Bard: A Unified Framework for Managing Soft Timing and Power Constraints

Connor Imes
Department of Computer Science
University of Chicago, Chicago, IL, USA
ckimes@cs.uchicago.edu

Henry Hoffmann
Department of Computer Science
University of Chicago, Chicago, IL, USA
hankhoffmann@cs.uchicago.edu

***Abstract*—Embedded systems are subject to timing and power constraints. To support both, software currently must integrate multiple tools, resulting in additional complexity. We address this problem with a unified, portable framework called Bard which uses control theory to meet the primary constraint and linear programming to optimize the other. We evaluate Bard on two embedded platforms that exhibit different performance and power/energy characteristics and show that it achieves less than 2% error in meeting power constraints while maintaining nearly 95% of optimal performance. Additionally, Bard supports changing the primary constraint type at runtime while still achieving similar results.**

I. Introduction

Application performance and system power consumption are important factors influencing embedded system design. Embedded applications have hard and soft real-time performance demands [37]. Managing power consumption is important for maximizing battery life and for meeting thermal design limits to prevent overheating and irreparable system damage [11, 35, 38].

Meeting timing or power constraints are, individually, well-studied areas, though the problems become more complex when meeting one and optimizing the other. For example, it is common to meet a timing constraint while trying to minimize power or energy consumption [6, 19, 20, 31, 42, 43]. Similarly, systems may need to meet a power target while attempting to maximize performance [8, 9, 15, 23, 25]. Researchers have proposed a wide variety of solutions, but unfortunately they do not offer the ability to change the primary constraint, *i.e.,* from performance to power and vice versa.

This can negatively impact software and systems that need to adapt at runtime due to of any number of considerations, including user preferences, pre-determined use cases, or environmental factors. For example, an application on a smartphone may need to meet timing constraints when executing in the foreground to deliver a sufficiently high level of performance to satisfy users. The same application may have tasks that execute when running in the background that are not time-sensitive and would benefit the user by keeping power consumption low so as not to drain the battery. Other systems that normally provide timing guarantees must continue operating with reduced capacity during hardware failures or during periods of extremely low energy reserves.

A naive approach to solving this problem is to integrate two different solutions into the application – one that meets timing constraints, and another for power. If the libraries are not properly managed, they may compete for control of system resources. This can result in failing to meet either goal, causing poor performance and high power consumption. Furthermore, it adds additional complexity to the program and increases the overhead in development and validation testing.

Exacerbating the problem is that many existing solutions are specific to particular applications or systems, *i.e., they are not portable*. For example, controlling power consumption has become so essential that Intel now has hardware support for guaranteeing power consumption [8]. This approach obviously only works on Intel hardware. Software developers must not only integrate multiple software packages to meet multiple constraints, they must integrate a different set of packages on different platforms.

In summary, there is no existing package that (1) meets either timing or power constraints, (2) optimizes the other, and (3) remains portable across systems. We address these challenges with Bard. Bard extends prior work that met soft real-time constraints by adding the ability to track both performance and power behavior and allow the user to decide which constraint/optimization scheme to use. *The key technical contribution of this paper is a demonstration that a control system that guarantees timing and minimizes energy can be expanded to guarantee power while maximizing performance.*

We evaluate Bard with eight parallel benchmarks on two different embedded platforms – an ARM big.LITTLE SoC and an Intel mobile Haswell processor. These two systems have very different features and power/performance tradeoffs. Despite these differences, Bard meets timing constraints while minimizing energy and meets power constraints while maximizing performance. Additionally, we demonstrate that Bard rapidly switches between performance and power constraints.

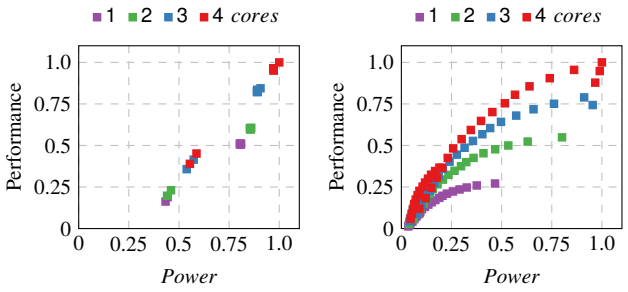
This paper makes the following contributions:

- Motivates the need for a unified framework for managing performance and power constraints on embedded systems.
- Design and implementation of Bard[1].
- Evaluation of Bard on two embedded platforms with different power/performance tradeoffs using 8 applications.

The effort on this project is funded by the U.S. Government under the DARPA BRASS program, by the Dept. of Energy under DOE DE-AC02-06CH11357, by the NSF under CCF 1439156, and by a DOE Early Career Award.

[1] Available at http://poet.cs.uchicago.edu/

978-1-5090-3076-7/16/$31.00 ©2016 IEEE



(a) Intel Haswell (b) ARM big.LITTLE

Fig. 1: Normalized power/performance tradeoffs.

II. Background and Motivation

Embedded hardware exposes resources exhibiting a wide range of power and performance tradeoffs. For example, Figure 1 illustrates the power/performance tradeoffs for two platforms while running a video encoding application. The x-axes show normalized power and the y-axes show normalized performance for each resource configuration. There are clear differences in the tradeoffs' shapes and sizes even though they are running the same application – the Intel is close to linear while the ARM presents a wider, more complicated space and contains more resource configurations.

