Chrysso: An Integrated Power Manager for Constrained Many-Core Processors

Sudhanshu Shekhar Jha
Universitat Politécnica de Catalunya, Spain

Wim Heirman
Intel Corporation
Belgium

Ayose Falcón
HP
Spain

Trevor E. Carlson
Uppsala University
Sweden

Kenzo Van Craeynest
Ghent University
Belgium

Jordi Tubella
Universitat Politécnica de Catalunya, Spain

Antonio González
Universitat Politécnica de Catalunya, Spain

Lieven Eeckhout
Ghent University
Belgium

ABSTRACT

Modern microprocessors are increasingly power-constrained as a result of slowed supply voltage scaling (end of Dennard scaling) in conjunction with the transistor density scaling (Moore's Law). Existing many-core power management techniques such as chip-wide/per-core DVFS, and core and cache adaptation are quite effective in isolation at moderate to high power budgets. However, for future many-core chip, the existing techniques do not scale well to large core counts, small time slices and stringent power budgets. We need a new solution that combines different adaptation and reconfiguration techniques.

In this paper, we present Chrysso, an integrated, scalable and low-overhead power management framework. Chrysso consists of a three-step process: leveraging simple analytical performance and power models, pruning the search space early using local Pareto front generation, followed by global utility-based power allocation. This ensures scalable and effective dynamic adaptation of many-core processors at short time scales along multiple axes, including core, cache and per-core DVFS adaptations. By integrating multiple power management techniques into a common methodology, Chrysso provides significant performance improvements over isolated mechanisms within a given power budget without power-gating cores. On a 64-core system, Chrysso improves system throughput by 1.6× and 1.9× over core-gating at stringent power envelops for multi-program (SPEC) and multi-threaded (PARSEC) workloads, respectively.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Adaptable architectures

General Terms

Design, Performance, Experimentation

Keywords

Many-core processor, analytical modeling, power management, microarchitecture reconfiguration

1. INTRODUCTION

Modern microprocessors are quite power-constrained as a result of prominent trends in chip technology. Moore's Law [32] refers to the doubling of transistors on chip every 18 months, and has been a fundamental driver of computing. Unfortunately, because of the end of Dennard scaling [9] (slowed supply voltage scaling), we may become so power-constrained that we will no longer be able to power on all transistors at the same time—a problem referred to as *dark silicon* [13]. Moreover, run-time factors such as thermal emergencies [7] and power capping [16] further constrain the available chip power. Hence, an intelligent solution is needed to selectively power on transistors to maximize performance within a given power budget at any given time.

A number of mechanisms exist to manage power, including (DVFS) [21], core adaptation [3, 17, 35], and cache adaptation [1, 39]. Although these mechanisms are quite effective at managing power in isolation at high to moderate power budgets, systems had to revert to core-gating under constrained conditions [26, 27]. The stringent power requirements for future many-core systems therefore require an integrated, scalable and low-overhead approach that can select the best *combination* of power savings methods for each core tailored to each application phase, at small time scales. This includes dynamically re-allocating the power budget between cores (e.g., a compute-bound core should be allowed to 'steal' power from a memory-bound core), and, for multi-threaded applications, taking into account which threads are performance-critical to avoid wasting energy by speeding up non-critical threads. While many of these problems have been studied in isolation, combining all these requirements is essential yet quickly leads to an explosion in complexity of the power management algorithms.

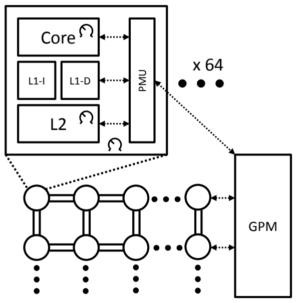


Figure 1: Adaptive many-core architecture with Chrysso, featuring per-core PMU and a GPM.

In this paper, we propose Chrysso[1], an integrated many-core power management methodology to quickly adapt the processor to workload characteristics and run-time conditions. Chrysso leverages analytical performance models and table-based power models to explore the complex optimization space comprising core, cache and per-core DVFS adaptations. Chrysso can operate at small time slices (10 ms) while simultaneously scaling to large core counts (multiple tens of cores). Chrysso exploits both inter-workload variability as well as intra-workload phase behavior to optimize power-performance trade-off over time under varying workload conditions.

Overall, we make the following contributions in this paper:

- We propose a novel two-tier local/global optimization algorithm, combining local prediction and Pareto front selection with a global utility-based power distribution, to quickly search the adaptation space and partition the available power budget among cores.
- We integrate the concept of thread criticality into the power allocation to avoid wasting energy by speeding up non-critical threads.
- We evaluate Chrysso on a many-core processor and show that our integrated approach achieves optimized performance while meeting stringent power constraints. Chrysso's low complexity enables adaptation at fine granularity.

On a 64-core system, Chrysso outperforms DVFS, core-gating, core and cache adaptation in isolation by a significant margin over a broad range of power envelops without power-gating any cores. Chrysso improves system throughput by 1.6× and 1.9× on average over core-gating for multi-program and multi-threaded workloads, respectively. Chrysso incurs little execution time overhead—less than 1%, while requiring limited hardware overhead—roughly 3 KB per core to assist adaptation decisions. This makes it scalable both in time (small time scales) and space (large core counts).

