6 RELATED WORK

6.1 Overview

In this chapter we briefly discuss related work in characterization and governance of reconfigurable computers. These include:

- Characterizations of energy efficiency (Section 6.2). We describe our work in this area in Chapter 2.
- Descriptors for system power-performance states (Section 6.3). We propose using Π-states and the Π-dashboard ([192], Chapter 2).
- Power-performance goals targeted by governors (Section 6.4). We discuss the SLAs that we target in Chapter 3.
- Analytical models for cache performance (Section 6.5). We develop new models ([191], Chapter 4) that are based on cache reuse distances.
- Examples of system reconfiguration studies and reconfigurable knobs (Section 6.6). We describe the knobs that we study and their governance mechanisms in Chapters 3 and 5.

Finally, in Section 6.6.1 we propose a new classification system for governance studies that is based on the behavioral semantics of the reconfiguration capabilities instead of on the components that are reconfigured. Semantics-based classification enjoys the advantages of being more compact and perhaps more insightful than the other classifications.

6.2 Energy Efficiency Characterization

Energy proportionality has been extensively studied [22, 106, 122, 141, 184, 213, 231, 232]. Barroso and Hölzle [22] introduced the concept and argued that energy proportionality should be one of the main design goals. They compute power efficiency as $\frac{\text{Utilization}}{\text{Power}}$. We use the same definition for efficiency, but relabel utilization as percentage of peak performance.

David Lo et al. [141] proposed a relaxed model of energy-proportionality, called Dynamic Energy Proportionality, that ignores idle power. This corresponds to the Dynamic EP line in Figure 2.1. This linear model has also been studied in other prior works [223, 232]. Daniel Wong and Murali Annavaram [232] name the region that we call Sub-Linear as Superlinear. We prefer to use "Sub-" in the sense that operating in this region lowers efficiency compared to that of Linear (Dynamic EP).

A number of metrics for characterizing energy efficiency exist. Efficiency ($\frac{\text{Performance}}{\text{Power}}$) can be computed at individual loads [22], or as $\frac{\text{Total Performance}}{\text{Total Power}}$ over all loads [205]. Metrics based on the dynamic power range compute the ratio between the idle and peak power consumptions [223]. Other metrics consider the deviation of the power curve from an ideal curve, e.g., maximum relative power difference with respect to Dynamic EP [223], area enclosed by the power curve relative to that by Dynamic EP [232] or EP [184, 232], power used in excess to that by EP [232], etc. These metrics continue to be useful with the new ideals, EOP and Dynamic EO, replacing the conventional ideals.

Wong and Annavaram [232] introduced the notion of the EP Wall that needs to be overcome. They proposed leveraging heterogeneity to improve energy efficiency at low utilization. Our work with prefetch control and cache power budgeting will further help to overcome the wall. Wong and Annavaram [231] also investigated techniques for quantifying energy proportionality of a cluster of servers.

We observe that both Pegasus [141] and Knightshift [232] report data appearing to show occasional incursions into super-proportional regions. Chung-Hsing Hsu and Stephen W. Poole [106] observed real machines doing better than the conventional "ideal" system that assumes linear proportionality. They proposed quadratic proportionality (Power(u) $\propto u^2$, where u is the load level) as the new ideal model. However, this makes ideal system efficiency load-dependent $\left(\frac{u}{\text{Power}(u)} = \frac{1}{u}\right)$, with higher efficiency at lower loads than at higher loads.

Our view is that the design ideal, EOP, will have maximum efficiency (η_{max}) independent of load and will consume power linearly proportional to load, as proposed in the original EP model, but the constant of proportionality is different: it is defined by the most efficient configuration instead of by the configuration achieving the maximum performance. The Pareto frontier (Dynamic EO) is the operational ideal for the system and its efficiency is load-dependent. The most efficient configurations lie at the intersection of the EOP and Dynamic EO curves.

Song et al. [204] proposed Iso-energy-efficiency (EE) as the energy ratio between sequential and parallel executions of a given application. Our CPUE, LUE and RUE metrics do not use specific execution modes (e.g., sequential/parallel, homogeneous/heterogeneous, speculative/non-speculative, cache-conscious/cache-oblivious, etc.) for reference, but compare system states to the Pareto frontier (Dynamic EO) or to EOP. The definitions of our metrics are oblivious to which configurations created the frontier.

The EE model focuses on maintaining equal efficiency as systems and applications scale up. In contrast, the EP and EOP models focus on maintaining equal efficiency under changing loads. So our metrics include load, along with the configuration, as a parameter for quantifying excess energy used. On the other hand, EE does not quantify its dependence on load.

Barroso and Hölzle [104] compute datacenter energy consumption as PUE $\times$ SPUE $\times$ energy to electronic components. While PUE [16] accounts for non-compute overheads in datacenter building infrastructure, SPUE (Server PUE) accounts for overheads, e.g., power supply losses, to computing energy. Our RUE and LUE metrics do not separate SPUE losses from computing energy but separate energy-wasting operating configurations and loads from optimal ones.