Timing and power constraints are conflicting by nature. The key challenge for portability is finding a common approach for effectively navigating diverse tradeoff spaces, like those in Figure 1. Thus, many different resource management systems have been proposed. Some meet power constraints while maximizing performance. Others meet performance constraints while minimizing power or energy. Both capabilities may be necessary, however, especially for a system in the field which must deal with changing environmental conditions. We review related work of both types and demonstrate the need for a unified system that can meet either constraint while optimizing the other.

A. Meeting Power Constraints

As thermal dissipation limits multicore scaling, power constraints are becoming stricter [11, 38]. For example, the Exynos 5 processor is a system-on-chip based on the ARM big.LITTLE architecture [24]. It is typical of the type of heterogeneous design that has emerged for embedded systems, and it is critically power constrained — its 5.5W peak power is nearly twice its sustainable heat dissipation, limiting time at peak speed to less than one second [35].

To support systems with strict power budgets, several approaches guarantee power consumption while maximizing performance. Examples include those that manage DVFS for a processor [25], per-core DVFS in a multicore [23], processor idle-time [15], and DRAM [9]. Intel recently commercialized a hardware power controller which accepts a power cap from software and enforces it with DVFS [8]. The availability of such commercial power capping techniques signifies their importance. Other approaches coordinate multiple system components to meet a power goal. Examples include processor and DRAM speed [5, 12], DVFS and core allocation [7, 33, 40], and recent techniques coordinate arbitrary sets of resources [18, 44]. *Whether managing single or multiple resources, all of these approaches are concerned strictly with power consumption. None are capable of providing performance guarantees.*

B. Meeting Performance Constraints

Embedded systems must meet timing constraints. Many approaches provide timing guarantees through management of a single resource; *e.g.,* processor speed [41], processor duty cycle [46], caches [1], DRAM [47], and disks [26]. Other studies have shown that it is more energy-efficient to coordinate multiple resources than to manage any one alone [6, 42, 43]. Li et al. manage memory and processor speed [27], Dubach et al. coordinate several microarchitectural features [10], and Maggio et al. coordinate core allocation and clock speed [28]. Petrucci et al. coordinate thread scheduling and the use of heterogeneous cores [31]. Still other approaches focus on managing a general set of system-level components [19, 20, 32, 36, 45]. *None of these approaches, however, can provide power consumption guarantees.*

C. Need for a Unified Framework

None of the above approaches are flexible enough to support constraints on both performance and power, which is essential for embedded systems that need to operate in dynamically changing environments. We consider two examples: a system experiencing fan failure and another using harvested energy.

Consider an embedded system with a fan that provides real-time guarantees. If the fan fails, the system will need to switch from providing timing guarantees to providing power guarantees so that it does not destroy itself. In this case, the system will miss deadlines, but we would still expect it to maximize performance under the safe operating power.

The second example concerns a system using harvested energy. Such systems have batteries that store excess energy. When the battery is near full, the system will want timing guarantees. When the battery runs low, it must switch to a conservative mode that guarantees power to extend battery life. The system continues in a best-effort capacity until more energy is available, then switches back to timing guarantees. Such a system may switch between timing and power guarantees constantly during deployment.

These are just two of many possible examples demonstrating how an embedded system benefits from having support for both power and timing guarantees. However, current practices use separate solutions for each problem. These solutions must be validated against each other to ensure that they are compatible. This additional engineering effort could be saved by developing a single system capable of meeting performance or power constraints and switching between the two.

Some prior approaches have come close, but not quite fulfilled this need. CoAdapt provides timing and power guarantees, but only for *approximate* computations which can change their output accuracy [16]. JouleGuard meets energy guarantees through a combination of power and performance management [17]. Fu et al. provide real-time and temperature guarantees [14]. This approach addresses the fan failure example, but is not sufficient for the harvested energy case as

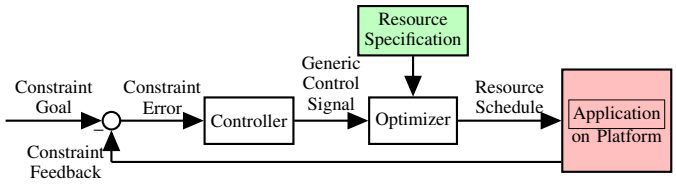


Fig. 2: Overview of the Bard runtime.

it only reacts to rising temperature – it cannot handle power explicitly. Furthermore, the Fu approach requires tuning for a specific system, which is not portable. Bard, in contrast, provides a unified, portable approach that can manage arbitrary sets of system components to (1) meet a timing constraint while minimizing energy, (2) meet a power constraint while maximizing performance, and (3) alternate between the two.

III. A Unified Framework

Bard's predecessor, POET, provides soft real-time guarantees with minimal energy by manipulating application resource usage. However, its design is independent of any particular set of resources or tradeoff space, making it portable. Bard extends this functionality to (1) meet power constraints while maximizing performance and (2) seamlessly switch between performance and power constraints as necessary. This section explains how we formulate these problems and describes Bard's design and implementation.