2. CHRYSSO

We propose Chrysso in the context of a many-core processor, as shown in Figure 1. Each core has a number of configuration knobs that together define distinct operating points, each with a different power-performance trade-off. A per-core power/performance monitoring unit (PMU) keeps track of core activity and controls the core configuration in response to requests made by the global power manager (GPM).

[1]Chrysso is a spider genus whose color is variable. By analogy, Chrysso adapts the many-core processor configuration to variable workload and run-time conditions.

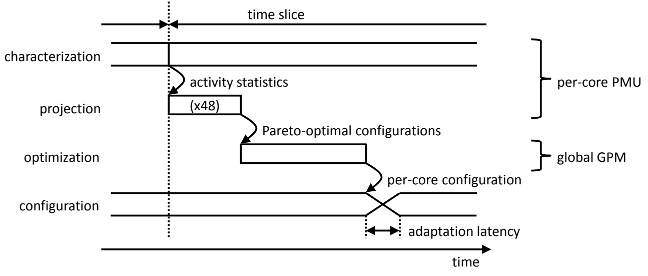


Figure 2: Event flow in Chrysso.

This GPM combines information from all cores, and performs the global power/performance optimization. By being knowledgeable about differences in per-core behavior, the available power budget can be dynamically distributed across cores.

Figure 2 depicts the Chrysso event flow. Time is divided into fixed-sized time slices, typically few milliseconds. During each time slice, per-core PMU tracks the activity statistics using the hardware counters. At the end of a time slice, the PMU uses analytical performance models along with table-based power models to predict/project the performance and power of all possible core reconfigurations (48 configurations in our setup). Each core then sends a list of Pareto-optimal configurations to the GPM, which globally optimizes the many-core configuration within the given power budget. Finally, the GPM instructs each core to reconfigure itself based on the configuration that was decided upon. Using a 10 ms time slice, we find that the Chrysso event flow has low overhead (less than 1%).

2.1 Chrysso Optimization Algorithm

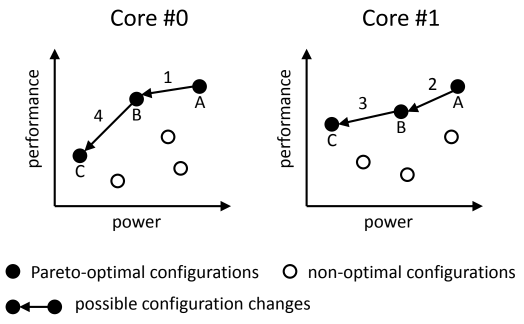
The aim of Chrysso's optimization algorithm is to find a combination of per-core configuration settings that maximizes performance within the imposed chip-wide power budget. The search space is multi-dimensional with local minima that is not trivially navigated. However, by combining per-core Pareto frontier generation with a *utility*-based power budget allocation, Chrysso can quickly navigate this search space and converge to a (near-)optimal configuration at small time slice. Chrysso's scalability and adaptivity depends on three key features: *local Pareto front generation*, *utility-based optimization* and *critical-thread awareness.*

Per-core prediction and Pareto front generation

Chrysso first predicts performance and power for all possible configuration tuples (w, c, f) for each core, with w denotes the resized core micro-architecture, c the number of cache-ways enabled, and f the core frequency-voltage setting. This is done by the PMUs for the respective cores using the projection models which we discuss in detail in Section 2.2. Once per-core PMU has computed the projected performance and power values for each possible configuration, it discards all non Pareto-optimal configurations. This significantly reduces the number of configurations that have to be taken into account in the global optimization round: out of 48 per-core configurations, typically only 6–12 are Pareto-optimal, depending on the workload characteristics. These configuration points are then sent to the GPM for global optimization.

Utility-based optimization

The global optimization algorithm uses the current many-core configuration as a starting point. As long as the projected power consumption of the current core configuration exceeds



Core #0
Core #1
performance
power
A
B
C
1
2
3
4
● Pareto-optimal configurations
○ non-optimal configurations
possible configuration changes

Figure 3: Chrysso algorithm: Pareto front selection followed by utility-based optimization.

the power budget, cores are successively selected to reduce the micro-architectural configuration. All settings occur only along each core's Pareto frontier—by using Pareto-optimal points only, we have already linearized the per-core selection process without loss in optimality.

At each iteration, the core that is selected to perform a *step-down* (move to a lower-power configuration) is the one that will provide the highest *utility*—this is the core that can achieve the highest reduction in power consumption while loosing least amount of performance along the core's Pareto frontier. To speed up the algorithm and to reduce the chance of ending up in local minima, cores can also be selected to *step-down* by multiple steps at once. The algorithm works by constructing a list of down-steps, between 1 and K steps along the Pareto frontier for each core. It then sorts these steps by *utility.* The first *step-down* in this list is applied to the current configuration, after which the algorithm performs the next iteration with another *step-down* as long as the predicted power consumption exceeds the available power budget. Once the power limit has been reached, or if the initial configuration falls below the power limit, the *step-up* phase of the algorithm starts in which cores can be stepped up. This allows Chrysso to take up any spare power budget that was made available by the last *step-down* phase, if any.

Figure 3 illustrates this process: for two cores #0 and #1, it shows the various configurations with the Pareto-optimal design points shown as black dots. Down-steps are selected based on utility, i.e., by the largest reduction in power for the least reduction in performance: Chrysso selects down-steps 1, 2, 3 and 4 for cores #0, #1, #1, and #0, respectively.

