

Scalably Verifiable Dynamic Power Management

Opeoluwa Matthews, Meng Zhang, and Daniel J. Sorin

Department of Electrical and Computer Engineering
Duke University

Abstract

Dynamic power management (DPM) is critical to maximizing the performance of systems ranging from multicore processors to datacenters. However, one formidable challenge with DPM schemes is verifying that the DPM schemes are correct as the number of computational resources scales up. In this paper, we develop a DPM scheme such that it is scalably verifiable with fully automated formal tools. The key to the design is that the DPM scheme has fractal behavior; that is, it behaves the same at every scale. We show that the fractal design enables scalable formal verification and simulation shows that our scheme does not sacrifice much performance compared to an oracle DPM scheme that optimally allocates power to computational resources. We implement our scheme in a 2-socket 16-core x86 system and experimentally evaluate it.

1 Introduction

For the computer systems of today and tomorrow, the limiting constraint is power. Computer architects strive to achieve the greatest possible performance within a given power budget. One way in which computers maximize their power-efficiency (performance per watt) is through the use of *dynamic power management* (DPM). At runtime, computers dynamically re-allocate power to hardware resources. DPM may involve dynamic voltage and/or dynamic frequency scaling, dynamic power gating, dynamic clock gating, etc. DPM is performed at many granularities—among cores on a multicore processor chip and among processors within a datacenter—although the DPM schemes at each level may vary.

Designing an effective DPM scheme is challenging, and this challenge is exacerbated by the increasing scale of computer systems. Multicore processors contain increasing numbers of cores, and datacenters contain increasing numbers of processors. With more hardware resources to which to allocate power, the DPM algorithm has many more options for how to allocate power. For a DPM scheme to be useful, it must be scalable to large-scale systems.

DPM is a well-studied field with many published techniques [10][8][6], but one major concern with implementing a DPM scheme is verifying that the scheme behaves correctly in all possible situations. Furthermore, as system sizes grow, verification becomes more difficult. DPM protocols are similar to cache coherence protocols in their complexity and in the difficulty of verifying them correct. Verification is critical, because a bug in a DPM scheme can lead to a chip or a rack that overheats and

damages itself or to a system that is far less power-efficient than it could be. However, there is currently no automated way to verify that a DPM scheme is correct for an arbitrary number of computing resources. The only prior approach is to divide a multicore processor into small groups of cores (e.g., 3 cores per group) and verify that each group manages its own power correctly [9].

In this paper, we develop a new DPM scheme that we design specifically to be scalably verifiable with fully automated formal verification tools. Our approach creates a DPM scheme that is hierarchical and, more importantly, *fractal*, i.e., has the same behavior at every level. We leverage the fractal nature of the DPM scheme to enable an inductive proof that the scheme is correct for any number of computing resources (i.e., for any number of levels of hierarchy). Our DPM scheme borrows the fractal idea from a recent paper in cache coherence [14] and adapts and extends it to the new context of dynamic power management. To the best of our knowledge, our fractal DPM scheme is the first DPM scheme that is scalably verifiable with fully automated formal tools.

In this paper, we first present our system model (Section 2) and explain why it is difficult to verify DPM schemes for this model (Section 3). Motivated by the verification challenge, we present our fractal DPM scheme (Section 4) and show how we can verify it at any scale (Section 5). We explain how the fractal DPM potentially sacrifices performance in order to maintain its fractal behavior (Section 6) and evaluate an abstract system to show there are no fundamental performance limitations (Section 7). We then describe our software implementation of the fractal DPM scheme on a real x86 system with 16 cores (Section 8) and experimentally evaluate it (Section 9). Lastly, we compare fractal DPM to prior work (Section 10) and conclude (Section 11).

2 System Model

We assume a system with an arbitrary number of computing resources, C . These computing resources can be cores or multicore processors, and we intentionally treat them abstractly to highlight the generality of our approach. (In our implementation that we present later in the paper, each computing resource is a pair of cores.) The computing resources are homogeneous in how they interact with DPM but can otherwise be heterogeneous. Each computing resource C_i individually and dynamically requests power that is directly proportional to X_{max_i} , where X_{max_i} is the current performance of computing resource C_i if C_i is allocated its maximum possible power.

Table 1. Averaging Power Settings of Children. Bracketed entries are symmetric (e.g., L:ML and ML:L).

| Node Power | Possible Power Settings of Children |
|------------|---|
| L | L:L |
| ML | {L:ML, ML:L}, {L:M,M:L}, ML:ML |
| M | {L:MH, MH:L}, {L:H, H:L}, {ML:M, M:ML}, {ML:MH, MH:ML}, M:M |
| MH | {M:MH, MH:M}, {ML:H,H:ML}, {M:H,H:M} MH:MH |
| H | {MH:H, H:MH}, H:H |

Table 2. Notation for Describing DPM Actions

| label | action |
|-------|---|
| a | send request for L (ReqL) request to parent |
| b | send request for ML (ReqML) request to parent |
| c | send request for M (ReqM) request to parent |
| d | send request for MH (ReqMH) request to parent |
| e | send request for H (ReqH) request to parent |
| f | send GrantPowerReq response to left child |
| g | send DenyPowerReq response to left child |
| h | send Ack to parent |
| z | stall request |

2.1 DPM Model

The DPM scheme dynamically assigns to each computing resource C_i a power allocation, P_i . A computing resource’s performance, X_i , is a function of its power allocation and its unconstrained performance at that time, $Xmax_i$. That is, $X_i = f(P_i, Xmax_i)$. Later in this paper, we explore specific performance functions, but we intentionally keep this function abstract for now.

The goal of the DPM scheme is to allocate a fixed system-wide power budget, B , to the computing resources, in response to their requests, so as to maximize the performance of the system. That is, the DPM scheme seeks to maximize $\sum X_i$ under the constraint that $\sum P_i < B$.

2.2 Power and Performance Model

Without loss of generality, we assume that there are five possible power settings for each computing resource: Low (L), Medium-Low (ML), Medium (M), Medium-High (MH), and High (H). These power settings can correspond to different voltage/frequency settings, different power gating settings, etc. For example, setting a processor core to the Low power setting could mean setting the core to a low-voltage and low-frequency or it could mean powering down the core. The mapping from abstract power states to concrete configurations of computing resources is orthogonal to our work.

We further assume that each computing resource’s $Xmax_i$ has five possible values, also labeled L, ML, M, MH, and H. A computing resource will thus request a power setting equal to its $Xmax_i$.

3 Verification Scalability Problem

As computer architects, we seek to design our DPM scheme to “fit” existing tools rather than attempt to develop new tools. In this work, we focus only on *fully automated*

formal verification methodologies. We do not consider informal simulation-based validation (i.e., simulating extensively to try to uncover design bugs), because it is fundamentally incomplete (i.e., cannot find all bugs). We also do not consider formal verification methodologies such as theorem proving [12] and parametric verification [3], both of which are scalable but require substantial manual effort from verification experts.

For our automated tool, we choose the well-known Mur ϕ [5] model checker. A model checker exhaustively searches the reachable state space of a design and checks that specified invariants are maintained in every possible reachable state.

Model checking is an exhaustive technique for verification but it suffers from the well-known state space explosion problem. Exhaustively searching the state space of a non-trivial system is generally infeasible. Consider our system with DPM. If each computing resource can be in one of five states, then the number of states in the system is on the order of 5^C . Clearly, there is some value of C beyond which the state space exceeds the capability of the model checker.