Figure 2 shows Bard's feedback control design. A user provides a performance or power *constraint goal* for an application. A *period* is a fixed-size work interval, composed of a predetermined number of jobs. After a period completes, Bard computes the error between the goal and the results from the last period and passes the result to the **controller**. The controller computes a *generic control signal* which the **optimizer** uses along with a system-agnostic **resource specification** to compute a *resource schedule* for the next period. A platform-specific component matches this schedule to actual system resources and applies it. When all the jobs in the next period are complete, the process repeats.

A. Controller

The **controller** computes the error between desired and actual performance or power, denoted as $e_x(t)$. It also estimates the application's base speed, $b_s(t)$, and base power, $b_p(t)$, using Kalman filters [39]. These are the application's estimated job performance and power consumption when using the minimal resource allocation. Base values can change at runtime for a number of reasons, including phases in the application input or environmental changes like temperature fluctuations. *The key to rapidly switching from a performance to a power constraint and vice versa is that Bard continually estimates both base values*. The controller then generates its generic control signal, $x(t)$, called *speedup* or *powerup*:

$$x(t) = x(t-1) + (1-\alpha) \cdot \frac{e_x(t)}{b_x(t)} \tag{1}$$

The $x(t-1)$ term is the speedup or powerup computed in the previous period, and α is a user-configurable pole value with the restriction $0 \leq \alpha < 1$. The pole determines how reactive the controller is to external changes, where a low value makes it very responsive and a high value makes it less responsive. $b_x(t)$ represents either base speed or power, depending on the constraint.

Algorithm 1 Finding an Optimal Configuration Schedule.

Require: C ▷ system configurations
Require: W ▷ workload size
Require: *constraint* ▷ PERFORMANCE or POWER
Require: $x(t)$ ▷ speedup/powerup, depending on *constraint*
  $U = \{c \mid x_c \leq x(t)\}$
  $O = \{c \mid x_c > x(t)\}$
  $candidates = U \times O = \{\langle u, o\rangle \mid u \in U, o \in O\}$
  $cost = \infty$
  $optimal = \langle -1, -1\rangle$
  $schedule = \langle -1, -1\rangle$

  for $\langle u, o\rangle \in candidates$ **do** ▷ loop over all pairs
    $j_u = W \cdot \frac{x_u \cdot (x_o - x(t))}{x(t) \cdot (x_o - x_u)}$ ▷ jobs spent in each config
    $j_o = W - j_u$
    if $constraint = POWER$ **then** ▷ cost of this pair
      $newCost = \frac{W}{j_u \cdot s_u + j_o \cdot s_o}$ ▷ normalized latency
    else
      $newCost = \frac{j_u \cdot p_u + j_o \cdot p_o}{W}$ ▷ normalized energy
    end if
    if $newCost < cost$ **then** ▷ compare cost to best so far
      $cost = newCost$
      $optimal = \langle u, o\rangle$
      $schedule = \langle j_u, j_o\rangle$
    end if
  end for

  return *optimal* ▷ pair of configurations with minimal cost
  return *schedule* ▷ jobs to spend in each configuration

B. Optimizer

The **optimizer** translates the generic control signal into a configuration schedule that optimizes energy or performance. There is a set C of possible configurations, where each $c \in C$ has a speedup s_c and a powerup p_c. By convention, $c = 0$ uses the minimal amount of resources. Speedup and powerup values are normalized to $c = 0$, such that $s_0 = p_0 = 1$. The formulation for minimizing energy under a timing constraint remains unchanged from POET [20]. We formulate maximizing performance under power constraint P_r as:

$$\textit{maximize} \quad \sum_{c=0}^{C-1} \tau_c \cdot s_c \tag{2}$$

$$\textit{s.t.} \quad \sum_{c=0}^{C-1} \tau_c \cdot p_c \cdot b_p(t) \leq P_r \tag{3}$$

$$\sum_{c=0}^{C-1} \tau_c = 1 \tag{4}$$

$$\tau_c \geq 0, \qquad \forall c \in \{0, \ldots, C-1\} \tag{5}$$

where τ_c is the proportion of time spent in configuration c.

Algorithm 1 computes a minimal-energy or maximal-performance schedule. It takes as input the configuration set C,

TABLE I: Configurability of our two embedded systems.

Platform	Processor	Cores	Core Types	Speeds (GHz)	TurboBoost	HyperThreads	Configurations
SVT11226CXB	Intel Haswell	2	1	.6–1.5	yes	yes	44
ODROID-XU3	Samsung Exynos5 Octa	8	2 (A15 & A7)	.2–2.0 (A15) .2-1.4 (A7)	no	no	128

```
#id spdup pwrup
0   1     1
1   1.55  1.06
2   2.11  1.11
3   2.16  1.12
4   2.66  1.17
5   3.36  1.22
6   4.51  1.31
7   5.11  1.37
```

```
#id cores frequencies
0   0x01  200000,-,-,-,200000,-,-,-
1   0x01  300000,-,-,-,200000,-,-,-
2   0x01  400000,-,-,-,200000,-,-,-
3   0x03  200000,-,-,-,200000,-,-,-
4   0x01  500000,-,-,-,200000,-,-,-
5   0x07  200000,-,-,-,200000,-,-,-
6   0x0F  200000,-,-,-,200000,-,-,-
7   0x07  300000,-,-,-,200000,-,-,-
```