Critical thread awareness

The utility-based global optimization strategy as just described allows for giving a larger share of the total power budget to cores/threads that benefit more than others. In a multi-program workload environment this helps overall system performance by giving a relatively larger power budget to compute-intensive applications over memory-intensive applications. For multi-threaded workloads, not all threads have equal impact on system performance (application runtime): speeding up threads that are on the critical path of the application improves run-time while allocating power to speed up non-critical threads is inefficient—unless they are slowed down too much at which point they become critical.

Chrysso leverages the notion of thread criticality proposed in [10] to compute thread criticality and allocate power accordingly. The criticality value quantifies how much time a thread is performing useful work (active/running) and how many threads are concurrently waiting within a given time slice. Intuitively, a thread that is active while other threads are waiting due to synchronization is more critical

Knob	Parameter	Values			
Core (w)	Width	1	2	3	4
	ROB size	16	32	64	128
	RS entries	4	8	16	32
	LQ entries	6	12	24	48
	SQ entries	4	8	16	32
Cache (c)	L2 cache ways	4	8	12	16
	Capacity (KB)	128	256	384	512
DVFS (f)	Frequency (GHz)	0.8	1.0	1.2	—
	Vdd (V)	0.7	0.75	0.8	—

Table 1: Configuration knobs and corresponding architectural parameters and values.

and therefore receives a larger criticality value. We make Chrysso critical-thread aware by identifying the thread with the largest criticality value and we prevent the core running this thread from entering the *step-down* phase. Instead, during the *step-up* phase it gets allocated a higher power budget whenever possible. This way we achieve a more balanced execution, and hence reduce overall run-time. Note that we recompute the criticality for all threads in each time slice, and we thus dynamically determine the most critical thread and repartition the power budget accordingly. Computing thread criticality can be done in hardware with minimal overhead (65 bits per core [10]) or using an OS kernel.

2.2 Chrysso Projection Models

Chrysso uses model-based prediction to explore the optimization space at run time. This allows Chrysso to scale much more easily to architectures with multiple configuration knobs, while incurring little run-time overhead, in contrast to sampling-based methods as done in prior work [3, 17, 35]. The inputs to the models are the activity statistics collected using hardware counters during the current time slice. Assuming workload behavior is at least representative for the next time slice, this information is then used to predict performance and power for all possible core configurations (w, c, f) during the next time slice.

Performance modeling

Performance is projected based on the notion of a CPI stack, which breaks down the average number of cycles executed per instruction into individual CPI components representing cycles 'lost' due to branch and memory stalls, in addition to a *base* component.

$$CPI = CPI_{core} + CPI_{mem} \tag{1}$$

$$CPI_{core} = CPI_{base} + CPI_{branch} \tag{2}$$

$$CPI_{mem} = CPI_{L1} + CPI_{L2} + CPI_{dram} \tag{3}$$

To account for the frequency changes, we use the seconds per instruction (SPI) metric. SPI is a function of CPI and clock frequency f:

$$SPI = CPI/f \tag{4}$$

We use the CPI stack of the $Current$ (C) configuration to predict performance of other configurations by rescaling individual CPI stack components. In the discussion to follow, we will compute the performance of a $Target$ (T) configuration tuple (w^T, c^T, f^T), given performance information obtained from the previous slice which ran at the $Current$ (C) configuration tuple (w^C, c^C, f^C). As mentioned in Section 2.1, we consider three degrees for adaptation: within the core, the last-level cache, and through per-core DVFS (see Table 1). We now describe in detail the performance models for each of these adaptations.

Core knob. The core knob affects the core's execution width and the size (quadratically with core width) of various supporting structures (ROB, reservation station, load and store queues). None of the memory-related components nor the branch predictor are assumed to be affected, so their CPI components are kept constant. We observe in our experiments that changing the core width along with the respective buffers has a saturating effect on ILP. Using the relation between processor width and ILP, the effect of changing the core knob is predicted by:

$$CPI_{base}^{T} = CPI_{base}^{C} \cdot \frac{w^{C}}{w^{T}} \tag{5}$$

Cache knob. Cache projections are based on data obtained from auxiliary tag directories (ATDs) [36], which we use to estimate what the cache miss rate would be for each of the possible $Target$ (T) configurations. To gauge the performance impact incurred by a change in miss rate, we assume constant DRAM access time and memory-level parallelism. The CPI_{dram} component can therefore be assumed to scale with the miss rate estimated by the ATDs:

$$CPI_{dram}^{T} = CPI_{dram}^{C} \cdot \frac{ATD(c^{T})}{ATD(c^{C})} \tag{6}$$

When the current miss rate $ATD(c^{C})$ is zero—for application phases in which the working set fits in c^{C}—we avoid a division by zero by assuming a fixed cost per DRAM access and estimate CPI_{dram}^{T} by multiplying the (uncontended) DRAM latency with the number of expected LLC misses.

