SmartBalance: A Sensing-Driven Linux Load Balancer for Energy Efficiency of Heterogeneous MPSoCs

Santanu Sarma, T. Muck, Luis A. D. Bathen, N. Dutt, and A. Nicolau

Department of Computer Science, University of California Irvine, CA, USA
{santanus, tmuck, lbathen, dutt, nicolau}@ics.uci.edu

Abstract

Due to increased demand for higher performance and better energy efficiency, MPSoCs are deploying heterogeneous architectures with architecturally differentiated core types. However, the traditional Linux-based operating system is unable to exploit this heterogeneity since existing kernel load balancing and scheduling approaches lack support for aggressively heterogeneous architectural configurations (e.g. beyond two core types). In this paper we present SmartBalance: a sensing-driven closed-loop load balancer for aggressively heterogeneous MPSoCs that performs load balancing using a sense-predict-balance paradigm. SmartBalance can efficiently manage the chip resources while opportunistically exploiting the workload variations and performance-power trade-offs of different core types. When compared to the standard vanilla Linux kernel load balancer, our per-thread and per-core performance-power-aware scheme shows an improvement in energy efficiency (throughput/Watt) of over 50% for benchmarks from the PARSEC benchmark suite executing on a heterogeneous MPSoC with 4 different core types and over 20% w.r.t. state-of-the-art ARM's global task scheduling (GTS) scheme for octa-core big.Little architecture.

General Terms Embedded Software, Operating System, OS Kernel, Power-Aware Systems.

Keywords Task Allocation & Scheduling, Load Balancing, Linux Scheduler, Energy Efficiency, Heterogeneous Multiprocessors (HMP), System-on-Chip (SoC), MPSoC.

1. Introduction

Emerging embedded devices (e.g., mobile platforms) face diverse multi-threaded workloads along with conflicting needs of high energy efficiency and performance, necessitating a move towards heterogeneous MPSoC platforms with architecturally differentiated cores providing attractive power-performance benefits [6] [11] [21]. Architectures with different core types are already a reality (e.g., NVidia's Kal-El [21] and ARM's big.LITTLE [11]) and this trend towards heterogeneity is only expected to grow further in the future [6, 19]. Unfortunately, handling heterogeneity comes at the cost of a complex OS scheduler and load balancer. Both time-varying characteristics of the application and OS workloads as well as the heterogeneous features of the platform need to be smartly managed. Existing OS kernels' scheduling and load balancing schemes are openly accepted to be inefficient for such systems [12]. For instance, the vanilla Linux kernel load balancer evenly distributes the workload among cores even if the cores have distinct processing capabilities, which can result in serious performance and energy efficiency loss. Even though there have been some recent efforts to address this important issue (e.g., the IKS [23] and the GTS [2] Linux extensions), the solutions have been limited to the very specific case of ARM's big.LITTLE with two core types. This limitation not only discourages hardware vendors from introducing new architectural improvements with diverse core types, but also defeats the purpose of versatile OS support for heterogeneous MPSoC with seamless OS configurations without making major structural changes in the OS load balancer/scheduler.

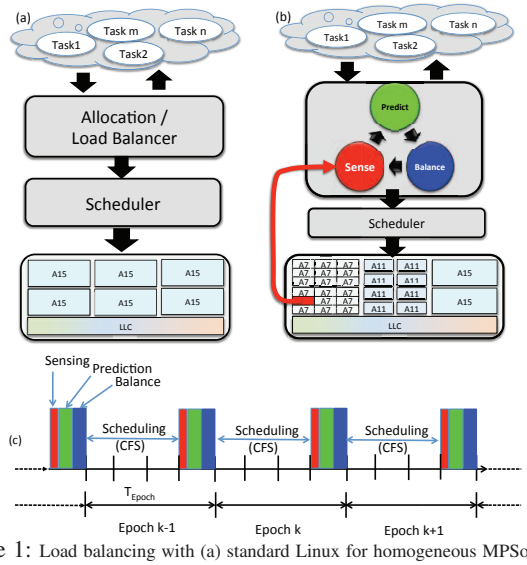


Figure 1: Load balancing with (a) standard Linux for homogeneous MPSoCs, (b) SmartBalance closed-loop sense-predict-balance approach for aggressively heterogeneous MPSoCs, (c) timing relations of the phases in SmartBalance. Each epoch covers several Linux CFS scheduling periods.

In this paper we propose SmartBalance, a closed-loop sensing-driven opportunistic load balancer that uses on-chip sensing, estimation and prediction, and global optimization for aggressively heterogeneous MPSoCs (with more than two core types, for instance with Big (A15), Medium (A11), and Little (A7) core types) as depicted in Fig.1 (b). SmartBalance consists of three phases: **sense**, **predict,** and **balance** that are executed at runtime in periodic *epochs*, where each epoch covers multiple Linux scheduling periods as shown in Fig. 1(c). Unlike the open-loop standard Linux load balancer seen in Fig.1 (a) which distributes the thread evenly, our closed-loop feedback-driven approach makes judicious decisions to distribute the threads smartly (i.e. matched to the core type) so as to best achieve the system goal(s) (e.g. energy efficiency) in a smart way. SmartBalance leverages existing OS code bases by reusing and refactoring the legacy Linux kernel code to improve energy efficiency. The key contributions of our approach are:

- We present SmartBalance: a closed-loop load balancing approach under a sense-predict-balance paradigm that efficiently manages chip re-

DAC'15, June 07 - 11, 2015, San Francisco, CA, USA.

http://dx.doi.org/10.1145/2744769.2744911

sources while opportunistically exploiting diverse workload and core performance-power characteristics.