Fig. 3: Snippets of actual Bard system-agnostic (left) and system-specific (right) configuration files.

workload size W, the *constraint* type, and the generic control signal $x(t)$. It then partitions C into two pairwise disjoint subsets, U and O. U contains the configurations with speedup or powerup values less than or equal to the signal computed by the controller with Eqn. 1. O contains the rest, with values greater than the controller's signal. The algorithm then loops over all possible pairs of configurations, with one chosen from each subset (the Cartesian product), and determines the schedule required to achieve the speedup or powerup given by the controller. Once the schedule is computed, its cost is determined. If the constraint is performance, energy is minimized; if the constraint is power, performance is maximized (latency is minimized). The algorithm remembers the pair with the lowest cost and returns the optimal pair and their schedule.

Algorithm 1 works because an optimal solution to the linear program in Eqns. 2–5 has no more than two non-zero τ_c [3]. The algorithm accounts for the two τ_c values with j_u and j_o, *i.e.,* number of jobs to complete in each configuration. This abstraction supports meeting both performance and power targets. Algorithm 1's complexity upper bound is $O(|C|^2)$ since each subset of configurations has at most $|C|$ configurations.

C. Implementation

A user provides four pieces of information to Bard – the set of configurations, runtime performance and power metrics, the performance or power target, and the constraint type.

System configurations are split into two data structures. The first data structure is system-agnostic, containing a configuration identifier along with that configuration's *speedup* and *powerup* values (normalized performance and power behavior). The second data structure is system-specific. While it can take any form a user desires for a particular system, the default format included with Bard contains a configuration identifier, a core mask, and a comma-delimited list of DVFS frequencies to apply[2]. This allows Bard to assign any possible subset of cores to an application, and if the system supports it, different cores may use different DVFS settings.

Examples of the data structures are presented in Figure 3. A dash indicates that a DVFS frequency does not need to be applied to a core, either because the frequency does not matter or it is already managed through another affected core. In the system-specific example on the right, state with *id* = 7 assigns cpu0, cpu1, and cpu2 (core mask 0x07) and sets the DVFS frequency on cpu0 to 300 MHz and cpu4 to 200 MHz. On this particular system, cores 0-3 are in one DVFS domain and cores 4-7 are in another. Therefore, applying a DVFS setting of 300 MHz on cpu0 sets the same frequency on cores 1-3, and applying a frequency of 200 MHz on cpu4 sets the same frequency on cores 5-7. We specify a dash for cores 1-3 and 5-7 to prevent setting their frequencies explicitly which could result in unnecessary overhead. Although cores 4-7 are not assigned, forcing a low DVFS setting reduces their power consumption while they idle.

[2]Bard is not limited to these components – users can write functions to manage any resources accounted for in the configurations.

TABLE II: System power characteristics.

System	Idle Power	Min Power	Max Power
Vaio	2.50 W	3.04 W	8.05 W
ODROID	0.21 W	0.19 W	6.37 W

To collect performance and power metrics, Bard relies on the Application Heartbeats API – recent updates to the API add power and energy tracking to the original, performance-based interface [20]. The user also provides the performance or power target by specifying minimum and maximum values through the Heartbeats API. Bard uses the average of these two values as the target.

We add the *constraint* type (`PERFORMANCE` or `POWER`) to the initialization function of the POET API. If `PERFORMANCE` is specified, Bard meets a performance target and minimizes energy. If `POWER` is specified, Bard meets the power target and maximizes performance. We then add a setter function to support changing the constraint type at runtime.

IV. Experimental Design

This section details the platforms and applications used to evaluate Bard.

A. Testing Platforms

The first system is the same model Sony Vaio tablet as POET was originally analyzed with, specifically a SVT11226CXB. The second is a newer model ODROID from Hardkernel, called an ODROID-XU3. Both systems run Ubuntu Linux 14.04. The Vaio uses mainline kernel 3.13.0 and the ODROID uses a modified kernel 3.10.58+. Table I demonstrates the variety of configurable resources we manage for each system and Table II shows their power characteristics.

Although not enforced in hardware, the DVFS frequencies on the Vaio must be the same across all cores to prevent non-deterministic behavior. We capture power data on the Vaio using the Intel Haswell processor's Model-Specific Register [34].

On the ODROID, applying a DVFS setting to any core on a cluster applies the setting to all cores on that cluster. Embedded

TABLE III: Application input and configurations.

Application	Input	Jobs	Period
blackscholes	10 million options	400 batches	20
bodytrack	sequenceB	261 frames	20
facesim	Storytelling	100 frames	20
ferret	corel:lsh	2,000 queries	20
x264	ducks_take_off	500 frames	20
dijkstra	input_small	1,000 paths	20
sha	in_file(1-16)	1,000 hashes	50
STREAM	self-generated	500 updates	50

INA-231 power sensors provide power data for the big Cortex-A15 cluster, the LITTLE Cortex-A7 cluster, the DRAM, and the GPU [21]. Unlike the ODROID model used previously, our ODROID's Exynos5 Octa SoC supports Heterogeneous Multi-Processing which allows any possible subset of cores to be allocated to a single task, with each cluster possibly running at different DVFS frequencies. To provide a fair comparison with POET, we do not execute on both clusters simultaneously, so we always set the unused cluster to its lowest DVFS setting to reduce power consumption.