6.3 Power-Performance States

ACPI: The Advanced Configuration and Power Interface (ACPI) specification [102] is an open standard that allows devices (resources) to specify discrete operating states identified by alphanumeric names. For example, P0, P1, P2,... represent processor performance states. ACPI enumerations lack quantification of system-wide power-performance impacts by not accounting for inter-resource interactions or dynamic execution profiles. A static enumeration of possible states for individual knobs, as in the ACPI [102] approach, is insufficient because it does not quantify power-performance impacts at the system level or take into account correlated effects across different knobs (e.g., prefetching). So, it is difficult to answer questions such as: which system configuration performs the best for a given power budget? which system configuration has the minimum energy-delay (ED) or ED^2 product?

New P states: Eckert et al. [70] proposed new processor P-states and L2 cache P-states, but did not provide a framework for optimal system configuration selection. The new processor states save power by reconfiguring pipeline front-end structures and mechanisms, e.g., register and fetch buffer sizing, simplified speculation control, limited checkpoint state, etc. (Sharkey et al. [193] also explored different fetch throttling mechanisms that differed in local (per-core) vs global chip information and per-core vs

chip-wide settings.) New states for L2 include drowsy and power-gating states. Dirty data from power-gated L2 ways is written to the L3 cache.

Π-states: Sen and Wood [192] proposed Π-states that overcome the limitations of ACPI state enumerations. This is motivated by the observation that individually ordered lists of operating states for different resources, as in ACPI, do not identify ordering for combinations of states across resources, required for system-level coordinated management. ACPI enumerations lack quantification of system-wide power-performance impacts by not accounting for inter-resource interactions or dynamic execution profiles.

Each Π-state is a 4-tuple (slowdown, dynamic energy, static power, work) that describes the effect of using a configuration of a system component. A centralized coordinator stitches these descriptors together to determine system-wide impacts if multiple components are reconfigured. The system computes a Π-dashboard consisting of Pareto-optimal Π-states that are numbered in decreasing order of performance. The user or operating system selects a desired Π-state, that corresponds to the optimum value of a metric (e.g., minimum energy, minimum EDP, etc.), causing the system to transition to the corresponding configuration.

6.4 Optimization Goals

There exists a variety of flavors of the power/energy management problem. The *power-budgeting* problem seeks to partition a maximum power budget among resources to maximize performance [115]; the *energy-minimization* problem seeks to find configurations that minimize energy consumption (equivalently, maximizes performance/watt); the *min-EDP* problem seeks to minimize the energy-delay product (EDP) [82] so that configurations that reduce energy but cause unacceptable delays are not chosen. Snowdon et al. [202] generalized this metric to include non-integral exponents for power and delay. Our

two-level governors, described in Chapter 3, can be easily retargeted to optimize system operations for these metrics.

A number of works have studied power/energy management while meeting tail latency and response time deadlines [122, 141, 152]. Targeting this SLA may require that workload-specific semantic knowledge be available to the objective selector. Our current governors do not have high-level knowledge about the workloads and we do not target this SLA in this work.

6.5 Cache Models

The goal of these models is to predict cache performance (miss ratio) as a function of cache organization and workload properties. Here we briefly describe various approaches to solve this problem.

Power-Law models: Chow [50], Hartstein et al. [100], and others used power laws based on cache capacity to predict miss ratios. One instance of such a power law predicts that the miss ratio reduces by $\sqrt{2}$ if the cache capacity doubles, and is popularly known as the 2-to-$\sqrt{2}$ rule. These models have practically zero overhead but may have large errors since they do not account for working-set sizes and cache access patterns.

Unique and absolute reuse distance models: Mattson [150] introduced the concept of predicting miss ratios from (unique) stack distances for caches that use replacement policies having the inclusion property. This technique has been subsequently used in many works [29, 55, 56, 65, 103, 140, 191, 196, 200, 242]. Guo and Solihin [91] proposed circular sequence profiles that are similar to stack distances in reuse intervals.

Stack distance distributions can be determined offline or online. In offline algorithms, stack distances are computed offline from an address trace. Previous work has extensively studied cache miss rate prediction using offline estimation of LRU stack/reuse

distances [12, 29, 103, 196, 200], but have limited applicability for online use.

Computation time to determine the distributions can be reduced with efficient algorithms [13] or by approximate analysis [242]. Hill and Smith [103] introduced techniques for estimating miss ratios for many different cache organizations from a single pass over an address trace. Shi et al. [196] perform single-pass stack simulation to project cache performance and to study the impact of data replication for various L2 cache configurations. Online determination of the stack distance distribution cannot directly apply techniques from offline methods due to constraints on computational state and complexity.

Tam et al. [220] use hardware mechanisms for address sampling and post-processing software for computing stack distance distributions. Since distribution estimation and hit ratio computation is offline, it cannot react to workload changes in real time.

