



Declarative Power Sequencing

Conference Paper**Author(s):**

Schult, Jasmin; [Schwyn, Daniel](#) ; [Giardino, Michael Joseph](#) ; Cock, David; Achermann, Reto; Roscoe, Timothy

Publication date:

2021-10

Permanent link:

<https://doi.org/10.3929/ethz-b-000508223>

Rights / license:

[In Copyright - Non-Commercial Use Permitted](#)

Originally published in:

ACM Transactions on Embedded Computing Systems 20(5), <https://doi.org/10.1145/3477039>

Declarative Power Sequencing

JASMIN SCHULT, DANIEL SCHWYN, MICHAEL GIARDINO, and DAVID COCK, ETH Zurich, Switzerland

RETO ACHERMANN, University of British Columbia, Canada

TIMOTHY ROSCOE, ETH Zurich, Switzerland

Modern computer server systems are increasingly managed at a low level by baseboard management controllers (BMCs). BMCs are processors with access to the most critical parts of the platform, below the level of OS or hypervisor, including control over power delivery to every system component. Buggy or poorly designed BMC software not only poses a security threat to a machine, it can permanently render the hardware inoperative. Despite this, there is little published work on how to rigorously engineer the power management functionality of BMCs so as to prevent this happening.

This paper takes a first step toward putting BMC software on a sound footing by specifying the hardware environment and the constraints necessary for safe and correct operation. This is best accomplished through automation: correct-by-construction power control sequences can be efficiently generated from a simple, trustworthy model of the platform's power tree that incorporates the sequencing requirements and safe voltage ranges of all components.

We present both a modeling language for complex power-delivery networks and a tool to automatically generate safe, efficient power sequences for complex modern platforms. This not only increases the trustworthiness of a hitherto opaque yet critical element of platform firmware: regulator and chip power models are significantly simpler to produce than hand-written power sequences. This, combined with model reuse for common components, reduces both time and cost associated with platform bring-up for new hardware.

We evaluate our tool using a new high-performance 2-socket server platform with >100W per socket TDP, tight voltage limits and 25 distinct power regulators needing configuration, showing both fast (<10s) tool runtime, and correct power sequencing of a live system.

CCS Concepts: • **Computer systems organization** → **Firmware; Reliability; Hardware** → *Power networks*; • **Software and its engineering** → *Specification languages*.

Additional Key Words and Phrases: power sequencing, board management controller, reliable firmware, declarative specification

ACM Reference Format:

Jasmin Schult, Daniel Schwyn, Michael Giardino, David Cock, Reto Achermann, and Timothy Roscoe. 2021. Declarative Power Sequencing. *ACM Trans. Embedd. Comput. Syst.* 20, 5s, Article 84 (September 2021), 21 pages. <https://doi.org/10.1145/3477039>

1 INTRODUCTION

In this paper, we present a novel solution to a surprisingly subtle problem: how to turn on a computer.

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Embedded Software (EMSOFT), 2021.

Authors' addresses: Jasmin Schult, schultj@student.ethz.ch; Daniel Schwyn, daniel.schwyn@inf.ethz.ch; Michael Giardino, michael.giardino@inf.ethz.ch; David Cock, david.cock@inf.ethz.ch, ETH Zurich, Department of Computer Science, Zurich, Switzerland; Reto Achermann, achreto@cs.ubc.ca, University of British Columbia, Department of Computer Science, Vancouver, Canada; Timothy Roscoe, troscoe@inf.ethz.ch, ETH Zurich, Department of Computer Science, Zurich, Switzerland.

© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *ACM Transactions on Embedded Computing Systems*, <https://doi.org/10.1145/3477039>.

It goes without saying that modern computer server systems are enormously complex. However, less well-known is the software complexity that is never visible to the operating system or firmware running on the CPU. Almost all modern servers today include hidden processors, known as baseboard management controllers (BMCs), whose purpose is to control power and clock distribution to every major component on the main server board. These processors are often powerful enough to run a full operating system themselves, such as Linux or Minix [44].

Almost no attention has been paid in the research community to rigorously engineering the software for BMCs despite the fact that a BMC has almost complete control over the server, is usually connected to the network, and runs completely independently of any OS, hypervisor, or firmware on the main CPU. While some open-source implementations of BMC software exist (such as OpenBMC [13]), for the most part development and deployment of this software is invisible to users, developers, and the research community.

This raises a number of serious issues. First, incorrect behavior of the BMC in controlling voltages on the board can render the hardware permanently unusable, essentially destroying the board. This, in turn, means that developing and validating such software for a new board is a cautious, time-consuming process.

Secondly, security holes in BMC software can be catastrophic [27, 41]. Since the BMC is independent of the main processor cores, and is connected to every major component via, for example, an I²C network, a compromise of this code can result in almost undetectable threats to the system.

We encountered the first problem ourselves in the process of bringing up BMC software for a new, large, server-class board intended for research, and the work in this paper was driven by that experience as well as numerous 3rd-party accounts of BMC-related problems.

Our aim is to create a rigorous foundation for engineering board control software that gives OS and firmware developers assurance that the underlying hardware can be trusted to behave as advertised. This is an ambitious goal, but the first step is to define the BMC’s hardware environment and the constraints that capture the machine’s “safe operation”.

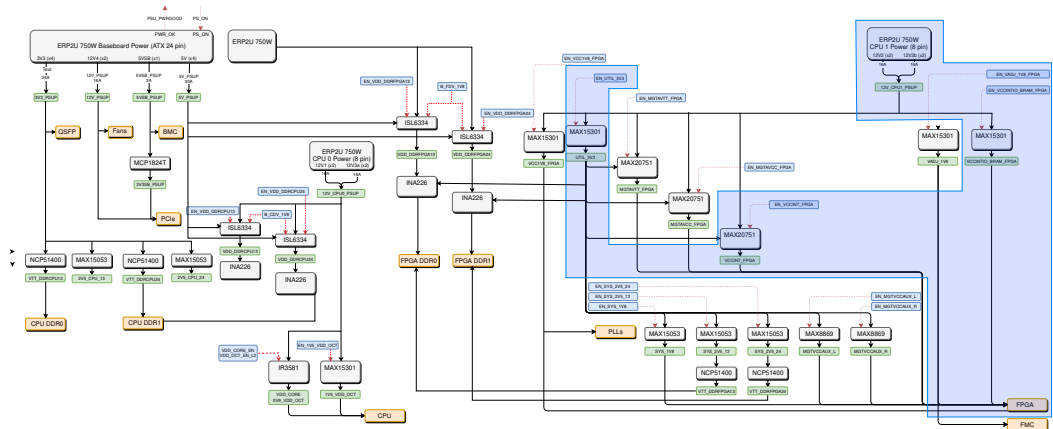


Fig. 1. The power tree of a modern two socket server system contains dozens of regulators, illustrated here with the one for Enzian. Individual regulators are shown in white boxes and their control signals are in blue with red dotted lines. The black lines connecting boxes are supplied voltages and the orange boxes are end consumers of one or more power sources. The blue highlighted region is used as an example for the remainder of this paper.

Our focus here is not on security but on complexity and correctness. We start by modeling the power trees of server machines and using these models to synthesize power-on sequences for the server given a specification of the desired powered-up state of the system.

The state of the art approach, as illustrated by publicly-available systems such as OpenBMC, is hand-written sequences derived from schematics and datasheets. In this paper we improve on this in the following ways:

- Identifying that platform description is a problem of *declarative specification*, and can (and should) be separate to the *mechanism* of generating imperative sequences.
- Recognizing that the important features of power tree nodes (e.g. regulators) are consistent across a wide range of components and topologies, allowing any system to be specified with a small set of *universal primitives*, expressed as a declarative specification language.
- Illustrating that the sequence generation mechanism maps well onto a well-known and widely-studied technique (constraint satisfaction problem (CSP)), with existing mature tools.
- Demonstrating that this implementation scales efficiently to the *largest, most complex* system we could find, and works on *real production hardware*.

