Undersubscribed Threading on Clustered Cache Architectures

Wim Heirman[1,2] Trevor E. Carlson[1] Kenzo Van Craeynest[1]
Ibrahim Hur[2] Aamer Jaleel[3] Lieven Eeckhout[1]

[1] *Ghent University, Belgium*
[2] *Intel, ExaScience Lab* [3] *Intel, VSSAD*

Abstract

Recent many-core processors such as Intel's Xeon Phi and GPGPUs specialize in running highly scalable parallel applications at high performance while simultaneously embracing energy efficiency as a first-order design constraint. The traditional belief is that full utilization of all available cores also translates into the highest possible performance. In this paper, we study the effects of cache capacity conflicts and competition for shared off-chip bandwidth; and show that undersubscription, or not utilizing all cores, often yields significant increases in both performance and energy efficiency. Based on a detailed shared working set analysis we make the case for clustered cache architectures as an efficient design point for exploiting both data sharing and undersubscription, while providing low-latency and ease of implementation in many-core processors.

We propose ClusteR-aware Undersubscribed Scheduling of Threads (CRUST) which dynamically matches an application's working set size and off-chip bandwidth demands with the available on-chip cache capacity and off-chip bandwidth. CRUST improves application performance and energy efficiency by 15% on average, and up to 50%, for the NPB and SPEC OMP benchmarks. In addition, we make recommendations for the design of future many-core architectures, and show that taking the undersubscription usage model into account moves the optimum performance under the cores-versus-cache area trade-off towards design points with more cores and less cache.

1 Introduction

Increasing core counts on many-core chips require processor architects to design higher-performing memory subsystems to keep these cores fed with data. An important design trade-off is the allocation of die area and power budget across cores and caches [9, 19]. Adding cores increases the theoretical maximum performance of the chip, but sufficient amounts of cache capacity must be available to exploit locality; if not, real-world performance will suffer. Yet, the notion of *enough* cache capacity depends on the application's working set characteristics, which varies widely across applications, input sets and even different kernels or phases within an application. Designing a processor architecture that maximizes performance for most benchmarks (by maximizing core count) but does not cause unacceptable degradation when the working set does not fit in cache, is a difficult balancing act.

Proposals have been made to make the cache architecture more adaptive. Shared caches can exploit dynamic capacity allocation across cores, as can hybrid approaches such as Cooperative Caching [3], Dynamic Spill-Receive [17] or Reactive NUCA [8]. These techniques exploit heterogeneity of threads running on different cores. Yet, the main use case for many-core processors is to run data-parallel applications, which are often highly structured and typically employ fork-join parallelism or other bulk-synchronous paradigms. Threads therefore act homogeneously, which invalidates the assumption that some cores have unused cache capacity that can be used by others.

This application behavior advocates a workload-adaptive approach, in which the only remaining degree of freedom is to reduce the number of (concurrently executing) threads [6, 21, 22]. We propose ClusteR-aware Undersubscribed Scheduling of Threads (CRUST) as a mechanism to *dynamically* reduce the number of application threads, such as to match the application's working set size and off-chip bandwidth demands with the available on-chip cache capacity and off-chip bandwidth. CRUST reduces thread count for improving performance through locality, as well as for improving energy efficiency of bandwidth-bound applications. In the latter case, extra cores that are stalled on off-chip memory requests do not contribute to application performance and can be disabled to save power and energy. We implement and evaluate CRUST in the context of OpenMP fork-join based applications, which allows it to exploit phase behavior by being aware of parallel loop boundaries. The concepts used by CRUST can also be applied to other parallel runtimes, e.g., by dynamically changing the number of worker threads in applications that exploit request-based parallelism.

In addition, we perform a detailed shared working set analysis for a number of data-parallel applications, and make the case for clustered cache architectures as an efficient design point for many-core architectures. Clustered caches, or multiple last-level caches each shared by a subset of cores, enable data sharing within each cluster, while providing low access latencies (comparable to private caches) and ease of implementation (simpler than dynamic allocation policies in fully shared NUCA caches). We find clustered caches to be particularly beneficial in the context of undersubscription by providing a constant total-chip cache capacity, thereby improving performance, while conserving energy when a number of cores per cluster are disabled — a critical property that private caches do not provide.

Going one step further, and assuming CRUST as a runtime substrate, we find that the optimum many-core design point shifts towards processors with more cores and less cache. This result demonstrates that considering a CRUST runtime at design time leads to a higher-performing processor compared to a design process ignorant of undersubscription. It also leads to a processor system that can better deal with variations in workload behavior. Consequently, CRUST improves performance and energy-efficiency in practice when running a wide variety of workloads.

The key contributions made in this paper are:

- We make the case for clustered caches in future large many-core architectures, based on a detailed shared working set analysis, and show they are the ideal middle ground between private and shared (NUCA) caches. Moreover, clustered caches support the use case of undersubscription by keeping all cache capacity accessible even when some cores are disabled.

- We show how undersubscription of clustered cache architectures can improve performance and save energy; and propose ClusteR-aware Undersubscribed Scheduling of Threads (CRUST), a set of simple yet powerful mechanisms for dynamically adapting thread count. CRUST leads to performance and energy efficiency improvements of 15% on average, and up to 50%, for the NPB and SPEC OMP benchmarks. It improves on existing methods for bandwidth-aware threading [21] by taking data sharing and its effect on off-chip bandwidth requirements into account, next to being cache capacity aware.