Online methods have severe restrictions on space and time complexity, but must achieve good accuracy. Way counters [127, 175, 215] exploit the LRU stack property to predict miss rates for configurations smaller than the current cache. Shadow tags [172] (or Auxiliary Tag Directories [175]) extend way counters to predict configurations with higher associativity than the active cache configuration. Dynamic set-sampling can reduce overheads [174]. Way counters have been extended to work with PLRU replacement [127] using a heuristic that estimates the LRU stack depth using the PLRU tree bits. Suh et al. [214], Qureshi et al. [175] proposed mechanisms for partitioning of shared caches (L2) among competing processes using way-counters. Suh et al. [215] also proposed set-counters in LRU order, with each counter tracking accesses to a group of sets. We discuss a limitation of set counters in Chapter 4, Section 4.6.2. Gordon-Ross et al. [84] used a hardware TCAM to track stack distances for LRU miss-ratio predictions. However, area overheads probably limit this approach to small caches. In contrast, our methods

([191], Chapter 4) have very low area and power overheads, so they can used for large caches. We predict miss ratios for caches that are configurable in both the number of sets as well as ways.

StatCache [28] and StatStack [71] used sampling on the cache access stream to estimate the absolute distance distributions, then use that to estimate miss ratios for fully-associative LRU or RANDOM caches. Estimating absolute distances is easier than estimating unique distances. We also use absolute distances for RANDOM cache estimations, but estimate average absolute distances from unique distances. We use unique distances for LRU caches. Pan and Jonsson [167] use absolute distance distributions and Markov models to estimate performance of set-associative caches. Inter-Reference Gaps in IRG models [169, 219] are basically absolute reuse distances.

Binomial and Poisson models: Smith [200] and Hill [103] introduced the technique of using Binomial distributions along with stack distances to model set-associative LRU caches. We use the same approach in our work ([191], Chapter 4) but can handle some implementable, non-LRU, policies as well. Agarwal et al. [3] also use Binomial models, in conjunction with other models, e.g., Markov models, for cache performance estimations. Their models also account for block sizes, degree of multiprogramming, and task switching intervals. Stone and Thiebaut [211], Falsafi and Wood [75] use binomial probability models to model cache reload transients due to context switches based on the footprints of the competing programs and cache size. The Binomial model assumes independent mapping of addresses to cache sets. Pan and Jonsson [167] improved prediction accuracy by considering the actual mapping achieved.

Cypher [55, 56] proposed using Poisson distributions to estimate cache miss ratios from estimated stack distances. We use a similar approximation in Chapter 4 to reduce the computational costs associated with using the Binomial model. Cypher also used

filter fraction metrics that reduce the effective distance to be tracked. Computing filter fractions can be expensive.

Markov models: These models consider the evolution of cache states with transition probabilities between states [90, 91, 167] but can be computationally expensive. Guo and Solihin [91] proposed Replacement Probability Functions (RPFs) that describe the probability that a line at a certain stack position will be replaced given that a miss happens to that cache set. (Our cache hit function, ϕ, describes the probability that a line at a certain stack position will hit given that an access happens to that line.) Guo and Solihin's Markov model tracks the evolution of states described by reuse intervals and stack positions. Grund and Reineke [90] proposed replacement policy tables that improve upon Guo and Solihin's approach by decoupling policy characterization from workload properties. Agarwal et al. [3] used Markov models to characterize spatial locality. Others have used Markov models to analyze the behavior of context switch misses [138].

Closed-form models: These models express cache performance in terms of easily-computable, non-recursive expressions having a small number of terms. The power-law models are examples of closed-form models (parametrized with an initial measurement). Cache Calculus [24] developed a system of differential equations, solving which produces closed-form expressions of (fully-associative) cache performance in terms of cache size and high-level data structures, e.g., array sizes in array access workloads. One challenge is to be able to formulate the cache access stream in terms of high-level data structures.

Worst-case models: Reineke and Grund [180] prove relations on best and worst-case bounds of cache performance for several replacement policies. Our work, in contrast, studies average case behavior.

A number of prior works [14, 89, 179] have explored applying static analysis techniques, such as abstract interpretation, to determine bounds on cache performance for real-time

systems. Lv et al. [144] provide a survey of the area.

Xiang et al. [235] developed a higher-order theory of locality (HOTL) that interrelates a number of locality metrics—reuse distance, fill time, footprint, miss ratio, and time between cache misses.

6.6 Reconfiguration Knobs

In this section we briefly discuss some microarchitectural and runtime knobs for power-performance management where a dynamic choice can be made on whether or not to execute using a certain system/component configuration or on the extent/degree of such reconfiguration. Our goal is to present different kinds of reconfigurable architectures/capabilities, not necessarily to catalog all prior studies within each class.

Core DVFS: Dynamic voltage and frequency scaling (DVFS) and dynamic frequency scaling (DFS) for cores are well-known techniques [86, 110, 111, 229].

Isci et al. [115] introduced the concept of maximizing performance for a given power budget, using a global power manager and per-core monitors to set per-core DVFS modes. Their MaxBIPS policy predicts the power and performance for each possible configuration. This can limit scalability to larger number of cores. A global manager selects the configuration predicted to have the highest throughput within the power budget.

Ma et al. [145] explored the problem of selecting different per-core DVFS levels for systems that have the capability. Their mechanism first uses feedback control to determines an aggregate frequency that does not exceed the specified power budget. Next, this is partitioned among groups of cores where all cores within the same group run threads of the same application. Finally, cores within the group are allocated their frequency settings based on thread criticality.