DVFS knob. Changing the core's clock frequency affects the behavior of the core itself as well as that of the L1 and L2 caches—which in our architecture are tied to the core clock. The speed of operations taken by the core or caches, when measured in clock cycles, will therefore not change. In contrast, DRAM access latency will stay constant when measured in absolute time. Thus, we can predict total core performance as:

$$SPI^{T} = (SPI_{core} + SPI_{L1} + SPI_{L2}) \cdot \frac{f^{C}}{f^{T}} + SPI_{dram} \tag{7}$$

Putting it all together. To estimate performance (in instructions per second, IPS—inverse of SPI) for a configuration of interest, Equations 5–7 are first applied to rescale the performance components of the current configuration to the new configuration; subsequently, application of Equations 1–4 yields a prediction for overall core performance. To dampen modeling errors, we perform an extra correction step by rescaling the calculated *IPS* by the ratio of *measured* IPS versus *estimated* IPS for the $Current$ (C) configuration.

$$IPS_{est.-scaled}^{T} = IPS^{T} \cdot \left(\frac{IPS_{measured}^{C}}{IPS^{C}}\right) \tag{8}$$

Power modeling

Power consumption prediction uses a table-based approach. For each core configuration (w, c, f), the table contains its static power consumption P_{static}, its dynamic energy consumption per instruction (E_{instr}), and its dynamic energy consumption per L2 cache access (E_{L2}). This look-up table is populated 'at the factory' through off-line analysis averaged over a broad set of applications, and can be part of the existing calibration and binning process of new chips.

To estimate power consumption for the next time slice, an estimate is needed for the dynamic instruction count and cache access count during the next time slice. The dynamic instruction count is estimated by multiplying the time slice length with the projected IPS; the cache access count is computed by scaling the dynamic instruction count with the cache access rate of the current time slice:

$$Icount^{T} = time\ slice \cdot IPS_{est.-scaled}^{T} \tag{9}$$

$$L2access^{T} = (L2access^{C} / Icount^{C}) \cdot Icount^{T} \tag{10}$$

Computing projected power is simply done by multiplying the projected instruction and L2 access counts with their respective energy costs, divided by the time slice length, and added to the static power for a given configuration T.

$$P_{est.}^{T} = P_{static}^{T} + \frac{(E_{instr}^{T} \cdot Icount^{T}) + (E_{L2}^{T} \cdot L2access^{T})}{time\ slice} \tag{11}$$

This estimate is subsequently corrected by rescaling it with the ratio of *measured* versus *estimated* power for the current configuration:

$$P_{est.-scaled}^{T} = P_{est.}^{T} \cdot (P_{measured}^{C} / P_{est.}^{C}) \tag{12}$$

$P_{est.}^{C}$ is computed using Equation 11 with the table values corresponding to the $Current$ (C) configuration, while $P_{measured}^{C}$ is obtained from hardware energy counters.

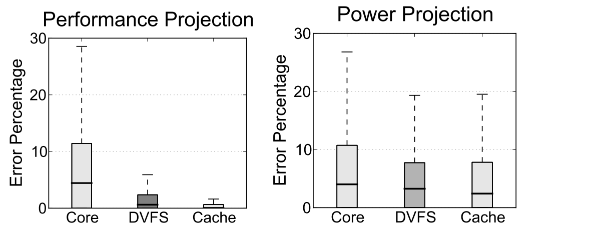


Figure 4: Per-knob modeling error of Chrysso's performance (*left*) and power (*right*) projection models.

Accuracy versus complexity

Note that for each of these knobs, more elaborate models can be constructed that may reduce modeling error. For instance, by taking average dependency distance into account, the saturating effect of processor width on ILP extraction could potentially be estimated. However, more complex models would incur more overhead for both collecting the required statistics and for running the projection models—this would compromise scalability for many-core system, while not necessarily yielding better scheduling decisions. Figure 4 plots the modeling error (using the evaluation methodology outlined in Section 3). While the absolute errors can be significant, we will later compare reconfiguration decisions based on these projections with those based on idealized models, and show that these simple models still allow the right scheduling decisions to be made.

Hardware support

Core-level projections rely on hardware support for collecting CPI stacks. On out-of-order cores, hardware collection can be complicated because of various overlap effects between miss events. Recent commercial processors such as the IBM Power5 [30] and current generation Intel processors [20] have support for computing memory stall components, making most of the required information already available.

Cache projections are based on data obtained from auxiliary tag directories (ATDs) [36], which keep track of what the cache miss rate would be given each configuration setting.

Since we use selective ways, only a single array of tags has to be maintained per cache set, corresponding to the largest possible configuration. Assuming LRU replacement policy, an access would be a hit when the cache was configured to be M ways *iff* its LRU position is less than M (with 0 being MRU and $M-1$ being LRU in an M-way cache). Set sampling can be employed to reduce hardware overhead with minimal impact on accuracy. In our experiments, we sample 32 randomly selected sets out of 512 sets in the LLC, incurring an overhead of 2,688 bytes per core[2].

For power projections, we employ simple models that predict the relative difference between the current configuration and other configurations of interest. Input to the models are activity statistics that count the total number of instructions and the number of LLC accesses. In addition, the current power consumption is used as a correction factor, which can be obtained from energy counters as available in current generation Intel processors [37]. The power characterization table itself could either be populated using data obtained at design time, or be filled in with per-core specific values after chip fabrication to take into account process variation. The table consists of three 16-bit numbers per configuration; for 48 configurations this amounts to 288 bytes of storage.