- Finally, we revisit the cores-versus-cache area tradeoff for many-core processors, and show how taking the interplay of clustered caches and undersubscription into account at design time leads to higher-performance, adaptive architectures that are more resilient to variations in workload characteristics.

2 Motivation

In this section we analyze multi-threaded application performance as a function of thread count to illustrate the potential of undersubscription. We use the Sniper [2] simulator to model a 128-core architecture, and simulate the execution of the SPEC OMP [1] and NAS Parallel Benchmarks (NPB) [11] suites while varying thread counts. Our simulated hardware architecture models a clustered design with private L1 instruction and data caches and a shared last-level cache (LLC) for every four-core cluster. We run each benchmark with 32, 64, 96 and 128 threads, equally distributing threads across all clusters to always keep the full LLC capacity available. Unused cores are power-gated. For each combination of benchmark and thread count, we measure execution time and energy consumption for a number of the benchmark's iterations. We then report performance and energy efficiency[1] relative to full subscription, see Section 5 for more methodological details.

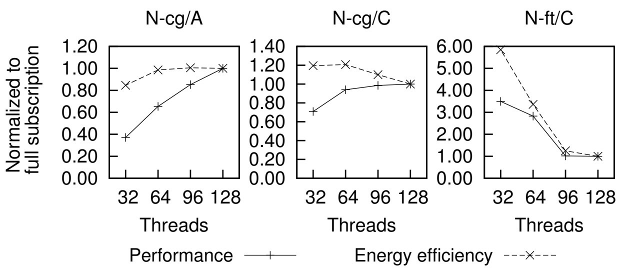

Figure 1. Performance and energy efficiency by thread count on a 128-core architecture.

As shown in Figure 1, applications can behave in a number of ways. The `N-cg/A`[2] benchmark (Figure 1 on the left) is compute-bound, reducing thread count leaves processor cores unused and degrades performance. Using the larger C input set, however, `N-cg/C` (Figure 1 in the middle) is off-chip bandwidth-bound when all cores are used. One can therefore reduce thread count by half without significant loss in performance, yet obtain energy savings of up to 20%. For other applications, reducing thread count can lead to a performance *increase*. The working set of `N-ft/C` (Figure 1 on the right) with 128 threads is too large for the available LLC capacity. Reducing thread count to only 32, leaving 75% of the chip's computational resources unused, results in a performance gain of over $3\times$, in addition to an increase in energy efficiency of almost $6\times$. These energy savings are achieved by both lowering runtime and reducing power consumption, caused by having fewer cores active and by avoiding off-chip memory accesses through improved data locality.

To summarize results over all benchmark and input set combinations, we first define what we consider the *optimum* thread count. As changing thread count has impact on both

[1]We define energy efficiency as $1/\text{energy}$, in useful work per Joule.

[2]We prefix applications from the NPB suite with `N-`, those from SPEC OMP with `O-`, and add the input set used separated with a `/`.

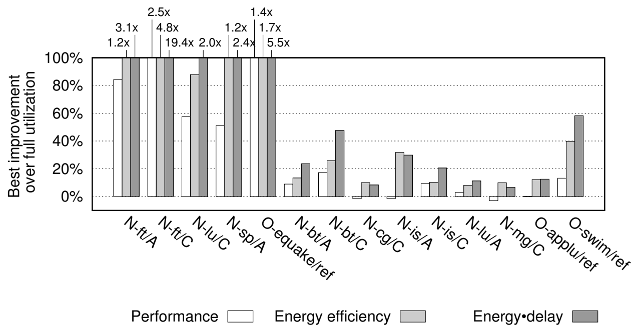

Figure 2. Potential improvements in performance, energy efficiency and energy-delay product (EDP) obtainable through undersubscription.

performance and energy efficiency, a single optimum may not readily exist. However, an undersubscribed chip always consumes less power than a fully subscribed one, simply because there are fewer active resources. Therefore, *if* an undersubscribed option is faster, it also consumes less energy. We further assume that in the high-performance compute space, users are interested in performance first, and will want to save energy whenever possible. We therefore define the optimum thread count, using a 5% guard band to account for application variability, as follows:

> An application's optimum thread count is the highest-performing option; or if no option is faster than full subscription by at least 5%, the most energy-efficient one that is no more than 5% slower than full subscription.

According to this definition, the optimum thread count for the workloads in Figure 1 becomes 128 (full subscription) for `N-cg/A`, 64 for `N-cg/C`, and 32 for `N-ft/C`.

Figure 2 summarizes the potential improvements in performance, energy efficiency and energy-delay product (EDP), when each application is run at its optimum thread count as defined above. The potential is significant. For the five leftmost benchmarks, which are cache capacity bound, we observe performance and energy efficiency improvements of over 50% caused by capacity effects. For the nine rightmost benchmarks, which are bandwidth-bound, undersubscription improves energy efficiency and EDP significantly with limited, if not a positive, impact on performance. On the other hand, for several applications no improvement is possible, i.e., using all cores provides the best option. This is the case for an additional 11 benchmarks (listed in Table 1 but not shown in Figure 2). It is therefore important to recognize the behavior of an application at runtime and select the optimum thread count through a dynamic algorithm.