Koala [202] developed a power-performance management framework in the OS. It uses DVFS to manage tradeoffs between performance and energy consumption. Koala uses performance counters to characterize an offline model to predict energy for all frequency settings. It then uses this model online to select the best setting. Koala proposed a new metric called generalized energy-delay ($P^{1-\alpha}T^{1+\alpha}$, $\alpha \in [-1, 1]$) that extends the ED metric by allowing non-integral exponents for both power (P) and delay (T).

Spiliopoulos et al. [207] proposed green governors that improve energy efficiency by controlling core frequency and voltage settings. The governors use performance counters (to obtain IPC and stall counts) and measurements of idle static power to predict the performance and energy consumption for different frequency settings. The governors choose the best frequency that optimizes a given metric, e.g., minimum EDP, minimum ED^2P, minimum EDP with a performance constraints. Our frequency governors, described in Chapter 3, use a profiling-based approach that interpolates power and performance from a few measurements. Our prefetching governor profiles prefetching modes and selects the best-performing mode. Our governor for SPECpower does not predict power/energy for any configuration. Its goal is to reduce the number of idle cycles while maintaining the current performance. It only profiles for prefetch modes but not frequencies. A profiling-based approach removes the need for building accurate performance and power models from performance counters, but is only possible for knobs that allow fast reconfiguration, e.g., processor frequency or prefetching. Our governor for cache sizing, described in Chapter 5, builds a performance and power model from performance counters.

Rubik [125] uses per-core DVFS to reduce power while still meeting latency constraints of work requests. It maintains two tables that describe distributions of per-request core and memory completion times in cycles. These tables are updated periodically (every

100ms). Every time a request completes or a new request is admitted, the tables are consulted and a frequency is calculated that meets completion time constraints for all current requests. Rubik uses a power model based on performance counters to guide its decisions. A proportional-integral (PI) controller runs over a longer interval (1 sec) to compare predicted with measured latencies and calculate adjustments. Rubik also studies consolidating batch and latency-critical applications together to reduce total idle power of servers.

Prekas at al. [171] propose mechanisms that control per-core DVFS and the number of logical cores (physical cores and hyperthreads) allocated to applications running on the IX [26] operating system. Their controller keeps track of network queueing delays to help in resource management and can migrate applications and network processing between cores. They propose two schemes, one for reducing computing energy and the other for reducing idle power through consolidation. In both schemes, latency constraints are maintained.

Rangan et al. [177] proposed maintaining different homogeneous cores at different voltage/frequency levels. Depending on its runtime characteristics, a workload can be rapidly migrated from one core to another so that it can be executed at a different frequency. This scheme aims to reduce DVFS transition times from microseconds, with voltage regulators, to nanoseconds by inter-core migration.

A number of prior works [141, 182, 213] have explored the use of RAPL [113] in specifying power caps. Rountree et al. [182] used RAPL to reduce power for HPC (High Performance Computing) applications and observed that power limits can transform power variations across machines into performance variations. Lo et al. [141] used RAPL to improve energy efficiency of OLDI (online data intensive) workloads. As we have discussed in Section 3.9, current RAPL implementations do not deal with reconfiguration

knobs such as cache prefetching.

Memory DFS/DVFS: Deng et al. (MemScale) [62] and David et al. [59] proposed using DFS/DVFS for main memory.

Core DVFS and memory DVFS: Subramaniam and Feng [213] explored using RAPL for both cores and DRAM to improve energy efficiency of enterprise workloads and found that limiting power for the core domain provides the most benefits compared to other domains.

CoScale [61] demonstrated coordinated management of core DVFS and memory DVFS. It uses a gradient-descent heuristic that is related to the greedy solution strategy for the integer knapsack problem. The approach in CoScale is to arrange operating states in decreasing order of marginal utility ($\Delta power/\Delta performance$) and greedily pick states till a maximum slack is not violated. Although theoretically the greedy strategy can have up to a 2-approximation factor [58], CoScale demonstrates a tighter approximation in practice. Their default epoch length is 5ms.

Number of cores/threads and core DVFS: Li and Martinez [136] explored simultaneous management of the number of cores and DVFS levels for the chip (all cores at the same level). Their algorithm performs a binary search on the number of processors, starting from the middle number. For each number, it tries lowering the DVFS level till the performance target is just missed. This process is repeated on the upper or lower halves of the search space on the number of processors till no improvement is found. Heuristics are used to speed up the search for the appropriate DVFS level for each setting of the number of cores. Overall, the complexity is $O(\alpha log(N))$ for N cores and $\alpha << L$ for L DVFS levels.

Vega et al. [224] explored coordinated management of core DVFS and the number of enabled cores. This is driven by heuristics based on core utilizations. The utilization is

averaged over a number of intervals, with the best choice of history length being 3. They consider a time interval of 1 second for reconfiguration decisions.

Curtis-Maury et al. [54] studied varying the number of threads, mapping of threads to cores, and core DVFS levels. This involves sampling the execution with a few configurations before a reconfiguration decision is made. The applications are instrumented around OpenMP parallel regions. Varuna [208] also changes parallelism dynamically, but does not need to make changes to application source code.