One typical approach to this state space explosion problem is to model check a small-scale instance of the desired system. For example, one might model check a system with three computing resources, because that is the limit of the state space that can be explored. However, model checking a system with three computing resources does not, in general, ensure that systems with more computing resources are correct.

Our goal in this work is to design a DPM scheme that we can verify for any arbitrary number of computing resources.

4 Fractal DPM Design

Given the state space explosion problem, our strategy for a scalably verifiable DPM scheme is to design it such that we can leverage the power of induction. The key insight is that a *fractal* design—a design in which the system behaves the same at every scale—enables an inductive verification. We need to verify two aspects of the design. First, we must verify that the base case of the induction, the smallest scale of the system, is correct (i.e., never exceeds its power budget). Second, we must verify the inductive step, i.e., that the

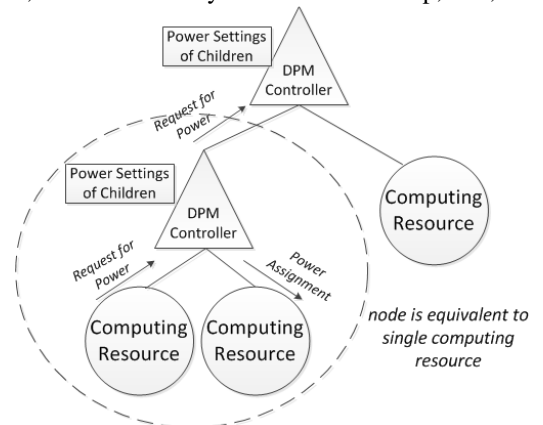


Figure 1. DPM scheme with 3 computing resources

Table 3. Specification of Behavior of Computational Resource

| State | X_{max_i} Demand | | | | | Responses from Parent Controller | |
|--------|--------------------|-----------|----------|-----------|----------|----------------------------------|------------------|
| | L | ML | M | MH | H | GrantReq | DenyReq |
| L | -- | b/pend-ML | c/pend-M | d/pend-MH | e/pend-H | | |
| ML | a/pend-L | -- | c/pend-M | d/pend-MH | e/pend-H | | |
| M | a/pend-L | b/pend-ML | -- | d/pend-MH | e/pend-H | | |
| MH | a/pend-L | b/pend-ML | c/pend-M | -- | e/pend-H | | |
| H | a/pend-L | b/pend-ML | c/pend-M | d/pend-MH | -- | | |
| pend-* | z | z | z | z | z | h/requested state | h/previous state |

design is indeed fractal. Critically, both of these verification steps are fully automated with Mur ϕ . We defer a discussion of the actual verification until Section 5, but we have outlined it here to provide intuition for our design.

4.1 Fractal System Organization

In Figure 1, we illustrate the smallest scale system, which has three computing resources. Our fractal systems are hierarchical, based on a binary tree organization. The leaves of the tree are the computing resources, and the intermediate nodes are *DPM controllers*. Each DPM controller is a simple finite state machine that records the power states of its children as well as some state regarding in-flight requests for power.

Each computing resource can request a new power setting by sending a request to its parent DPM controller. Depending on the request and its current state, the DPM controller either responds directly to the computing resource or sends a request to its parent DPM controller.

Because of the fractal design of the DPM scheme, we often reason about a DPM controller with its two children as a single “node” that behaves like a single computing resource. If the two children are at different power settings, we average them (and round up) to obtain the power setting of the node. The averaging/rounding process is listed in Table 1, where we denote the power settings of the children in the form $X:Y$, where X and Y are the power settings of the left and right child, respectively. For example, state L:M denotes that the left child is in state L and the right child is in state M.

4.2 Maintaining Fractal Power Invariants

To be fractal, our DPM scheme’s behavior must be fractal (described next in Section 4.3) *and* the invariants it maintains must be fractal. For example, we cannot specify as an invariant that the average power of all computational resources on the chip is a given power level (e.g., MH). An invariant must be applicable at all scales of the system, not just when considering the system as a whole. This need for fractal invariants distinguishes our DPM scheme from all prior DPM schemes of which we are aware. The fractal invariant we specify here for DPM also distinguishes this work from Fractal Coherence [14] because the coherence invariant is naturally fractal and power invariants are not.

The specific fractal invariant we choose is:

Fractal Invariant: It is impossible for both children of a DPM controller to be at the High power setting at the same time.

This invariant is useful for a system that has a power budget less than what would be drawn by all of its computing resources if they were all operating at their highest power setting. Such a system could be a multicore processor or a datacenter. As a concrete example, an Intel multicore processor with TurboBoost cannot let all of its cores operate at the highest power setting (in fact, only one core can be at the TurboBoost setting).

By maintaining this fractal invariant at every level, the DPM scheme limits system-wide power consumption. In Section 6, we analyze the relationship between our fractal invariant and system-wide power consumption and show that, as a result of this fractal invariant, the average power consumption of all computing resources asymptotically approaches a maximum of MH. For example, consider an 8-core chip with an 80W chip-wide power budget. If we equate a core at MH with a 10W per-core power budget, then our DPM scheme is provably guaranteed to enforce the chip-wide power budget. If we instead have a datacenter with 10,000 nodes and a 1MW power budget, then we would equate MH with a 100W per-node power budget.

We could have chosen another fractal invariant, such as “both children are never at H or MH at the same time” or “the average power of both children is never more than MH”, etc. Whatever fractal invariant we choose requires us to analyze the relationship between this fractal invariant and chip-wide power consumption, as we do for the invariant in this paper. Future work will explore other fractal invariants, but we do not believe that the choice of fractal invariant qualitatively affects the contributions or conclusions of our work.

4.3 Fractal DPM Scheme Specification

In this section, we precisely specify the behaviors of the computing resources and the DPM controllers. We use a table-based specification methodology [13] to specify these finite state machines. The rows of a table correspond to states, and the columns correspond to events. Each entry in the table corresponds to a state/event combination, and the entry specifies what happens in that situation. An entry has the form *Actions/NextState*. To keep the tables concise, we denote actions using a shorthand notation shown in Table 2. For example, a table entry of the form “d/MH” would denote that the finite state machine sends a request for MH (ReqMH) to its parent DPM controller and then changes its state to MH. A shaded entry in the table denotes that this entry is impossible; the given event cannot occur in the given state. An entry with “--” denotes that no action or state change occurs. There are some states and transitions that are required

Table 4. Specification of Behavior of Non-Root DPM Controller
Bolded entries show states/situations that are specially required to maintain fractal behavior.