The benefits of the approach we describe are allowing the engineer to separate “*what their platform looks like*” from “*how it is controlled*” allowing firstly a division of labor, and more significantly the opportunity to delegate the mechanical, time-consuming, and error-prone second problem to a mature, well-studied, and high-performance algorithm. This promises to reduce not only the time needed to derive a sequence for new hardware, but the maintenance effort as hardware evolves. In the end, we get both the correct sequence and the reasons (often buried in datasheets) why the sequence is the way it is.

2 BACKGROUND

We start by surveying the power sequencing problem, and how it has become so complex.

In the past, it was enough to use careful circuit design to power on a system which needed only a single 5 V or 3.3 V power rail, supplied directly from an AT or ATX power supply. As processor frequencies increased, however, the supply voltage decreased: because the power consumed by a CPU can be approximated by $P_{CPU} \propto V^2 \cdot f$ where V is voltage and f is clock frequency, the increased frequency of processors necessitated a decrease in voltage in order to keep power budgets reasonable. To obtain these lower voltages, *step-down regulators* take an input voltage from a power supply, and step it down to a stable voltage appropriate for the consuming device.

As boards become more complex, this leads to the situation in modern computing platforms where power and clock management is not a trivial matter, and the platform consists of numerous power and clock domains. For example, an AMD Zen processor requires three separate voltage domains (VDDCPU, VDDIO, and VDDSOC) [5], in some cases drawing hundreds of amps. Each channel of DDR4 memory requires two different voltages (1.2 V V_{DD} and 2.5 V V_{PP}) [24]. This means that for a two-socket system, each with two channels of DDR DRAM, a minimum of 14 different voltage regulators are needed to supply the necessary voltages, not to mention the needs of storage, networking, and accelerators. These devices each rely on a tree of regulators, each supplying voltage within a safe range, often needing to be supplied and activated in a specific order during the bring-up process. An example of a modern power tree is shown as Figure 1.

As most voltage regulators are able to accept ranges of input voltages and, depending on configuration, produce a range of output voltages, getting a correct configuration is critical. Misconfiguration of voltage levels is an obvious source of failure, but there are additional failure modes associated with power electronics. Latch-up is one such failure mode that can occur due to incorrect power sequencing, when the input to a CMOS device is greater than VDD, causing a low impedance

path that can cause the circuit to malfunction or be destroyed completely [23]. As the number of transistors increases and their size decreases the danger of latch-up and other sequencing failures increases [12]. Borderline power and clock levels can be exploited as security vulnerabilities [28, 39].

In order to deal with these more complex requirements, several solutions have been developed, beginning with specialized hardware for power sequencing [7, 18]. These integrated circuits, once configured, are reliable and inexpensive but still require configuration with the correct sequence when the platform is manufactured. Moreover, system designers, especially with an eye on the data center, demand more complete and configurable board management than simple power sequencing, such as monitoring and remote management, which is beyond the capabilities of hardware regulator controllers.

This, coupled with the need to orchestrate increasing numbers of regulators via software over networks such as low-speed serial buses (e.g. Inter-Integrated Circuit (I²C)) leads ultimately to modern servers using a BMC as a complete platform management system. The BMC includes NIC drivers and network protocols in addition to controlling power regulators, and typically runs a complete OS.

Narayanan *et al.* argue that the resulting inherent complexity of the firmware code routinely introduces bugs and vulnerabilities [26]. The general state of firmware running on BMCs, as much as is public, is at odds with the high level of privilege with which it executes [4]. This includes the critical parts dedicated to power and clock management. In 2018, the National Institute of Standards and Technology (NIST) of the United States released guidelines to improve the protection as well as the detection and subsequent recovery of platform firmware from malicious attacks [29].

To our knowledge, the current published state-of-the-art in industry does not go beyond manually coded point solutions and no attempts at auto-generating firmware, let alone formal verification, have been made. Indeed, until a few years ago, BMC firmware was proprietary and not publicly available at all [14].

Recent projects such as OpenBMC and u-bmc [42] have disclosed some implementations to the public, with the aim of providing more transparency and in the hopes of collectively finding and fixing bugs and vulnerabilities [15].

OpenBMC was the first major attempt at open-source BMC software, developed by Facebook engineers [13]. It has since become a Linux Foundation Project with founding members Microsoft, Intel, IBM, Google and Facebook, and is now part of the IBM OpenPower stack [30].

While openness helps with transparency and development, it does not eliminate vulnerabilities, and there have been significant critical flaws discovered in recent years [8–10]. Some of these vulnerabilities are related to OpenBMC itself, which is implemented by a large collection of Python and shell scripts running over the Linux D-Bus communications framework. While these underlying technologies are not insecure *per se*, the system as constructed does not provide high assurance of correctness or security.

Other recent vulnerabilities are related to weaknesses in the (mostly proprietary) Intelligent Platform Management Interface (IPMI) systems that BMCs rely on. uBMC attempts to address the inherent weaknesses of IPMI by replacing the weak security of IPMI with Google's gRPC [42]. The development of open standards for both BMC hardware and software indicates the need among system designers to coalesce around a secure, open, and well-understood system for platform management.

While fixing security vulnerabilities in BMC distributions is important, it does not help ensure the firmware's correct functionality Narayanan *et al.* propose a verified minimal OS *RedLeaf* that is "aimed at the needs of a diverse family of firmware subsystems" [26].

In this paper, we argue that such a verified software foundation like RedLeaf or seL4 [21] is necessary but not sufficient. To have confidence in the correctness of BMC software, we need a framework for reasoning in software about the behavior of the power network and its regulators.

3 EXPERIENCE

Our interest in this challenge is more than purely academic. Our work (and awareness of the problem) is motivated by our concrete experience building Enzian, a heterogeneous 2-socket server system designed for systems software research [2, 40]. Enzian is an eATX-format 22-layer board which has a per-socket thermal design power (TDP) of more than 100W, and a total system TDP of around 600W. The power tree consists of 25 discrete voltage regulators and more than 30 separate power rails with complex sequencing requirements. This complexity is increased by the heterogeneity of the system's main components leading to very different power sequencing constraints: One socket is populated with a 48-core Marvell Cavium ThunderX-1 server System-on-chip (SoC) the other with a Xilinx Ultrascale+ XCVU9P FPGA, each with 4 channels of DRAM.

Designing and building a system as large as Enzian has exposed us to a number of problems not apparent in the existing public literature concerning smaller, simpler systems, including the importance and difficulty of correct power sequencing in a modern computer.

Much early ad-hoc work led us to appreciate the importance of separating concerns: dividing the task of correct *sequence generation* from both the specification of *platform parameters*, and the labor-intensive task of faithfully modeling *regulator behavior*, including quirks.

Even though we (like most other groups who face this problem) are only working on a single design, both the design and our understanding of it have evolved over time. This persuaded us that a more generalizable approach was, indeed, worth the extra effort for us.

Platform parameters consist of minimum and maximum voltages on nets, power topology information, and the like. Some of these parameters we could control, but others were liable to change at short notice. For example, in our case an error in board layout led to some regulators being replaced with a different model partway between early prototypes and final hardware, requiring an updated power tree. Also, as we evaluated the prototypes, and the tightness of regulation (e.g. $I \cdot R$ losses) and transient behavior of the power nets, we gradually tightened (or occasionally loosened) the upper and lower voltage and current limits for specific nets relative to the datasheet values, again requiring updated parameters.

Between our first design and working hardware, the power sequences needed to be redesigned and/or adapted repeatedly, and doing this manually required a lot of time spent tediously changing one voltage after another and examining the resulting current levels in the system.

The modeling of regulators, particularly those for the critical processor core voltage supplies (our main chips draw more than 100 Amps at less than 1 Volt), was a very time-consuming process.

Device quirks and minor areas of non-compliance with the standard for Power Management Bus[38] (PMBus), the protocol stack used by the BMC to communicate with the regulators, can have dramatic consequences: In one case, due to an interaction between the CPU core regulator's PMBus interface and its more fine-grained proprietary control registers, it would default to an output of 1.2 V on enable, well outside the maximum rating of the CPU, even though it was notionally programmed to 0.9 V.