- We expose workload (performance, power) variability to the OS to exploit per-thread workload characteristics (IPC, power, and utilization) in each core for aggressively heterogeneous architectures. We introduce estimation and prediction models to calculate the performance and power impact of executing each thread on different heterogeneous cores without performing sampling at each core type.
- We demonstrate a working prototype of SmartBalance for the Linux 2.6.x kernel.
- When compared to the standard vanilla Linux kernel load balancer, SmartBalance improves energy efficiency by over 50% for PARSEC benchmarks executing on a Heterogeneous MPSoC with 4 core types and by over 20% w.r.t the state-of-the-art ARM GTS scheme.

2. Motivation and Related Work

Heterogeneous MPSoCs provide architecturally diverse cores with drastically different power-performance trade-offs that can be exploited by the OS. Consequently, heterogeneous MPSoCs and their run-time systems is an active area of research. Previous works focused on extracting computational performance with energy efficiency as a secondary benefit [3, 9] and recently the focus has shifted to energy efficiency [2, 7, 14, 19]. However, most of these works have been restricted to a special class of heterogeneous MPSoC architecture (e.g., homogeneous cluster of two limited core types as in ARM big.LITTLE [13, 21]) with the number of threads being scheduled limited by the number of cores [1, 3, 7, 9, 15, 19]. Within this restricted architectural model, many of them do not take into account the OS and associated issues in their simulation/analysis[1, 3, 7, 9, 15, 19]. Many of these techniques also lack thread-level awareness of both the performance and power characteristics for exploiting multithreaded workloads at a finer granularity[1, 7, 9, 14, 15, 19, 20, 23]. This limits the opportunity to perform extensive energy and performance optimizations. SmartBalance overcomes these restrictions by handling aggressive architectural heterogeneity and exploiting runtime performance-power variability. The key differences of SmartBalance with the existing state-of-the-art is summarized in Table 1.

Table 1: Comparative Summary of Related Work

Reference	Scheme Generality		Per-Thread Awareness		Per-Core Awareness			Integrated & Implemented in OS
	No Core Types >2	Thread-to-core-ratio >1	IPC	Power	Util.	IPC	Power	
Chen2009	Yes	No	No	No	No	Yes	Yes	No
Annamalai2013	No	No	No	No	No	Yes	Yes	No
Liu2013	Yes	Yes	No	No	No	Yes	Yes	No
Kim2014	No	Yes	No	No	Yes	No	No	Yes
Linaro IKS 2013	No	Yes	No	No	Yes	No	No	Yes
ARM GTS 2013	No	Yes	No	No	Yes	No	No	Yes
SmartBalance	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

The closest to our work are the approaches proposed in [1, 2, 14, 19, 20, 23]. The work by Kim et al. [14] improves Linaro IKS [23] by bringing the core utilization awareness to the scheduler but lacks support for architectures with more than two core types in addition to per thread awareness for finer balancing. The Global Task Scheduler (GTS) [2] proposed by ARM brings several improvements in comparison to Linaro IKS[23] both in terms of performance and generality. GTS allows a selection of either a big core or a little core instead of a core cluster as in IKS, thus enabling architecture exploitation at finer granularity. However, GTS can not directly support architectures with more than two core types without major re-engineering of the kernel. In addition, although GTS considers per-thread utilization awareness, it is unable to exploit thread level opportunity for instruction level parallelism (ILP) and the power consumption of the threads. Annamalai et al. [1] proposed a program phase-aware dynamic scheduling scheme limited to only two core types without considering the OS workload and implementation issues. Liu et al [19] proposed a dynamic thread scheduling scheme with the number of threads limited to the number of cores. They also do not provide per thread awareness for the load balancer to exploit and consider the OS implementation issues. Mogul et al. [20] considered OS issues and proposes an asymmetric architecture with special core types optimized to run OS code with improved energy efficiency, but are also limited to two core types in their scheduling and load balancing. In contrast with all of the above related work, SmartBalance considers aggressively heterogeneous architectures with more than two core types while exploiting per-thread and per-core level awareness in a closed loop manner.

3. Heterogeneous Computing Elements and Thread Model

We consider the case of heterogeneous multi-core systems in which all cores can have different capabilities. A core type r is defined by the combination of micro-architectural features and their nominal performance and power (voltage/frequency). For example, Table 2 shows four core types labeled as *Huge, Big, Medium*, and *Small* differing in seven architectural feature combinations such as issue width, instruction queue (IQ) size, reorder buffer (ROB) size, no. of registers, and cache size. In addition, even if the cores are identical in terms of microarchitecture but associated with different nominal frequencies, they can be considered as distinct core types. We define the set of cores as $C = \{c_1, c_2, ..., c_n\}$, the set of core types as $R = \{r_1, r_2, ..., r_q\}$, where $\gamma : C \rightarrow R$ gives type of a particular core. Each type r is characterized by a unique combination of the parameters $X = \{x_1, x_2, ..., x_p\}$, with an example shown in Table 2.

Table 2: Heterogeneous Core Configuration Parameters

Parameter	Symbol	Huge Core	Big Core	Medium Core	Small Core
Issue width	x_1	8	4	2	1
LQ/SQ size	x_2	32/32	16/16	8/8	8/8
IQ size	x_3	64	32	16	16
ROB size	x_4	192	128	64	64
Int/float Regs	x_5	256	128	64	64
L1$I size (KB)	x_6	64	32	16	16
L1$D size (KB)	x_7	64	32	16	16
Freq. (MHz)	F	2000	1500	1000	500
Voltage (V)*	V_{DD}	1	0.8	0.7	0.6
Peak Throughput*	IPC	4.18	2.60	1.31	0.91
Peak Power (W)*	P_{Total}	8.62	1.41	0.53	0.095
Area (mm^2)*	A	11.99	5.08	3.04	2.27

* Estimated by the Gem5 [5] and McPAT modeling framework [17] for a 22nm node based on PARSEC [4] benchmarks.

