Power Analysis of Interrupt-Driven and Multi-Threaded Programs *

Fang Yu
Department of Computer Science
University of California, Los Angeles

September 5, 2006

1 Abstract

We aim to combine software verification techniques to achieve static power analysis for interrupt-driven and multi-threaded programs, which are used in many networked embedded systems. The goal is achieved by 1) control flow analysis, 2) instruction-level power estimation/emulation, 3) thread-context model, and 4) counter-example guided refinement.

2 Introduction

One essential requirement for sensor networks is the reliability of applications since sensors are planned to deploy into the area once and unattended for a period of time without maintenance. An interesting focus is the lifetime of the sensor network. In some sense, the lifetime for a hardware configuration reflects the period from the beginning till it out of charge. Hence, how to achieve precise power analysis attracts more and more research attention. Previous researchers focus on simulation/emulation, in which they calculate the power consumption of their implementation along some specified execution.

Since concurrent interactions between components may make the behavior of applications hard to predict, it appears to be attractive to adopt formal verification to support power analysis.

The hurdle relies on handling the interaction between concurrent tasks and hardware interrupt, which raise exponential behaviors of the system and makes dynamic methods, e.g., testing/simulation [17, 23], run out of ability. While symbolic model checking shows some promise, previous works focused on explicit problems of programming languages, such as data race checking [14], and none of them address the power analysis, an emerging requirement for sensor networks.

To address this issue, we aim to extend symbolic model checking to power analysis of multi-threaded programs, such that most networked embedded systems, e.g., nesC/tinyOS and SOS applications, can be analyzed with guarantee.

The extension is achieved by combining the following techniques: 1)control flow analysis [19] 2)

*This is the draft for the summer research with Prof. Sarrafzadeh, Summer06

instruction-level power estimation [23]/emulation [17], 3) thread module abstraction [11,13] and 4)counter-example guided refinement [12, 14].

3 Power Model

Before proceeding power analysis, the first step is constructing the basic power information as a start point to analyze complex behavior. One idea is dividing the program into basic units such that the power consumption of each unit can be predicted precisely with existed simulation/emulation tools.

To achieve this, we induce *Control Flow Graph*(CFG). We serve a CFG as a representation of a program where contiguous regions of code without branches, known as basic blocks, are represented as nodes in a graph and edges between nodes indicate the possible flow of the program.

First we focus on C programs. The C Intermediate Language(CIL) is a high level representation along with a set of tools that make it easy to analyze and manipulate C programs, and emit them in a form that resembles the original source code. CIL has three basic concepts: expressions, instructions, and statements. Expressions represent functional computation, without side-effects or control flow. Instructions express side effects, including function calls, but have no local control flow. Statements capture local control flow. The program structure is captured by a recursive structure of statements, with every statement annotated with successor and predecessor control-flow information. Given a basic block, we need to estimate its power assumption. The precise power analysis is based on the execution of real codes in an simulator/emulator, e.g., Avrora [24] and Atmeu [20]. Landsiedel et al. extended AEON [17]to break the CPU consumption down to individual routines and blocks of the source code. 2) Rather than emulation, ShynayderčiteSHCAW04 map each basic block to the appropriate number of CPU instructions as executed by the Atmega128L on the Mica2 by compiling the TinyOS application C codes to a Mica2 binary and using a disassembler(avr-objdump) to determine the set of AVR instructions for each source code line. In general, their tool, PowerTOSSIM, is faster than emulation-based tool but sacrifices accuracy in some cases.

On the other hand, we may look at machine codes directly, i.e., construct the CFG of machine codes, and hence prevent mapping languages. Avrora [24] includes a tool to generate a control flow graph for AVR programs, which emits the structure and organization of machine code programs to dot files [6]. Some utilities are also implemented, e.g., collapse and group procedures, and may be used to do abstraction. Since Avrora stands on a cycle-accurate instruction-level simulator and can further analyze energy consumption, we may equip the output CFG with energy information.

4 Abstract-Check-Refine Paradigm

We aim to achieve precise power analysis by extending traditional simulation method to verification, such that we are able to consider all cases rather than look at specified ones.