| State | Messages from Left Child (messages from right child are symmetric) | | | | | | Messages from Parent (for requests from left child) | | Optional |
|----------------------------|---|---------------|--------------|--------------------------------|--------------------------------|--------------------|--|-----------|---------------------|
| | ReqL | ReqML | ReqM | ReqMH | ReqH | Ack | GrantReq | DenyReq | |
| L:L (L) | | b/pend-ML:L | b/pend-M:L | c/pend-MH:L | c/pend-H:L | | | | |
| L:ML (ML) | | f/block-ML:ML | c/pend-M:ML | c/pend-MH:ML | d/pend-H:ML | | | | |
| L:M (ML) | | c/pend-ML:M | c/pend-M:M | d/pend-MH:M | d/pend-H:M | | | | |
| L:MH (M) | | c/pend-ML:MH | d/pend-M:MH | d/pend-MH:MH | e/pend-H:MH | | | | |
| L:H (M) | | d/pend-ML:H | d/pend-M:H | e/pend-MH:H | g/block-L:H | | | | --/X:H ^F |
| X:H^F (M) | | d/pend-ML:H | d/pend-M:H | g/block-X:H^F | g/block-X:H^F | | | | --/L:H |
| ML:L (ML) | a/pend-L:L | | f/block-M:L | c/pend-MH:L | c/pend-H:L | | | | |
| ML:ML (ML) | f/block-L:ML | | c/pend-M:ML | c/pend-MH:ML | d/pend-H:ML | | | | |
| ML:M (M) | b/pend-L:M | | f/block-M:M | d/pend-MH:M | d/pend-H:M | | | | |
| ML:MH (M) | f/block-L:MH | | d/pend-M:MH | d/pend-MH:MH | e/pend-H:MH | | | | |
| ML:H (MH) | c/pend-L:H | | f/block-M:H | e/pend-MH:H | g/block-ML:H | | | | |
| M:L (ML) | a/pend-L:L | f/block-ML:L | | c/pend-MH:L | c/pend-H:L | | | | |
| M:ML (M) | b/pend-L:ML | b/pend-ML:ML | | f/block-MH:ML | d/pend-H:ML | | | | |
| M:M (M) | b/pend-L:M | f/block-ML:M | | d/pend-MH:MH | d/pend-H:M | | | | |
| M:MH (MH) | c/pend-L:MH | c/pend-ML:MH | | f/block-MH:MH | e/pend-H:MH | | | | |
| M:H (MH) | c/pend-L:H | f/block-ML:H | | e/pend-MH:H | g/block-M:H | | | | |
| MH:L (M) | a/pend-L:L | b/pend-ML:L | b/pend-M:L | | f/block-H:L | | | | |
| MH:ML (M) | b/pend-L:ML | b/pend-ML:ML | f/block-M:ML | | d/pend-H:ML | | | | |
| MH:M (MH) | b/pend-L:M | c/pend-ML:M | c/pend-M:M | | f/block-H:MH | | | | |
| MH:MH (MH) | c/pend-L:MH | c/pend-ML:MH | f/block-M:MH | | e/pend-H:MH | | | | |
| MH:H (H) | c/pend-L:H | d/pend-ML:H | d/pend-M:H | | g/block-MH:H | | | | |
| H:L (M) | a/pend-L:L | b/pend-ML:L | c/pend-M:L | f/block-MH:L | | | | | |
| H:ML (MH) | b/pend-L:ML | b/pend-ML:ML | c/pend-M:ML | c/pend-MH:ML | | | | | |
| H:M (MH) | b/pend-L:M | c/pend-ML:M | c/pend-M:M | f/block-MH:M | | | | | |
| H:MH (H) | c/pend-L:MH | c/pend-ML:MH | d/pend-M:MH | d/pend-MH:MH | | | | | |
| pend-* | z | z | z | z | z | | fh/block* | gh/block* | |
| block-* | z | z | z | z | z | --/requested state | | | |

to maintain fractal behavior, and they are in **bold** text in the tables.

4.3.1 Computing Resource Specification

In Table 3, we specify the behavior of each computing resource. The rows of the table are the states of the finite state machine, i.e., possible power settings. One state is labeled “pend-*”, which is shorthand for a family of pending states in which the computing resource has requested a new power state and is waiting for a response. For example, pend-L denotes waiting for Low power. The columns correspond to events that are either changes in demand ($Xmax_i$) from the computing resource or responses from the parent DPM controller.

The power management behavior of a computing resource is fairly simple. It responds to changes in demand by issuing requests for changes in power. It changes its power based on responses from its parent DPM controller.

4.3.2 DPM Controller Specification

We specify the behavior of each non-root DPM controller in Table 4, and we specify the behavior of the root DPM controller in Table 5. The specifications differ in that the root

DPM controller has no interactions with a parent. The state names are of the form $X:Y (Z)$, where X and Y are the power settings of the left and right children, respectively, and Z is the average power of the two children. The DPM controllers have two state names that are shorthand for families of states: pend-* and block-*. The block-* state family includes states such as block-L:ML, in which the DPM controller granted or denied a request to a child and is blocked waiting on the Ack from the child and will then go to state L:ML.

The tables are admittedly dense and likely hard to read, but our goal is to show a complete specification and to reveal that the entire finite state machine is not terribly complicated (i.e., fits on a dense page). The reader does not need to walk through each entry of each table but rather is encouraged to skim some entries to get a feel for how the protocol works. In an effort at conciseness, we include in the tables only the requests from the left child and responses to requests from the left child; the behavior with respect to the right child is identical.

The power management behavior of the DPM controllers is significantly more complicated than that of the computing resources. Notably, the non-root DPM controller must query

Table 5. Specification of Behavior of Root DPM Controller.
Bolded entries show states/situations that are specially required to maintain fractal.

| State | Requests from Left Child (requests from right child handled symmetrically) | | | | | Ack from Left Child | Optional |
|----------------------------|---|---------------|--------------|--------------------------------|--------------------------------|---------------------|---------------------|
| | ReqL | ReqML | ReqM | ReqMH | ReqH | | |
| L:L (L) | | f/block-ML:L | f/block-M:L | f/block-MH:L | f/block-H:L | | |
| L:ML (ML) | | f/block-ML:ML | f/block-M:ML | f/block-MH:ML | f/block-H:ML | | |
| L:M (ML) | | f/block-ML:M | f/block-M:M | f/block-MH:M | f/block-H:M | | |
| L:MH (M) | | f/block-ML:MH | f/block-M:MH | f/block-MH:MH | f/block-H:MH | | |
| L:H (M) | | f/block-ML:H | f/block-M:H | f/block-MH:H | g/block-L:H | | --/X:H ^F |
| X:H^F (M) | | f/block-ML:H | f/block-M:H | g/block-X:H^F | g/block-X:H^F | | --/L:H |
| ML:L (ML) | f/block-L:ML | | f/block-M:L | f/block-MH:L | f/block-H:L | | |
| ML:ML (ML) | f/block-L:ML | | f/block-M:ML | f/block-MH:ML | f/block-H:ML | | |
| ML:M (M) | f/block-L:M | | f/block-M:M | f/block-MH:M | f/block-H:M | | |
| ML:MH (M) | f/block-L:MH | | f/block-M:MH | f/block-MH:MH | f/block-H:MH | | |
| ML:H (MH) | f/block-L:H | | f/block-M:H | f/block-MH:H | g/block-ML:H | | |
| M:L (ML) | f/block-L:ML | f/block-ML:L | | f/block-MH:L | f/block-H:L | | |
| M:ML (M) | f/block-L:ML | f/block-ML:ML | | f/block-MH:ML | f/block-H:ML | | |
| M:M (M) | f/block-L:M | f/block-ML:M | | f/block-MH:M | f/block-H:M | | |
| M:MH (MH) | f/block-L:MH | f/block-ML:MH | | f/block-MH:MH | f/block-H:MH | | |
| M:H (MH) | f/block-L:H | f/block-ML:H | | f/block-MH:H | g/block-M:H | | |
| MH:L (M) | f/block-L:ML | f/block-ML:L | f/block-M:L | | f/block-H:L | | |
| MH:ML (M) | f/block-L:ML | f/block-ML:ML | f/block-M:ML | | f/block-H:ML | | |
| MH:M (MH) | f/block-L:M | f/block-ML:M | f/block-M:M | | f/block-H:M | | |
| MH:MH (MH) | f/block-L:MH | f/block-ML:MH | f/block-M:MH | | f/block-H:MH | | |
| MH:H (H) | f/block-L:H | f/block-ML:H | f/block-M:H | | g/block-MH:H | | |
| H:L (M) | f/block-L:ML | f/block-ML:L | f/block-M:L | f/block-MH:L | | | |
| H:ML (MH) | f/block-L:ML | f/block-ML:ML | f/block-M:ML | f/block-MH:ML | | | |
| H:M (MH) | f/block-L:M | f/block-ML:M | f/block-M:M | f/block-MH:M | | | |
| H:MH (H) | f/block-L:MH | f/block-ML:MH | f/block-M:MH | f/block-MH:MH | | | |
| block* | z | z | z | z | z | --/requested state | |

its parent DPM controller whenever a requested power setting change would change the node's state. For example, consider the case in which the DPM controller's state is L:L and the left child requests High power. Granting the left child High power would change the node's state from L to M (because H:L averages to M), and this change must be requested from the parent DPM controller. To maintain fractal behavior, the node must behave like a single computing resource, which would similarly issue a request to change its state from L to M.