SmartBalance assumes each core can run multiple threads and perform multitasking. We assume processes are encapsulating threads similar to the Pthread model, so there is no formal or explicit dependency between threads. Within the Linux scheduling subsystem, processes and threads are all treated as a *task entity* and scheduled independently. For uniformity, in this paper the term *thread* is used interchangeably for both single-threaded processes and for threads of the same process. We consider a total of m threads to be mapped to n cores without restricting the number of threads to the number of cores. Threads can enter and leave the system at any time and their total execution time is unknown. The set of threads to be optimized contains all threads active at the beginning of each SmartBalance Epoch.

Throughout the paper, we use the following notations:

- $V = \{t_i, 1 \leq i \leq m\}$ is the set of all the threads to be allocated during an epoch. The thread set $\Psi_j(k) = \{t_i, 1 \leq i \leq m_j\}, \forall t_i \in V = \{t_1, t_2, ..., t_m\}$ is m_j threads mapped to core c_j at the k^{th} epoch such that the "allocation"
$$\mathbf{\Psi(k)} = \{\Psi_j(k), 1 \leq j \leq n\} \quad (1)$$
where total threads $m = \sum_{j=1}^{n} m_j$.
- We define the throughput characterization matrix that is exposed to the OS as:
$$\mathbf{S(k)} = [\mathbf{s_i(k)}] = [ips_{ij}(k)] \quad (2)$$
as the average throughput (in instruction per second (IPS)) of threads executing on different cores at k^{th} epoch. $\mathbf{s_i(k)} = \{ips_{ij}(k), 1 \leq j \leq n\}$ represents the vector throughput when executing thread t_i on different cores. $ips_{ij}(k)$ represents the average throughput of thread t_i executing on core c_j. $IPS_j(k)$ is the average throughput of core c_j when executing its allocated threads during the epoch.
- Similarly, we define a power characterization matrix that is exposed to the OS as:
$$\mathbf{P(k)} = [\mathbf{p_i(k)}] = [p_{ij}(k)] \quad (3)$$

as the average power consumption of threads executing on different cores during an epoch k. $\mathbf{p}_i(k) = \{p_{ij}(k), 1 \leq j \leq n\}$ represents a power vector corresponding to a thread t_i executing on different cores and $p_{ij}(k)$ represents the average power of thread t_i executing on core c_j at the k^{th} epoch. $P_j(k)$ is the total average power consumed by the core c_j during the epoch while executing the threads allocated to it.

4. SmartBalance Approach

SmartBalance is closed-loop load balancing approach consisting of three stages: **sensing & measurement**, **estimation & prediction**, and **balance** that are executed at runtime in periodic epochs, where each smart balancing epoch covers multiple Linux CFS (completely fair scheduling) periods as shown in Fig. 2. Since we use the standard Linux CFS to perform scheduling of the threads allocated to the same core, we only focus on the three stages at the beginning of each epoch. At every epoch, SmartBalance first samples the power and the performance counters of each core. Second, using the measurements, it estimates the individual impact or contribution of each thread towards the performance and power of its current core. Third, the individual per-thread contribution in performance and power are used to predict the per-thread performance and power in other core types. These per-thread performance and power characteristics are represented in a systematic manner using two 2-dimensional matrices. These matrices are then used to find the allocation for the next epoch in the fourth step. Once the allocation is found, the threads are scheduled by the OS using the standard Linux CFS in the last step. This process is repeated in each epoch as shown in Fig.2.

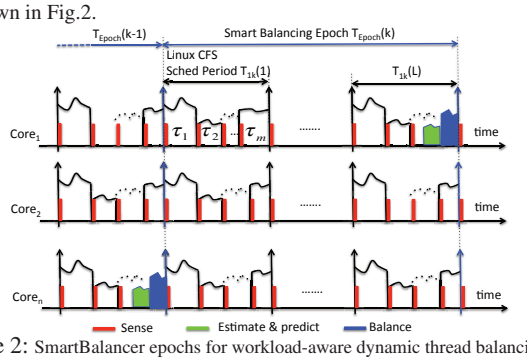


Figure 2: SmartBalancer epochs for workload-aware dynamic thread balancing and scheduling in heterogeneous MPSoCs. Each SmartBalance epoch covers L Linux CFS scheduling periods.

4.1 Sensing & Measurement

During the sensing and measurement phase, *hardware performance counters (HPCs)* and power are periodically sampled per thread (at the thread context switch as shown in Fig. 2) in order to trace the workload characteristics of the threads currently running in the system. We currently use following hardware performance counters (cycle, instruction and performance degradation events):

- **Cycle counters** sample the amount of *busy cycles* (cy_{Busy}), *idle cycles* (cy_{Idle}), and *sleep cycles* (cy_{Sleep}) of a core. Busy cycles represent the time a core spends doing computation. Idle cycles capture idling time due to pipeline stalls or cache misses. Sleep cycles capture the time a core spends in a quiescent state. In our current experimental platform and kernel implementation, a core enters this state when it has no threads to execute.
- **Instruction counters**. We sample the total amount of *committed instructions* (I_{total}), *committed load and stores* (I_{mem}), and *committed branches* (I_{branch}). These are used to calculate the share of memory instructions $I_{msh} = \frac{I_{mem}}{I_{total}}$ and the share of branch instructions $I_{bsh} = \frac{I_{branch}}{I_{total}}$.
- **Performance events counters.** We measure the following events which are known to drive the performance of a core[1]: mispredicted branches, which is used to compute the branch misprediction rate (mr_b); instruction/data L1 cache and TLB misses and hits, which are used to compute the L1 instruction miss rate ($mr_{\$i}$), L1 data cache miss rate ($mr_{\$d}$), instruction TLB miss rate (mr_{itlb}), and data TLB miss rate (mr_{dtlb}).

4.2 Estimation and Prediction of Performance and Power

As described in Section 3 threads are characterized using the performance and power matrices. Estimation and prediction of the performance and power matrices are possible since there is a direct correlation between the behavior of different core types. In this section, we derive analytical models to compute these matrices using a combination of measurements and prediction among the cores, and the experimental validation of each step is presented in Section 6. We propose a simple methodology to predict $ips_{il}(k)$ in a different core c_l by using the measurement $IPS_j(k)$ (instructions per second) and its throughput contribution $ips_{ij}(k)$ in the core c_j, $\forall j \neq l$ as discussed below.