In summary, hardware overhead required by Chrysso is limited. Either the required information is already available in existing systems; or can be obtained at low cost—less than 3 KB per core (ATDs, power table, and critical thread calculation).

Complexity and scalability

The computational cost of Chrysso reconfiguration is composed of two parts: projection models and global optimization. The projection models amount to less than 1,000 operations to be implemented using fixed-point arithmetic and run on each core's PMU, in parallel with normal execution. Filtering for Pareto-optimal points is done on the PMU as well, resulting in 6–12 points in practice to be sent to the GPM. The global optimization algorithm has complexity $O(N \log N)$, where N is the number of cores. In our experiments, we observe that in most cases less than half of the cores need a configuration change at any point in the execution, the adaptation is typically a minor modification from the current configuration. This further reduces the effective complexity of the global step.

3. EXPERIMENTAL SETUP

3.1 Methodology

Performance simulator. We use a modified version of the Sniper multi-core simulator [8], version 5.2, updated with a cycle-level core model with modifications to support for dynamically changing core and cache parameters. Both core adaptation and DVFS transitions take 2 µs during which no computations can be performed—a conservative approach. When reducing the number of cache ways, dirty lines are written back through the simulated memory subsystem, consuming NoC and DRAM bandwidth.

Power consumption. McPAT version 1.0 is used to estimate static and dynamic power consumption [28]. Power savings incurred by reconfiguration are modeled by running McPAT with the modified target parameters as per Table 1. Running McPAT along with the performance simulation allows us to emulate the behavior of hardware energy counters at simulated time slices of 10 ms.

[2]42-bit tags and 16-way maximum associativity.

Component	Parameters
Core count	64
Core type	4-way issue OOO, 128-entry ROB
Load/Store queue	48 load entries, 32 store entries
L1-I cache	32 KB, 4-way
L1-D cache	32 KB, 4-way
L2 cache	512 KB, 16-way, private per core
L2 prefetcher	stride-based, 8 independent streams
Coherence protocol	directory-based MESI, distributed tags
Network On-chip	16×4 mesh, 32 GB/s/link
Main memory	8 controllers, 45 ns latency, 128 GB/s total
Technology	22 nm, 660 mm^2 total area
Frequency	1.2 GHz
Vdd	0.8 V
TDP	120 W

Table 2: Base configuration.

Multi-program workloads. We construct a multi-program workload composed of SPEC CPU2006 benchmarks. There are 29 CPU programs in total, which along with all of their reference inputs leads to 55 benchmarks in total. Each of the 55 benchmarks is pinned to a core, the final nine cores run a randomly selected benchmark. We select representative simulation points of 750 million instructions each using PinPoints [34]. We run the simulation for a fixed amount of simulated time (500 ms). When a benchmark completes before this time, it is restarted on the same core. We quantify system throughput using the STP metric [14] (also called weighted speedup [38]) which quantifies the aggregate throughput achieved by all cores in the system.

Multi-threaded workloads. We evaluate 9 multi-threaded benchmarks from PARSEC [5]. To make the analysis meaningful, we use the *simlarge* input set. Benchmarks are executed with 64 threads in our 64-core processor. Each thread is pinned to a core. We run each benchmark to completion and report total execution time.

3.2 Adaptive many-core architecture

The architecture on which we evaluate Chrysso is a large 64-core processor; see Table 2 for more details. The configuration knobs for *core*, *cache* and *DVFS* adaptations (Table 1) are down-scaled versions of the base architecture. We define the chip's maximum thermal design power (TDP) using the average power consumption of a full-feature chip (each core at maximum width, maximum cache ways and highest DVFS setting) while running the average SPEC workload, which was 120 W. Results will be shown for power budgets as a percentage of this value.

Core configuration. The first configuration knob adapts the core itself. The core width can be adapted, along with the size of various structures. We maintain a quadratic relation between execution width and size of microarchitectural buffers [15]. Unused components are power-gated to reduce both static and dynamic power consumption.

Cache configuration. For cache adaptivity, we use a flushing, selective-way LLC implementation as described in [1]. By controlling which ways are on and off, we can power-gate portions of the cache to reduce its capacity and lower power usage. We use selective-ways because of their sim-

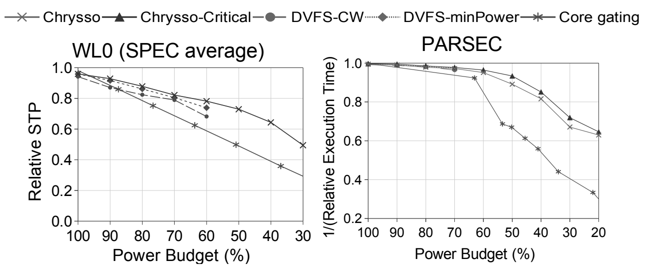


Figure 5: Relative STP vs. power budget: comparing Chrysso against alternative power managers.

ple design, as selective-sets require changes to the number of tag bits used [39]. By using the flushing cache policy when shrinking to a smaller number of ways, we can turn off the corresponding ways sooner, reducing the static power consumption.

DVFS configuration. Finally, we assume the availability of on-die voltage regulation to enable fast per-core DVFS [21, 25] with a range between 0.8 GHz at 0.7 V to 1.2 GHz at 0.8 V, which is in line with Intel Xeon Phi [23].

