ABSTRACT
Complex and sophisticated power management strategies are a commonplace design policies today in order to manage the power consumption of complex low power digital integrated circuits. These global power management strategies are implemented in software/firmware and are used to orchestrate the switching between power states of multiple power domains in local power controllers which resides in hardware. In this paper, we propose a methodology of verifying such global power management softwares with safety linear temporal logic (LTL) properties using bounded model checking based verification approach. We present our results on several test cases of significant complexity to demonstrate the feasibility of the proposed framework.

Categories and Subject Descriptors
D.2.8 [Software Engineering]: Software/Program Verification—Model Checking

General Terms
Embedded Software, Verification

Keywords
Power Intent Verification, Assertion, Bounded Model Checking, Temporal Logic

1. INTRODUCTION
The rise in the use of embedded systems over the last decade has forced the developers to integrate embedded software in such systems in a rapid way. Thus the need of formally verifying the embedded software is of fundamental importance in the present scenario in order to ensure the correctness of the functionality of embedded systems. However, when a large software design is considered, formal property verification using model checker often suffers from the state space explosion problem. The main challenge for any embedded software is in verifying the temporal properties over the variables and functions of the software module. The authors of [18] proposed two new approaches to integrate assertions for verifying embedded software using simulation-based verification.

In the domain of low power embedded system design, the task of verifying that the implementation of the power domains and the architectural power management logic meets the micro-architectural specs has become important. This has prompted the design community to adopt standards like Unified Power Format (UPF) [17] and Common Power Format (CPF) [1] for specifying the power domains and their control points at a higher level of abstraction. The power management fabric is broadly divided into the controllers for the individual power domains and the global power management logic which orchestrates the power state transitions in the local power domains. The global power management logic is responsible for implementing the architectural power management strategy, but the per-domain controllers specified using UPF and CPF control analog interfaces, like level shifters.

A recent research work has attempted to leverage per-domain UPF / CPF specifications for translating architectural power intent into assertions for enabling formal verification of the global power management logic [8]. Our work developed on the same approach, but with an important difference, namely, that the global power management logic is specified in software, as opposed to the assumption in [8] that the logic is in hardware. The significance of our contribution is based on the fact [11] that most recent designs use a combination of software and micro-instructions in firmware to implement the global power management logic, and it is increasingly likely that in future this trend will become ubiquitous. The work of [16] models the synchronization between the software which implements the global power management logic with the per-domain controllers that are implemented in hardware. It then verifies the composition of the software and hardware power managers against the architectural power intent.

There are several important differences in verifying a power management fabric that is partially controlled by software/firmware, as opposed to ones that are implemented solely in hardware. Typically global power management (GPM) software is driven by software/hardware interrupts. For example, if an interrupt comes then the GPM executes a sequence of instructions in order to determine the next state of itself as well as the local power managers. The task of the GPM is to determine the global power management strategies and
provide directives to the local power managers (LPM) (typically in hardware) which are responsible for actual execution of these strategies into hardware.

Our objective in this paper is to model such software power management units by capturing the execution semantics in terms of the interrupts that drive the software, specifying the architectural intent (safety requirements) of the power management strategies in terms of these interrupts, and formally verifying the architectural intent on the software using state of the art formal software verification techniques.

An incorrectly implemented global power controller might issue illegal power state transition sequence and GPM directives thereby violating the desired power-intent. It is therefore imperative to consider the timing and functionality of the GPM software in verification. Thus the verification of the global power management strategies in software is important in guaranteeing the functional correctness of the directives that it issues over the entire power-management fabric. The main contributions of this paper are as follows:

1. We allow the specification of the global power management strategy in a procedural language like C.
2. Given the LTL properties capturing architectural power intent, we automatically translate the temporal logic properties into safety properties of given depth using unfolding technique.
3. The GPM software is automatically instrumented with the assertions generated in step 2.
4. The instrumented code is then passed to the CEGAR loop for model checking the GPM software against the safety properties.

2. RELATED WORK

In the past, LTL [19] has been the most acceptable logic for describing temporal properties in the model checking environment such as SPIN [10]. Assertion-based verification methodology is useful for hardware verification technique [7]. However, this technique of verification is not suitable for verifying embedded software which contains complex programming constructs like complex data types, pointers, recursive functions etc. and has no notion of timing behaviors.