4.2.1 Performance and Power Estimation of Each Thread on a Core

In order to estimate the thread-specific performance and power characteristics, we define the following notations and timing relations as shown in Fig. 2. We define the load balancing epoch T_{Epoch} consisting of L CFS scheduling periods T_{jk}. Let $T_{jk}(l), l = 1..L$ be the l^{th} CFS scheduling period in the k^{th} epoch $T_{Epoch}(k)$. Let there be $m = \sum_{j=1}^{n} m_j$ threads i.e. $\Psi_j = \{t_i, 1 \leq i \leq m_j\}, \forall t_i \in V = \{t_1, t_2, ..., t_m\}$ scheduled in a scheduling period $T_{jk}(l)$ in the core c_j. Let i^{th} thread get time share τ_{ijl} in the l^{th} scheduling period such that the scheduling period and smart balancing epoch is given by $T_{Epoch}(k) = \sum_{l=1}^{L} T_{jk}(l)$ where $T_{jk}(l) = \sum_{i=1}^{m_j} \tau_{ijl}$. Let I_{ijl} be the measured counter values for the number of instructions executed by the i^{th} thread in the l^{th} scheduling period, $\dddot{ips}_{ijl}$, $\ddot{p}_{ijl}$, and $\ddot{\epsilon}_{ijl}$ be the respective measured throughput, power, and energy consumed by the thread during the execution duration τ_{ijl} is given by $ips_{ijl} = I_{ijl}/\tau_{ijl}$ and $\ddot{p}_{ijl} = \ddot{\epsilon}_{ijl}/\tau_{ijl}$ respectively. The average total throughput and power in the k^{th} epoch for the i^{th} thread is :

$$ips_{ij}(k) = \sum_{l=1}^{L} I_{ijl} / \sum_{l=1}^{L} \tau_{ijl} = \frac{1}{L} \sum_{l=1}^{L} \dddot{ips}_{ijl} \quad (4)$$

$$p_{ij}(k) = \sum_{l=1}^{L} \ddot{\epsilon}_{ijl} / \sum_{l=1}^{L} \tau_{ijl} = \frac{1}{L} \sum_{l=1}^{L} \ddot{p}_{ijl}. \quad (5)$$

Total average throughput IPS_j and power P_j in the k^{th} epoch for all the threads for the core c_j:

$$IPS_j(k) = \sum_{i=1}^{m_j} \left(\sum_{l=1}^{L} I_{ijl} / \sum_{l=1}^{L} \tau_{ijl} \right) = \frac{1}{m_j} \sum_{i=1}^{m_j} ips_{ij}(k) \quad (6)$$

$$P_j(k) = \frac{1}{m_j} \sum_{i=1}^{m_j} p_{ij}(k) \quad (7)$$

Average core performance in terms of instruction per second for the k^{th} epoch is $IPS_j(k) = IPC_j(k) * F_j = I^j_{total} * F_j / (cy^j_{Busy} + cy^j_{Idle})$ where cy^j_{Busy}, cy^j_{Idle}, and I^j_{total} are the counters values sampled for core c_j at epoch k, and F_j is the frequency of the core.

4.2.2 Performance and Power Prediction for Different Core Types

In the previous section we described how we compute $ips_{ij}(k)$ for all threads t_i executed on core c_j in the k^{th} epoch. The next step updates the remaining values of the $\mathbf{S}(k)$ matrix with predictions of performance for different core types. Some approaches use a sampling-based method[3], in which every thread is periodically executed on all core types in order to collect performance and power statistics. This imposes a high overhead in the system. Instead, we employ a prediction-based approach. The key idea behind the prediction of the throughput matrix relies on the fact that the average throughput behavior of a thread on one core is correlatable to the throughput on another core (with same ISA and memory hierarchy) with a good degree of accuracy [9]. By collecting performance information of the thread in one core we can predict the throughput in the other cores. In Section 4.1, we identified a set of performance counters that can be used to characterize the workload being executed by a thread. Given the aforementioned set of counters, the performance of a thread t_i that has run on core c_j at the the k^{th} epoch can be predicted on every other core c_l, $\forall j \neq l$ as follows:

$$\widehat{ipc}_{ij}(k) = \mathbf{\Theta} * X^T_{ij} \quad (8)$$

such that $\widehat{ips}_{ij}(k) = \widehat{ipc}_{ij}(k) * F_j$ where $X_{ij}^T = [x_1^{ij}, x_2^{ij}, ..., x_N^{ij}]^T$ represents a characterization vector of the workload of thread t_i during epoch k-1. $\mathbf{\Theta}$ is the coefficient matrix which defines impact of each metric when predicting from core c_j of type $\gamma(c_j)$ to core c_l of type $\gamma(c_l)$. In order to obtain Θ, we employ standard linear regression using the least squares method in a similar approach to [1].

The final step of the estimate/predict phase of SmartBalance is to obtain the power characterization matrix $\mathbf{P}(k)$. We use the observation that the power p_{ij} a thread t_i is linearly correlated to its ipc_{ij} [18]. We use linear interpolation to relate the power consumption of the thread t_i running a core c_j of type $y = \gamma(c_j)$ to the predicted ipc_{ij} as:

$$\widehat{p}_{ij} = \alpha_1 * \widehat{ipc}_{ij} + \alpha_0 = \mathbf{\Theta_1} * X_{ij}^T + \alpha_0 \quad (9)$$

where α_0, α_1 are constants that provide the performance–power relationship for core type y and are obtained from offline profiling.