3.3 Alternate power management policies

We implemented two DVFS-based power management policies. Chip-wide DVFS *(DVFS-CW)* runs all cores at the same frequency. In some commercially available processors, this is referred to as P-states [26]. Minimum-power DVFS *(DVFS-minPower)* represents state-of-the-art per-core DVFS, which iteratively reduces the DVFS of the core with the lowest power [21]; memory-bound cores will not see much performance loss with reduced frequency. In addition, we also compare Chrysso against power-gating cores until the power budget is met. Under core-gating, an equal-time scheduler is used to time-share all 64 applications on the active cores. We assume a time slice of 10 ms, unless mentioned otherwise.

4. RESULTS AND DISCUSSION

We start our discussion with Figure 5 which plots performance as a function of the available power budget as obtained under Chrysso and for a number of alternative power management schemes. Focusing on multi-program results first (left graph), using DVFS alone, power consumption cannot be reduced below 60%. At this setting, chip-wide DVFS (*DVFS-CW*) achieves just 68% of nominal system throughput (measured at 100% TDP) while per-core DVFS (*DVFS-minPower*) increase this to 73%. Core-gating allows any power budget to be reached, but comes at a near-linear cost in performance as it blindly turns off resources irrespective of their utility. In contrast, Chrysso can select the power savings technique that is best suited to each application phase. This allows it to outperform the DVFS-only techniques at moderate power settings, and allows the various techniques to be combined to reach the most stringent power settings at an acceptable loss in performance.

Integrated optimization. The key benefit of Chrysso model-based approach is that it easily allows different adaptation methods to be combined and achieve higher performance within the same power budget. Figure 6(left) confirms this by plotting results for Chrysso in comparison to isolated adaptation (*core*, *cache* and *DVFS*) for the multi-program workload. Using adaptation techniques in isolation, power budget can be reduced by 40% at most; on the other hand,

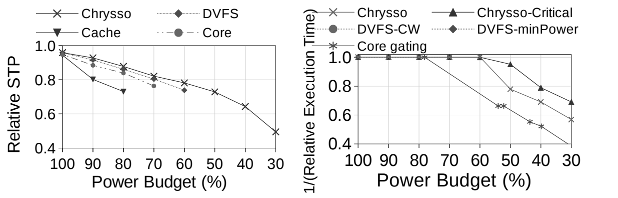


Figure 6: (left) Isolated versus integrated optimization using Chrysso for the multi-program workload. (right) Performance of blackscholes with critical thread prioritization.

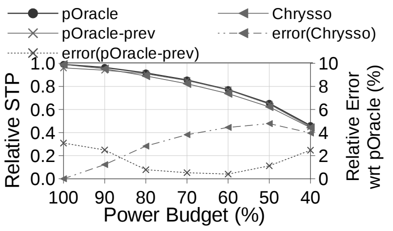


Figure 7: Effectiveness of Chrysso compared to oracle projection models.

combining all three knobs can reduce power budget by 70%. Even at moderate power budgets Chrysso can outperform isolated adaptation: at 80% of TDP Chrysso improves system throughput by 16.8% over cache adaptation, by 17% over core-gating, by 6.3% over chip-wide DVFS, by 4.5% over core adaptation, and by 2% over per-core DVFS. Furthermore, at modest power budgets, e.g. 60%, isolated adaptation (e.g., DVFS) incurs a performance hit of at least 5.5% compared to combined adaptation using Chrysso.

Critical thread awareness. The impact of critical thread awareness is illustrated in Figure 6(right). The blackscholes workload has a relatively low baseline power consumption so to reach even the 60% power setting (corresponding to 72W) no resources need to be turned off and hence there is no performance impact. From 50% onwards, however, Chrysso needs to start managing power by selecting resources to tune down. When treating all cores equally, the application slowdown is significant (over 20% at 50% power). In contrast, the criticality-aware Chrysso variant is able to detect which threads are performance-critical and increases their power budget, while decreasing that of non-critical threads. This has a positive effect on application run-time, leading to a slowdown of just 5% at the same 50% power setting.

Projection model accuracy. Even though Chrysso uses very simple projection models, it is able to make the right scheduling decisions—more complex models would improve the relative and absolute power and performance projections, but do not significantly change the resulting configurations and hence have little impact on system performance. To illustrate this we set up an experiment where per-core performance and power are no longer modeled, but are taken from a database populated with periodic measurements taken from simulations of each workload run at all 48 configuration settings. Figure 7 plots the performance of Chrysso compared to two oracle schemes. *pOracle-prev* forgoes the modeling step and uses actual performance and power data corresponding to the previous time slice, then runs the Pareto front selection and global scheduling based on these values. *pOracle* uses actual performance and power from the trace for the *upcoming* time slice, removing both projection error

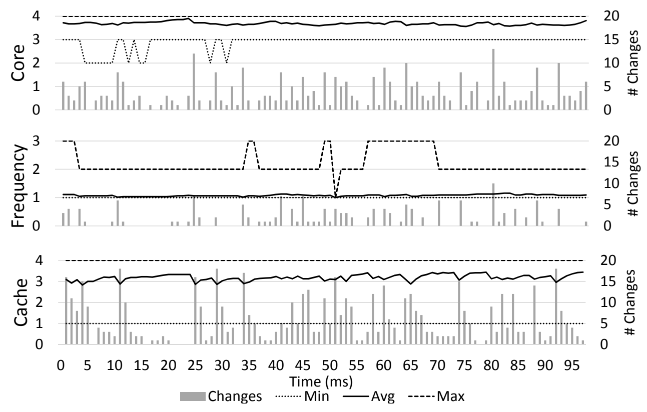


Figure 8: Chrysso configuration changes through time at 50% power setting and a 1 ms (reconfiguration) time slice.