The result was a nail-biting moment when we realized that a component costing more than a thousand dollars was dissipating 100W more than its rated power. The solution here was to reprogram the output voltage *after* the regulator's logic supply was enabled, but *before* its output was enabled. This is now incorporated as a sequencing requirement in this device's model, guaranteeing that future generated sequences automatically incorporate this hard-won knowledge, which would not be explicit in a hard-coded power-up sequence.

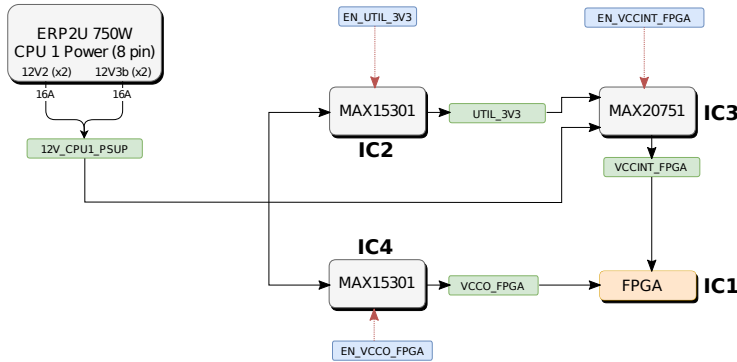


Fig. 2. Detail view of power tree (highlighted in blue in Figure 1) that illustrates sequencing requirements and high currents: IC1 (FPGA), IC2 (MAX15301) that supplies UTIL_3V3, IC3 (MAX20751) which supplies VCCINT_FPGA, IC4 (MAX15301) that supplies VCCO_FPGA, and the 12V_CPU1_PSUP rail. The regulators are controlled by enable signals: IC2 is enabled by EN_UTIL_3V3, IC3 is enabled by EN_VCCINT_FPGA, and IC4 is enabled by EN_VCCO_FPGA.

The large investment we had to make in faithfully modeling the behavior of these components is preserved, and applied automatically to any updated sequence for this board, or indeed wherever the same components are used in future designs.

Enzian is designed with redundant configuration mechanisms: While the current firmware sequences its power tree in software over PMBus, the hardware itself also supports a traditional ‘hardwired’ power sequence where the enable input of a regulator is driven (via a complex programmable logic device (CPLD)) by some logical combination of control signals and the ‘power-good’ signals of other regulators.

However, whether the sequence is programmed in software, or wired-in directly, exactly the same problems exist, namely: what should the sequence be, and how do we know that it is safe?

4 MODEL

In this section we develop our model of a power tree, with reference to the full power tree of Enzian in Figure 1. Figure 2 shows a representative section of this full tree, and will serve as our running example. We consider the power tree as a directed graph, with two types of nodes: components (IC1–4) and nets (12V_CPU1_PSUP, UTIL_3V3, VCCINT_FPGA, VCCO_FPGA, EN_UTIL_3V3, EN_VCCINT_FPGA, EN_VCCO_FPGA). A directed edge exists from a net to the input of either a regulator or load component (e.g. CPU). Likewise, an edge exists from a regulator output to the net it supplies. Each net may only be driven by a single output.

While there are a huge variety of regulators, load devices and system designs, the basic electrical laws, together with practical considerations, mean that at the level of detail we need all regulators and devices are essentially equivalent: Regulators convert a small number of input voltages (power and logic) into a small number of outputs, and with only limited exceptions (e.g. USB OTG charging), power only ever flows from source to sink. Devices (e.g. CPUs) accept some range of voltages, and require some ordering between the rails

Except for systems with rechargeable batteries, which are beyond the scope of the current work, the power rails thus always form a directed acyclic graph. In all systems of which we are aware, regulators can be characterized by a range of permissible input and output voltages and currents,

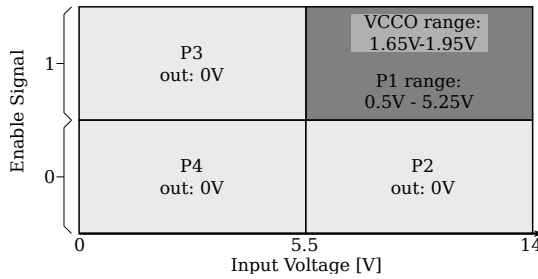


Fig. 3. IC output voltage range as a function of inputs, for LVC MOS18 IO.

and load devices by their allowed supply voltages and a partial order between them that power-up must respect. This model is applicable to a large range of systems, from embedded devices with only a handful of regulators, up to the large server-class system on which we evaluate our approach in Section 6.

The state of a net is its current nominal voltage (the model only considers its DC value and ignores $I \cdot R$ drops). The state of a regulator is the combined state of its inputs, both physical (supply voltage, on-chip enable pin) and logical (PMBus-commanded output voltage or enable signal).

The output of a regulator is a function of its inputs. Figure 3 illustrates this for regulator IC4 which supplies net VCCO_FPGA (the I/O pin supplies), whose value depends on the I/O standard in use, generally either 1.2 V (e.g. for DRAM), 1.8 V or 3.3 V (e.g. for legacy CMOS ICs). Here, the IO standard is LVC MOS18 which requires an I/O bank voltage between 1.65 V and 1.95 V. The inputs to this regulator are two-dimensional: supply voltage and enable signal. With enable deasserted, the regulator outputs 0 V. With enable asserted, it can generate any voltage between 0.5 V and 5.25 V. Thus, its output range (the set of outputs that we can instruct it to generate) is $[0V] \cup [0.5V, 5.25V]$. The valid ranges for the output driving a net and all inputs supplied by the net are intersected to compute the range of target voltages for that net, in this case $[1.65V, 1.95V]$. If this regulator were able to supply only up to, say, 1.90 V, the target interval would be reduced to $[1.65V, 1.90V]$.

From the target range for a net (the *static* constraint) and the state diagram for a regulator, we can infer *dynamic* constraints on its inputs which in turn become the target for regulators higher in the tree, possibly after intersection with the input requirements of other regulators/loads sharing the same net. These constraints are then iteratively filtered back toward the root nodes, such that every net has a target range. In this example, to produce an output in the 1.65 V–1.95 V range, the supply net for IC4 (12V_CPU1_PSUP) must be between 5.5 V and 14 V, and its enable signal must be asserted. In this instance the supply net’s requirement is satisfied by the EPS12V power supply’s guaranteed output range of $[11.4V, 12.6V]$.

Looking again at Figure 2, IC3 introduces a different type of constraint: ordering. This regulator has two supply inputs: one for the supply net to be regulated down, and the other for its internal logic. Until its internal logic is powered, it’s impossible to communicate with the regulator, and thus enable its output, even if the main supply is available. In this case, we’ve introduced a recursive element to sequence construction once this logic supply is produced by another regulator that we must configure. This case is easily handled by the model: this regulator’s equivalent of Figure 3 has an additional axis, corresponding to its logic supply. This in turn generates a dynamic constraint on the logic regulator (IC2) output, forcing us to enable it before attempting to enable IC3.

The final constraint imposed by the model is an ordering between the voltage rails for a given IC. While the supply nets for IC3 may be safely enabled in any order, this is not true for the CPU and the FPGA. As already mentioned, incorrect power sequencing can lead to latch-up and

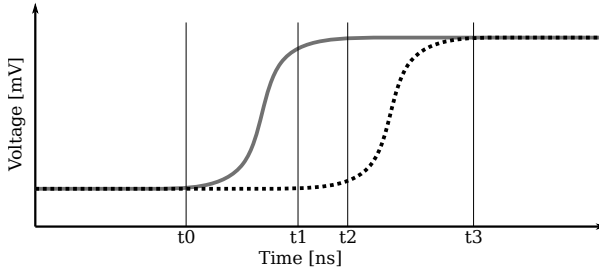


Fig. 4. Illustrative example of a manufacturer-supplied sequencing graph for an IC. The voltage signal represented by the dotted line must only be ramped up once the other signal has stabilized i.e. $t_2 > t_1$.

instantaneous destruction for ICs whose *normal* power consumption is in the hundreds of amps: The core regulators will happily supply 200 A into a short circuit.

The sequencing requirements for an IC are generally supplied by the manufacturer, either as a recommended/mandated power sequence or in a diagram such as Figure 4.