4.3 Thread Balancing and Allocation Problem

The thread mapping problem consists of finding an optimal allocation of threads $V = \{t_i, 1 \leq i \leq m\}$ on the cores $C = \{c_1, c_2, ..., c_n\}$ of each type $R = \{r_1, r_2, ..., r_q\}$ such that an objective or cost function is optimized. We call an assignment of all threads $V = \{t_1, t_2, ..., t_m\}$ to available cores $C = \{c_1, c_2, ..., c_n\}$ an *allocation* $\mathbf{\Psi(k)}$. An allocation as defined in Eq.(1) results in an objective or cost function to be optimized while accounting for the heterogeneity of processing elements and executing threads. An objective or a cost function for the allocation problem can be defined in several ways according the desired optimization goals. In this paper, we focus on **maximizing overall energy efficiency** (i.e., IPS/Watt or Instructions per Joule) as defined below:

$$\underset{\mathbf{\Psi(k)}}{Maximize} \quad (J_E) \quad (10)$$

$$J_E = \sum_{j=1}^{n} \omega_j \frac{IPS_j(k)}{P_j(k)} \quad (11)$$

where the objective J_E is the weighted sum of energy efficiency (IPS/watt) of all the cores, $IPS_j(k)$ can be calculated using Eq.(6) and $P_j(k)$ is given by Eq.(7) or the characterization matrices in Eq.(2) and Eq.(3). The weights ω_j are ideally set to 1, but can be tuned to give preference to certain cores or core types.

Optimization Methodology Finding the optimal thread allocation is an NP-hard combinatorial problem [10] that requires heuristics exploiting specific characteristics of the problem to achieve acceptable solutions within a reasonable amount of time. In our SmartBalance context, we have a further constraint of achieving online optimization at the start of each SmartBalance scheduling epoch (Fig. 2). We achieve this through the run-time optimization outlined in Algorithm 1 that deploys a modified online Simulated Annealing (SA)-based approach. SA has been been shown to produce nearly-optimal global solutions while accommodating problem-specific changes without significant modifications [10]. Furthermore, SA provides tunable parameters to trade-off computational complexity for solution quality (e.g. number of iterations and precision of probabilistic functions), enabling a run-time light-weight implementation. The tunable input parameters in Algorithm 1 can be used to trade-off computational complexity for solution quality (e.g. number of iterations and convergence speed). While a straightforward floating-point implementation of Algorithm 1 may lead to long execution times due to the high cost of computing the probabilistic functions, we use custom fixed-point implementations of $rand$ and e^x that trade-off performance with uniformity ($rand$) and precision (e^x) without significantly compromising the quality of the final solution. The computation of the objective function is also optimized by keeping track of previous computations and obtaining a new evaluation only by performing computations induced by the latest swap on Ψ. In Section 6.3 we show that our run-time optimization algorithm is able to find solutions online (at the beginning of each scheduling epoch) without imposing significant overheads on the operating system.

5. Experimental Setup and Implementation

We created an experimental simulation platform (shown in Fig. 3) comprising the heterogeneous cores described in Table 2 using the *Gem5* performance simulator [5]. *Gem5* includes cycle-accurate models for various CPU architectures, as well as peripherals models that allow us to run full system

Algorithm 1 Smart_Balance() Run-time Optimization

Input Parameter Options: Max. no. of iterations Opt_{max_iter}, perturbation schedule $Opt_{\triangle perturb}$, solution acceptance rate $Opt_{\triangle accept}$, initial perturbation $Opt_{perturb}$ and acceptance rate $Opt_{\triangle accept}$
Input Data: Throughput Matrix $\mathbf{S}$, Power Matrix $\mathbf{P}$ (core dynamic and leakage power vectors P_D and P_L), thread utilization vector $\mathbf{U}$, initial allocation $\mathbf{\Psi}_0$
Output: Allocation $\mathbf{\Psi}$

0: $\mathbf{\Psi} \leftarrow \Psi_0$ {Implemented as uni-dimensional array}
0: $iteration \leftarrow Opt_{max\_iter}$
0: $perturb \leftarrow Opt_{perturb}$
0: $accept \leftarrow Opt_{accept}$
0: **while** $iteration > 0$ **do**
0: $\quad \mathbf{\Psi'} \leftarrow \mathbf{\Psi}$
0: $\quad pos \leftarrow randi(0, n * m)$ {permute for new index position}
0: $\quad pos\_new \leftarrow pos + \sqrt{perturb} * randi(-pos, n * m - pos)$
0: $\quad$ swap($\mathbf{\Psi'}, pos, pos\_new$)
0: $\quad diff \leftarrow J_E(\mathbf{S}, \mathbf{P}, \mathbf{\Psi'}) - J_E(\mathbf{S}, \mathbf{P}, \mathbf{\Psi})$
0: $\quad$ **if** $diff > 0$ **then** {Always accept if new solution is better than previous}
0: $\quad\quad \mathbf{\Psi} \leftarrow \mathbf{\Psi'}$
0: $\quad$ **else**{Accept worse solution with a probability proportional to $accept$}
0: $\quad\quad probability \leftarrow e^{\frac{-diff}{accept}}$
0: $\quad\quad$ **if** $randi()$ mod $\frac{1}{probability} = 0$ **then**
0: $\quad\quad\quad \mathbf{\Psi} \leftarrow \mathbf{\Psi'}$
0: $\quad\quad$ **end if**
0: $\quad$ **end if**
0: $\quad iteration \leftarrow iteration - 1$ {Update perturbation and acceptance rate}
0: $\quad perturb \leftarrow perturb * Opt_{\Delta perturb}$
0: $\quad accept \leftarrow accept * Opt_{\Delta accept}$
0: **end while**
$\{randi()$ generates an uniformly distributed integer number in the interval $[0, 2^{32})$, while $randi(x, y)$ generates a number in the interval $[x, y)\}$ =0

simulations. We use *Gem5* to create heterogeneous cores by changing the architectural simulation parameters of the original Alpha 21264 superscalar architecture. Our approach is not limited by the voltage and frequency of the cores, however to show the effect of architectural heterogeneity, we fix all cores' voltages and frequencies to predefined values. All L1 and L2 caches are private and the cores are connected to the main memory through a shared bus. For obtaining power data, we integrated the *McPAT* power model [17] directly with the *Gem5* simulation framework, which allows us to obtain power sensor data at runtime. *Gem5* is also extended with a *sensing interface* which exports *McPAT* power information and other *Gem5* statistics (specifically the hardware counters) to the kernel at run-time.