The main issues are a) what is a model for the program that simultaneously 1) abstract enough to permit efficient checking and 2) precise enough to preclude false positives as well as yield real traces, and

b)how can we derive such model automatically. We follow abstract-check-refine paradigm [14]. We start at some coarse abstraction which overapproximates system behaviors, and iteratively check and refine the model according to infeasible counter examples.

5 Single Thread Analysis

We first consider one thread case. In the following we propose two methods to analyze single thread program power consumption.

5.1 Control Flow Graph

We focus on the thread itself behavior, and ignore the environment interaction and system interrupt. This is the most coarse model but can be done efficiently. (I think the time complexity may be polynomial to the number of branch points)

5.1.1 Coarse Abstraction

The power analysis is achieved by 1) according to the C code, generating Control Flow Graph(CFG) of basic blocks(no branching) (Using C Intermediate Language, a toolkit can build high-level representation of the structure of the C programs) 2) for each basic block, estimating the power consumption by emulating its corresponding instructions with Atmeu [20]. 3) reducing the CFG to a directed graph $G = (V, E)$, where each $v \in V$ is a branching point and $e \in E, e = (v, c, v')$, $v, v' \in V$, and c is the cost reflecting the power consumption to complete the calculations between branch points v to v'.

Then finding the min/maximum power consumption to finish a task could be reduced to finding the shortest/longest path form s to t in G, where s and t stand on the points the task initialized and finished respectively. Without loss of generality, s and t could be a set of states.

Note that in this abstraction, we do not trace the values of variables and discard all branching conditions of programs. As a result, we over approximate system behaviors.

5.1.2 Check-Refine Paradigm

Since we overapproximate system behaviors, the path we found might be infeasible in real programs. For example, we may allow a path jumping into infinite loops with arbitrary high costs. To address this problem, we adopt the counterexample guided refinement. [12]. We simulate the path in real programs, if it's feasible, we are done and may get the precise power consumption along the path. If it's not, we refine G to avoid finding this path (e.g., change the cost of infeasible edge to $\infty/0$), and repeat the algorithm.

5.2 Control Flow Automata

While the control flow analysis is efficient, it might be imprecise, e.g., it ignores the environment. Though the defect could be covered by adopting check-refine paradigm, it may lead inefficient computation. We use discrete finite automata with synchronizers to simulate the interactions between environment in this section.

5.2.1 Coarse Abstraction

We generate Control Flow Automata(CFA) $A = (Q, X, S, Q_0, \delta)$ from the CFG $G = (V, E)$, where Q is the set of states, X is a set of discrete variables, S is a set of sychronizers, Q_0 is the initial states, $\rightarrow$ is the transition relation. For each $e = (v, c, v') \in E$, we generate an edge (v, a, v'), denoted as $v \rightarrow^a v'$,where a is the action $engr := engr + c$. To simulate environment, e.g., system interrupt, we also induce extra discrete automata to randomly and periodically generate different events by sending/receving synchronizers. Initially, X only contains an auxiliary variable engr to record the power consumption during computation.

Since we ignore values of variables in original programs, we keep on overapproximating system behaviors. If the program is safe, then we are done. O.w., a counterexample is generated.

Again we have to simulate the path in real programs. If it is feasible, we find a witness that the system does not have sufficient resource to finish the task. If it is not feasible, we adopt lazy abstraction [12] to refine the model.

5.2.2 Check-Refine Paradigm

The CFA could be more precise with predicate abstraction, in which we serve the branch-point conditions as predicates and trace how its value changes. Instead of tracing the exact values of variables, we trace values of Boolean formulae over a finite set of predicates P over the variables. Each $p \in P$ is a boolean variable standing for the value of branch point condition. At each iteration, according to the infeasible path, we expand X to $X \bigcup \mathcal{P}$, where $\mathcal{P}$ is the set of predicates of branch conditions which make the path infeasible.

Here comes the problem. Since we choose power analysis to prevent tracing arithmetic operations, adding predicates to refine models weakens the advantage and might increase the complexity dramatically. When I look at the whole C program of some nesC applications, I am worried if we can handle its complexity.