To model this we associate each state change of a net with an *initiate event* (e.g. enable signal asserted) which triggers the state change and a *complete event* (e.g. N consecutive voltage measurements within range after which the conductor has stabilized in the new state).

For the CPU’s main power supplies, the `VCCINT_FPGA` initiate event must happen after the `UTIL_3V3` complete event:

$$T(\text{Initiate}(\text{VCCINT_FPGA})) > T(\text{Complete}(\text{UTIL_3V3}))$$

Additionally, we have the natural condition that complete events always happen after the initiate event for the same net:

$$T(\text{Complete}(\text{net})) > T(\text{Initiate}(\text{net}))$$

This gives us a partial order on events.

The goal of a power sequence is to place leaf components — those with no outputs e.g. the CPU — in a specified power state. We term these leaf components *consumers*. To generate a valid power sequence we must:

- (1) Find a platform state that satisfies all consumer constraints and all other input constraints (static and dynamic).
- (2) Find an order of actions that transitions the platform into that state while observing the partial ordering of all relevant events.

5 ALGORITHMS

As discussed in Section 4 there are two correctness criteria for a power-up sequence that we tackle separately. First, we compute a valid platform state by casting the problem as a constraint satisfaction problem (CSP) and using a constraint solver. After that, we use the partial order on the events to derive a sequence that transitions the platform from its current state to the new state. If no such sequence exists, we compute a new solution for the platform state and try again.

5.1 Computing the platform state

Given a set of consumer constraints that describe the desired power states of the consumers (CPU and FPGA in the case of Enzian), we first need to compute a platform state that satisfies these constraints. This means we need to propagate these constraints back through the tree and for each output, select a state that satisfies the static and dynamic input constraints for the attached net.

A CSP is defined by a set of variables, a set of domains that define what values each variable can take, and a set of constraints that define relations between subsets of variables that any assignment for the variables must satisfy. We cast the problem of computing a valid platform state as a CSP as follows: The set of variables consists of a variable w_i , $1 \leq i \leq (\text{number of nets})$ for each net in the platform, that represents the state of the output connected to that net. The variables can take on integer values that represent the voltage of the output in millivolts (mV) or 0 and 1 in the case of logical signals. The set of constraints is composed of static constraints, dynamic constraints, and consumer constraints.

The static constraints ensure that all the maximum ratings for the connected inputs are observed, i.e., for each net we have a constraint

$$\bigwedge_{j=1}^{\text{number of inputs}} \text{low}(M_j) \leq w_i \leq \text{high}(M_j)$$

For each input j connected to net i this ensures that the net's state is within the range of the input's maximum rating, i.e. at least its minimum rating $\text{low}(M_j)$ and at most its maximum rating $\text{high}(M_j)$.

The dynamic constraints connect the components' outputs to their inputs: they ensure that for each output that gets configured into a specific state the inputs of the same component have the appropriate values. In the example in Figure 3 showing the possible states for IC2, this means the output can fall into one of four regions. Each one of these regions then in turn imposes dynamic constraints on the inputs if the output value falls within the region. In general for each component we add constraints of the following form:

$$\bigvee_{P_i \in \text{state regions}} \text{low}(P_i) \leq w_i \leq \text{high}(P_i) \wedge D_{P_i}$$

Each element in the disjunction represents the situation wherein the output connected to net w_i is in a particular state region P_i , i.e. between $\text{low}(P_i)$ and $\text{high}(P_i)$. The term D_{P_i} in each element represents the region-specific dynamic constraints on the inputs: they impose the requirements of the state region on the component's inputs and propagate them back up the tree. Each D_{P_i} is of the following form:

$$\bigwedge_{j=1}^{\text{number of inputs}} \text{low}(I_j)_{P_i} \leq w_j \leq \text{high}(I_j)_{P_i}$$

Every element of the conjunction takes care of propagating the dynamic requirements to one of the inputs I_j by ensuring that it will be configured inside its required bounds for state region P_i , i.e. between $\text{low}(I_j)_{P_i}$ and $\text{high}(I_j)_{P_i}$.

Finally, we add the consumer constraints which for each consumer constrain the values of the nets that they are connected to according to their desired power state:

$$\bigwedge_{j=1}^{\text{number of inputs}} \text{low}(I_j) \leq w_j \leq \text{high}(I_j)$$

Now that we have encoded the problem of finding a platform state that conforms with the consumer constraints for a specific power state into a CSP, we can use standard CSP solving techniques to obtain a state for every output. We now compute a set of actions that ensure every output is configured into this state and all sequencing requirements are observed.

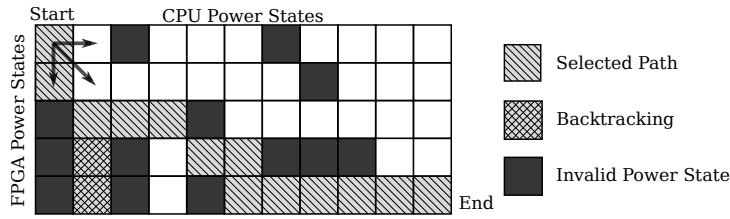


Fig. 5. Finding a path through the state table.

5.2 Computing the sequence

We have seen in Section 4 why it is important for a sequence that transitions the platform from a current state into a new state to observe the sequencing requirements. We have also seen that every change of output state is associated with an initiate and a complete event and that there is a partial ordering between those events. Every initiate event then translates to an action that triggers a state change and every complete event to a check that a state change has successfully completed, e.g. reading a voltage sensor. By topologically sorting the events we obtain a sequence that transitions the platform from the current state into the new state, observes the partial order between the events and therefore also conforms with the sequencing requirements of the platform.

5.3 Full power-up sequence

Given a power state for each consumer in the platform and corresponding consumer constraints we can now compute a platform state that satisfies these constraints. Given two of these states we can also compute a correct sequence that transitions the platform from one to the other. The primary chips on Enzian transition through multiple intermediate power states between being off and fully operational or vice versa. This is illustrated in Figure 5.

The columns are the CPU’s power states and the rows are the FPGA’s. Each tile of the grid then represents a potential platform state which satisfies the consumer constraints for the corresponding power state of both CPU and FPGA. The upper left corner is the state when both chips are off and in the lower right one both chips are fully operational. We can compute such a platform state using the approach in Section 5.1.

To transition the platform from powered off to fully operational now means finding a path through the grid. In each step we have the choice of only advancing in one of the sockets’ power state sequence or both at once. In the grid this corresponds to going right or down vs. diagonally right and down. Some combinations of consumer constraints might be mutually exclusive, i.e. there is no platform state that can satisfy both consumer constraints at once. These combinations are illustrated with a dark-colored square in the figure. This means we can also get stuck and need to backtrack to the last tile in the grid where we still had a choice and try a different path. Such a situation is illustrated in the figure towards the lower left with the cross-hatched tiles. If we end up with no choices left, it means that the state transition is infeasible. This can either be sign of a badly designed platform or a bug in the model.

Once we have found a path from the origin state to the desired state, in our example from “off” in the upper left to “fully operational” in the lower right, we can compute the partial sequences that transition the platform between tiles using the approach in Section 5.2.

We obtain the full sequence by concatenating all the partial sequences from the individual steps.

6 EVALUATION

We begin our evaluation by showing that using our model and a conventional constraint-satisfaction algorithm, we can indeed generate a working power sequence which successfully configures the voltage regulators of Enzian (Section 6.1). We then show that this is not only possible, but that the event sequence can be derived efficiently from the model with reasonable computational effort (Section 6.3 and Section 6.2). Finally, we provide a user-experience report about the efforts needed in using our tool to support a new hardware platform topology (Section 6.4) and adapting it to a new power management interface provided by the BMC (Section 6.5).

We built an implementation prototype in Python which is capable of deriving correct power sequences for Enzian. The tool converts the event graph produced by the constraint solver operating from the platform description into a sequence of power management API invocations on the BMC. We execute all performance experiments on a desktop machine with an Intel Core i7-6700K CPU @ 4 GHz and 32 GB DDR4 RAM. We then run the generated Python program on Enzian's real BMC to configure the power sequence and bring up the processors of the board.