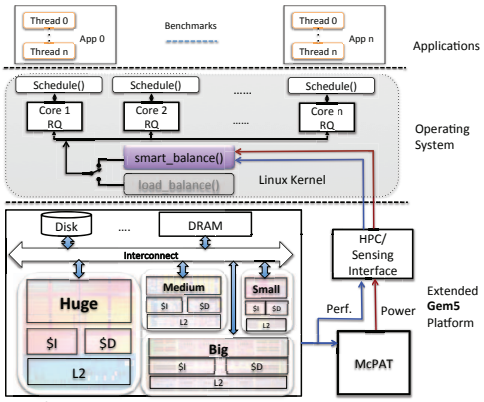


Figure 3: SmartBalance experimental platform using extended Gem5.

5.1 SmartBalance Implementation

SmartBalance replaces Linux's existing load balancing mechanism via *smart_balance()*. In the vanilla kernel, load balancing is triggered by the function `rebalance_domains()`. We have reimplemented this function in order to call SmartBalance instead of the standard load balancing according to the epoch length. We sample performance counters on a thread-by-thread basis. The sampling of these counters is done at the granularity of Linux's `schedule()` function. At the beginning of each epoch, the values of all thread-counters are collected and the performance and power matrices are built for use in the subsequent phases. Values that are unavailable are

predicted as described in Section 4.2. It is worth mentioning that SmartBalance can optimize both the user and kernel threads jointly. Since the impact of the user level threads dominates that of the kernel threads, we focus on the user-level threads by identifying and marking them during their creation in the `sched_fork()` function. We assume that each user thread is allowed to run on any core, however special constraints can easily be included by modifying the objective function in Eq. (11). Once a new allocation Ψ is found, threads may be migrated to balance the system according to the new allocation. The migration process is performed using the `set_cpus_allowed_ptr()` function already provided by the kernel.

Table 3: Benchmarks and their Mixes

PARSEC Mixes	Mix1	Mix2	Mix3	Mix4	Mix5	Mix6
	$x264_n$crew $x264_n$bow	$x264_t$crew $x264_t$bow	$x264_t$crew $x264_n$bow	$x264_n$crew $x264_t$bow	Bodytrack $x264_n$crew	Bodytrack $x264_n$crew $x264_t$bow

6. Experimental Results

The objective of our experiments is to evaluate the energy efficiency and effectiveness of the SmartBalance Linux kernel w.r.t the baseline vanilla Linux kernel and the state-of-the-art ARM GTS policy [2] using a varied sets of benchmarks and their mixes. Our first set of experiments shows results for generic HMP architectures with four core types, while our second set of results in section 6.1 performs a comparison with state-of-the-art ARM GTS policy for big.Little architectures with two core types. In order to do a comprehensive evaluation of SmartBalance, we use PARSEC benchmarks and their combinations for different levels of parallelization (2,4,8 threads) as shown in Table 3. We select a set of multithreaded benchmarks that have diverse characteristics (compute and memory intensive) from the PARSEC [4] benchmark suite as representative applications as well as create sets of synthetic benchmarks with attributes that reflect interactive/IO dependent applications.

We call these sets of multithreaded synthetic benchmarks *interactive microbenchmarks* (IMB) that provides the ability to control the load, phasic behavior, and interactivity (sleep and wait periods). The IMBs can be configured to have throughput (T) and interactivity (I) that controls the sleep/wait periods for high (H), medium(M), and low(L) values. As an example, HTHI corresponds to the high throughput and high interactiveness of the IMB configuration and all the other 8 combinations are similarly labeled in our experiments. Note that we use the x264 benchmark with different configurations (high(H)/low(L) frame processing rate) and input videos (crew/bowing) to show that a single benchmark can have different characteristics (both in IPS and power) as shown in Table 3.

Fig. 4 shows the energy efficiency of the SmartBench approach for different sets of benchmarks when the benchmarks are run using the *Gem5* experimental platform in Fig.3(a) in full system mode in a cluster computing environment. We observe that the SmartBalance kernel performs 50.02 % on average better with the interactive benchmarks (Fig.4(a)), 52 % with the PARSEC benchmarks (Fig. 4(b)) and their mixes respectively when running 2, 4, and 8 threads of each benchmarks. Overall, SmartBalance Linux kernel achieves an energy efficiency of over 50 % across all the benchmarks in comparison to the vanilla Linux kernel.

6.1 Comparison with state-of-the-art

SmartBalance provides a simple yet tunable runtime optimization engine that overcomes the limitations of balancing scheme generality for emerging HMPs with several core types. In order to demonstrate the relative advantage of the SmartBalance approach even for specific HMP architectures such as ARM big.Little [13] with two core types, we compare our approach with that of state-of-the-art ARM's global task scheduling policy[2]. We create an octa-core big.Little HMP using *Gem5* and modify the Linux 2.26.x kernel to implement the ARM GTS policy. Note that the GTS policy works only for big.Little type of HMP as the policy makes a fixed utilization threshold-based binary decision to either select *a big* or *a little* core [2]. In fact, the lack of joint per-thread (finer granularity) and per-core accurate power as well as performance awareness, limits GTS from achieving (near) optimal energy efficiency by as much as ˜20 % in comparison to SmartBalance as shown in Fig. 5 for several benchmarks. On the contrary, instead of using core utilization as a proxy for deciding thread allocation for energy efficiency, SmartBalance uses accurate energy efficiency *measurements directly* –both at fine and coarse granularity during thread balancing to provide additional improvements while providing a generic and fairly scalable solution as shown in Section 6.3.