One way to reduce the complexity is preventing jumping into standard routines. In other words, we can use the *depth* of unwinding routines as the abstract parameter. However, we still unavoidably need to trace predicates to exclude infeasible paths.

Another way is discovering predicates efficiently. Henzinger et al. [13] used Craig interpolation [4] to construct, from a given error trace, the facts which need to be known at the cut-point of the trace in order to prove infeasibility. They have extended BLAST with predicates discovered by Craig interpolation, and applied it to C programs with more than 130,000 lines of code.

We also propose another abstract-refine way in discussion section.

6 Multi-threaded Analysis

6.1 Informal Semantics

We informally describe the semantics of our multi-thread system. For simplicity, assume all threads running the same program. A thread is designed to run to complete, but may be preempted due to system interrupts. When a thread is in some atomic state, it will not be preempted till it leaves to states not atomic. During the computation, a thread may invoke new threads. Hence, there may be an arbitrary number of threads.

6.2 Thread Module Reasoning

Here we sketch the idea of thread module reasoning. the formal model could be found in [**?**]. Consider a 2-threaded program $\Pi = (Q, Q_0, T_1, T_2, E)$. The program Π is safe if there is no $(T_1||T_2)$-trace that ends in an error state.

- The program Π is safe iff $R(T_1||T_2) \cap E = \emptyset$
- The program Π is safe in a *strongly thread module way* iff $R(T_1||\sigma[T_2]) \cap R(T_2||\sigma[T_1]) \cap E = \emptyset$.
- The program Π is safe in a *thread module way* iff there are enviromental assumptions G_1 and G_2 such that $G_1 \supseteq \rho_1[T_1, G_2], G_2 \supseteq \rho_2[T_2, G_1], R(T_1||G_2) \cap R(T_2||G_1) \cap E = \emptyset$.

If a program is safe in a thread-module way, then it can be proved without considering the product transition relation $T_1||T_2$; it suffices to reason about thread 1 together with an environment assumption about thread 2 that concerns only the shared state nd vice versa.

The main idea of thread modularity is that the state space of *one* thread is explored at a time, making assumption about how the environment can interfere. Henzinger et al. [?] extend thread modular reasoning to abstract multi-threaded programs. The system contains a main thread(system) and a context(environment) which is an abstraction of all the other threads. Then they verify that 1) this composed system is safe("assume") and 2) the context is a sound abstraction ("guarantee"). We adopt their model to analyze multi-threaded programs.

6.3 Coarse Abstraction

A multi-threaded program is a set of threads where each thread is represented by a CFA. Instead of tracking the control location of each thread separately, Henzinger et al. proposed the context automata in which they count the number of threads at each control location.

For our power analysis, the main thread can be constructed as a CFA in a similar way as in previous section. (Note that If we do not consider branch conditions, then there is no interaction between other threads)

The context thread could be constructed from the main thread by 1) discarding power information, 2) considering only predicates involved global variables (overapproximation), and 3) optimizing the automata, e.g., remove null transitions and unreachable states.

6.4 Check-Refine Paradigm

Again, we iteratively add predicates to refine the main thread and derive the context from the refined main automata.

6.5 Correctness

We here need to prove our abstraction follows the assume -guarantee reasoning. Then the correctness follows [13].

7 Implementation

Basically I am thinking using Avrora [24] to construct the power model, and then modify modern model checkers, e.g., Red or BLAST, to take the rest.

8 Application

8.1 Verify energy-aware algorithms

In a sensor network, the data being sensed must be transmitted to base station so that the end-user can access the data. To conserve power, the data is usually being relayed multiple times towards destination. Coleri and Ergen [5] performed the power analysis of one node as a function of the distance from the base station. They first estimated energy consumptions of the instruction set in tinyOS, and then modelled each component as a Linear Hybrid Automaton (LHA), such that each component records its power consumption during the computation. The distance is served as a parameter of LHA to adjust the marginal rate of cost. The assumption behind Coleri's model is that the nodes closer to the base station will relay more packets compared to the far away one. While they assume the flow of a node has an opposite relation to the distance between the node and base station, in real sensor networks, the flow usually depends on its routing algorithm. In particular, researchers have proposed many routing algorithms to minimize the power consumption of either the entire system or each node, in a sense to extend the lifetime as long as possible.