Cache size: Albonesi [10] introduced mechanisms for disabling cache ways to save energy. Dropsho et al. [67] proposed the accounting cache that uses way counters to evaluate configurations with different associativities. Yang et al. proposed cache reconfiguration in the number of sets [238] and in both sets and ways [237]. Kaxiras et al. [126] introduced cache decay that exploits generational behavior of cache line usage to reduce cache leakage with (area-expensive) power-gating control per line and a (small) performance hit. Instead of fully disabling cache lines and losing state, Flautner at al. [77] introduced the drowsy cache where cache lines can be put into a low-power state-saving mode for saving energy. Our work (Chapters 4 and 5, [190]) studies reconfiguring the cache for both associativity and the number of sets, but not enabling/disabling at the granularity of cache lines, and considers power gating the disabled regions for maximum power savings.

Intel processors/microarchitectures such as the Core Duo [163], and Ivy Bridge [181] dynamically size caches based on activity. In contrast, reuse-based predictors track locality and hence select a smaller cache for highly cache-active but cache-insensitive workloads whereas activity-based models would select a large cache. Some recent Intel processors allow core-wise monitoring and partitioning of the LLC capacity [53].

Yang et al. [237, 238] compare the number of misses to a bound/threshold to drive

reconfiguration decisions. Keramidas et al. [128] compared the number of misses and conflict misses to bounds/thresholds to decide reconfigurability in the number of sets and ways of LRU caches. Sundararajan et al. [216] considers both the LRU stack distance and number of dead sets to determine the configuration for the number of sets and associativity.

Gordon-Ross et al. [84] proposed a one-shot cache reconfiguration scheme. It uses a hardware TCAM to determine stack distances of addresses and tracks stack hit counts for various cache sizes, assuming stack inclusion. It (time-)samples the address stream to reduce the average processing time. The hardware TCAM results in a 12% area overhead for the ARM920T that has 16KB instruction and data caches. As we have discussed in Chapter 4, tracking stack distances for large multi-megabyte caches is prohibitively expensive.

Albericio et al. [8] proposed the reuse cache, an LLC organization where the data array is sized depending on the amount of data that is reused. The tag array is decoupled from the data array. On a miss in the tag array, data is fetched from memory but not placed in the LLC data array. On a hit to the tag array for a line that is not present in the data array (indicates reuse), that line is fetched from memory and placed in the data array. This scheme requires modifications to the coherence protocol, forward pointers in the tag array, reverse pointers in the data array, and uses a different replacement policy for the tag and data arrays. Traditional cache resizing, in the form that we have studied, does not need these changes but incurs overheads during resizing due to subsequent warmup.

Cache compression: Compression helps to store more data in a cache of a given size compared to storing in uncompressed form. Compression may reduce misses but makes hits more costly due to the decompression latency. Alameldeen and Wood [5] proposed

an adaptive compression scheme that uses stack distance information to decide whether to allocate cache lines in compressed or uncompressed form. Further benefits can be derived by combining compression with adaptive prefetching [7].

Cache size and compression: Hajimiri et al. [97] studied cache size reconfiguration along with code compression.

Cache partitions: The fraction of shared cache space available to each core or application can also be dynamically controlled. Cache partitioning [49, 178] can be application-utility-aware [175, 214], LLC-bank-aware [124], MLP-aware [159], spatial-locality-aware [93], cooperative [217] etc. The partitions may be coarse-grained [49, 178, 217] or fine-grained [146, 186].

Cache hierarchy: Albonesi [9] studied simultaneous reconfiguration of L1 and L2 cache sizes in an exclusive hierarchy. Balasubramonian et al. [20] studied reconfigurable L1, L2, and TLBs for energy-efficient performance. A single large cache organization serves as a configurable 2-level non-inclusive hierarchy. The optimal cache configuration is selected by exploration—successive cache sizes are chosen till the miss rate is sufficiently small. The reconfiguration intervals for the cache and TLB are 100K cycles and 1M cycles respectively.

Cache content replication: Chang and Sohi [45], Beckmann et al. [23] explored replicating cache blocks from a remote LLC bank in the private or local LLC bank. The advantage is that hits to local banks are faster than those to remote banks. However, replicating blocks reduces effective capacity of the cache that in turn can potentially increase the miss rate. The degree of replication can be varied based on cost-benefit tradeoffs.

Cache insertion/promotion/eviction: Caches are of finite size and hence must eventually evict some existing data to make room for new data when needed. A number

of works have proposed new management policies that decide whether or not to insert new data [79, 94, 165], where (in terms of recency) to insert new data [79, 118, 119, 120, 123, 173, 234], and how to promote [119, 120, 131, 135, 236] or protect [69] or evict [169, 176, 185, 219] data.

Access granularity:Veidenbaum et al. [225], Yoon et al. [239] proposed dynamically reconfiguring the width/granularity for cache and memory accesses.

Cache size and core DVFS: Meng et al. [155] explore cache resizing (in the number of ways) and changing core DVFS levels. They use way counters and analytic power-performance models for power management. In contrast to MaxBIPS, they use a greedy search strategy to decide the final configuration.