6.1 Generating working power sequences

In this qualitative evaluation we demonstrate that our tool is able to generate a working power sequence capable of bringing up the Enzian platform.

We model the power tree of Enzian using the above-described semantics and use the tool to generate a power sequence. We then ensure that the platform is turned off, i.e. all voltage rails are off except for the stand-by power. Next, we execute the generated Python program containing the command sequence to power up Enzian. This transitions the platform from the off-state to the on-state where both sockets are fully powered on and operational. We then verify that the voltages are correctly set according to the specification.

First, we analyze the generated power sequence and the resulting Python program. The initial lines of the power sequence can be seen in Listing 1. Overall, the generated sequence consists of 28 discrete steps. These are implemented as several calls to the power management API that executes initiate and wait-for-completion events. This results in a Python program with a total of 74 lines for Enzian. The majority of these lines (61 in total) are directly executing power sequencing actions on the voltage regulators. The remaining 13 lines are initializing the power management framework of the BMC and creating required software objects.

When executing the generated Python program, we observe that the power rails of Enzian are configured correctly and the CPU and FPGA are brought into an operational state.

Comparing the generated power sequence to the manually derived one we observe three key differences:

Ordering The order of the executed steps differs between the manually written program and the generated one. This difference is due to the partial ordering of the sequence steps. Consequently, there are multiple correct sequences to bring up a platform. Our tool is even capable of generating multiple correct sequences by selecting a slightly different path through the power states resulting in a different power sequence. However, the end state is always the same and thus for the purpose of this work all sequences are equivalent.

Checks The manual sequence always inserts checks to verify that the complete events actually have happened before proceeding to the next step. The generated sequence will do multiple steps in parallel if the sequencing requirements allow it and only then insert checks to verify the steps have completed.

```

init_device('isl6334d_dds_v')
init_device('pac_cpu')
init_device('pac_fpga')
gpio.set_value('C_RESET_N', False)
gpio.set_value('C_PLL_DCOK', False)
gpio.set_value('B_PSUP_ON', True)
wait_for_voltage('3v3_psup', v_min=3.135, v_max=3.465,
    device='pac_cpu', monitor='VMON3_ATT')
wait_for_voltage('12v_cpu0_psup', v_min=4.702,
    v_max=5.197, device='pac_cpu', monitor='VMON1_ATT')
wait_for_voltage('12v_cpu1_psup', v_min=4.702,
    v_max=5.197, device='pac_fpga', monitor='VMON1_ATT')
wait_for_voltage('5v_psup', v_min=4.750, v_max=5.250,
    device='pac_fpga', monitor='VMON2_ATT')
wait_for_voltage('5v_psup', v_min=4.750, v_max=5.250,
    device='pac_cpu', monitor='VMON2_ATT')
init_device('clk_main')
init_device('clk_cpu')
init_device('ir3581')
init_device('ir3581_loop_vdd_core')
init_device('ir3581_loop_0v9_vdd_oct')
power.device_write('ir3581_loop_vdd_core',
    'VOUT_COMMAND', 0.96)

# more lines follow ...

```

Listing 1. First lines of the generated power sequence

Default States The manual sequence explicitly sets the voltage for every regulator. The generated sequence omits this if our tool could infer from the model that the regulator was already configured correctly at that stage in the sequence.

Based on the results obtained in this evaluation we can conclude that the tool is indeed capable of generating a working power sequence that correctly powers up Enzian.

6.2 Efficient state generation

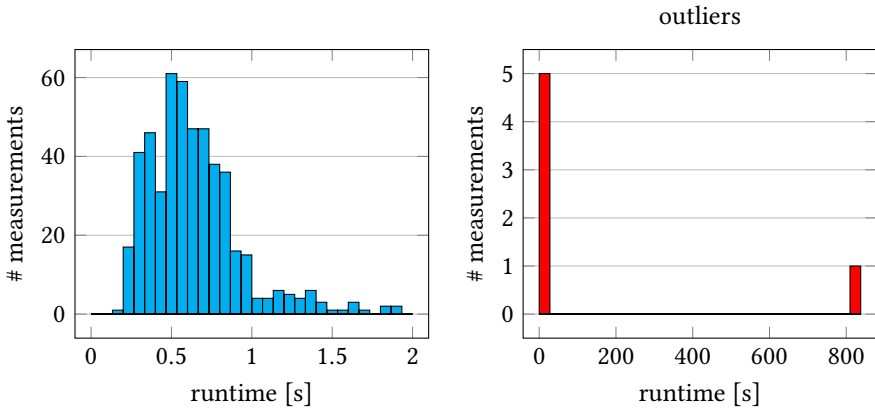
We now quantitatively evaluate the state generation process and show that computing a platform state satisfying a set of consumer demands is efficient and feasible within acceptable time limits.

We populate our model with the power tree description of Enzian. We then measure the time it takes to evaluate the model and to compute the new platform state for the three combinations of consumer demands listed in Table 1. This evaluates the algorithm of Section 5.1.

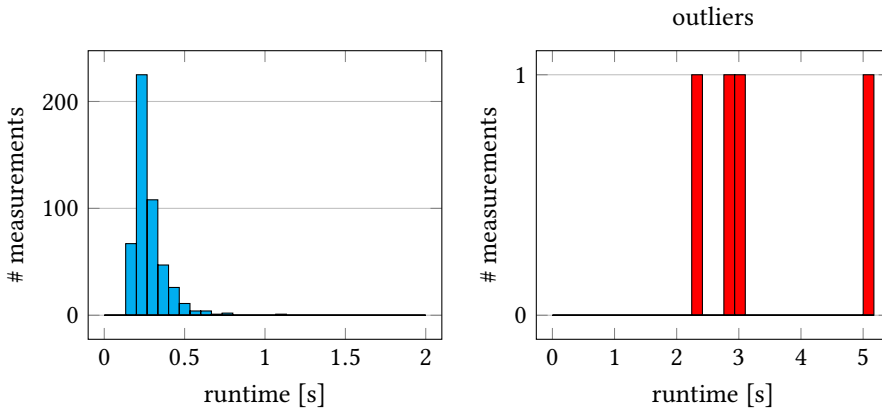
Problem	CPU power state	FPGA power state
P1	Powered on	Powered on
P2	Powered on	Powered off
P3	Powered off	Powered off

Table 1. An overview of problem instances P1 to P3

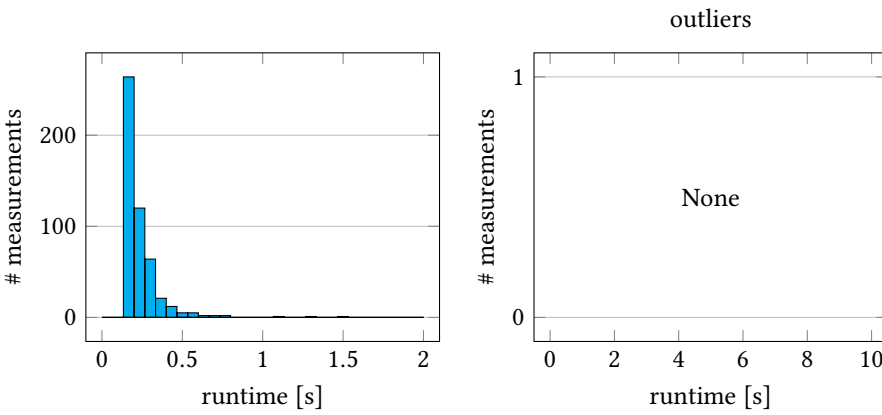
For each problem P1 to P3, we measure 500 runs of the experiment. Note that the constraint solver uses backtracking and thus may explore the search space in a different order each time, depending on the order in which the constraints are presented. To get a better representation of the expected runtime we randomize the order of constraints to compensate for better and worse paths through the search space.



(a) Solving times for problem P1



(b) Solving times for problem P2



(c) Solving times for problem P3

Fig. 6. Histograms of solving times (to find one solution) for the three problems P1, P2, and P3.