The crux in the verification of embedded software lies in monitoring the valuation of the program variables, functions, conditional loops and complex pointer handling based on temporal property specification. Formal methods prove to be efficient for verifying temporal properties over medium sized embedded software. However, for larger software design, an abstraction technique is used in order to avoid the state-explosion problem for the back-end model checker. The tool named BLAST [9] verifies temporal safety properties on software using predicate abstraction technique. BLAST uses Counter-Example Guided Abstraction Refinement (CEGAR) framework for model checking software. It iterates the whole process by first constructing an abstract model of the software based on predicates, then checks the safety properties, and if the property fails, it refines the model based on the counter-example. The software model checking approaches can be categorized as follows:

1. Bounded Model Checking (BMC) [3] based verification
2. Predicate abstraction based model checking using a theorem prover or SAT solver [9, 4, 2]
3. Modeling the semantics of the software using suitable abstraction technique and then verify the abstract model using BDD-based and SAT-based model checkers.

The tool CBMC [5] performs the formal verification of ANSI-C programs using bounded model checking based approach. However, the boundedness restricts the tool for proving the correctness of the software. Therefore, the verification of temporal properties of embedded software is still a challenge for the verification community.

A low-power digital integrated circuit can be viewed as the composition of several individual power-domains constituting the design, a set of power control circuitry (PCC), and the power control logic (PCL). PCC provides a power control interface which is driven by the PCL so as to make power state transitions within the power managed domain. A PCC consists of various isolation cells, retention cells, level shifters and voltage regulators. Further, PCL can be decomposed into two separate power managers for larger circuits, namely the local power manager (LPM) and the global power manager (GPM). LPM’s are mainly per-domain local power managers which issues valid control signals to the corresponding power domain so as to make a valid power state transition. The GPM implements various global power management properties. It orchestrates the LPM’s to implement the architectural power management strategy. The architectural view of such power management logic is popular among architects [12, 13]. Figure 1 shows the architectural components of a low-power digital integrated circuits.

![Figure 1: Architectural Power Managed Design Component](image-url)
and low-level control signals over which the PCL is defined, and performs the formal verification of the Power Control Logic (PCL) at the register transfer level.

In [8], all controllers are given in Verilog and so it is implemented using standard model checkers from the hardware domain. In our problem the global power management logic is in software and the local controllers are in Verilog. Hence we propose a verification framework for the purpose of verifying the correctness of the global power management logic against the architectural power intent.

Software-based power management strategy tries to exploit explicit power management hooks provided by the underlying architecture that are of particular interest to a software at higher level in an intelligent manner. Power management software employs efficient algorithms to ensure that the system and devices maintains low power state as long as possible without degrading system performance. In order to make maximum utilization of the CPU, modern day operating system like Linux implements dynamic voltage and frequency scaling (DVFS) based on the system’s utilization via heuristic algorithms. Recently, Intel’s Nehalem microarchitecture uses a dedicated on-die microcontroller [11], called the power control unit (PCU) that runs embedded firmware, used for making power management decisions based on global inputs like temperature, current, power and OS requests. Thus, the migration from hardware to firmware / software based power management in modern day processors provides efficient platform for managing system’s power.

3. FORMAL MODEL

The GPM software is triggered by software/hardware interrupts. On waking up, the GPM issues one or more directives and goes back to sleep. The directives are executed by the local controllers to actuate the power state transitions. Our objective is to verify architectural assertions over the global power management software. The GPM software is used to implement the overall power management strategies of the embedded system and orchestrate the switching between power states of multiple power domains in local power controllers. Unlike conventional software verification techniques, the main aspect of verifying the power management software lies in the fact that the program states (comprising valuation of the program variables and program locations) and the power states defining the per-domain power states are two different entities. The architectural properties are specified relating the power states and its transitions.

In order to formally capture the execution semantics of the GPM, we model the interrupts that drive the GPM with a single clock, that is the time period between any two consecutive interrupts define a time cycle. Figure 2 depicts the scenario for a typical GPM software. We take an example in English language describing a typical safety requirement. The architectural assertions are defined over the GPM power state and may be used to capture the temporal nature of the power management behavior. With the GPM software execution model described in this section, it is possible to encode the above requirement using a temporal logic property as follows:

For example, let us consider an architectural property comprising three power domains namely CORE, MEMORY and TRANSMITTER. The properties may be temporal in nature to capture the architectural power intent or it may be safety properties as well.