Considering these energy-aware algorithms, we can extend our work to perform the power analysis of one node as a function of its flow by adjusting the environment automata. Roozbeh et al. [15] proposed an ϵ-optimal multi-hop routing technique such that given the network topology with a set of sources and destinations, the optimal flow, with respect to load balance among nodes, can be calculated in a centralized manner. They reduce the problem to the min-cost problem, and hence achieve polynomial time complexity by solving linear programming.

Incorporating Roozbeh's work, we can serve the optimal flow as the input configuration for each node, and verify whether a given hardware configuration is sufficient to match the workload. In other words, we aim to check whether all nodes have sufficient resources to support the calculated optimal flow. If the answer is no, a counter example is generated to depict how a node fails even for this optimal balanced flow, i.e., the possible best load distribution/the best lifetime of a node. On the other hand, since all

nodes are identical, the bottleneck of the flow can be identified easily. Instead of checking all nodes, we may assert that once the node with the heaviest load can take the work, the system will not crash.

We summarize our ideas as below.

- Given a directed graph $G = (V, E)$,$S = (s, q, t)|s \in V, q \in N$, we first calculate the ϵ-optimal flow f.
- Let $f_M = \max\{f(i)|i \in V\}$ denote the max flow into a node. Assume each node is implemented by a C program P. We then generate the thread-context automata A which could be verified against the risk property, "the node is out of charge before finishing the workload."

9 Discussion

Our main contribution is applying verification techniques to power analysis, such that we can handle complex concurrent behaviors. The hurdle relies on how to apply thread-context reasoning and abstract-check-refine paradigm to achieve precise power analysis of multi-thread programs.

When we look at C codes, once we find a counter-example but is infeasible in original codes, we refine our abstract model by adding extra predicates such that we may exclude the infeasible path in the refined model after tracking their values. As I mentioned, we might be struggling in the refinement cycle , especially when we look at a detailed C program, e.g., the one generated by nesC compiler. We might relieve the pain by further abstracting the program, e.g., enforcing the unwind depth of routines. But it's not clear what this means for the original program.

Another way is using the inherited hierarchical structure of nesC applications. This poses an attractive abstract-check-refine paradigm. The abstraction of thread context(environment) can be done by modelling the interface despite the inner behaviors. The refinement can be done by exploring one/more inner level when the counter example is infeasible.

Looking at nesC codes also follows some advantages: 1) the patent of interactions between components is clear, 2) applying counter-example guided refinement by hierarchical structure is new, and 3) applying thread-context reasoning to sensor network systems is also new.

When starting at nesC codes, the analysis can be done: 1) generate energy profiles of predefined components (Again, we can use emulators) 2) parse nesC codes 3) generate the main automata (target, detailed to power model) 4) generate the context automata (environment, start at the highest hierarchy, allow all possible results of inner behaviors, ignore power info, record the number of threads at each location) 5) verify the property. If it is safe, then we are done. O.w., a counter-example is generated. 6) simulate the path in real programs. If the counter example is feasible, we are done . O.w., refine the context. repeat 5,6 till the context overapproximate environment.

10 Conclusion

We propose a framework to proceed power analysis of multi-threaded programs, in which we extend simulation-based power analysis to verification-based power analysis. While considering intractable in-

terleaving behaviors, our model poses an attractive way to analyze complex concurrent systems.

References

[1] R. Alur, T. A. Henzinger, P.-H. Ho, Automatic Symbolic Verification of Embedded Systems. In Proc. of Real Time Systems Symposium. IEEE Computer Society Press, 1993.

[2] R. Alur, T. A. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In Proc. of the Tenth Int. Conference on Computer Aided Verification, LNCS 1427, pages 521-525. Springer-Verlag, 1998.

[3] M. Broy, Modular hierarchies of models for embedded systems, Formal methods and models for system design: a system level perspective, pages 3-32, 2004.

[4] W. Craig, Linear Reasoning, J. Symbolic Logic, vol 22. pp. 250-268, 1957.