The results are shown in Figure 6. For better visibility of the data, we show two histograms: one with the regular measurements showing runtime on the x-axis and the number of runs on the y-axis and one with the outliers above two seconds execution time. For all problems, we observe that the majority of experimental runs completed in less than 1.0 second with just a few outliers.

The complexity of the problems decreases from P1 to P3 as fewer components need to be powered, thus reducing the total number of constraints in the system (P1 has both processors on, while P3 has both switched off). This is reflected in the results of P1 to P3, where the median execution time decreases with fewer powered-on components.

As mentioned in the experimental setup we randomize the order in which the constraints are presented to our solver. The outliers correspond to runs where the solver happened to explore the search space in a particularly inefficient way, such that it had to backtrack more often. We also see a larger number of outliers with more components turned on. This is due, in part, to the DRAM voltages being included in those configurations. While they are showing up as leaves in the power tree, exploring their state is mostly irrelevant to finding a power sequence for initializing the board. However, the general algorithm is not aware of this and can spend time exploring the DRAM voltage regulators resulting in runtime outliers shown in Figure 6. A possible solution to this would be to add additional constraints to avoid these types of situations, but we did not explore this option further as the number of outliers is very small.

We have shown that the evaluation of the power state space search algorithm from Section 5.1 is efficient, usually taking less than a second, and thus it is feasible to evaluate during runtime in response to user demands for re-configuring the power state of the platform.

6.3 Efficient Sequence Generation

We now quantitatively evaluate the time it takes to compute a complete boot sequence or a partial reconfiguration using the algorithm from Section 5.3. In other words, we show that it is possible to efficiently compute a sequence from one platform state to another.

We express the states of the main processors as either powered on or off which we call the initial state of the system. Then we toggle the power state of one or both of the chips to obtain the target state. Note that the underlying model captures all intermediate power states including the initial and target states. By enumerating all possibilities we obtain a total of 16 power states, four of which do not change the power state at all and are not of interest. From the remaining twelve states, we further eliminate the ones where the FPGA is powered without the CPU being powered. This was a constraint on Enzian at the time the experiment was performed. This leaves us with six total combinations of initial and target states shown in Table 2 to evaluate. For each configuration P1-P6 we measure the time to generate the transition sequence between the two states. We repeat each measurement three times.

We present the runtime measurements for all six configurations in Table 3. Overall we observe that in tendency the runtime grows with the number of transitions, and that ON transitions are more expensive than OFF transitions. Additionally, finding a transition sequence for the FPGA is more expensive than for the CPU. However, even in the worst case, the execution time is less than three seconds.

The dependence on the number of transitions is intuitive: When only one chip has to be transitioned, the state table illustrated in Figure 5 collapses to a single dimension and the problem is reduced to computing intermediate platform states. ON transitions are more expensive than OFF transitions as the components on Enzian have more ordering constraints when turning on than when turning off, hence it is more likely backtracking is required. Finally, the power sequence for the FPGA involves more components on the board and finding a correct sequence for it is therefore harder.

Problem	Consumer	Initial state	Target state
P1	CPU	Powered off	Powered on
	FPGA	Powered off	Powered on
P2	CPU	Powered off	Powered on
	FPGA	Powered off	Powered off
P3	CPU	Powered on	Powered on
	FPGA	Powered off	Powered on
P4	CPU	Powered on	Powered off
	FPGA	Powered off	Powered off
P5	CPU	Powered on	Powered on
	FPGA	Powered on	Powered off
P6	CPU	Powered on	Powered off
	FPGA	Powered on	Powered off

Table 2. Overview of problem instances P1 to P6

Problem	Measured runtime [s]	#Transitions
P1	2.7	2x ON
P2	1.3	1x ON
P3	1.7	1x ON
P4	0.4	1x OFF
P5	1.3	1x OFF
P6	1.5	2x OFF

Table 3. Measurements (average of three runs) obtained for the six different combinations of consumer transitions possible on the Enzian platform

With a measured worst case execution time of three seconds, pre-computing sequences offline is certainly feasible. Even online calculation at boot would be acceptable compared to other boot steps such as RAM initialization which can take a couple of minutes to complete.

In this evaluation we have shown that the entire power sequence of a platform can be generated within a few seconds and thus presents a viable option for both offline and online evaluation.

6.4 Re-computing sequences for new revisions

New board revisions or platforms have different power trees which must be modeled to generate a power sequence. We now elaborate on our experiences in expressing the Enzian platform using the modeling language.

There are essentially two steps involved: 1) obtaining the constraints of the different voltage regulators on the board, and 2) capturing the regulator topology in the power tree. We had to do both steps for both the manually-derived sequence and the model population.

For populating the model, we can independently focus on specifying power-tree topology and the voltage-regulator constraints. In contrast, when manually deriving the power sequence we had to pay attention to timing requirements and other constraints, as well as the power-tree topology. Adapting an existing platform to a new revision can be done by simply replacing the description of a voltage regulator with the new one in isolation without worrying about the effects it has on the power sequencing commands to bring up the board.

When using our tool we can express each regulator in isolation and form the topology step-by-step without worrying about timing and voltage constraints. We expect this approach to be less susceptible to errors than designing the sequence manually.

6.5 Adapting the tool

In this part of the evaluation we qualitatively evaluate the user experience, specifically the efforts needed to adapt our tool to the BMC-specific power management interface.

In its initial version, the tool was built against a different firmware image. When upgrading the firmware, the tool was no longer compatible with the Enzian BMC's firmware, and thus needed to be adapted to support this major change in the power management API. We adapted both the manually derived sequence and our tool to the new interface.

Adapting the sequence manually, required consulting various datasheets to obtain knowledge about the sequencing requirements of the various components and how they can be expressed using the API provided by the firmware. The previous sequence did not provide enough information to adapt it and required a significant amount of work reading the datasheets and carefully examining the schematics.

In contrast, we did not have to adapt the model itself for this change in the power management API because all knowledge about the power tree (with its components and constraints) were already encoded in the model. This completely avoids consulting the platform datasheets. All that is left to do is adapting the code generator of the tool to the new API.

Thus, adapting the code generator took roughly a single person-day, while understanding and adapting the entire power sequence manually to the new API consumed over three person-weeks. Our experience shows that supporting a new power management interface resulted in significantly less work than to manually deriving and adapting the bring-up sequences to the new firmware.

7 RELATED WORK

As we remarked in Section 2, there is a dearth of published work on board management software. Nevertheless, our work is closely related to other, neighboring fields which we discuss here.

The problem of deriving a correct power-up sequence bears some similarity with the problem of a device driver correctly initializing and controlling the operation of a device. Device drivers not only contribute a large amount of code to systems software [20], but are also a significant contributor to bugs and errors [6]. Dingo formalizes driver protocols to make the interaction with devices unambiguous [32].

Writing device drivers is inherently tied to the operating system architecture, but device driver synthesis [33, 34, 45] enables the generation of OS-specific device driver code based on a specification, and thus automatically generating the right control sequence for the device. Beyond drivers, there is early work on synthesizing most of the hardware-specific parts of an OS based on specifications [19]. Our work similarly applies program synthesis techniques to derive power sequences for the BMC.

Inside the OS (as opposed to BMC firmware), constraint solving has been applied to a variety of OS techniques both online and offline to select a whole-system configuration which satisfies current requirements.

For example, the problem of configuring PCI Express devices under a set of root complexes has been expressed in Prolog and solved using constrained logic programming techniques [35]. Similar methods have been applied to data center network configuration [25] or synthesizing cluster management code in distributed systems [37]. Spex [48] goes further by attempting to infer configuration constraints from the program source code, which is not possible in our case.

Cocoon [31] uses a hierarchical design process to specify the configuration of software defined networks to obtain a correct-by-construction initialization of the network controller.

Outside the field of OS design, software-based industrial control systems consist of a controller and a plant forming a closed loop system; software running on the controller must correctly configure the plant. QKS synthesizes correct-by-construction control software from the specification of a plant model, an implementation specification and the safety and liveness requirements [1, 22].