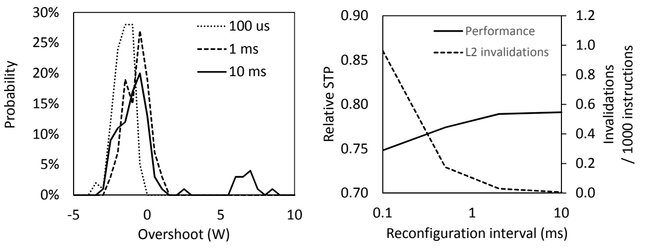


Figure 9: Chrysso at different reconfiguration time slice at 50% power setting.

and workload variability from the equation.

In the figure, solid lines plot STP relative to the full-feature base configuration (left y-axis) obtained by each algorithm for the different power budgets, while dashed lines report the difference with *pOracle* (right y-axis). *pOracle-prev* has an error of up to 3% showing that at this time scale (10 ms), workload variability is still significant and is the cause for some of the suboptimal configuration decisions. Chrysso itself deviates from the oracle by up to 5%, indicating that more elaborate projection models would only improve system performance by another 2% at most.

Dynamic behavior. The dynamic behavior of Chrysso over a 100 ms execution interval is shown in Figure 8, at a 50% power setting. For each configuration knob (core, frequency and cache), the minimum, average and maximum settings over all 64 cores are shown at each 1 ms time slice, in addition to the number of cores for which each knob was changed in that time slice. At this power level, most cores have their frequency reduced to the lowest setting. Core reconfiguration is not used much since it usually has a high performance impact and isn't yet needed at this power setting. In contrast, cache reconfiguration is performed often since it has the most power impact, and is also most affected by application phase behavior.

Reconfiguration time slice. Chrysso is designed to enable fast reconfiguration. Figure 9 explores the characteristics of time slices between 100 µs and 10 ms. The left plot displays the distribution of actual power consumption relative to the specified power limit in 1 ms intervals. Due of workload variability, a 10 ms reconfiguration time slice is often too slow to ensure the limit is always met. Using 1 ms and 100 µs time slice progressively reduces the number of violations.

However, when looking at relative STP (Figure 9, right), it turns out that faster reconfiguration can be detrimental to performance. The main reason is that, especially at this power setting, frequent cache configuration changes result in a large amount of invalidations and writebacks from the L2 cache, and subsequent misses once the application needs this data again later. This phenomenon occurs when the reconfiguration time slice becomes shorter that the typical time between reuse of data in the caches: whereas the ATDs predicted that a smaller cache will not harm performance for a given time slice, the application does exhibit reuse at longer time scales causing the caches to be resized too aggressively. Core and frequency reconfiguration do not have this problem, as their reconfiguration cost is much smaller (we model both with 2 µs penalties).

We conclude from this experiment that for efficient execution, the reconfiguration time slice should never be shorter than the corresponding time scales of application behavior *on a per-knob basis*. At time scales of 1 ms and shorter, a single reconfiguration time slice no longer suffices as some knobs (core, frequency) can be re-tuned every time slice whereas other knobs (caches) will need to be kept constant for a number of time slices to avoid excessive switching costs.

5. RELATED WORK

Micro-architecture Adaptation. A variety of prior work has explored how to improve power-efficiency by adapting microarchitecture structures. Prior works in [3, 17] adapt the instruction window and issue logic to provide greater power/energy efficiency while showing a small reduction in application performance. [19] propose the ForwardFlow core to trade off performance for power. [1, 39] evaluate shutting down portions of the cache, either a number of ways or a combination of ways and sets for improved energy efficiency. These techniques evaluate adapting micro-architectural structures to trade off performance for power and energy. [12] uses drowsy caches with front-end pipeline-gating to demonstrate better performance-power scaling than DFS and even DVFS in some cases. Although their work shows that one can reconfigure the system to perform better than DVFS, they do not perform run-time optimizations of large many-cores in power-constrained environments. [11] use machine-learning models (trained using profiling) to perform on-line adaptation of a single core at a time. [29] proposed DVFS adaptation along with cache adaptation for 4-core system. None of these previous works have evaluated integrated power management including fine-grain adaptation of the core microarchitecture, cache, and per-core DVFS settings for many-core processors at stringent power budgets.

Dynamic Power Management. [21] propose a global power controller to determine different per-core DVFS settings to maximize chip-wide MIPS. [22] propose global DVFS with per-core adaptation based on neural networks to reach the power budget. On similar grounds, [6] formulate global resource allocation using machine learning. [35] proposes to dynamically adjust the capabilities of an out-of-order core at coarse-grained time slice (100 ms) using sampling-based global genetic algorithm to improve performance compared to core-gating at moderate power budgets. RCS [18] proposes SVM-based machine-learning mechanisms to obtain the number of active cores (8/10/12) with reduced micro-architectural size to exploit application variability at a fixed power budget. In contrast, Chrysso uses low-overhead analytical models to provide a more scalable adaptation scheme while exploring a broader adaptation space (including cache and DVFS along with core adaptation).