3 Clustered Cache Architectures

Current many-core architectures mostly use either private caches for fast access to private data (e.g., Intel Xeon Phi [5]), or a globally shared last-level (NUCA) cache which avoids data duplication, thereby maximizing useful LLC capacity at the expense of access latency (e.g., Tilera TILE-Gx [18]). To trade off some duplication for faster private data access, NUCA architectures are often augmented by a management layer such as D-NUCA, at the cost of increased complexity.

In this section we perform a detailed working set analysis of a set of HPC-oriented multi-threaded benchmarks. We argue for clustered cache architectures as a good fit for this working set behavior. We then study the impact of undersubscription and show that it naturally aligns with clustered cache architectures.

3.1 Measuring shared working set sizes

To efficiently characterize working set sizes, we use the Cheetah cache simulator [20], which allows miss rates for a range of cache sizes to be computed simultaneously. A Pin tool [16] is used to instrument the workload and send each thread's memory access stream to a number of Cheetah cache models, allowing both private and shared working sets to be measured in a single run of each application, thread count and input set combination. We instantiate one set of cache models as being private to each thread; these characterize the per-thread working sets. Extra cache models are fed by the combined memory reference stream generated by groups of (adjacent) threads, in addition to a global model that sees all memory accesses.

Figure 3 plots the results of this analysis. All cache models are configured for a 16-way set-associate cache with LRU replacement. (We use the reduced iteration counts as outlined in Table 1.) The graphs show, for a given application, input set and thread count combination, the miss rate for various cache sizes. The *per-thread* line represents the private cache model, and plots the miss rate of a single thread when accessing a private cache. Reading each graph from left to right, by increasing cache size, sudden drops in miss rate are observed followed by plateaus of stable miss rates. At each drop, the cache became large enough to fully hold a new working set. In the case of `N-bt/C` (Figure 3(a)), this happens at 4 KB which corresponds to per-thread stack data; at 256 KB for working sets related to inner loops; and at 32 MB which is the total amount of data touched by a single thread in each iteration.

When comparing the per-thread miss rates with those of the combined *by-2*, *by-4* or *by-8* measurements, the amount of data shared between threads becomes apparent. For `N-bt/C`, the combined working set grows linearly with the number of combined threads, indicating that no data is shared. In contrast, `N-cg/C` (Figure 3(b)) has a 2 MB working set which does not grow when combining threads.

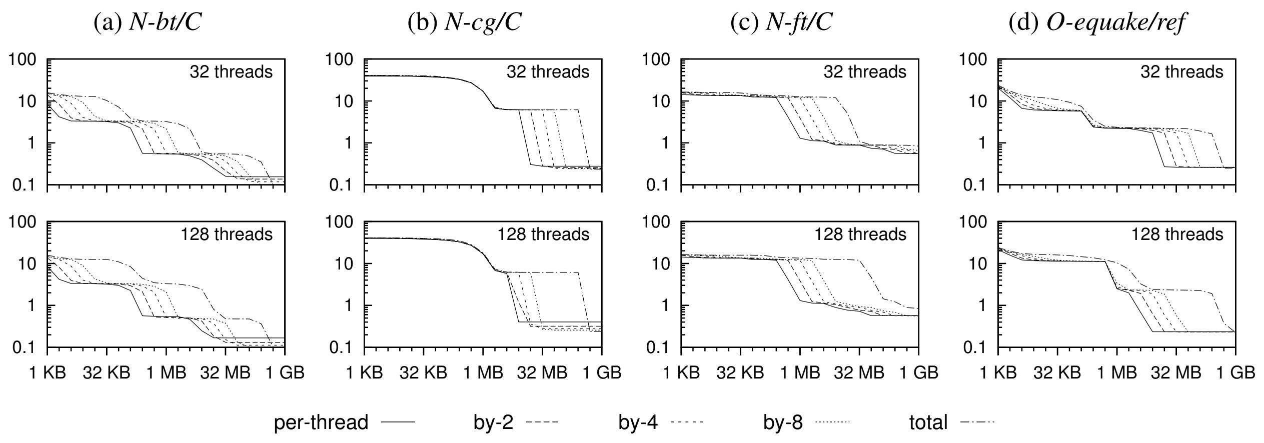

Figure 3. Working set analysis: miss rates for a 16-way set-associative cache with LRU replacement, with one cache per thread, per N threads, and a globally shared cache.

This working set is completely shared among all threads, which indicates that `N-cg/C` will favor those architectures where a shared cache capacity of at least 2 MB is available.

Dependence on thread count. As undersubscription changes the number of threads, it is important to know how working set sizes change with thread count. By comparing the top and bottom rows of Figure 3, which correspond to runs using 32 and 128 threads, respectively, we can observe a number of different scenarios. For the `N-ft/C` benchmark (Figure 3(c)), each thread has a constant 1 MB working set, and only the total working set size increases (from 32 MB to 128 MB) as more threads are used. In the case of `N-cg/C`, the shared 2 MB working set does not change between the 32-thread and 128-thread cases. However, there is a larger per-thread working set which shrinks when the number of threads is increased: 16 MB when using 32 threads, and under 8 MB with 128 threads. This behavior is a result of the constant *total* working set of 512 MB being data-partitioned across the different threads.