1. Property 1: We may be interested to verify a temporal property which specifies that:

   Once the MEMORY domain is ON, the TRANSMITTER domain has to be ON within 1 to 2 cycle.

   The property can be encoded as follows:

   \[ \text{G}(\text{M}_\text{ON}_\text{T}_\text{OFF} \rightarrow F[1:2] \text{M}_\text{ON}_\text{T}_\text{ON}) \]

2. Property 2: Global safety properties may also be used to capture power management strategies which specifies that:

   CORE and TRANSMITTER domains are never ON together.

   The property can be encoded as follows:

   \[ \text{G}(\neg \text{C}_\text{ON}_\text{T}_\text{OFF} \land \text{C}_\text{OFF}_\text{T}_\text{ON} \land \text{C}_\text{OFF}_\text{T}_\text{OFF}) \]

4. VERIFICATION FRAMEWORK

Let PD be the total number of power domains and M be the total number of states. Then, we have \( N = \log_2 M \) number of state variables. The inputs of the system are as follows:

1. Software code of the Global Power Manager (GPM)
2. GPM specification as a set of Temporal Logic properties (\( \varphi^{i,j} \)) encoded in terms of \( N \) state variables \( \{S_0, S_1, \ldots, S_N\} \).

We use \( \varphi^{i,j} \) to denote that \( \varphi \) is true at the \( i^{th} \) time step and \( \varphi \) is to be unfolded further for \( j \) time steps. We interchangeably use \( \varphi' \) to denote \( \varphi^{0,0} \), that is \( \varphi \) is true at the \( i^{th} \) time step and is not unfolded further.

4.1 Procedure

For performing the verification task we unfold the LTL properties up to certain depth (either determined from the properties if they are bounded or chosen by the verification engineer) in order to express them as safety properties of the software. This is facilitated by maintaining state information up to the chosen depth. The steps of the proposed approach shown in Figure 3 work as follows.

1. We store \( j \)-depth of state information. Thus, we need to expand the property \( \varphi \) up to \( j \)-depth.
2. The recursive unfolding of \( \varphi^{i,j} \) is defined as follows:
We save 4 levels of state bits (only up to a depth of 3) is:
\[ \varphi^3 \equiv S^3 \lor (S^1 \land S^2) \lor (S^1 \land S^3) \lor (S^1 \land S^2 \land S^3) \]

Let us now consider the semantics of the global power management controller software. When the program is in a particular power state and some external interrupt arrives, the power management strategy determines the next state and the signals asserted during the process. Each state of the global power manager controller corresponds to the set of states of the individual local power managers. Our GPM software has the following form:

```c
int state;
void controller() {
    ...
    state = new_state;
    ...
}
```

The global controller is instrumented with the unfolded safety properties. The `update_saved_state_information()` procedure maintains state information up to certain depth of the temporal property which is either determined by the property if they are bounded or specified by the verification engineer. The instrumented code is as follows:

```c
int state;
int S0, S1, S2, S3;
void controller() {
    ...
    state = new_state;
    update_saved_state_information();
    // state change takes place :
    unfolded assertion is inserted -
    S1 || (!S1 && S2) || (!S1 && !S2 && S3)
    || (!S1 && S2 && S3);
}
```

### 4.3 Case Study

In order to outline the syntax for specifying the global power management (GPM) logic, we consider an example consisting of three power domains, namely CORE, MEMORY and TRANSMITTER. The GPM description starts with an enumeration of the power states of the design:

```c
enum STATE {
    ST0, //Core-Off, Trans-Off, Mem-PartiallyOn
    ST1, //Core-Off, Trans-Off, Mem-FullOn
    ST2, //Core-StandBy, Trans-Off, Mem-PartiallyOn
    ST3, //Core-Active, Trans-Off, Mem-FullOn
    ST4, //Core-Off, Trans-Low, Mem-FullOn
    ST5, //Core-Off, Trans-High, Mem-FullOn
};
```

The following C code snippet for the GPM logic implements the following aspect of the power management strategy:

> On getting an interrupt signal, the GPM drives a transition from ST0 to ST2 while retaining the...
same state for other two domains. It also drives a transition from ST0 to ST1 if either of the following two conditions hold:
\[
(\text{finish} \&\& \text{setbit}) \lor (\text{finish} \&\& \text{signal} \_\text{transmit})
\]