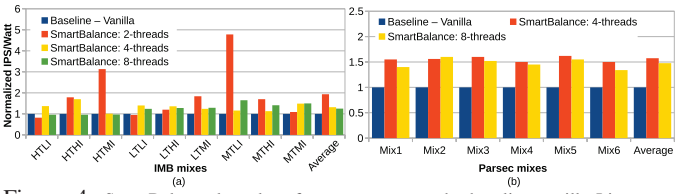


Figure 4: SmartBalance kernel performance w.r.t. to the baseline vanilla Linux kernel (running the same benchmarks with same no of threads) using (a) interactive microbenchmarks (b) PARSEC benchmark and their mixes. The optimization goal is set to maximize overall energy efficiency (IPS/Watt).

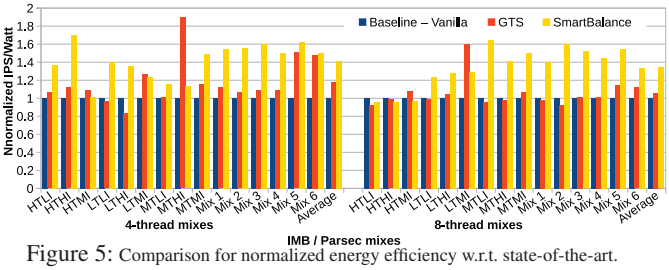


Figure 5: Comparison for normalized energy efficiency w.r.t. state-of-the-art.

6.2 Estimation and Prediction Evaluation

The runtime prediction of performance and power incurs an average error of 4.2 % and 5 % respectively as shown in Fig.6 corresponding to the predictor coefficient matrix Θ in Table 4. The average percentage error is about 70% smaller than the one reported by [1], which proposes a similar predictor. The predictor in [1] doesn't include the share of memory and branching instructions in its expression, which might explain the higher average error. The work of [19] has also employed a similar performance predictor based on the offline profiling of PARSEC benchmarks. However, differently from our work, [19] implements a binning approach in which each thread is categorized as their nearest neighbor in the PARSEC benchmarks. Compared to [19], our approach yields 71% smaller average error.

Table 4: Predictor coefficient matrix

Predictor IPC	FR^*	$mr_{\$i}$	$mr_{\$d}$	I_{msh}	I_{bsh}	mr_b	mr_{itlb}	mr_{dtlb}	$ipc_{i,src}$	$const$
Huge-¿Big	-0.006	-2.110	-0.135	-0.975	1.213	0.385	-2.928	-9.552	0.416	0.691
Huge-¿Medium	-0.007	-3.610	-0.106	-0.751	0.906	0.841	-0.881	-9.661	0.130	0.668
Huge-¿Small	-0.005	1.786	1.139	-2.017	0.125	0.035	4.958	-8.913	0.142	0.657
Big-¿Huge	-0.007	1.870	0.393	1.787	-2.213	-1.134	0.000	5.717	2.084	-1.017
Big-¿Medium	-0.003	-2.286	-0.057	-0.416	0.455	1.351	0.000	-5.173	0.326	0.415
Big-¿Small	-0.004	-0.046	1.394	-1.235	0.607	1.060	0.000	-23.62	0.194	0.432
Medium-¿Huge	-0.053	3.064	-0.385	3.882	-3.987	-6.458	-249.4	7.210	4.503	-1.835
Medium-¿Big	-0.016	2.145	0.025	1.401	-1.460	-1.739	-6.416	9.666	2.731	-0.987
Medium-¿Small	-0.008	0.995	1.080	-0.258	0.105	2.719	208.5	3.290	1.040	-0.467
Small-¿Huge	0.042	-1.835	-9.291	6.021	-1.238	-19.81	0.000	104.00	2.745	0.235
Small-¿Big	0.003	-0.630	-4.087	1.185	-0.277	-9.902	0.000	39.354	1.545	0.829
Small-¿Medium	-0.001	-1.035	-1.122	-0.517	0.357	-2.639	0.000	2.639	0.594	0.776

$^*FR$ if the ratio of the frequencies of source to target core

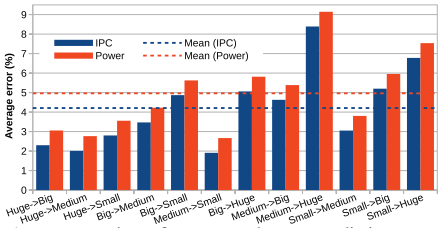


Figure 6: Average error in performance and power prediction across PARSEC.

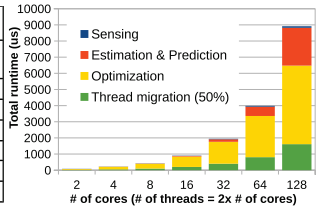
6.3 Overheads and Scalability

Fig. 7 shows average runtime measurements for each SmartBalance phase on our 4-core platform as well as extrapolated results for systems scaling from 2 to 128 cores with 4 to 256 threads. Most of the overhead originates from the optimization algorithm and thread migration (assuming 50% of the threads are migrated). However, for typical embedded platforms (e.g., quad-core mobile devices) with 2 to 8 cores, the average overhead of using SmartBalance is negligible with respect to the 60ms epoch length (less

than 1%). Recall that we used the runtime light-weight SA-based optmization described in Section 4.3, which enables us to control the overhead. As shown in Fig. 8(a), for larger configurations we limit the number of iterations to avoid excessive overhead, therefore trading off solution quality for scalability.