Summarizing the behavior observed across all benchmarks (see also Table 1), we consider the following categories of per-thread working set characteristics:

- Per-thread working set size reduces when spawning more threads: this behavior is typical for data partitioning, and usually happens for the larger working sets (e.g., the largest working set for `N-cg/C`).

- Per-thread working set size does not depend on thread count: this mostly occurs for smaller working sets, per-thread local data such as stacks, or globally shared structures (e.g., private 2 KB and 256 KB working sets of `N-bt/C` and the shared 2 MB working set of `N-cg/C`).

- Per-thread working set size increases with thread count: this behavior is less common, but occurs when optimizations are made that privatize data at the cost of increased memory usage.

An example of the last category can be found in the shared working set of `O-equake/ref` (Figure 3(d)). Here, a matrix-vector product is parallelized by having each thread compute partial sums for each element of the result vector, using only its private part of the matrix. These partial sums are later combined by the main thread which leads to this data set being shared.

Translating these different types of behavior into cache size requirements, we see that the smallest working sets usually do not change with thread counts. Miss rates in first-level caches are therefore not expected to change as a result of undersubscription.[3] The largest working sets are limited by the total data set size — which depends on input data but not on how this data set is partitioned across threads. For realistic input sizes, these working sets usually do not fit in on-chip caches, although their behavior can affect the effectiveness of large off-chip caches. More variation can be seen in the intermediate working set range. Here, both private and shared working sets appear, and correlation between thread count and working set size can be positive, negative or neutral.

3.2 A case for clustered caches

From the above analysis, a number of desirable properties for many-core cache architectures become apparent. When data sharing occurs, this should be exploited to avoid duplication, hence increasing effective cache capacity. This provides an argument for shared last-level caches. In ad-

[3]When co-scheduling threads on an SMT processor core, in which threads share L1-I, L1-D and TLBs, pressure on these resources does change which may be affected by undersubscription.

Benchmark	Iterations			Working set size			
	from :	*to* /	*total*	32t /	64t /	128t	type
NAS Parallel Benchmarks — class A input set							
`bt`	1 :	200 /	200		128 KB		private
`cg`	1 :	15 /	15	1 MB /	∼724 KB /	512 KB	private
`ft`	1 :	6 /	6		512 KB		private
`is`	1 :	10 /	10	8 MB /	4 MB /	2 MB	private
`lu`	1 :	250 /	250		4 MB		private
`mg`	1 :	4 /	4	32 MB /	16 MB /	8 MB	private
`sp`	20 :	39 /	400	8 MB /	∼6 MB /	4 MB	private
`ua`	1 :	200 /	200	1 MB /	∼724 KB /	512 KB	private
NAS Parallel Benchmarks — class C input set							
`bt`	2 :	2 /	200		256 KB		private
`cg`	2 :	2 /	75	16 MB /	∼12 MB /	8 MB	private
`ft`	2 :	2 /	20		1 MB		private
`is`	3 :	4 /	10	64 MB /	32 MB /	16 MB	private
`lu`	2 :	2 /	250	1 MB /	512 KB /	256 KB	private
`mg`	2 :	2 /	20		>16 MB		private
`sp`	2 :	2 /	400	32 MB /	16 MB /	8 MB	private
`ua`	2 :	2 /	200	16 MB /	8 MB /	4 MB	private
SPEC OMP(M) 2001 — reference input set							
`ammp`	2 :	2 /	200		1 MB		shared
`applu`	2 :	2 /	50	2 MB /	1 MB /	512 KB	private
`apsi`	2 :	2 /	50		128 KB		private
`equake`	8 :	14 /	3334	256 KB /	512 KB /	1 MB	shared
`fma3d`	2 :	2 /	522	32 MB /	16 MB /	8 MB	private
`gafort`	2 :	2 /	250	64 MB /	32 MB /	16 MB	private
`mgrid`	2 :	2 /	9		8 MB		private
`swim`	10 :	19 /	1200	64 MB /	32 MB /	16 MB	private
`wupwise`	2 :	2 /	200	64 MB /	32 MB /	16 MB	private

Table 1. Benchmarks, input sets and their relevant working set sizes.

dition, we want to support the use case where the combined working set from a number of threads does not fit the available LLC capacity. Here, thread count, and thus aggregated working set, can be reduced by disabling cores; yet all cache capacity must remain accessible to the active cores. At the same time, private accesses to relatively large working sets still make up for a large fraction of first-level cache misses. This provides an argument against large NUCA caches where hit latency depends on address assignment and can be very large. While dynamic techniques have been proposed to mitigate this effect, their additional variability and complexity are not always appreciated.

Clustered cache architectures on the other hand are easy to implement from a hardware point of view, are completely transparent to software, and require no active management. By keeping the cluster size C relatively small,[4] hits to private data can always be satisfied at low latency, while duplication is still reduced up to a factor C. In addition, undersubscription can be applied to clustered architectures by disabling up to $C-1$ cores per cluster while still being able to access all cache capacity. This effectively increases the available per-thread cache capacity by up to a factor of C.