Their work has several limitations as follows. First, they assume true LRU replacement, which is impractical to implement for highly associative last-level caches. This is critical, because practical implementations such as PLRU do not have the stack property and thus a single tag array cannot both provide replacement decisions and miss-rate predictions for larger sized caches. While Meng et al.'s work could probably be extended to use shadow tags and dynamic set sampling [172, 175], the extra area and power would far exceed that of our reuse sampling approach. Second, their study evaluates 600µsecs observation intervals, which are far to short to amortize the reconfiguration overhead of large, e.g., 32MB LLCs. Finally, their approach only handles limited cache reconfigurability (number of ways, not sets), does not consider thermal effects on leakage power, and optimizes for the lowest-power configuration, not the highest-performance configuration.

Cache size, memory bandwidth, and core DVFS: Bitirgen et al. [31] used a machine learning approach for resource management. They construct a per-application artificial neural network (ANN) that takes as input the power budget, bandwidth, cache size, and various performance counter values to predict performance as output. These ANNs are

queried by a global resource manager that uses a stochastic hill-climbing algorithm to select a configuration predicted to achieve the highest performance.

Chip-wide DVFS and thread packing: Cochran et al. [51] explored simultaneous management of DVFS levels and packing of threads on to a subset of cores. They use an offline characterization process to analyze performance counters and power caps to create a lookup table. This is queried at run time for all potential configurations to determine the optimal setting. The online lookup uses performance counter values including information from thermal sensors, but does not require power information. The control activation period is in the order of seconds.

Core organization: Reconfiguring various components in both the frontend and backend of cores has been well studied. Albonesi [9] proposed Complexity-Adaptive Processors (CAP) with reconfigurable resources (instruction queue size, L1 and L2 cache sizes). Manne et al. [147] proposed mechanisms that reduce energy consumption by adapting control speculation to prevent potentially wrong-path instructions from being dispatched. Ghiasi et al. [80] proposed switching between in-order and out-of-order executions. Bahar and Manne [18] explored changing the issue width and the number of enabled functional units. Buyuktosunoglu et al. [39, 40] studied issue queue reconfiguration and fetch gating. Forwardflow [81] dynamically tracks dataflow dependencies in the instruction stream and uses a dataflow queue whose capacity can be dynamically reconfigured, thereby changing the instruction window size. The dataflow queue is organized into banks that can be independently activated/deactivated by system software.

Core throttling and memory throttling: Felter et al. [76] demonstrated performance benefits through power-shifting between the core and memory within a power budget. They achieve this by throttling core and memory operations depending on workload

characteristics. Core throttling is done at the instruction dispatch unit. Memory throttling is done by limiting the number of memory request per unit time (allowed bandwidth). Their throttling mechanism affects shifting of active power. They studied interval sizes in the range of 5μ sec to 1ms.

Core organization and number of cores: Ipek et al. proposed core fusion [114] where resources of independent cores are dynamically grouped/fused together to form a larger core resulting in asymmetric CMPs. WiDGET [228] dynamically changes the number of instruction engines and the number of execution units allocated to each engine. Instructions are steered from the instruction engines to the execution units dynamically allocated to it.

Cache size and core organization: Albonesi et al. [11] explored adaption of the L1 cache sizes and core resources. Dropsho et al. [68] also studied dynamic reconfiguration of these resources in GALS (Globally Asynchronous, Locally Synchronous) microprocessors. The L1I, L1D, and L2 caches have reconfigurable associativities. The branch predictor has a configurable history length. The L1D and L2 caches are sized together. Similarly, the L1I cache and branch predictor are sized together. The sizes of the integer and floating point issue queues are configured depending on the ILP that is available.

Core organization and DVFS: Sasanka et al. [189] explored changing the DVFS level along with core instruction window size, issue width, and the number of functional units. This study aims to reduce energy consumption for multimedia applications while still meeting deadlines. The DVFS decision seeks to eliminate idle time between processing different frames whereas reconfiguring the other resources aims to reduce processing energy within each frame without affecting execution time.

Core organization and memory idle states: Li et al. [137] studied reconfiguration of both core resources (instruction window size, issue width, number of functional units)

and memory idle states (standby, nap, powerdown). Each application phase is profiled, with the number of profiling intervals per phase being equal to the number of processor configurations. This overhead is amortized over multiple occurrences of each phase for long running applications. The decision algorithm aims to optimally distribute the target slack between the core and memory.

Dynamic task reassignment: Chakraborty et al. [43, 44] proposed computation spreading in over-provisioned multicore systems (OPMS). This uses a light-weight virtual machine monitor (VMM) to assign similar computation fragments (parts of software threads) to cores. For example, OS code is executed on a different core than user code. The idea is to improve energy efficiency through dynamic specialization (e.g., predictive structures such as branch predictors, etc., can work better if similar code is executed on the same core) as well as manage peak power consumption by limiting the number of simultaneously active cores in OPMS.

Prefetching: Data prefetching is a well-known speculative technique [17, 41, 74, 209] for improving performance. However, inaccurate prefetching will use more energy either due to unneeded or inadequate fetches or due to lack of timeliness. Both the number of blocks prefetched [57] as well as the lookahead distance [85] may be dynamically configured. Gornish and Veidenbaum [85] combined hardware prefetching with software (compiler-directed) prefetching. Guo et al. [92] developed a power-aware prefetch engine that uses analysis information from the compiler to decide on the prefetch mechanism.