B. Applications

We test Bard with eight parallel applications, none of which were originally written to provide predictable timing or power behavior.

Five applications are from the PARSEC benchmark suite [2] – `blackscholes`, `bodytrack`, `facesim`, `ferret`, and `x264`. Blackscholes prices financial investment portfolios using partial different equations, bodytrack and x264 both process video input, facesim creates animation of a human face from a model and a time sequence of muscle movements, and ferret performs content-based similarity searches of non-text data. `Dijkstra` and `sha` are from the ParMiBench benchmark suite [22]. Dijkstra computes single-source shortest paths in graphs and sha is an efficient encryption algorithm. `STREAM` [29] is a synthetic benchmark that represents memory-bound applications. Each of these applications performs a task that can reasonably be expected to run on embedded systems.

We instrument the applications with the Heartbeats API and add Bard calls, which requires only a few additional lines of code. Table III provides details on the inputs, the total number of jobs, and the size of the workload (period) for each Bard iteration. STREAM was previously shown to have low variance and therefore its runtime is safely shortened to save unnecessary execution time. Conversely, the blackscholes input is increased from 1 million to 10 million options, which is the default input for the application in the PARSEC benchmark suite. Higher performance and relatively slow sensor refresh intervals make it difficult to get accurate power readings on the ODROID when 1 million options are used – see Section V-A for more details. All inputs are included with the benchmarks, except the x264 input which is from a standard test set.

V. Experimental Evaluation

This section presents an empirical evaluation of Bard. Since POET optimizes energy consumption under timing constraints, we do not present a standalone analysis for the capability here. It is demonstrated, though, in switching the constraint type from performance to power at runtime (Section V-C).

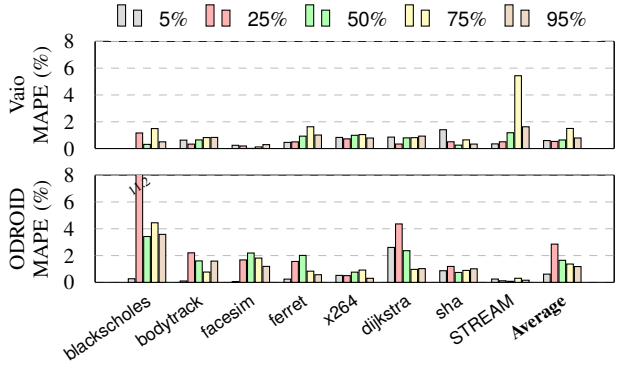


Fig. 4: Error for power targets (lower is better, 0 is optimal).

Prior to testing Bard, we characterize the applications on our systems by executing each in all configurations. We then derive an oracle that computes the optimal, dynamic resource usage for all applications and constraints. The oracle's optimal solution always meets the target, but requires perfect knowledge of the future.

A. Meeting Power Targets

Based on the characterizations, for each application i we determine the minimum power p_i^{min} and maximum power p_i^{max} consumed, measured in Watts. We then set five power targets within this range, from 5% to 95% of maximum over the minimum. For example, if application j has $p_j^{min} = 1W$ and $p_j^{max} = 5W$, then a 5% target is $1.2W$, a 50% target is $3.0W$, and a 95% target is $4.8W$.

To quantify Bard's ability to meet power targets, we compute the Mean Absolute Percentage Error (MAPE), a standard metric in control theory [13]. We formulate MAPE such that it increases when the measured power consumption $P_m(t)$ for job t exceeds the power requirement P_r. For an application with n jobs:

$$MAPE = 100\% \cdot \frac{1}{n} \sum_{t=1}^{n} \begin{cases} P_m(t) > P_r : & \left| \frac{P_m(t) - P_r}{P_r} \right| \\ P_m(t) \leq P_r : & 0 \end{cases} \quad (6)$$

Figure 4 presents the MAPE results for each application on the Vaio and the ODROID. During the first period, Bard just observes application behavior, after which the first action is taken. The second period is the initial adjustment phase – the end of this period is the earliest we can expect the controller to converge. For a fair analysis, the first two periods are ignored. An additional period of adjustment is required in some cases for the controller to converge at the 5% target. To prevent artificially inflating the performance results, the third period is also ignored in these cases.

Bard achieves, on average for all power targets, 0.81% MAPE on the Vaio and 1.53% MAPE on the ODROID. The higher error on the ODROID is due in part to the low refresh rate of its power sensors (264 ms). The most difficult execution is the 25% target for blackscholes on the ODROID, resulting in 11% error. Blackscholes' power/performance tradeoff space has an unusually large gap in power between Pareto-optimal LITTLE and big-core states. The 25% power target falls in this gap, so Bard must transition between the big and

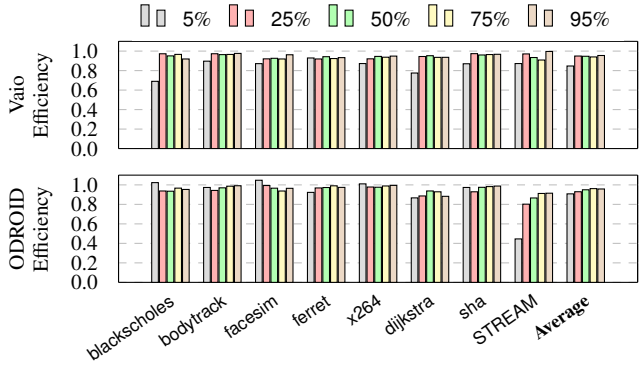


Fig. 5: Performance efficiency (higher is better, 1 is optimal).