Huh et al. [10] make a similar analysis w.r.t. data duplication, and find that cluster sizes of around four to eight are most optimum. Li et al. [13] favor four-core clusters as they find this configuration to optimize the energy-delay-area product. Lotfi-Kamran et al. [15] propose the scale-out processor architecture for datacenter workloads which clusters 16 to 32 (smaller) cores around a shared LLC, serving mostly instructions. Here, in this section, we made the case for a clustered cache architecture for scientific data-parallel workloads on many-core processors.

[4]The cluster size C is defined as the number of cores sharing each LLC.

4 Undersubscribed Thread Scheduling

As discussed in Section 2, the optimum thread count depends on the application and its input set, making it important to recognize the behavior of each application at runtime and select the optimum thread count through a dynamic algorithm. We propose ClusteR-aware Undersubscribed Scheduling of Threads (CRUST), a method that leverages hardware performance counter information to make this decision for each application phase. CRUST is implemented in the OpenMP runtime, and operates at the granularity of OpenMP parallel sections as identified in the application's source code by `#pragma omp parallel`. When a given parallel section is first encountered, statistics are collected during its execution. Different undersubscription levels may be explored during the following occurrences. After one or more executions of each section, CRUST will stabilize on an optimum thread count which is used for all of future occurrences of that section.

We propose two simple yet effective mechanisms that can quickly find the optimum thread count. The *descend* algorithm needs no knowledge of the architecture it is running on but can take a number of iterations to converge. The *predict* algorithm is able to converge much more quickly but instead requires some architecture-specific modeling.

4.1 *Descend* algorithm

The *descend* algorithm builds on the insight that, when starting from full subscription and incrementally reducing thread count, the following performance profiles may occur. Assuming an off-chip bandwidth-bound application, lowering thread count reduces the rate of requests made to off-chip memory, but as long as off-chip bandwidth is saturated, application performance will stay constant. At the same time, power and thus energy consumption reduce as fewer active cores are needed to obtain the same level of performance. For those benchmarks for which the working set is larger than the available cache capacity, reducing thread count will reach a point where suddenly the working set does become cache-fitting, and performance spikes up. Reducing thread count even further does not provide any additional gains beyond this step function in per-core performance while chip performance goes down as there are fewer active cores. Another possible case occurs when there is no change in LLC miss rates, but the number of active cores is reduced so far as to no longer fully saturate

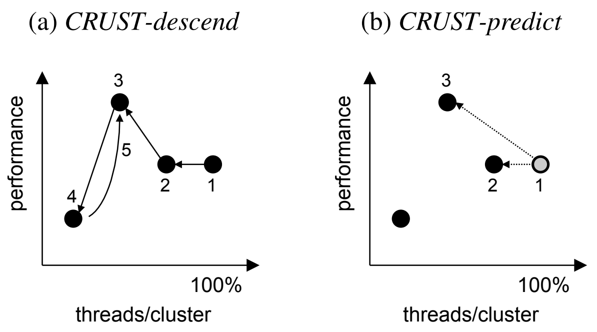

Figure 4. Flow of CRUST algorithms. *Descend* **starts at full subscription and reduces thread count as long as performance increases.** *Predict* **uses heterogeneous subscription to reduce the search space.**

off-chip bandwidth, again reducing performance. Degenerate cases occur when full subscription is not off-chip bandwidth bound, which implies that miss rates are low and that undersubscription will not yield any capacity effects either; or when the working set is too large to fit in cache even for the smallest possible thread count making all options off-chip bandwidth-bound. In summary, starting from full subscription and gradually reducing thread count, performance first stays constant as long as the application is off-chip bandwidth-bound, optionally makes an upwards step triggered by capacity effects, and finally reduces as too few active cores contribute to computation.

The *descend* algorithm exploits this behavior by starting with full subscription, then reduces thread count as long as either performance increases, or stays level but off-chip bandwidth utilization is high. (See Figure 4(a) for a schematic overview.) Each parallel section is tuned individually. It is initialized to use full subscription, while each subsequent occurrence of the section is run with one fewer thread per cluster. During each occurrence, hardware performance counters measure runtime, instruction count, and the number of DRAM accesses, from which aggregate IPC and DRAM bandwidth utilization are computed. By using aggregate IPC as a performance metric, the technique is resilient to occurrences of the same parallel section with different instruction counts.[5] Once performance is observed to decrease compared to the previous occurrence, calibration for that section ends and thread count is fixed to the previous (highest performing) setting. If full subscription was not bandwidth-bound (measured DRAM bandwidth was below 80% of its theoretical maximum), no improvement is to be expected from undersubscription and the *descend* algorithm immediately exits and selects full subscription.

[5]We disable spin loops in the runtime library using the environment option `OMP_WAIT_POLICY=passive`. Other ways exist to exclude spin loops from performance counter measurements.

In addition, LLC miss rates are monitored across occurrences of the same section, using misses per 1,000 instructions (MPKI) as a metric. An increase in miss rate while reducing thread count signifies a growing per-thread working set. This happens when this working set is the result of data partitioning, and a constant data set is partitioned over ever fewer threads. In this case, the increase in per-thread working set size negates the benefit of undersubscription, and full subscription will be used instead.