The non-root DPM controller either satisfies the request directly (if doing so does not change the node's state) or passes along the appropriate request to its parent DPM, with only four exceptions. These four exceptions are situations in which satisfying the request would violate our invariant (i.e., both children cannot be in state H). These four exceptions are requests for High power when the other child is already in state H. In these situations, the DPM controller denies the request.¹

One unusual state (X:H^F) and its usage is required for verification purposes. We refer readers interested in this subtle issue to Appendix A.

4.4 Design Scalability

For purposes of verification, our DPM scheme is arbitrarily scalable. For purposes of performance, there are

¹ A more efficient solution would treat the request for H as a request for MH, but we sacrificed that optimization for simplicity.

possible scalability issues due to its structure. With a binary tree organization and many computing resources, a request that must be communicated to an upper level of the tree requires many hops and a potentially long latency. A higher-degree tree would mitigate this problem, but the verification tools we use are incapable of verifying the smallest scale DPM scheme with a higher-degree tree.

Despite the binary tree structure, there are three reasons why scalability is not a major concern. First and foremost, the latency of DPM itself is not critical. The computing resources continue to execute while waiting on outstanding DPM requests. Second, many requests can be satisfied without traveling far up the tree. Third, our experimental results on a real system (Section 9.4) show that, at least for a modestly sized system (16 computing resources), latencies are reasonable.

5 Verification of Fractal DPM

The motivation for our DPM scheme is to enable scalable verification. That is, we can scale the verification to any arbitrary number of computing resources, and the effort to verify the DPM scheme is independent of the number of computing resources.

In this section, we show the inductive verification process of our DPM scheme, which is based on a one-time proof that the induction is complete (Section 5.1). The verification process includes two steps:

1. Base case: Verify that the minimum system satisfies its power constraints (Section 5.2).

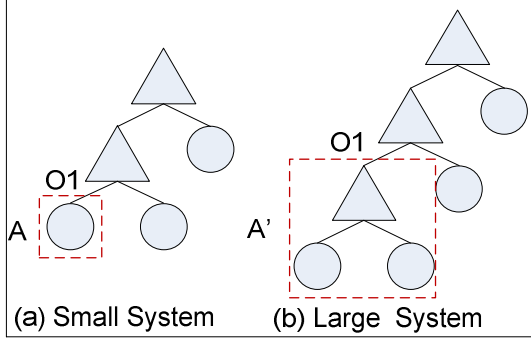


Figure 2. Observational Equivalence: “looking down”

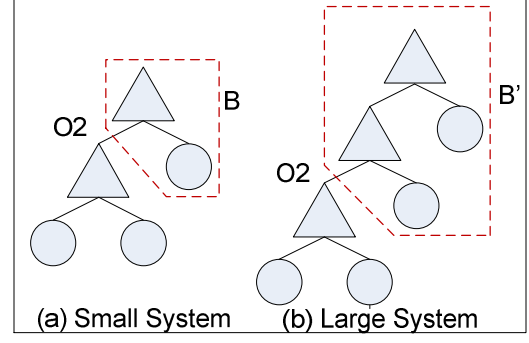


Figure 3. Observational Equivalence: “looking up”

2. Inductive step: Verify that larger systems are equivalent to smaller systems (Section 5.3).

Both verification steps are completed using the same automated tool, Mur ϕ . This single-tool verification methodology is an important improvement over Fractal Coherence [14]—which employs Mur ϕ for the base case and an equivalence checker for the inductive step—because the use of a uniform tool avoids the need for mistake-prone translation from one tool to another. Based on the results of these two verification steps, we can then prove the fractal DPM scheme is correct for any arbitrary number of computing resources.

5.1 Proof of Verification Completeness

There is a one-time proof (i.e., independent of the number of computing resources) that shows the above two verification steps are sufficient to inductively verify the correctness of DPM schemes with any arbitrary number of computing resources. The proof is very similar to that of Fractal Coherence, and we refer the reader to that proof [14] instead of replicating it here.

5.2 Base Case: Minimum System Verification

The base case of the inductive proof is minimum system verification. For our DPM scheme, the minimum system includes two computing resources, one internal DPM controller, and one root DPM controller, as shown in Figure 1. The minimum system is not chosen arbitrarily; it must have all different kinds of components in the system to ensure the completeness of verification. An example of an incomplete minimum system would be a system that includes only two computing resources and one root DPM controller. Although this system is even smaller than the correct minimum system, any further proof based on this system is incomplete because a non-root DPM controller may actually have spurious actions and those situations would have been missed in this incomplete base case.

The verification of the minimum system is straightforward. We describe the DPM scheme using the expressive language in Mur ϕ , and we specify the properties we are interested in as invariants. In our model, the invariant is that no two computing resources are in state H at the same time. Mur ϕ automatically traverses all possible reachable

states through explicit state enumeration and checks whether the property is maintained throughout the entire state space.

5.3 Inductive Step: Equivalence Verification

After verifying the power management of the minimum system is correct, we need to show that at each scale the system has exactly the same behavior in order to prove that the scheme is also correct for larger scale systems. This self-similarity feature is called fractal behavior and it is the guarantee of correctness when the system scales.

To check fractal behavior, we need to perform equivalence verification. Specifically, we verify a form of equivalence called “observational equivalence,” because we care only that each scale of the system behaves the same when observed from the outside world; internal actions are ignored. Observational equivalence is transactional in that it considers how systems react to inputs but not their timing.

We verify observational equivalence from two perspectives. We refer to one equivalence as “looking down,” because it is an equivalence between two children observed by a DPM controller parent. We illustrate the “looking down” equivalence in Figure 2. This equivalence is to ensure that, when observed from the O1 point in the figure, the systems inside the dashed boxes in Figure 2(a) and Figure 2(b) behave the same. The “looking down” equivalence enables us to scale the system downward while maintaining the illusion that any larger system behaves the same as the single computing resource A in the figure.

We illustrate the other equivalence, which we call the “looking up” equivalence, in Figure 3. When observed from the O2 point, the systems inside the dashed boxes in Figure 3(a) and Figure 3(b) behave the same. The “looking up” equivalence enables us to scale the system upward with the guarantee that any larger system behaves the same as sub-system B in the figure.

The overall impact of verifying these two equivalences is that any scale of the system behaves the same.