LITTLE clusters. High variability in this particular transition's power overhead in combination with blackscholes' unmanaged threading makes it difficult for the controller to converge on the target.

B. Performance Optimization

Figure 5 demonstrates Bard's performance efficiency, *i.e.,* the actual performance compared to optimal as determined by our oracle. Since the applications are launched in the highest performance/power system state, the same periods described in Section V-A are ignored so as not to overstate Bard's performance.

Bard achieves near-optimal performance on both systems for all applications – efficiency is 93% on the Vaio and 94% on the ODROID. In a few 5% target cases on the ODROID, the performance efficiency exceeds optimal. The small error in meeting power targets sometimes pushes the actual performance over the oracle's computed optimal when too many resources are allocated.

C. Responding to Changing Goals

Bard supports both power and timing constraints in a single, portable framework. To demonstrate this capability, we launch applications with an initial timing constraint and then switch to a power constraint. We achieved similar results switching from power to timing constraints, but due to space limitations we omit that analysis and present charts for only four of the applications.

Performance targets are derived in the same way as power targets, where for each application i there is a minimum speed s_i^{min} and a maximum speed s_i^{max}. On both systems, we start with a performance target of 75%. About halfway through the execution, we switch to a power target. We set the power target to 25% on the Vaio. The extreme convexity of the ODROID's power/performance tradeoff space means that switching from a 75% performance target to a 25% power target does not result in as significant a change in system behavior as it does on the Vaio (see Figure 1). Instead, we set a more challenging power target of 0.5 Watts on the ODROID to illustrate taking advantage of the low-power LITTLE cores.

Figure 6 has four columns, one for each application. The top portion has performance data, normalized to the target from the first half of the execution. The bottom portion of the charts shows power data, normalized to the target set in the second half of the execution. The dashed vertical lines indicate when the switch from performance to power constraints is made. The solid black horizontal lines represent the goals.

Bard first keeps the performance at or above the target, then keeps power at or below the new target. MAPE quantifies performance deficit in the first half and power cap violations in the second half. Similarly, our oracle computes the minimal-energy schedule, then the maximal-performance schedule to determine the total efficiency of the execution. On the Vaio, Bard achieves 1.58% MAPE and 92% efficiency on average. On the ODROID, it achieves 1.97% MAPE and 91% efficiency. As Figure 6 shows, the time taken to switch between the timing and power constraints is quite small – just one period. The small fluctuations seen in the first portion of some executions (like x264) are not caused by Bard, but rather by the variability of the application inputs which makes them difficult to control [20]. These results demonstrate that Bard achieves its design goal of providing a *single, portable framework* for meeting either timing or power constraints and *dynamically switching* between the two.

D. Comparison with DVFS

DVFS alone is often used to meet performance and power constraints as it is nearly ubiquitous and is well-understood. Bard supports both timing and power constraints, and prior work has already shown that a DVFS-only approach for minimizing energy under timing constraints is not optimal. In this section, we limit Bard to DVFS-only configurations for meeting power targets, which also results in sub-optimal behavior. This analysis highlights the benefits of Bard's support for multiple configurable components.

For brevity, we focus on low power targets, where the discrepancy between a DVFS-only approach and Bard's support for managing multiple resources is most pronounced. These low power targets are particularly important, as we motivated in Section II, *e.g.,* in case of fan failure. It is more energy-efficient to use processor cores rather than let them idle [4], so we test while running on all cores on the Vaio and using the big cluster on the ODROID, which supports a wide power range. We use the same targets and analysis techniques as in Sections V-A and V-B and present the results for 5% and 25% power targets in Figure 7.

In most cases, the 5% power target is not achievable, hence the high MAPE values which are off the charts. Performance efficiency above optimal is meaningless in these cases since the primary constraint is violated. The ODROID is able to meet more of the 5% targets than the Vaio, but is still not always successful. For applications on the ODROID that can actually achieve this target, the average efficiency is only 52% of optimal. The LITTLE cores often achieve better performance for the same power, but they are not available using only DVFS.

In Sections V-A and V-B we showed that Bard meets power targets with low error and near-optimal efficiency by manipulating multiple resources. It is clear here that a DVFS-only approach is sub-optimal (and often unachievable) for these low-power targets, highlighting the importance of managing multiple components.