Prefetching, DVFS: In Chapter 3, we demonstrate governors that improve energy efficiency by simultaneously controlling DVFS levels and enabling/disabling of L2 cache prefetching.

Prefetching, DVFS, and number of cores/threads: Kamruzzaman et al. [121] proposed using helper threads to prefetch data needed by the main computation. A number

of helper threads can be used, each of which can prefetch a different chunk of data. Once the data is fetched, the compute thread is migrated to that core whereas the helper thread is shifted elsewhere. The helper threads can run at lower frequencies whereas the compute thread can run at higher frequencies.

Frequency control and sleep states: SleepScale [139] investigated coordinated management of frequency levels and sleep state selection. Running at higher frequencies consumes more power, but finishes tasks earlier leaving more time to enter and remain in deep sleep states. The best policy is determined by factors such as job size and desired average response time. They logically partition the execution into epochs of 5mins. Once every epoch, the best policy is determined based on the estimated interarrival times, response times, and utilization. Considering each policy requires ~6ms for a total of < 1s to consider all policies.

Networks: Abts et al. [2] proposed making datacenter networks consume energy in proportion to their traffic by using a flattened butterfly topology and dynamically changing the frequency of the communication links. Changing the link frequency changes both its data rate and its power consumption. RAFT [157] dynamically varies the frequencies and number of virtual channels in routers. PowerNetS [241] explores joint optimization of workload consolidation and network traffic routing to minimize total power.

Kim et al. [130] use spatial speculation to reduce the number of flits/bit-flips transmitted thereby reducing interconnect energy. On a miss, L1 caches fetch block-sized data from the next level. However, not all words within the block may be used in future and hence need not be fetched. It uses a predictor to determine which parts of the requested block is likely to be reduced. A misprediction causing a required word not to be fetched would cause it to be fetched when the word is requested resulting in latency overheads.

Accelerators: Being specialized, accelerators (including GPGPUs) are more energy-efficient than general purpose processing elements. The DySER [87] accelerator uses specialized circuit switching between computation units to eliminate instruction execution overheads. Venkatesh et al. [226] proposed specialized processors called conservation cores (c-cores) to reduce energy and energy-delay of hot code paths. The CPU and c-cores communicate through scan chains. KnightShift [232] uses a heterogeneous server architecture that adds a low power compute node along with the primary server. DreamWeaver [154] proposed using a co-processor called the Dream processor, that monitors and suspends incoming work requests along with a Weave scheduler that aligns and increases idle times.

6.6.1 Classification

We will now present a new classification system for power-performance management studies by the semantics of reconfiguration capabilities. Our classification system has five main semantic types, each of which have several subtypes with a total of twelve subtypes.

1. **Computation**:

    - *Organization*: Number and organization of structures, such as functional units, instruction windows, issue queues, pipelines, etc. that make up or control how processing elements (CPU cores, network routers, etc.) do computations. Examples: [9] [18] [40] [39] [81] [114] [228] [11] [68] [189] [137] [157].
    - *Speed*: Time to process and transform information. Core DVFS and RAPL studies are included in this class. We also include network router DVFS in this class, but place memory controller DVFS in the Storage class (see below).

Examples: [229] [86] [115] [145] [202] [207] [125] [171] [177] [182] [213] [141] [61] [192] [136] [224] [54] [190] §5 [155] [31] [51] [189] §3 [121] [139] [157].

- *Concurrency*: Number of processing elements or threads to do a computation. This includes core parking, hyperthreading, task creation/stealing, and workload consolidation studies.

  Examples: [171] [136] [224] [54] [208] [51] [114] [228] [44] [43] [241].

2. **Communication**:

   - *Latency*: Time to transmit a bit of information over an interconnect. This includes frequency scaling of memory and network links.

     Examples: [213] [62] [59] [61] [2].

   - *Bandwidth*: Number of bits of information that are moved per unit time. This includes fetch bandwidth and memory bandwidth throttling.

     Examples: [213] [62] [59] [61] [192] [31] [76] [2].

3. **Storage**:

   - *Size*: Number of storage cells available to applications. Not all cells may be available—they can be shut down or allocated to other applications. This includes cache power-gating and partitioning studies.

     Examples: [192] [10] [67] [238] [237] [126] [77] [190] §5 [128] [216] [84] [8] [97] [49] [178] [214] [175] [124] [159] [93] [217] [186] [146] [9] [20] [155] [31] [11] [68].

   - *Content*: Information stored in allocated storage cells. This includes cache replication, replacement, bypass, and compression studies.

     Examples: [5] [97] [45] [23] [79] [94] [165] [123] [173] [118] [119] [120] [131] [236] [69] [176] [185] [135] [234] [169] [219] [7].

- *Latency*: Time to read or modify storage cells. This include memory DVFS/RAPL studies.

  Examples: [213] [62] [59] [61] [192].

4. **Scheduling**:

   - *Spatial*: Particular processing elements on which to do the computation. This includes scheduling work on specific cores or accelerators.

     Examples: [177] [54] [80] [44] [43] [121] [87] [226] [232].

   - *Temporal*: When to do the computation. It may be scheduled later for coordinated management with sleep states.