SmartBalance phases runtime (µs)				
Number of threads	Sensing	Estm. & Pred.	Optim.	Thread Migr.
4	9.4	4.5	161.3	50.4
8	11.9	12.0	295.9	100.8
16	13.5	20.0	420.0	201.6
32	15.3	28.0	530.0	403.2
64	16.8	36.8	645.0	806.4
128	18.5	55.0	750.0	1612.8

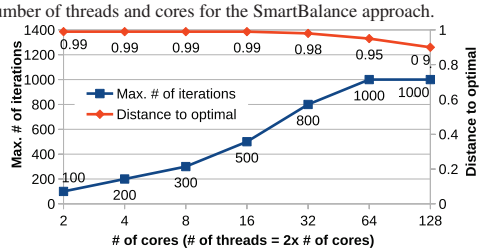
(a)



(b)

Figure 7: (a) Overhead with the quad-core HMP and (b) scalability analysis with increasing number of threads and cores for the SmartBalance approach.

(a)



(b)

Opt parameter	perturb	Δperturb	accept	Δaccept
Value	3670	0.98	28	0.91

Figure 8: Maximum number of iterations (Opt_{max_iter} parameter) for each scalability scenario (a). The *distance to optimal* is obtained by running our optimization algorithm for synthetic cases whose optimal solution is known. (b) shows the values used for the remaining optimization parameters.

6.4 Limitations and future work

One limitation of the SmartBalance approach may be argued to be the dependence on additional counters and sensors for fine-grained awareness of performance and power. We used as many as 10 counters and per-core power sensors for predictions. Although this may be viewed as a serious limitation on certain architectures, current trends and future projections suggest the inclusion of per-core power sensors, many counters, and on-chip monitors already in several existing platforms [16, 22]. For example, the recent Samsung Exynos big.Little board [22] already has power sensors for each core, GPU, and DRAM as well as extensive support for all the counters in the Linux kernel using oprofile [8]. In addition, a sparse virtual sensing mechanism [24] guaranteeing a minimal number of counters and sensors can be used to overcome this perceived limitation.

7. Conclusion

Single-ISA heterogeneous MPSoCs with architecturally differentiated cores are an attractive computing paradigm achieving higher performance and energy efficiency compared to the homogeneous counterparts. However, aggressively heterogeneous MPSoCs pose a serious challenge to the traditional Linux operating system scheduling as existing load balancers do not fully exploit widely heterogeneous architectural configurations (core types, core strength, and their combinations) and are not capable of adapting to new processor architectural changes without significant engineering efforts. In this paper we proposed SmartBalance: a closed-loop approach to load balancing under a sense-predict-balance paradigm that can efficiently manage the chip resources while opportunistically exploiting the diverse workload and performance-power characteristics of different cores for energy efficiency. Our SmartBalance approach deploys an efficient run-time optimization executed at the beginning of each scheduling epoch (covering multiple Linux scheduling periods) that effectively predicts the per-thread power and performance characteristics for each core, allowing efficient online load balancing. Our approach shows an improvement in energy efficiency of over 50% for a Heterogeneous MPSoC with 4-cores executing benchmarks from the PARSEC benchmark suite, as compared to the standard vanilla Linux kernel load balancer and over 20% improvement w.r.t. state-of-the-art ARM's global task scheduler (GTS).

Acknowledgment

This work was partially supported by the NSF award CCF-1029783 (Variability Expedition).

References

[1] A. Annamalai et al. An opportunistic prediction-based thread scheduling to maximize throughput/watt in AMPs. In *PACT'13*, Sept 2013.

[2] ARM. big.LITTLE technology: The future of mobile. 2013. http://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf.

[3] M. Becchi et al. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *CF '06*, pages 29–40. ACM, 2006.

[4] C. Bienia et al. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT'08*, pages 72–81. ACM, 2008.

[5] N. Binkert et al. The Gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[6] S. Borkar et al. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[7] J. Chen et al. Efficient program scheduling for heterogeneous multi-core processors. In *DAC '09*, pages 927 –930, july 2009.

[8] W. E. Cohen. Tuning programs with oprofile. 2004. http://oprofile.sourceforge.net/news/.

[9] K. V. Craeynest et al. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). ISCA'12, 2012.

[10] K. Deb. Multi-objective optimization using evolutionary algorithms. *John Wiley & Sons, Ltd, Chichester, England*, may 2001.

[11] P. Greenhalgh. Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7: Improving energy efficiency in high-performance mobile platforms. Technical report, ARM Ltd., 2011.

[12] G. Grey. big.LITTLE software update. http://www.linaro.org/blog/hardware-update/big-little-software-update/, 2013.

[13] B. Jeff. Advances in big.LITTLE technology for power and energy savings. Technical report, ARM Ltd., 2012.

[14] M. Kim et al. Utilization-aware load balancing for the energy efficient operation of the big.little processor. In *DATE'14*, March 2014.

[15] R. Kumar et al. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA'04*, june 2004.

[16] C. R. Lefurgy et al. Active guardband management in POWER7+ to save energy and maintain reliability. *Micro, IEEE*, 33(4):35–45, 2013.

[17] S. Li et al. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO-42*, pages 469–480, 2009.

[18] T. Li et al. Run-time modeling and estimation of operating system power consumption. *SIGMETRICS Perform. Eval. Rev.*, 31(1):160–171, June 2003.

[19] G. Liu et al. Dynamic thread mapping for high-performance, power-efficient heterogeneous many-core systems. In *ICCD'13*, Oct 2013.

[20] J.C. Mogul et al. Using asymmetric single-isa cmps to save energy on operating systems. *Micro, IEEE*, 28(3):26 –41, May-June 2008.

[21] NVidia. Variable SMP - a multi-core CPU architecture for low power and high performance. 2011.

[22] Oroid. Odroid-xu3 single board computer. 2014. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127&tab_idx=2.

[23] M. Poirier. In Kernel Switcher: A solution to support ARM's new big.LITTLE technology. https://events.linuxfoundation.org/images/stories/slides/elc2013_poirier.pdf, 2013.

[24] S. Sarma et al. Minimal sparse observability of complex networks: Application to mpsoc sensor placement and run-time thermal estimation amp; tracking. In *DATE'14*, pages 1–6, March 2014.