There are a few tools that perform equivalence checking, such as the bisimulator [1] in the CADP toolset [7] used in Fractal Coherence. However, those tools usually do not use the same language as Mur ϕ , and the language translation process is error-prone. Being able to use the same tool to verify both the base case and the inductive step is preferable.

Therefore, we leverage Park et al.’s aggregation checking idea [11] to perform the equivalence verification with Mur ϕ . The idea was originally introduced to check that an implementation of a protocol is consistent with its specification. The implementation is a fine-grained description of the execution, and the specification is an abstraction of the protocol with coarse-grained atomicity. The key idea is to use an aggregation function to map an implementation state to a specification state by completing any committed but incomplete transactions. Then an invariant is checked about this mapping to ensure that the two are actually consistent.

We find Park et al.’s aggregation method [11] to be a good match for verifying observational equivalence because, by executing all committed but incomplete transactions, it hides the internal transitions and leaves us only the transitions we are interested in. In our DPM scheme, the small system can be considered the specification and the large system can be considered the implementation. For example, as shown in Figure 2, computing resource A is the specification and sub-system A’ is the implementation. Computing resource A always has atomic transitions, and sub-system A’ has many internal transitions. The commit point in sub-system A’ is when any request or reply message arrives at the internal DPM controller. After a message passes through this DPM controller, the message is in its post-commit stage and needs to be processed until the end. The aggregation function is designed in a way that it drains out all the committed messages in all buffers inside sub-system A’.

We perform each equivalence verification using the Mur ϕ model of the larger subsystem. Mur ϕ performs the equivalence verification automatically. It is worth mentioning that Mur ϕ enables the checking of the equivalence to be performed “on-the-fly,” which means the verification does not incur any increase in the state space compared to the state space of the larger subsystem. That is to say, if we can verify the correctness of the larger subsystem without a state explosion problem, it is guaranteed that the equivalence checking will not incur state explosion either.

6 Power Management Efficiency

Our fractal invariant guarantees that a DPM controller’s two children are never both at the High power setting. The implication of this invariant at the system level is that the system-wide power consumption is upper bounded. The most power that a system with C computational resources can consume while maintaining this invariant is $(C-1)MH + H$. That is, $C-1$ computing resources are at power level MH and one computing resource is at power level H. This result implies that, as the number of computing resources approaches infinity, the maximum average power of the computing resources approaches MH. We prove this in Appendix B.

Maintaining a fractal invariant leads to some situations in which our DPM scheme sacrifices performance that a non-fractal DPM scheme could achieve. In Appendix B, we

prove that our system can end up with the average computing resource power approaching MH and, in fact, it is legal for all computing resources to be in state MH. Our DPM scheme allows this situation, but it does not permit certain other situations in which the system uses the same power. Clearly, if all computational resources are allowed to be in MH, then this system-wide power consumption should be legal in all situations.

Consider the examples in Figure 4. On the left side, we have a system in which the average computing resource power is MH, and the invariant is maintained. On the right side, the average computing resource power is still MH but this system violates the invariant because the bottom-left DPM controller has both of its children in state H.

This illegality is the price we pay for the verifiability of our fractal DPM scheme. These suboptimal situations exist, but fortunately they are rare and the inefficiency itself is small. Our DPM scheme would force one of the two leaf nodes on the bottom-left of the figure to be in state MH instead of H, which is a relatively small performance inefficiency. We quantify this performance impact in our evaluation in Section 7.

7 Evaluation of Abstract System

In this section, we evaluate an abstract system that consists of generic computing resources that request power from the DPM scheme. This evaluation enables us to isolate fundamental characteristics of the fractal DPM scheme without obscuring them with implementation details. We will experimentally evaluate our fractal DPM scheme as implemented in a real system in Section 9.

The goal of the evaluation in this section is to determine whether our fractal DPM scheme does a good job of allocating a fixed power budget to computing resources. One can easily design a DPM scheme that never exceeds a power budget by simply turning off all of the computing resources; clearly, there is more to DPM than just staying within the power budget.

7.1 Simulation Methodology

We wrote a simple simulator to model a system with a parameterizable number of abstract computing resources. Each abstract computing resource periodically changes its $Xmax_i$ and, as a result, requests a new power setting from its parent DPM controller. The simulator randomly chooses the $Xmax_i$ values for each computing resource at each time step

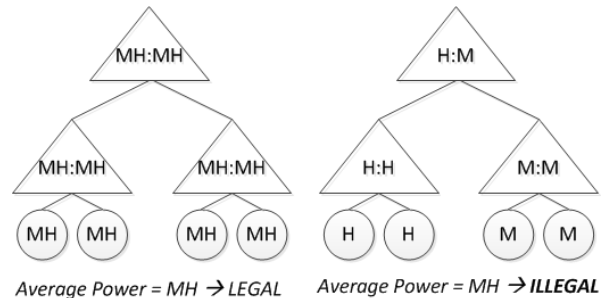


Figure 4. Example Inefficiency Due to Fractal Invariant

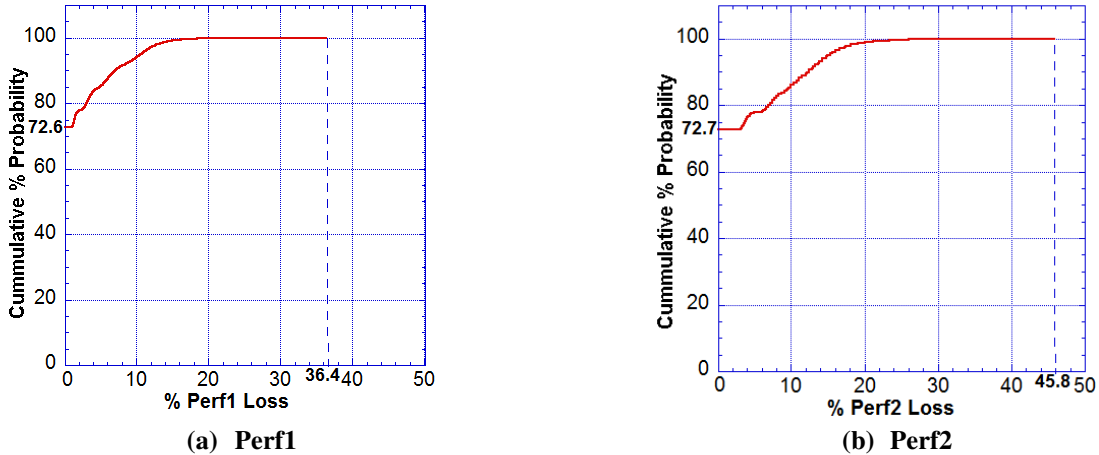


Figure 5. Percentage performance loss of fractal DPM compared to oracle DPM

and models the behaviors of the computing resources and the DPM controllers that were specified in Table 3-Table 5 (e.g., changing states, sending requests, granting/denying requests, etc.). The simulator computes the performance of each computing resource as a function of the power it is granted by the DPM scheme per time step. We simulate for millions of time steps to obtain statistically significant results.

To quantify our results, we must assign numerical values to the power settings (Low-High) and $Xmax_i$. We map the power settings as: L=5, ML=10, M=15, MH=20, and H=25. For $Xmax_i$, we set it equal to the requested power. For example, if a computational resource requests ML, its $Xmax_i=10$.