Closer to our goal of platform power management, for modern servers and phones it is usually the OS's responsibility to implement power management *policies* – deciding which components to power on or turn off is important to minimize the power requirements. Xu *et al.* argue for a centralized power management agent [46, 47] which decides when devices should switch between the discrete enabled or disabled states based on quality-of-service (QoS) requirements and specification of power states. QoS can also be used by agents in embedded systems to automatically find appropriate dynamic power states [16]. Benini *et al.* [3] provide a survey of design techniques for system-level dynamic power management.

In contrast our work is not trying to decide *when* devices should transition to different power states but provide help in *how* these transitions are implemented at a lower level.

We are also not the first ones to apply more formal techniques to power management: Gupta *et al.* applied formal methods to dynamic power management with the goal to minimize the overall power consumption, which as already stated above is policy that could be implemented using our mechanisms [17]. p-FSMs model system-level power management including control mechanisms and operating states [36]. Like us, the authors argue that the application of formal methods is essential to cope with the complexity of system-level power management in order to meet energy, power, and thermal constraints. They focus on the design of power management systems and it is not clear whether their model is able to handle individual regulators as is required for our work. While they model-check their representation of an SoC, they do not demonstrate controlling physical hardware with their technique.

Other approaches focus on providing an interface between the main OS and the BMC or other power sequencing functionality. The Advanced Configuration and Power Interface (ACPI) [43] defines mechanisms to control the power state of the entire system e.g. transitioning to sleep or waking up. Moreover, the ACPI tables include information about the power states of the motherboard devices and their connections, including methods to change the power state of the devices. Like the work mentioned above, ACPI operates at a higher level than our tool: it provides the OS on the CPU(s) of a platform with information about the power management capabilities of the platform but does not deal with how those capabilities are implemented.

Similarly, Devicetree [11] provides information about the hardware platform such as device addresses, amount of memory, processors, and existing power and clock domains to system software. The OS uses this information to find the device and its power and clock domains. However, information about power domains encoded in the devicetree does not include voltage levels or supported input/output voltages and is thus not suitable for our purpose.

8 CONCLUSION

In this paper, we have applied computer science techniques to a somewhat understudied problem in building and operating a computer: how to turn the machine on in a safe and efficient manner.

While not well-known in the systems community, the power sequencing problem is real and becoming more significant as systems become increasingly complex, and the consequences of getting it wrong become more serious (whether these consequences are security vulnerabilities or permanent damage to the hardware).

We have shown that generating a correct power-on sequence can be reduced to a constraint satisfaction problem, and that even a relatively unoptimized solver can compute a solution for a realistic, complex server in a relatively short amount of time – to the extent that it would be practical online at boot time.

However, this solution can only be achieved if the problem can be posed to the solver in a suitable manner. Consequently, we have presented a representation of a machine’s power tree that captures both the detailed topology of a modern server platform, and the behavior of individual power regulators and other components at a sufficient level of detail to generate useful results. We look forward to the release of more well-documented open hardware on which to evaluate our approach.

Even with a single system design though, our direct experience has been that this effort to create a more general solution has already paid off. We were prompted to explore it by the effort (and nerve) required to create a power-on sequence manually and interactively, and by the lack of any existing automated solutions to this problem, regardless of whether the resulting sequence was to be executed in software by a BMC or programmed into hardware in a CPLD. Having done the work to model hardware platforms, we are confident that applying it to another server design would be both valuable (in time saved) and low-effort.

Generating correct power sequences is only a first step to bringing rigorous engineering discipline to the problem of BMC firmware. We have laid the foundations to use our model for use-cases like online power and thermal management, which we are exploring in the context of Enzian.

All of our code, including details of the platform used, is available as open source¹.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their feedback and helpful comments. We would also like to acknowledge the generous support of Arm Ltd. and VMware for this work.

REFERENCES

- [1] Vadim Alimguzhin, Federico Mari, Igor Melatti, Ivano Salvo, and Enrico Tronci. 2012. On Model Based Synthesis of Embedded Control Software. In *Proceedings of the Tenth ACM International Conference on Embedded Software (Tampere, Finland) (EMSOFT '12)*. Association for Computing Machinery, New York, NY, USA, 227–236. <https://doi.org/10.1145/2380356.2380398>
- [2] Gustavo Alonso, Timothy Roscoe, David Cock, Mohsen Ewaida, Kaan Kara, Dario Korolija, David Sidler, and Zeke Wang. 2020. Tackling Hardware/Software co-design from a database perspective. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. CIDR, [www.cidrdb.org](http://cidrdb.org), 6 pages. <http://cidrdb.org/cidr2020/papers/p30-alonso-cidr20.pdf>
- [3] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. 2000. A Survey of Design Techniques for System-Level Dynamic Power Management. *IEEE Trans. Very Large Scale Integr. Syst.* 8, 3 (June 2000), 299–316. <https://doi.org/10.1109/92.845896>
- [4] Anthony Bonkoski, Russ Bielawski, and J. Alex Halderman. 2013. Illuminating the Security Issues Surrounding Lights-Out Server Management. In *7th USENIX Workshop on Offensive Technologies (WOOT 13)*. USENIX Association, Washington, D.C., 9 pages. <https://www.usenix.org/conference/woot13/workshop-program/presentation/bonkoski>
- [5] Thomas Burd, Noah Beck, Sean White, Milam Paraschou, Nathan Kalyanasundharam, Gregg Donley, Alan Smith, Larry Hewitt, and Samuel Naffziger. 2019. “Zeppelin”: An SoC for Multichip Architectures. *IEEE Journal of Solid-State Circuits* 54, 1 (2019), 133–143. <https://doi.org/10.1109/JSSC.2018.2873584>
- [6] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An Empirical Study of Operating Systems Errors. *SIGOPS Oper. Syst. Rev.* 35, 5 (Oct. 2001), 73–88. <https://doi.org/10.1145/502059.502042>
- [7] Altera Corporation. 2014. Enpirion Power Datasheet ES1020QI Power Sequencing Controller. https://eu.mouser.com/datasheet/2/612/es1020qi_10041-1299392.pdf
- [8] CVE. 2019. CVE-2019-4169. Available from MITRE, CVE-ID CVE-2019-4169. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-4169>