Finally, once the optimum thread count has been determined for a particular parallel section, it will be used for all future occurrences of that section. CRUST continues to monitor each section's LLC miss rate, and compares it to the miss rate present during its corresponding calibration phase. When significant changes in miss rate occur,[6] CRUST re-enters calibration for that parallel section. As the threading algorithm already specializes for specific loops (indicative for application phases) recalibration does not occur often, but may still happen as a result of data-dependent behavior.

4.2 *Predict* algorithm

While the *descend* algorithm assumes almost no architectural knowledge, convergence can take multiple iterations and moreover scales unfavorably when a large number of undersubscription choices exist, i.e., many cores share a last-level cache. We now provide an alternative algorithm that is guaranteed to converge in at most three steps, at the cost of a more architecture-specific model. The *predict* algorithm starts off by applying each of the possible per-cluster thread counts to one cluster each (sampling in space), i.e., one cluster runs with just a single thread, the second cluster has two threads, etc. In our baseline architecture, which has 32 clusters of four cores each, this uses up four clusters; the remaining 28 clusters can be fully subscribed to maximize performance during the calibration phase. The undersubscribed clusters, in contrast, will allow the miss rate for each of the possible thread counts to be measured. This direct measurement has to its advantage that no modeling is required, and that effects such as sharing inside a cluster, threads competing for cache capacity, and traffic generated by hardware prefetchers are all taken into account automatically.

At the end of a parallel section, miss rates are collected from each cluster. Per undersubscription setting t (denoting t active threads per C-core cluster), the algorithm now knows the expected miss rate m^t. In addition, the average off-chip memory latency L, and the *base* and *memory* CPI components are collected (CPI_{base} and CPI_{mem}, respectively). Latency and CPI components are averaged across the chip and need not be collected per cluster. We assume

[6]We use a relative difference in LLC MPKI of over 30% as the trigger point, in addition to an absolute difference of at least 3.0 MPKI to avoid recalibration when miss rates are insignificant.

Symbol	definition
Architectural parameters	
C	Cores per cluster
N	Number of clusters per chip
L_0	Uncontended off-chip memory latency (in cycles)
BW	Off-chip bandwidth (in number of accesses per cycle)
Calibration measurements	
t	Active threads per cluster
m^t	LLC miss rate for t threads (in misses per instruction)
$CPI_{base,mem}$	Base and memory CPI stack components (cycles per inst.)
L	Measured average off-chip memory latency (in cycles)
Computed values	
$AIPC$	Aggregate IPC of active cores (instructions per cycle)

Table 2. Variable definitions used in the *predict* **algorithm.**

these metrics are available as hardware performance counters, or can be derived from existing counters with sufficient accuracy. (See Table 2 for a summary of the variables used.)

From this measurement, the *predict* algorithm estimates the maximum performance under both bandwidth and compute bounds, for each thread count t. The available off-chip bandwidth determines the maximum number of off-chip requests that can be serviced per clock cycle (BW). Using the miss rate per instruction m^t, we estimate maximum chip performance (expressed as aggregate instructions per clock cycle, AIPC) when bound by off-chip bandwidth:

$$AIPC_{BW}{}^t = BW/m^t.$$

This differs from Bandwidth-Aware Threading [21] in that, rather than using linear extrapolation assuming constant miss rates, CRUST is able to use specific miss rates for each thread count, implicitly taking data sharing into account.

If the application cannot saturate off-chip bandwidth, it will be compute-bound and experience a DRAM latency close to the uncontended latency L_0. We estimate per-core CPI by first calculating the application's memory-level parallelism (MLP) by dividing the observed memory latency L by the cost per miss (with m the global miss rate under heterogeneous subscription):

$$MLP = \frac{L}{CPI_{mem}/m}.$$

We assume that memory-level parallelism, or how much memory latency can be overlapped versus how much of it directly impacts execution time, is constant. This is true for long-latency accesses that stall the reorder buffer, so the determining factor here is the number of concurrently outstanding memory references determined by the application's dependency chain. We then estimate $CPI_{mem}{}^t$ using the expected miss rate m^t and the average cost per memory access L_0/MLP:

$$CPI_{mem}{}^t = m^t \cdot \frac{L_0}{MLP}.$$

Combining both equations yields the following per-core CPI estimate:

$$CPI^t = CPI_{base} + CPI_{mem} \cdot \frac{L_0 \cdot m^t}{L \cdot m}.$$

As t threads are active on each of the N clusters, aggregate performance for compute-bound thread counts can therefore be estimated as:

$$AIPC_{comp}{}^t = \frac{t \cdot N}{CPI^t}.$$

We can now determine which thread counts t are bandwidth-bound ($AIPC_{BW}{}^t < AIPC_{comp}{}^t$) and which are compute-bound. For all bandwidth-bound options, performance will be equal, hence we prefer the lowest thread count as it activates the fewest cores and thus has the lowest energy consumption. Following the intuition on which the *descend* algorithm is built, the highest-performing option will be either the last bandwidth-bound option (defined as $t = t_{BW}$), or the next option down which will be compute-bound (with $t = t_{BW} - 1$). However, directly comparing $AIPC_{BW}{}^{t_{BW}}$ with $AIPC_{comp}{}^{t_{BW}-1}$ is problematic as $AIPC_{comp}$ is usually an overestimation — even applications that are not off-chip bandwidth-bound on average can still perceive DRAM latencies higher than L_0 because of bursty behavior and instantaneous bandwidth-induced bottlenecks. While it may be possible to better estimate the effective latency experienced by the compute-bound option, the required modeling quickly explodes in complexity and will be very architecture-specific. Instead, we opt to have the *predict* algorithm simply try out both t_{BW} and $t_{BW} - 1$ during the next two occurrences of the parallel section and select the highest-performing one.