[5] S. Coleri, M. Ergen, Verification and Power Analysis of an Event-Based System (TinyOS) and Sensor Network with Hybrid Automata. SCI Orlando, July, 2002.

[6] Graphviz - Graph Visualization Software, http://www.graphviz.org/

[7] E. A. Emerson and A. P. Sistla, Symmetry and model checking, In Proc. of the International Conference on Computer Aided Verification (CAV93), LNCS 697, pp. 463-478, Elounda, Greece, 1993.

[8] D. Gay, P. Levis, R. Behren, M. Welsh, E. Brewer, and D. Culler, The nesC Language: A Holistic Approach to Networked Embedded Systems. In Proc. of Programming Language Design and Implementation (PLDI) 2003, June 2003.

[9] W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan, Energy-Efficient Communication Protocol for Wireless Microsensor Networks, In Proc. of the 33rd Annual Hawaii International Conference on System Sciences(HICSS), pp. 3005-3014, Jan. 2000

[10] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi, HyTech: A Model Checker for Hybrid Systems, Software Tools for Technology Transfer, vol 1, pp. 110-122, 1997.

[11] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In Proceedings of the International Conference on Programming Language Design and Implementation (PLDI), ACM Press, 2004.

[12] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy Abstraction. Proceedings of the 29th Annual Symposium on Principles of Programming Languages, ACM Press, pp. 58-70, 2002.

[13] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions frm Proofs. Proceedings of the 29th Annual Symposium on Principles of Programming Languages, ACM Press, pp. 232-244, 2004.

[14] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular Abstraction Refinement. In Proceedings of the 15th International Conference on Computer-Aided Verification (CAV), LNCS 2725, Springer-Verlag, pages 262-274, 2003.

[15] Roozbeh Jafari, Foad Dabiri, Majid Sarrafzadeh, ϵ-Optimal Minimal-Skew Battery Lifetime Routing in Distributed Embedded Systems, In Proc. of the Journal of Low Power Electronics (JOLPE), vol 1, no. 2, pp 97-107, September 2005.

[16] R. P. Kurshan, V. Levin, M. Minea, D. Peled and H. Yenigun, Static Partial Order Reduction, In Proc. of Tools and Algorithms for Construction and Analysis of Syatems(TACAS), LNCS 1384, pp. 345-357, Lisbon, 1998.

[17] O. Landsiedel, K. Wehrle, Aeon: Accurate Prediction of Power Consumption in Sensor Networks In Proceedings of The Second IEEE Workshop on Embedded Networked Sensors (EmNetS-II).

[18] Kedar Namjoshi, Lifting Temporal Proofs Through Abstractions, In Proc. of the Fourth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI03), New York, January 9-11, 2003.

[19] G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In Proc. CC, LNCS 2304, pages 213-228. 2002.

[20] Jonathan Polley, Dionysys Blazakis, Jonathan McGee, Dan Rusk, John S. Baras, and Manish Karir, ATEMU: A Fine-grained Sensor Network Simulator. In Proceedings of SECON'04, The First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, 2004.

[21] C. Praun and T. R. Gross, Static Conflict Analysis for Multi-Threaded Object-Oriented Programs, In Proceeding of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp.115-128, 2003.

[22] A. Perrig, R. Szewczyk, V. Wen, D. Culler, J. D. Tygar, SPINS:Security Protocols for Sensor Networks, In Proc. of Mobile Computing and Networking, Italy, 2001.

[23] V. Shnayder, M. Hempstead, B. rong Chen, G. W. Allen, and M. Welsh, "Simulating the power consumption of large-scale sensor network applications," in SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems. New York, NY, USA: ACM Press, pp. 188-200, 2004.

[24] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: Scalable Sensor Network Simulation with Precise Timing. In Proceedings of the Fourth International Conference on Information Processing in Sensor Networks, April 2005.

[25] P. Volgyesi a, M. Maroti b, S. Dora b, E. Osses b and A. Ledeczi Software Composition and Verification for Sensor Networks. Science of Computer Programming, Vol. 56 , Issue 1-2, pages 191 -210, April 2005.