Critical Thread Acceleration. Several prior works have proposed techniques to identify critical threads for acceleration, either by running serial parts at higher clock frequency [2, 31], by running serial code and synchronization bottlenecks on a big core in a heterogeneous multi-core [24, 33] or by speeding up critical threads in barrier-synchronized applications based on cache behavior [4]. Chrysso integrates criticality stacks [10] to accelerate critical threads and improve multi-threaded application performance under constrained conditions.

6. CONCLUSION

An integrated and scalable many-core power management is clearly needed as we move towards even tighter power budgets. Chrysso leverages its scalability and effectiveness from (i) using analytical performance models and table-based power models for core, cache, and per-core DVFS adaptation, (ii) a search process that identifies Pareto-optimal per-core configurations to prune the global optimization space, and (iii) utility-based optimization which reallocates power to the cores/threads that benefit the most, e.g., critical threads in multi-threaded workloads and power-hungry applications in multi-program workloads. Chrysso outperforms isolated power adaptation techniques by a significant margin at moderate power budgets, and outperforms core-gating in system performance by 1.6× and 1.9× for multi-program and multi-threaded workloads at stringent power budgets, respectively. Chrysso incurs limited run-time overhead (less than 1%) and hardware overhead (roughly 3 KB per core).

Acknowledgements

This work was supported in part by the European Research Council under the European Community's 7th Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295 and the Spanish Ministry of Economy and Competitiveness under grants TIN2010-18368 and TIN2013-44375-R. The experiments were run on computing infrastructure at the ExaScience Lab, Leuven, Belgium.

7. REFERENCES

[1] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *MICRO*, 1999.

[2] M. Annavaram et al. Mitigating Amdahl's law through EPI throttling. In *ISCA*, 2005.

[3] R. I. Bahar and S. Manne. Power and energy reduction via pipeline balancing. In *ISCA*, 2001.

[4] A. Bhattacharjee and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *ISCA*, 2009.

[5] C. Bienia et al. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[6] R. Bitirgen et al. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO*, 2008.

[7] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *HPCA*, 2001.

[8] T. E. Carlson et al. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *SC*, 2011.

[9] R. Dennard et al. Design of ion-implanted MOSFET's with very small physical dimensions. *ISSCC*, 1974.

[10] K. Du Bois et al. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. In *ISCA*, 2013.

[11] C. Dubach et al. A predictive model for dynamic microarchitectural adaptivity control. In *MICRO*, 2010.

[12] Y. Eckert et al. Something old and something new: P-states can borrow microarchitecture techniques too. In *ISLPED*, 2012.

[13] H. Esmaeilzadeh et al. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.

[14] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 2008.

[15] S. Eyerman et al. A mechanistic performance model for superscalar out-of-order processors. *TOCS*, 2009.

[16] X. Fan et al. Power provisioning for a warehouse-sized computer. In *ISCA*, 2007.

[17] D. Folegnani and A. González. Energy-effective issue logic. In *ISCA*, 2001.

[18] H. R. Ghasemi and N. S. Kim. RCS: Runtime resource and core scaling for power-constrained multi-core processors. In *PACT*, 2014.

[19] D. Gibson and D. A. Wood. Forwardflow: A scalable core for power-constrained CMPs. In *ISCA*, 2010.

[20] Intel. 2nd gen. Intel Core vPro processor family, 2008.

[21] C. Isci et al. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *MICRO*, 2006.

[22] R. Jayaseelan and T. Mitra. A hybrid local-global approach for multi-core thermal management. In *ICCD*, 2009.

[23] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Newnes, 2013.

[24] J. A. Joao et al. Bottleneck identification and scheduling in multithreaded applications. In *ASPLOS*, 2012.

[25] W. Kim et al. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *HPCA*, 2008.

[26] J. Leverich et al. Power management of datacenter workloads using per-core power gating. *CAL*, 2009.

[27] J. Li and J. F. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *HPCA*, 2006.

[28] S. Li et al. McPAT: An integrated power, area and timing modeling framework for multicore and manycore architectures. In *MICRO*, 2009.

[29] K. Meng et al. Multi-optimization power management for chip multiprocessors. In *PACT*, 2008.

[30] A. Mericas. Performance monitoring on the POWER5 microprocessor. In *Performance Evaluation and Benchmarking*. CRC Press, 2006.

[31] T. N. Miller et al. Booster: Reactive core acceleration for mitigating the effects of process variation and application imbalance in low-voltage chips. In *HPCA*, 2012.

[32] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 1965.

[33] T. Morad et al. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *CAL*, 2006.

[34] H. Patil et al. Pinpointing representative portions of large Intel Itanium; programs with dynamic instrumentation. In *MICRO*, 2004.

[35] P. Petrica et al. Flicker: A dynamically adaptive architecture for power limited multicore systems. In *ISCA*, 2013.

[36] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, 2006.

[37] E. Rotem et al. Power-management architecture of the Intel microarchitecture code-named Sandy Bridge. *IEEE Micro*, 2012.

[38] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for simultaneous multithreading processor. In *ASPLOS*, 2000.

[39] S.-H. Yang et al. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *HPCA*, 2002.