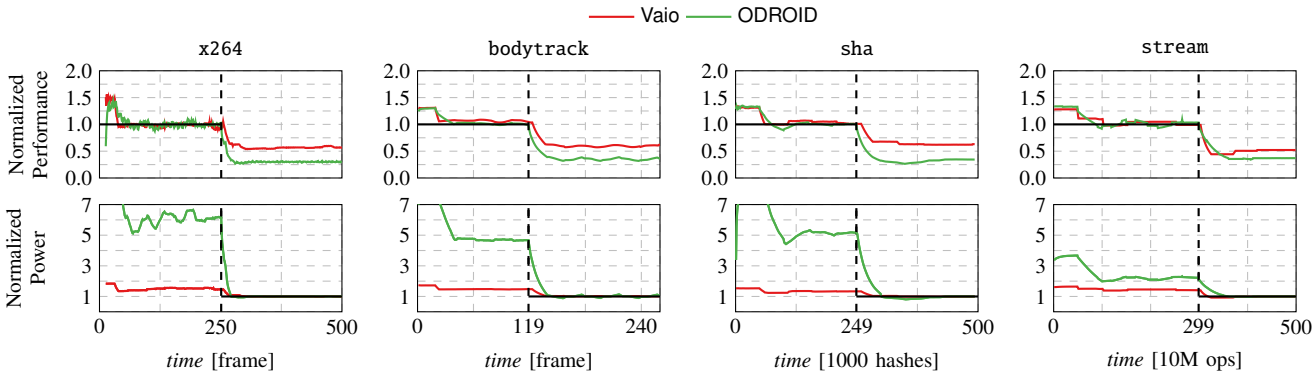
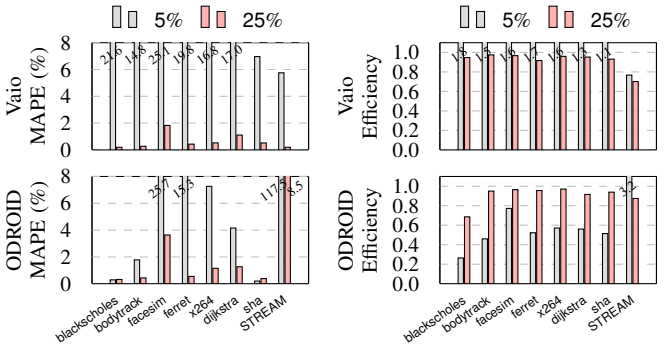


Fig. 6: Changing constraints from timing to power.



(a) Power error (b) Performance efficiency

Fig. 7: Sub-optimal results using a DVFS-only approach.

E. Discussion of Results and Limitations

Bard provides a unified, portable framework capable of delivering predictable timing with minimum energy or predictable power while maximizing performance. Our results hold despite the fact that our test platforms expose different resources and have very different power/performance tradeoffs. We stress that the same Bard runtime is executed on both systems with no code changes. The only required change is modifying the configuration files indicating what resources are controllable on the different platforms.

Many systems allow for small violations in performance and power constraints, so Bard is designed to provide soft guarantees rather than hard ones. Bard is also sensitive to the accuracy of the resource specifications provided to it – the controller can handle large errors, but it is still better to classify applications by their behavior, *e.g.,* compute-bound or memory-bound, and use separate configurations. Alternatively, Bard could be coupled with a learning engine (*e.g.,* LEO [30]) which could construct the models on the fly. Bard also does not explicitly model the overhead of changing system configurations, which can be non-trivial for some types of resources not examined in this work, like spinning up or down hard drives. Currently, these overheads are modeled as inaccuracy in the system configurations and are adapted to by the controller. Finally, Bard assumes that it has exclusive control over the system resources. Future work could extend Bard to support multiple applications and share control with other actors in the system.

VI. Conclusion

This paper establishes the need for a unified approach to managing timing and power constraints while optimizing energy and performance. In response, we build on prior work to design and evaluate a portable library called Bard. Bard uses feedback control to meet performance or power constraints, and linear optimization to either minimize energy or maximize performance, depending on the primary constraint. Furthermore, users or applications can switch the primary constraint at runtime to adapt to changing conditions. We evaluate Bard on two modern embedded systems and demonstrate that it meets its constraints with low error and is near-optimal in its optimization. Finally, we release Bard as an open-source C library, along with the configurations and benchmark patches used in its evaluation.

References

[1] R. Balasubramonian et al. “Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures”. In: *MICRO*. 2000.

[2] C. Bienia et al. “The PARSEC Benchmark Suite: Characterization and Architectural Implications”. In: *PACT*. 2008.

[3] S. Bradley et al. *Applied mathematical programming*. 1977.

[4] A. Carroll and G. Heiser. “Mobile Multicores: Use Them or Waste Them”. In: *HotPower*. 2013.

[5] J. Chen and L. K. John. “Predictive coordination of multiple on-chip resources for chip multiprocessors”. In: *ICS*. 2011.

[6] H. Cheng and S. Goddard. “SYS-EDF: a system-wide energy-efficient scheduling algorithm for hard real-time systems”. In: *IJES* 4.2 (2009).

[7] R. Cochran et al. “Pack & Cap: adaptive DVFS and thread packing under power caps”. In: *MICRO*. 2011.

[8] H. David et al. “RAPL: Memory Power Estimation and Capping”. In: *ISLPED*. 2010.

[9] B. Diniz et al. “Limiting the power consumption of main memory”. In: *ISCA*. 2007.

[10] C. Dubach et al. “A Predictive Model for Dynamic Microarchitectural Adaptivity Control”. In: *MICRO*. 2010.

[11] H. Esmaeilzadeh et al. “Dark silicon and the end of multicore scaling”. In: *ISCA*. 2011.