¹<https://github.com/Sockeye-Project/decl-power-req>

- [9] CVE. 2019. CVE-2019-4621. Available from MITRE, CVE-ID CVE-2019-4621. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-4621>
- [10] CVE. 2020. CVE-2020-14156. Available from MITRE, CVE-ID CVE-2020-14156. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-14156>
- [11] devicetree.org. 2020. Devicetree Specification, Release v0.3. <https://github.com/devicetree-org/devicetree-specification/releases/download/v0.3/devicetree-specification-changebars-v0.3.pdf>
- [12] Krzysztof Domanski. 2018. Latch-up in FinFET technologies. In *2018 IEEE International Reliability Physics Symposium (IRPS)* (Monterey, CA, USA). IEEE, Piscataway Township, NJ, USA, 2C.4–1–2C.4–5. <https://doi.org/10.1109/IRPS.2018.8353550>
- [13] Tian Fang. 2015. *Introducing “OpenBMC”: an open software framework for next-generation system management*. Facebook Engineering. <https://engineering.fb.com/open-source/introducing-openbmc-an-open-software-framework-for-next-generation-system-management/>
- [14] Jessie Frazelle. 2019. Open Source Firmware. *Commun. ACM* 62, 10 (Sept. 2019), 34–38. <https://doi.org/10.1145/3343042>
- [15] Jessie Frazelle. 2020. Opening up the Baseboard Management Controller. *Commun. ACM* 63, 2 (Jan. 2020), 38–40. <https://doi.org/10.1145/3369758>
- [16] Michael Giardino, Eric Klawitter, Bonnie Ferri, and Aldo Ferri. 2020. A Power- and Performance-Aware Software Framework for Control System Applications. *IEEE Trans. Comput.* 69, 10 (2020), 1544–1555. <https://doi.org/10.1109/TC.2020.2978468>
- [17] R.K. Gupta, S. Irani, and S.K. Shukla. 2003. Formal methods for Dynamic Power Management. In *ICCAD-2003. International Conference on Computer Aided Design (IEEE Cat. No.03CH37486)* (San Jose, CA, USA). IEEE, Piscataway Township, NJ, USA, 874–881. <https://doi.org/10.1109/ICCAD.2003.159778>
- [18] Jeff Heath and Akin Kestelli. 2005. *Flexible Power Supply Sequencing and Monitoring*. Analog Devices. <https://www.analog.com/en/technical-articles/flexible-power-supply-sequencing-monitoring.html>
- [19] Jingmei Hu, Eric Lu, David A. Holland, Ming Kawaguchi, Stephen Chong, and Margo I. Seltzer. 2019. Trials and Tribulations in Synthesizing Operating Systems. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems* (Huntsville, ON, Canada) (*PLOS’19*). Association for Computing Machinery, New York, NY, USA, 67–73. <https://doi.org/10.1145/3365137.3365401>
- [20] Asim Kadav and Michael M. Swift. 2012. Understanding Modern Device Drivers. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) (*ASPLOS XVII*). Association for Computing Machinery, New York, NY, USA, 87–98. <https://doi.org/10.1145/2150976.2150987>
- [21] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Trans. Comput. Syst.* 32, 1, Article 2 (Feb. 2014), 70 pages. <https://doi.org/10.1145/2560537>
- [22] Federico Mari, Igor Melatti, Ivano Salvo, and Enrico Tronci. 2014. Model-Based Synthesis of Control Software from System-Level Formal Specifications. *ACM Trans. Softw. Eng. Methodol.* 23, 1, Article 6 (Feb. 2014), 42 pages. <https://doi.org/10.1145/2559934>
- [23] W. Morris. 2003. Latchup in CMOS. In *2003 IEEE International Reliability Physics Symposium Proceedings, 2003. 41st Annual*. IEEE, Piscataway Township, NJ, USA, 76–84. <https://doi.org/10.1109/RELPHY.2003.1197724>
- [24] Janani Mukundan, Hillery Hunter, Kyu-hyoun Kim, Jeffrey Stuecheli, and José F. Martínez. 2013. Understanding and Mitigating Refresh Overheads in High-Density DDR4 DRAM Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) (*ISCA ’13*). Association for Computing Machinery, New York, NY, USA, 48–59. <https://doi.org/10.1145/2485922.2485927>
- [25] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. 2008. Declarative Infrastructure Configuration Synthesis and Debugging. *J. Netw. Syst. Manage.* 16, 3 (Sept. 2008), 235–258. <https://doi.org/10.1007/s10922-008-9108-y>
- [26] Vikram Narayanan, Marek S. Baranowski, Leonid Ryzhyk, Zvonimir Rakamarić, and Anton Burtsev. 2019. RedLeaf: Towards An Operating System for Safe and Verified Firmware. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) (*HotOS ’19*). Association for Computing Machinery, New York, NY, USA, 37–44. <https://doi.org/10.1145/3317550.3321449>
- [27] Charlie Osborne. 2019. *OpenBMC caught with ‘pantsdown’ over new security flaw*. <https://www.zdnet.com/article/bmc-caught-with-pantsdown-over-new-batch-of-security-flaws/>
- [28] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. 2019. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-Core Frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) (*CCS ’19*). Association for Computing Machinery, New York, NY, USA, 195–209. <https://doi.org/10.1145/3319535.3354201>
- [29] Andrew Regenscheid. 2018. *Platform Firmware Resiliency Guidelines*. Technical Report Special Publication (NIST SP) - 800-193. National Institute of Standards and Technology, Gaithersburg, MD, USA. <https://doi.org/10.6028/NIST.SP.800-193>

193

- [30] Todd Rosedahl, Martha Broyles, Charles Lefurgy, Bjorn Christensen, and Wu Feng. 2017. Power/Performance Controlling Techniques in OpenPOWER. In *High Performance Computing*, Julian M. Kunkel, Rio Yokota, Michela Taufer, and John Shalf (Eds.). Springer International Publishing, Cham, 275–289. https://doi.org/10.1007/978-3-319-67630-2_21
- [31] Leonid Ryzhyk, Nikolaj Bjørner, Marco Canini, Jean-Baptiste Jeannin, Cole Schlesinger, Douglas B. Terry, and George Varghese. 2017. Correct by Construction Networks Using Stepwise Refinement. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 683–698. <https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/ryzhyk>
- [32] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. 2009. Dingo: Taming Device Drivers. In *Proceedings of the 4th ACM European Conference on Computer Systems (Nuremberg, Germany) (EuroSys '09)*. Association for Computing Machinery, New York, NY, USA, 275–288. <https://doi.org/10.1145/1519065.1519095>
- [33] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. 2009. Automatic Device Driver Synthesis with Termite. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (Big Sky, Montana, USA) (SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 73–86. <https://doi.org/10.1145/1629575.1629583>
- [34] Leonid Ryzhyk, Adam Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm, and Mona Vij. 2014. User-Guided Device Driver Synthesis. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 661–676. <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/ryzhyk>
- [35] Adrian Schüpbach, Andrew Baumann, Timothy Roscoe, and Simon Peter. 2011. A Declarative Language Approach to Device Configuration. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (Newport Beach, California, USA) (ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 119–132. <https://doi.org/10.1145/1950365.1950382>
- [36] Mirela Simonović, Vojin Živojnović, and Lazar Saranovac. 2017. Formal model for system-level power management design. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017 (Lausanne, Switzerland)*. IEEE, Piscataway Township, NJ, USA, 1599–1602. <https://doi.org/10.23919/DATE.2017.7927245>
- [37] Lalith Suresh, João Loff, Nina Narodytska, Leonid Ryzhyk, Mooly Sagiv, and Brian Oki. 2019. Synthesizing Cluster Management Code for Distributed Systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems (Bertinoro, Italy) (HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 45–50. <https://doi.org/10.1145/3317550.3321444>
- [38] System Management Interface Forum (SMIF), Inc. 2010. PMBus™ Power System Management Protocol Specification, Revision 1.2. <https://pmbus.org/specification-archives/>
- [39] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2017. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1057–1074. <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang>
- [40] Enzian Team. 2021. Enzian. <http://enzian.systems>
- [41] Liam Tung. 2019. Intel warning: Critical flaw in BMC firmware affects a ton of server products. Retrieved October 10, 2020 from <https://www.zdnet.com/article/intel-warning-critical-flaw-in-bmc-firmware-affects-a-ton-of-server-products/>
- [42] u-bmc. 2015–2020. u-bmc. <https://github.com/u-root/u-bmc>
- [43] Inc UEFI Forum. 2020. Advanced Configuration and Power Interface (ACPI) Specification, Version 6.3. https://uefi.org/sites/default/files/resources/ACPI_6_3_final_Jan30.pdf
- [44] Steven J. Vaughan-Nichols. 2017. MINIX: Intel's hidden in-chip operating system. Retrieved October 10, 2020 from <https://www.zdnet.com/article/minix-intels-hidden-in-chip-operating-system/>
- [45] Shaojie Wang and Sharad Malik. 2003. Synthesizing Operating System Based Device Drivers in Embedded Systems. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (Newport Beach, CA, USA) (CODES+ISSS '03)*. Association for Computing Machinery, New York, NY, USA, 37–44. <https://doi.org/10.1145/944645.944655>
- [46] Chao Xu, Felix Xiaozhu Lin, Yuyang Wang, and Lin Zhong. 2015. Automated OS-Level Device Runtime Power Management. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (Istanbul, Turkey) (ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 239–252. <https://doi.org/10.1145/2694344.2694360>
- [47] Chao Xu, Felix Xiaozhu Lin, and Lin Zhong. 2014. Device Drivers Should Not Do Power Management. In *Proceedings of 5th Asia-Pacific Workshop on Systems (Beijing, China) (APSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 11, 7 pages. <https://doi.org/10.1145/2637166.2637233>
- [48] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do Not Blame Users for Misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (Farmington, Pennsylvania) (SOSP '13)*. Association for Computing Machinery, New York, NY, USA,

244–259. <https://doi.org/10.1145/2517349.2522727>

Received 9 April 2021; revised 11 June 2021; accepted 5 July 2021