Degenerated cases occur when the calibration phase concludes that no option is bandwidth-bound (full subscription is selected for highest performance) or when all options are bandwidth-bound (a single thread per cluster is selected for lowest energy usage). As with *descend*, the *predict* algorithm stores the miss rate m^t on which it based its decision and re-enters calibration when significant deviations occur.

5 Methodology

We now detail the methodology that was followed to evaluate the effectiveness of CRUST.

5.1 Baseline architecture

We evaluate CRUST on a clustered-cache 128-core processor as depicted in Figure 5. The chip consists of 32 tiles with four cores each. Each core has private L1 instruction and data caches and an L1 data prefetcher. L2 caches are shared by all four cores on the tile; L2s on different tiles are kept coherent through a MESI protocol with distributed tag directories. On-chip communication occurs over a 2-D mesh network with 512-bit wide links in each direction.

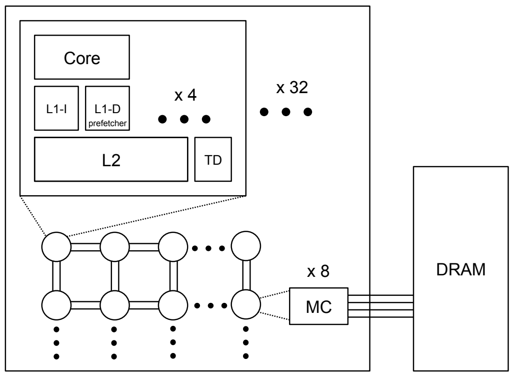

Figure 5. Simulated architecture. Four cores with private L1s share a last-level cache. 32 clusters are connected through a 2-D mesh network.

Component	Parameters
Core	1 GHz, 2-way issue OOO, 32-entry ROB
Branch predictor	hybrid local/global predictor
L1-I	32 KB, 4 way, 1 cycle access time
L1-D	32 KB, 8 way, 1 cycle access time
L1-D prefetcher	stride-based, 8 independent streams
L2 cache	1024 KB shared per tile, 16 way, 6 cycle
Coherence protocol	directory-based MESI, distributed tags
On-chip network	8×4 mesh, 4 cores per tile, 1 cycle per hop 64 GB/s per link per direction
Main memory	8 memory controllers, 45 ns access latency 64 GB/s total off-chip memory bandwidth

Table 3. Simulated architecture details.

Eight DRAM controllers are placed at the chip's edge, providing an aggregate 64 GB/s off-chip bandwidth. More micro-architectural parameters are listed in Table 3.

5.2 Simulation infrastructure

Sniper simulator. We use a modified version of the Sniper multi-core simulator [2], version 5.1, updated with a cycle-level core model. Sniper exploits parallelism allowing it to model our 128-core system at simulation speeds of up to 2 MIPS on modern multicore hardware.

Power consumption. McPAT 0.8 [13] is used to estimate power consumption. The McPAT integration available in Sniper was extended with a course-grain power-gating model, assuming the chip will power-gate individual cores once they have been idle for at least 10 µs. Most idle periods are much longer; on average, cores can be power-gated for over 80% of all idle time.

Applications. Workloads from the SPEC OMP [1] and the NAS Parallel Benchmarks (NPB) [11] suites were used to evaluate our dynamic threading method. As realistic working sets, and thus large input sizes, are required to

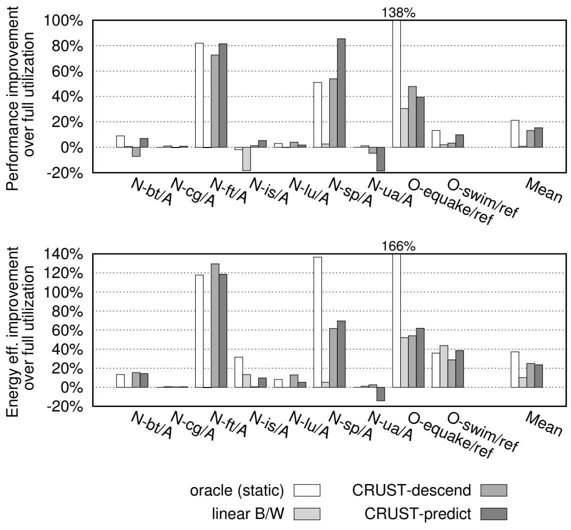

Figure 6. Improvement obtained in performance and energy efficiency through undersubscription over full subscription.

make this analysis meaningful, we use the reference inputs for SPEC OMP, and both class A and class C inputs for NPB, but we reduce the number of iterations that each benchmark executes to keep simulation times manageable. Table 1 lists the benchmarks that were used, and the iterations simulated in detail. The initial iterations (preceding those listed in Table 1) are used to warm up the caches.