     Examples: [137] [154] [139].

5. **Speculation**:

   - *Control*: This includes adapting structures that affect branch prediction and dealing with the predicted results.

     Examples: [147] [68].

   - *Data*: Amount of extra/less data to access than requested. This includes prefetching and access granularity studies.

     Examples: [225] [239] [17] [57] [85] [92] §3 [121] [130] [7].

Tables 6.1–6.3 summarize how the studies can be classified according to this system. The Count column in the tables show tuples of the form (n_1, n_2) where n_1 is the number of types and n_2 is the number of subtypes considered by the corresponding study. In these examples, at most three of the five possible types and at most four of twelve possible subtypes have been considered in any single study. This suggests the existence of potentially unexplored combinations. For example, simultaneous adaptivity

of communication and scheduling (such as, how can Varuna [208] coordinate with interconnect frequency scaling), communication and speculation (such as, how can MemScale[62] coordinate with adaptive prefetching), or storage and scheduling (such as, how can ICP [121] coordinate with cache resizing) does not seem to have been well explored.

Refs.	Computation			Communication		Storage			Scheduling		Speculation		Count
	Org.	*Speed*	*Conc.*	*Latency*	*BW*	*Size*	*Content*	*Latency*	*Spat.*	*Temp.*	*Control*	*Data*	
[18]	✓												1,1
[39]	✓												1,1
[40]	✓												1,1
[81]	✓												1,1
[141]		✓											1,1
[115]		✓											1,1
[86]		✓											1,1
[229]		✓											1,1
[125]		✓											1,1
[145]		✓											1,1
[182]		✓											1,1
[202]		✓											1,1
[207]		✓											1,1
[189]	✓	✓											1,2
[157]	✓	✓											1,2
[241]			✓										1,1
[208]			✓										1,1
[228]	✓		✓										1,2
[114]	✓		✓										1,2
[171]		✓	✓										1,2
[136]		✓	✓										1,2
[224]		✓	✓										1,2
[51]		✓	✓										1,2
[76]					✓								1,1
[2]				✓	✓								1,2
[67]						✓							1,1
[10]						✓							1,1
[8]						✓							1,1
[178]						✓							1,1
[216]						✓							1,1
[159]						✓							1,1
[217]						✓							1,1
[20]						✓							1,1
[84]						✓							1,1
[77]						✓							1,1

Table 6.1: Classification, by semantic types, of system reconfiguration capabilities.

Refs.	Computation			Communication		Storage			Scheduling		Speculation		Count
	Org.	*Speed*	*Conc.*	*Latency*	*BW*	*Size*	*Content*	*Latency*	*Spat.*	*Temp.*	*Control*	*Data*	
[128]						✓							1,1
[93]						✓							1,1
[126]						✓							1,1
[238]						✓							1,1
[186]						✓							1,1
[237]						✓							1,1
[214]						✓							1,1
[124]						✓							1,1
[49]						✓							1,1
[175]						✓							1,1
[146]						✓							1,1
[9]	✓					✓							2,2
[11]	✓					✓							2,2
[190]		✓				✓							2,2
§5		✓				✓							2,2
[155]		✓				✓							2,2
[31]		✓			✓	✓							3,3
[5]							✓						1,1
[169]							✓						1,1
[120]							✓						1,1
[236]							✓						1,1
[135]							✓						1,1
[79]							✓						1,1
[165]							✓						1,1
[219]							✓						1,1
[69]							✓						1,1
[173]							✓						1,1
[185]							✓						1,1
[131]							✓						1,1
[234]							✓						1,1
[119]							✓						1,1
[176]							✓						1,1
[94]							✓						1,1
[118]							✓						1,1
[45]							✓						1,1

Table 6.2: Classification (cont.), by semantic types, of system reconfiguration capabilities.

Refs.	Computation			Communication		Storage			Scheduling		Speculation		Count
	Org.	*Speed*	*Conc.*	*Latency*	*BW*	*Size*	*Content*	*Latency*	*Spat.*	*Temp.*	*Control*	*Data*	
[123]							✓						1,1
[23]							✓						1,1
[97]						✓	✓						1,2
[62]				✓	✓			✓					2,3
[59]				✓	✓			✓					2,3
[213]		✓		✓	✓			✓					3,4
[61]		✓		✓	✓			✓					3,4
[192]		✓			✓	✓		✓					3,4
[80]									✓				1,1
[232]									✓				1,1
[87]									✓				1,1
[226]									✓				1,1
[177]		✓							✓				2,2
[43]			✓						✓				2,2
[44]			✓						✓				2,2
[54]		✓	✓						✓				2,3
[154]										✓			1,1
[137]	✓									✓			2,2
[139]		✓								✓			2,2
[147]											✓		1,1
[68]	✓					✓					✓		3,3
[85]												✓	1,1
[225]												✓	1,1
[239]												✓	1,1
[57]												✓	1,1
[92]												✓	1,1
[130]												✓	1,1
[17]												✓	1,1
§3		✓										✓	2,2
[7]							✓					✓	2,2
[121]		✓							✓			✓	3,3

Table 6.3: Classification (cont.), by semantic types, of system reconfiguration capabilities.