7.2 Performance Modeling

One challenge in evaluating a DPM scheme is determining the performance of a computing resource at a given power setting. For a given computing resource, there are many ways it could use its power allocation. Two computing resources at the same power setting could run the same software but achieve different performance based on how they use that power. For example, consider two processor cores, both of which are allocated a low power setting. One core could stay within its power allocation by using all of its resources but at a lower clock frequency, while the other core stays within the same power allocation by keeping a higher clock frequency but disabling some of its resources.

To avoid muddying the evaluation with the specific details of how each computing resource uses its power allocation, we abstract away the relationship between performance and power. We consider a computational resource’s performance to be a function of its power setting and its $Xmax_i$, and we consider two functions that are representative of typical performance/power relationships. The two functions are:

$$perf1 = \max\left(Xmax_i, \frac{Xmax_i P}{P + K}\right), \text{ where } K = 10$$

$$perf2 = \max(Xmax_i, P)$$

The first performance equation represents a system in which adding power leads to decreasing marginal

performance benefit (e.g., using more power to enable a faster core clock frequency helps performance but eventually performance becomes memory-bound). The second equation offers linear performance benefit (e.g., ideal voltage/frequency scaling). In real systems, performance would likely fall between these two performance curves.

7.3 Comparisons

Rather than compare against a vast number of prior DPM schemes—and try to match a wide range of assumptions made in these schemes—we compare against the ideal case of an (unimplementable) oracle. The oracle exhaustively searches for the best possible allocation of power settings for all of the computing resources. The oracle satisfies the same system-wide power invariant that results from the fractal DPM’s fractal invariant, but the oracle is not constrained by the fractal invariant. The oracle can perform both power allocations shown in Figure 4, whereas the fractal DPM can only perform the allocation on the left side of the figure. Our fractal DPM scheme obviously cannot perform as well as the oracle, but our goal is to show that its performance is close to the oracle.

7.4 Results

In Figure 5, we plot the CDF of the percentage performance loss of fractal DPM, with respect to the oracle DPM. Figure 5a and Figure 5b correspond to *perf1* and *perf2*, respectively. The modeled system has 8 computational resources.²

We observe that, in the majority of the time steps (>72% for both *perf1* and *perf2*), fractal DPM achieves the exact same performance as the oracle. When fractal DPM does fall short of the oracle’s performance, the performance gap is never more than 37% for *perf1* and 46% for *perf2*. The discrepancy between fractal DPM and the oracle is somewhat greater for *perf2*, because *perf2* models greater performance

² We cannot, in a reasonable amount of time, simulate the oracle DPM for more computational resources.

at higher power states, and thus being at a lower power state (to maintain the fractal invariant) is somewhat more costly. Overall, these results confirm that there are few situations in which fractal DPM sacrifices performance and that, in these situations, the amount of performance sacrificed is relatively small.

8 Implementation

An abstract implementation and evaluation can provide insight, but the true merit of a DPM scheme can only be confirmed by implementing it in a real system and evaluating the implementation. We already know that the scheme is correct and has no fundamental performance limitations, yet real systems can reveal practical issues—such as latencies and bandwidths—that are ignored in our abstract evaluation.

We have implemented our fractal DPM scheme across two 8-core x86 machines from AMD, as illustrated in Figure 6. There are many different ways in which our DPM scheme could change the power allocations to cores. For our experiments, we chose dynamic voltage/frequency scaling (e.g., as in Isci’s well-known scheme [8]). That is, when a core requests a change in power, the DPM adjusts the core’s voltage and frequency accordingly. Dynamic voltage/frequency scaling (DVFS) offers perhaps the most intuitive relationship between power allocation and performance, which is why we chose it despite some recent studies showing that it might not be the best mechanism for power management [4]. We map L, ML, M, MH, and H to the following frequencies (all in GHz): L=1.4, ML=2.1, M=2.7, MH=3.3, and H=3.6.

Each machine is divided into 4 voltage/frequency domains, i.e., voltage/frequency can be adjusted at the granularity of a pair of cores but not on a finer, per-core granularity. A computing resource is thus a pair of cores in this implementation. Recall, though, that our fractal DPM can operate at any granularity, so operating at a 2-core granularity poses no problems. Each machine runs Linux, and we implement the DPM controllers as daemons that run on the machines. (Note that there is some asymmetry in mapping DPM controllers to machines.) Communication between cores and DPM controllers and between DPM controllers is performed over sockets.

We consider one minor optimization of the fractal DPM implementation in which a core pair that has a power request denied re-requests a power setting that is one level below the denied request (instead of just staying at its current setting). In our experimental results, we denote this scheme as OptFractalDPM. It is important to note that making core pairs re-request power settings does not involve changing the DPM scheme; the DPM scheme is orthogonal to the decisions the core pairs make when requesting power levels. Hence, OptFractalDPM preserves the scalable verifiability of FractalDPM.

9 Evaluation of Implementation

We have several goals in this experimental evaluation.

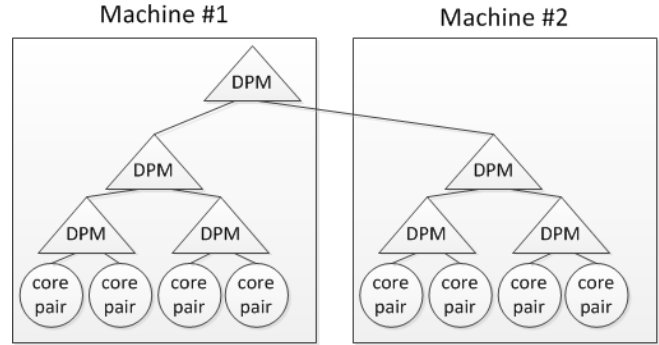


Figure 6. Implementation on Real System

- Compare the power and performance of fractal DPM against an unimplementable oracle DPM scheme that always assigns the optimal power levels to core pairs.
- Compare the power and performance of fractal DPM against a provably correct power management scheme that statically sets all cores to a given power level.
- Determine the latency to service requests for new power levels

9.1 Experimental Methodology

In all of our experiments, we run multithreaded benchmarks on both machines. On our particular machines, the power efficiency is, perhaps surprisingly, largely independent of the benchmark itself. That is, the optimal power setting for a core pair is almost entirely a function of the duty cycle of that core pair rather than which benchmark the core pair is running when it is not idle [4]. That is, if a core pair runs benchmark *B1* for 40% of its time and is idle the other 60% of its time, it will have the same optimal power setting if it instead ran benchmark *B2* for 40% of its time and was idle for the other 60%.

For experimental consistency, we chose a single 2-threaded application, bodytrack, from the Parsec suite [2], and we run this application on every core pair. We make each core pair a “cset” and pin applications to csets so that applications run to completion on the desired core pairs. Over the duration of each experiment, we vary the duty cycle of each core pair by varying the idle time between arrival times of new bodytrack jobs. We experimented with more sophisticated workloads with different benchmarks and combinations of different benchmarks, but the results (not shown) were nearly identical.