The GPM retains the same state if none of the above conditions occur.

```c
GPM\_core\_mem\_transmitter()
{
    const enum STATE curSt;
    int interrupt\_new, setbit\_new,
    signal\_transmit\_new, signal\_str\_new ; // Output of GPM
    if(curSt == ST0)
    {
        interrupt\_new = interrupt; //saves initial configuration
        setbit\_new = setbit;
        signal\_transmit\_new = signal\_transmit;
        if(interrupt == 1)
        {
            setbit\_new = 0;
            signal\_transmit\_new = 0;
            state = ST2; // Transition :: 'ST0 - ST2'
        }
        else if((finish \&\& setbit) ||
        (finish \&\& signal\_transmit))
        {
            if(finish \&\& setbit)
            {
                signal\_transmit\_new = 0;
                signal\_str\_new = 1;
            }
            else if(st0 \&\& signal\_transmit)
            {
                signal\_str\_new = 0;
                signal\_transmit\_new = 1;
            }
            state = ST1; // Transition :: 'ST0 - ST1'
        }
        else
        {
            signal\_str\_new = 1;
            finish\_new = 1;
            state = ST0; // Transition :: 'ST0 - ST0'
        }
    }
}
```

The signals `interrupt`, `setbit`, `signal_transmit`, and `finish` are inputs to the GPM. These variables are modified by operating system events or by hardware events. The variables `curSt` and `state` contain the current state and the next state values respectively. Writing the `state` variable acts as a directive to the local per-domain power controllers to change the power states to match with this variable. Let us consider the following bounded LTL property [6].

\[
G((ST2 \rightarrow F_{[0:2]} ST1) \lor (ST3 \rightarrow F_{[1:2]} ST0))
\]

Since this is a safety requirement, so the unfolding of the property into safety property is shown as follows. The state level information in each iteration \(i\) of the GPM execution is updated and captured using the variable \(L_i\).

\[
( (L_0 == ST2) \&\& (L_0 == ST1)) \lor (L_0 == ST2) \lor (L_1 == ST1) \lor (L_0 == ST2) \lor (L_2 == ST1))
\]

These assertions are automatically instrumented inside the GPM software and the level informations are updated whenever some state change takes place.

Figure 4: Power Domains of eleon3 processor

5. EXPERIMENTAL RESULTS

In this section, we present experimental results on six test cases, namely PowerTrans, PowerCounter, Prometheus and low-power Enhanced Leon3 benchmarks [14, 15], namely, eLeon3-SingleCore, eLeon3-DualCore and eLeon3-QuadCore. We have used Linear Temporal Logic (LTL) to express the temporal safety architectural properties and invoked SATABs [4], a verification tool for ANSI-C and C++ programs, to perform the model checking of the GPM software. The results presented in this section are performed on a 2.33 GHz INTEL(R) XEON(TM) CPU with 16 GB RAM.

Figure 4 shows the different power domains of the single-core eleon3 processor, namely primary integer unit, a secondary integer unit, a multiplication unit, a division unit, a memory controller unit, a cache unit, and a storage elements unit. Each domain has its own power modes, like ACTIVE, IDLE, OFF, FULL\_ON and PARTIAL\_ON, as decided by the architects. These power states are further specified using three flags, namely voltage, frequency, bias flag. There also exists potential transient states between the OFF\_state and on\_state of a power domain, namely Isolation, Retention, or between two on\_states by changing the voltage-frequency pair in particular order.

The GPM design details for various benchmark circuits are summarized below in Table 5. Column 2 denotes the size of the global power manager in terms of number of lines of code. The number of power states present in each global power manager of the power control logic is listed in Column 3. We also report the verification results using the Model checking tool SATABS [4] in Table 5. Column 4 denotes the number of global assertions. The maximum unfold depth among all global architectural properties is reported in Column 5. Column 6 shows the verification time and Column 7 reports the memory usage by the proposed methods.

6. CONCLUSION

This paper demonstrates the verification of global power management strategies implemented in software with respect to safety LTL properties using reachability analysis. The LTL properties are automatically translated into safety properties of a given depth and the generated assertions are automatically instrumented inside the GPM software. The instrumented code is then passed to the CEGAR framework for model checking the GPM software with respect to safety properties and thereby ensuring the correctness of the global power management strategies in software. Thus, the proposed verification framework is suitable for proving the
correctness of the embedded software against the temporal architectural assertions and the experimental results on embedded power management strategies for ELEON processor shows the feasibility of the proposed verification approach.

7. REFERENCES