully in industry or college research center, our works could be a small contribution to the hardware simulation field. On the other hand, with the reason that there are many controllers in SoC and a hardware simulator must include other key simulated controllers, our approach is also valuable and worth considering for simulating other controllers.

REFERENCES

[1] ARM. *ARM926EJ-S Technical Reference Manual*. ARM Limited, www.arm.com, r0p5 edition, 2008.
[2] C. Helmstetter, V. Joloboff, and H. Xiao. Simsoc: A full system simulation software for embedded systems. In *Open-source Software for Scientific Computation (OSSC), 2009 IEEE International Workshop on*, pages 49–55. IEEE.
[3] T. Andersson and P. Magnusson. Powerpc mmu simulation. Bachelor's project, Karlstad University, Sweden, http://www.cs.kau.se/cs/, 2001.
[4] B. Egger, S. Kim, C. Jang, J. Lee, S.L. Min, and H. Shin. Scratchpad memory management techniques for code in embedded systems without an mmu. *Computers, IEEE Transactions on*, 59(8):1047–1062, 2010.
[5] J.R. Haigh, M.W. Wilkerson, J.B. Miller, T.S. Beatty, S.J. Strazdus, and L.T. Clark. A low-power 2.5-ghz 90-nm level 1 cache and memory management unit. *Solid-State Circuits, IEEE Journal of*, 40(5):1190–1199, 2005.
[6] S.K. Agun and J.M. Chang. Design of a reusable memory management system. In *ASIC/SOC Conference, 2001. Proceedings. 14th Annual IEEE International*, pages 369–373. IEEE, 2001.
[7] X. Zhang, G. Yang, and D. Zheng. Component-based model for simulating the mmu coprocessor. In *Information Engineering and Computer Science (ICIECS), 2010 2nd International Conference on*, pages 1–4. IEEE.
[8] DC McLernon. Properties for state-transition matrix of lptv two-dimensional filter. *Electronics Letters*, 38(25):1748–1750, 2002.
[9] Y.C. Lee and A.Y. Zomaya. A novel state transition method for metaheuristic-based scheduling in heterogeneous computing systems. *IEEE Transactions on Parallel and Distributed Systems*, pages 1215–1223, 2008.
[10] A. Chander, D. Dean, and J.C. Mitchell. A state-transition model of trust management and access control. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 27–43. Citeseer, 2001.
[11] E. Clarke. Model checking. In *Foundations of Software Technology and Theoretical Computer Science*, pages 54–56. Springer, 1997.
[12] B. Nicolescu, N. Gorse, Y. Savaria, E.M. Aboulhamid, and R. Velazco. On the use of model checking for the verification of a dynamic signature monitoring approach. *Nuclear Science, IEEE Transactions on*, 52(5):1555–1561, 2005.
[13] T. Han, J.P. Katoen, and B. Damman. Counterexample generation in probabilistic model checking. *IEEE transactions on software engineering*, pages 241–257, 2009.
[14] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, et al. Vis: A system for verification and synthesis. In *Computer Aided Verification*, pages 428–432. Springer, 1996.
[15] H. Peng, S. Tahar, and F. Khendek. Comparison of spin and vis for protocol verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(2):234–245, 2003.
[16] J. Yoo, E. Jee, and S.S. Cha. Formal modeling and verification of safety-critical software. *IEEE Software*, pages 42–49, 2009.
[17] Q. Zhao and B.H. Krogh. Formal verification of statecharts using finite-state model checkers. In *American Control Conference, 2001. Proceedings of the 2001*, volume 1, pages 313–318. IEEE, 2001.
[18] M. Bourahla. Distributed ctl model checking. In *Software, IEE Proceedings-*, volume 152, pages 297–308. IET, 2005.