Before running our experiments, we ran each benchmark (i.e., interval of time with bodytrack running with a given duty cycle) to determine the optimal power setting for a core pair running that benchmark. We define the optimal power setting as the power setting that maximizes the $\text{energy} \times \text{delay}^2$ product. Hence, we developed 5 benchmarks, each of which has a unique power setting under which it runs optimally. We generate a random sequence of these benchmarks for each core pair before our experiments. We then use the same sequences of benchmarks for our experiments on each of the DPM schemes to ensure that the

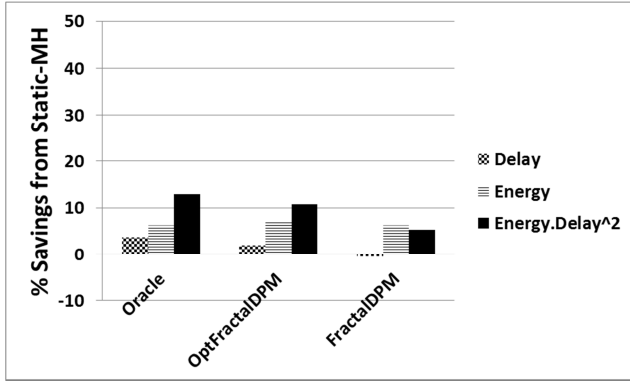


Figure 7. Comparison to Oracle

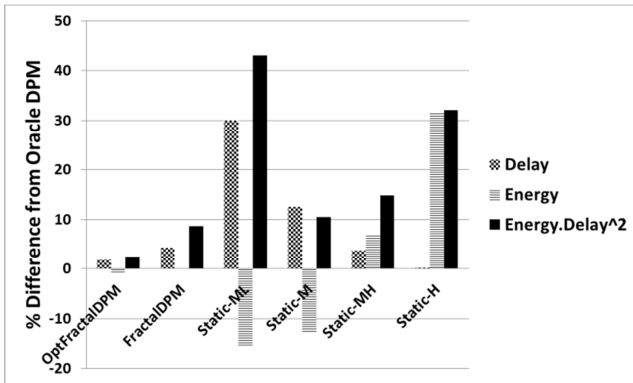


Figure 8. Comparison to Static DPM

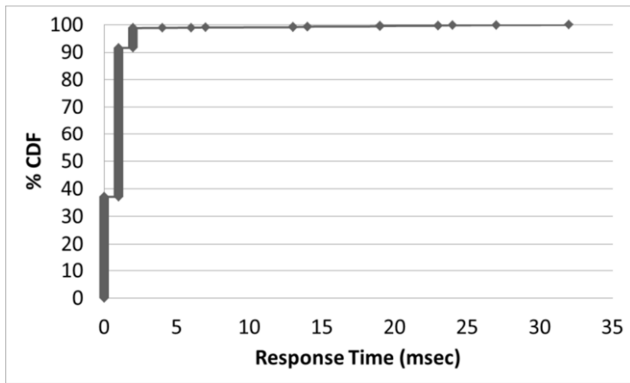


Figure 9. Fractal DPM Response Time

same amount of work is done in each experiment. We disabled c-states on the machines, to prevent the hardware from choosing to run at a lower frequency than the one we wish to run at.

Because the results are sensitive to how the core pair decides what power level to ask for and because this decision policy is orthogonal to our work here, we eliminate its impact by having core pairs ask for the pre-determined optimal power setting of a benchmark before running it.

In all experiments, we measure power with a “WattsUp?” power meter between the outlet and our machines.

9.2 Comparison to Oracle Power Management

As in our abstract evaluation in Section 7, we seek to compare fractal DPM to an unimplementable oracle DPM. The oracle always makes the ideal allocation of power to the core pairs, and the oracle has a pre-memoized set of decisions so that at runtime it consumes minimal latency and power. (Fractal DPM takes time and power to make decisions at runtime.) Comparing fractal DPM to this oracle shows how much the fractal DPM sacrifices in order to be both implementable and verifiable.

We plot the results of this experiment in Figure 7. We compare oracle, fractal DPM, and the optimized fractal DPM, and the comparisons are with respect to delay, energy, and ED^2 . All results in the figure are normalized to a trivially correct DPM scheme that statically sets all cores to the MH power level. The results show that there is indeed a gap between Oracle and fractal, but that the ED^2 gap is small (approximately 8% for fractal and 2% for the optimized fractal). These results corroborate the results from our earlier abstract evaluation.

9.3 Comparison to Static Power Management

In order to illustrate the efficacy of fractal DPM in optimizing power-efficiency, we compare it to schemes that statically allocate fixed power levels to all cores. Static allocation schemes are trivially correct in maintaining system-wide power invariants. We are unaware of any other scalable power management scheme that is provably correct.

The results of this comparison are in Figure 8. The figure compares fractal DPM and the optimized fractal DPM to static settings of ML, M, MH, and H. (The static-L scheme does so poorly that, if included in the graph, it obscures the more interesting trends.) The comparisons are with respect to the oracle DPM, i.e., smaller values are better. The figure shows that, in terms of ED^2 , the fractal and optimized fractal achieve better results than the static schemes. The static-ML and static-M schemes achieve impressive energy savings, but their performance is quite poor. The static-H scheme achieves excellent performance but at a steep energy cost.

9.4 Latency

One possible concern with fractal DPM is whether requests will take too long to be serviced. This latency includes the communication time between the core and its parent DPM controller and possibly between the parent DPM controller and its ancestors. This latency also includes the time required for the DPM controller daemons to wake up and determine what actions to take. Response time is not terribly critical since it is never on the critical path of execution (i.e., a core pair never stops executing while waiting for a new power level), but this latency is on the critical path to changing the power level of a core pair.

In Figure 9, we plot the results of performing a large number of power setting requests in our implementation. These results are admittedly a function of our particular hardware platform and operating system, but they give some idea of what response times are likely to be. The figure is the

CDF of the response time, and we observe that the vast majority of service times are within 1 msec. Almost 100% of requests are serviced within 3 msec. For perspective, a round-trip of datagram messages on our system—sending and receiving a message—takes 0.6 msec on average.

10 Related Work

There are two pieces of prior work that are most related to our work here. The first is Lungu et al.’s research on verifiable DPM for multicore processors [9]. They observed that DPM schemes (prior to our work here) cannot be verified with model checkers for more than a handful of cores. They showed the relationship between the number of power settings and the state space explosion, and they proposed designing the DPM scheme at the granularity of a handful of cores. There have been many other DPM schemes, including [10][8][6], but we are unaware of any other DPM scheme that considers verification.

The other work that inspires this work is fractal coherence [14]. The authors showed how to use the fractal design to enable scalably verifiable cache coherence protocols for multicore processors. A DPM protocol has similarities to a coherence protocol, but there are also key differences including, most notably, the invariants to be verified. We have leveraged this prior work for the idea of fractal design and for the proof of how a fractal design enables an inductive verification in two steps. We chose a simpler and more robust methodology for performing the equivalence verification (inductive step).

11 Conclusions

We have shown how to design the first dynamic power management protocol that can be formally verified for any number of computational resources. The key to scalable verification is the use of a fractal design methodology that enables an inductive proof of correctness. We have developed one concrete implementation of the fractal DPM, and one avenue of future work is to extend fractal DPM to larger scale systems, including datacenters.

Our analytical and experimental results both show that the fractal DPM protocol sacrifices only a small amount of performance compared to an oracle that makes optimal decisions at every time step, and we believe this sacrifice is worthwhile in order to achieve confidence in the DPM protocol’s behavior.

Acknowledgments

This material is based upon work supported by the National Science Foundation under grant CCF-0811290.

References

[1] D. Bergamini, N. Descoubes, C. Joubert, and R. Mateescu, “BISIMULATOR: A Modular Tool for On-the-Fly Equivalence Checking,” in *Proceedings of TACAS’05, volume 3440 of LNCS*, 2005, pp. 581–585.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC Benchmark Suite: Characterization and Architectural Implications,” in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[3] C.-T. Chou, P. Mannava, and S. Park, “A Simple Method for Parameterized Verification of Cache Coherence Protocols,” in *Formal Methods in Computer-Aided Design*, 2004, pp. 382–398.