[12] W. Felter et al. “A performance-conserving approach for reducing peak power consumption in server systems”. In: *ICS*. 2005.

[13] A. Filieri et al. “Automated Design of Self-adaptive Software with Control-theoretical Formal Guarantees”. In: *ICSE*. 2014.

[14] Y. Fu et al. “Feedback thermal control of real-time systems on multicore processors”. In: *EMSOFT*. 2012.

[15] A. Gandhi et al. “Power capping via forced idleness”. In: *Workshop on Energy-Efficient Design*. Austin, TX, 2009.

[16] H. Hoffmann. “CoAdapt: Predictable Behavior for Accuracy-aware Applications Running on Power-aware Systems”. In: *ECRTS*. 2014.

[17] H. Hoffmann. “JouleGuard: energy guarantees for approximate applications”. In: *SOSP*. 2015.

[18] H. Hoffmann and M. Maggio. “PCP: A Generalized Approach to Optimizing Performance Under Power Constraints through Resource Management”. In: *ICAC*. 2014.

[19] H. Hoffmann et al. “A Generalized Software Framework for Accurate and Efficient Managment of Performance Goals”. In: *EMSOFT*. 2013.

[20] C. Imes et al. “POET: A Portable Approach to Minimizing Energy Under Soft Real-time Constraints”. In: *RTAS*. 2015.

[21] T. Instruments. http://www.ti.com/product/ina231.

[22] S. Iqbal et al. “ParMiBench - An Open-Source Benchmark for Embedded Multiprocessor Systems”. In: *Computer Architecture Letters* 9.2 (2010).

[23] C. Isci et al. “An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget”. In: *MICRO*. 2006.

[24] B. Jeff. “Big.LITTLE system architecture from ARM: saving power through heterogeneous multiprocessing and task context migration”. In: *DAC*. 2012.

[25] C. Lefurgy et al. “Power capping: a prelude to power shifting”. In: *Cluster Computing* 11.2 (2008).

[26] X. Li et al. “Performance directed energy management for main memory and disks”. In: *Trans. Storage* 1.3 (2005).

[27] X. Li et al. “Cross-component energy management: Joint adaptation of processor and memory”. In: *ACM Trans. Archit. Code Optim.* 4.3 (2007).

[28] M. Maggio et al. “Power Optimization in Embedded Systems via Feedback Control of Resource Allocation”. In: *Control Systems Technology, IEEE Transactions on* 21.1 (2013), pp. 239–246.

[29] J. D. McCalpin. “Memory Bandwidth and Machine Balance in Current High Performance Computers”. In: *IEEE TCCA Newsletter* (1995).

[30] N. Mishra et al. “A Probabilistic Graphical Model-based Approach for Minimizing Energy Under Performance Constraints”. In: *ASPLOS*. 2015.

[31] V. Petrucci et al. “Lucky Scheduling for Energy-Efficient Heterogeneous Multi-Core Systems”. In: *HotPower*. 2012.

[32] R. Rajkumar et al. “A resource allocation model for QoS management”. In: *RTSS*. 1997.

[33] K. K. Rangan et al. “Thread motion: fine-grained power management for multi-core systems”. In: *ISCA*. 2009.

[34] E. Rotem et al. “Power management architecture of the 2nd generation Intel Core microarchitecture, formerly codenamed Sandy Bridge”. In: *Hot Chips*. 2011.

[35] Y. Shin et al. “28nm High-K Metal-Gate Heterogeneous Quad-Core CPUs for High-Performance and Energy-Efficient Mobile Application Processor”. In: *ISSCC*. 2013.

[36] M. Sojka et al. “Modular software architecture for flexible reservation mechanisms on heterogeneous resources”. In: *Journal of Systems Architecture* 57.4 (2011).

[37] A. Tannebaum. *Modern Operating Systems*. Pearson-/Prentice Hall, 2008.

[38] G. Venkatesh et al. “Conservation cores: reducing the energy of mature computations”. In: *ASPLOS*. 2010.

[39] G. Welch and G. Bishop. *An Introduction to the Kalman Filter*. Tech. rep. TR 95-041. UNC Chapel Hill, Department of Computer Science, 2006.

[40] J. A. Winter et al. “Scalable thread scheduling and global power management for heterogeneous many-core architectures”. In: *PACT*. 2010.

[41] Q. Wu et al. “Formal online methods for voltage/frequency control in multiple clock domain microprocessors”. In: *ASPLOS*. 2004.

[42] C.-Y. Yang et al. “System-Level Energy-Efficiency for Real-Time Tasks”. In: *SOCRTD*. 2007.

[43] H. Yun et al. “System-Wide Energy Optimization for Multiple DVS Components and Real-Time Tasks”. In: *ECRTS*. 2010.

[44] H. Zhang and H. Hoffmann. “Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques”. In: *ASPLOS*. 2016.

[45] R. Zhang et al. “ControlWare: A middleware architecture for Feedback Control of Software Performance”. In: *ICDCS*. 2002.

[46] X. Zhang et al. “A Flexible Framework for Throttling-Enabled Multicore Management (TEMM)”. In: *ICPP*. 2012.

[47] H. Zheng et al. “Mini-rank: Adaptive DRAM architecture for improving memory power efficiency”. In: *MICRO*. 2008.