[4] G. Dhiman, K. K. Pusukuri, and T. Rosing, “Analysis of Dynamic Voltage Scaling for System Level Energy Management,” in *Proceedings of the 2008 Conference on Power Aware Computing and Systems*, 2008.

[5] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, “Protocol Verification as a Hardware Design Aid,” in *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1992, pp. 522–525.

[6] A. Efthymiou and J. D. Garside, “Adaptive Pipeline Depth Control for Processor Power-Management,” in *Proceedings of the IEEE International Conference on Computer Design*, 2002.

[7] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu, “CADP - A Protocol Validation and Verification Toolbox,” in *Proceedings of the 8th International Conference on Computer Aided Verification*, 1996, pp. 437–440.

[8] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, “An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget,” in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[9] A. Lungu, P. Bose, D. J. Sorin, S. German, and G. Janssen, “Multicore Power Management: Ensuring Robustness via Early-Stage Formal Verification,” in *Proceedings of the Seventh ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2009.

[10] R. Maro, Y. Bai, and R. I. Bahar, “Dynamically Reconfiguring Processor Resources to Reduce Power Consumption in High-Performance Processors,” in *Proceedings of the Workshop on Power-Aware Computer Systems*, pp. 97–111, Nov. 2000.

[11] S. Park, S. Das, and D. L. Dill, “Automatic Checking of Aggregation Abstractions Through State Enumeration,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 10, pp. 1202–1210, Nov. 2006.

[12] S. Park and D. L. Dill, “Verification of FLASH Cache Coherence Protocol by Aggregation of Distributed Transactions,” in *Proceedings of the Eighth ACM Symposium on Parallel Algorithms and Architectures*, 1996, pp. 288–296.

[13] D. J. Sorin, M. Plakal, M. D. Hill, A. E. Condon, M. M. K. Martin, and D. A. Wood, “Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 6, pp. 556–578, Jun. 2002.

[14] M. Zhang, A. R. Lebeck, and D. J. Sorin, “Fractal Coherence: Scalably Verifiable Cache Coherence,” in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

Appendix A: Special State in DPM Controller

In Section 4.3.2, we specified the DPM controller behavior, which includes an unusual state and transition. Namely, the controller can, at any time, optionally choose to transition from state L:H to state X:H^F and vice versa. This seemingly useless state and transition are required for purposes of verifying the “looking up” equivalence (refer to Section 5.3).

Consider the example in Figure 10. In both halves of the figure, the bottom left computational resource requests High power. Its parent DPM then requests MH, because a combination of H with the right child’s M averages to MH. In Figure 10a, the shaded DPM controller grants the request because its new state, MH:H, is legal. In fact, in Figure 10a,

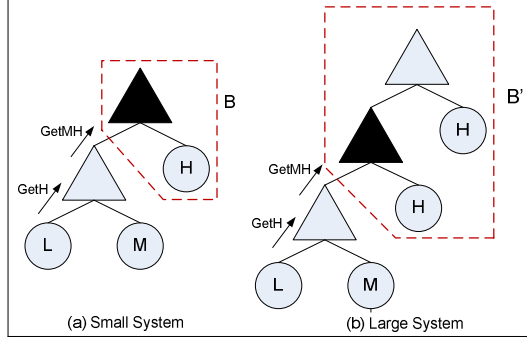


Figure 10. Observational Equivalence: “looking up”

there exists no state in which the shaded controller (and thus subsystem B) denies a request for MH.

In Figure 10b, however, the shaded DPM controller *denies* the request for MH. If it granted the request, its state would be MH:H, which averages to H. Then the shaded DPM’s parent would have two children in state H, which is illegal. Thus subsystem B’ has a state—the state in which it rejects a request for MH—that subsystem B does not have.

To make B and B’ equivalent, we add a new state to DPM controllers, called X:H^F. This state denotes a DPM controller whose right child is in H and that will deny a request for MH from its left child. By setting the shaded DPM controller in Figure 10a to this new state, B is equivalent to B’.

A crucial insight is that we added this DPM controller state to enable verification and *not* because we ever want to use it in the small system of Figure 10a. However, it has to exist and be reachable in order for subsystem B to have a state that is equivalent to the state of B’ in Figure 10b.

Appendix B: Chip-Wide Power Consumption

Definitions: Let T_n be the set of all possible binary trees with n computational resources that that satisfy our fractal invariant. For a tree $t \in T_n$, let $C(t) = \{c_1, \dots, c_n\}$ be the set of computational resources in t . The power consumption of tree $t \in T_n$ is: $P(t) = \sum_{i=1}^n P(c_i)$, and we denote the power of a computational resource as one of $\{L, ML, M, MH, H\}$. We define $P_{equiv}(t)$ as the average power of the children of t as specified in Table 1. We define $P_{max}(n) = \max_{t \in T_n} P(t)$. We denote replacing a computational resource c_i in tree t with an arbitrary tree s with m computational resources as: $t(c_i \rightarrow s \in T_m)$. Lastly, we define $\hat{t}_n \in T_n$ such that $P(c_1) = H$ and $P(c_2, \dots, c_n) = MH$.

Now we inductively prove that $\forall n P_{max}(n) = P(\hat{t}_n)$.

Base case: $P_{max}(1) = H$, which satisfies the constraint that $P_{max}(1) = P(\hat{t}_1)$.

Inductive step: Assume $P_{max}(k) = P(\hat{t}_k)$.

Let t' be an arbitrary element of T_{k+1} . Then, $\exists t \in T_k$ such that $t' = t(c_i \rightarrow \bar{t} \in T_2)$ and such that $P_{equiv}(\bar{t}) = P(c_i)$. This is true by the “looking down” observational equivalence because, for any tree t' , there exists an observationally equivalent tree t from which it can be scaled—by replacing a computational resource c_i with a

tree \bar{t} with two computational resources—such that none of the states of the nodes of t are changed. That is, \bar{t} has an average power equal to that of c_i .

There are two cases to consider:

Case #1: $P(c_i) \neq H$:

From Table 1, the possible power settings that average to any non-H state always sum to be less than or equal to twice the state they average to. So, since the power of the computational resources of \bar{t} average to $P(c_i)$, the sum of their power states will always be less than or equal to twice $P(c_i)$. Thus:

$$P(t') \leq P(t) - P(c_i) + 2P(c_i)$$

$$P(t') \leq P(t) + P(c_i)$$

Given that the highest power state that $P(c_i) \neq H$ can be in is MH, and by the inductive step,

$$P(t') \leq P(t) + MH \leq P_{max}(k) + MH = P(\hat{t}_{k+1})$$

$$P(t') \leq P(\hat{t}_{k+1})$$

Because t' was an arbitrary element in T_{k+1} , then $P(\hat{t}_{k+1}) = P_{max}(k+1)$.

Case #2: $P(c_i) = H$:

From Table 1, for a tree satisfying our invariant, the only power settings that average to $H = P(c_i)$ are H:MH and MH:H. Thus:

$$P(t') = P(t) - H + MH + H$$

By the inductive step,

$$P(t') = P(t) + MH \leq P_{max}(k) + MH = P(\hat{t}_{k+1})$$

$$P(t') \leq P(\hat{t}_{k+1})$$

Again, because t' was an arbitrary element in T_{k+1} ,

$$P_{max}(k+1) = P(\hat{t}_{k+1}) = (n-1)MH + H. \blacksquare$$