6 Results and Analysis

Figure 6 summarizes performance and energy efficiency through undersubscription compared to full subscription. To allow for calibration, we ignore the first five iterations of all benchmarks. This means we are restricted to showing results for those benchmarks where iterations are short enough so that at least six iterations can be simulated in reasonable time. According to Table 1, this includes `O-equake` and `O-swim` from SPEC OMP, and all benchmarks from the NPB suite with the A input set except for `N-mg`. This set of benchmarks includes representatives of each type of behavior identified earlier (capacity effects, off-chip bandwidth-bound, and no advantage). In addition to our CRUST algorithms introduced in Section 4, we implement a variant of Bandwidth-Aware Threading [21]. This method, denoted as *linear B/W* in Figure 6, first runs a single thread on each cluster, then extrapolates observed bandwidth linearly to determine how many threads are needed to saturate off-chip bandwidth.

Analyzing the results, we can see that both CRUST algorithms are able to provide performance improvements of around 15% (harmonic mean of speedup), and improvements in energy efficiency of over 20%. As expected by our analysis in Section 2, the result is application-dependent.

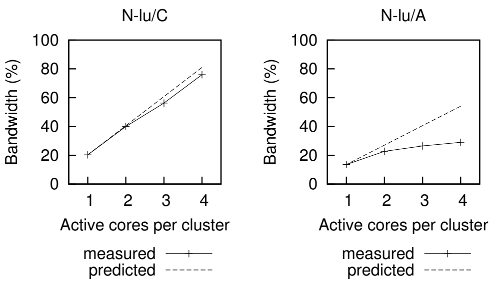

Figure 7. Scaling of DRAM bandwidth utilization by thread count can be non-linear.

Some applications do not show any benefit, and CRUST is able to predict that for these cases, full subscription is the best option. Other applications see significant capacity effects (most notably `N-ft`, `N-sp` and `O-equake`). In some cases, such as `N-sp/A`, CRUST achieves benefits that are even larger than the static *oracle* case, as CRUST can determine the optimum thread count for each parallel section individually. For one application, `N-ua/A`, CRUST degrades performance because this application has a large number of small parallel sections, each of which is tuned to a different thread count. Yet, because each loop is so short, `N-ua`'s performance is best when data sharing *across* parallel sections can be exploited, which is no longer the case when consecutive parallel sections use a different data partitioning. A simple solution to this problem is to aggregate parallel sections until a minimum size is reached, and change the thread count only at this larger granularity. Using an aggregation size of 50 ms we could avoid degradation of `N-ua` while not affecting the benefits for other applications.

Comparing CRUST with Bandwidth-Aware Threading (BAT), it is clear that BAT is not aware of the cache capacity effects that occur in many of the benchmarks. In fact, even for bandwidth-bound applications BAT can make suboptimal decisions by not taking data sharing into account. Figure 7 plots off-chip bandwidth utilization as a function of thread count. For `N-lu/C` (Figure 7 on the left), bandwidth indeed shows linear scaling so BAT can make the correct decision. When running the smaller class A input (Figure 7 on the right), however, a clustered cache can exploit data sharing between threads that execute on the same cluster. Bandwidth requirements now scale more slowly, causing BAT to overestimate bandwidth utilization and select a thread count that is too low resulting in lower performance. The opposite happens when the benchmark experiences capacity effects, and increasing thread count magnifies the combined per-cluster working set beyond its LLC capacity. Now, miss rates suddenly go up significantly, and so do bandwidth requirements. BAT is not aware of this superlinear bandwidth increase, and selects a thread count with poor locality. In contrast, CRUST is aware of both sharing and capacity effects, and can determine the optimum thread count taking these effects into account. It is able to do this by measuring miss rates for varying thread counts directly, either by running this thread count on the complete chip (*descend*) or on a select sample of clusters (*predict*).

7 Implications for Many-Core Design

So far, we evaluated undersubscription on a given many-core processor with a fixed architecture. We now go one step further and explore the following question: *How should one design a next-generation many-core processor architecture, given that dynamic undersubscription will be its usage model?* A key decision to be made during the design of a many-core processor is the allocation of area and power budgets between cores and caches. There is a natural tension between both, as each core's compute power contributes directly to application performance; yet, assuming the presence of data locality, sufficient cache capacity is required to keep these cores fed with data.

Studies that explore the cores versus cache area trade-off often assume a power-law relationship between cache capacity and performance [9, 19]. Its effect is that when devoting more area to cores, application performance initially goes up; but at some point caches become too small and average performance starts to decline. Krishna et al. [12] extend this relationship to take data sharing into account, and they observe that traditional methods can overestimate required cache capacity by not removing duplicate data from this equation, potentially shifting the optimum towards architectures with a smaller cache area ratio.

Undersubscription has a similar effect. The reason why performance decreases once the cache area ratio drops below a given point, is that the working set for *some* applications suddenly exceeds cache capacity. A gradual reduction of cache area makes progressively more applications fall into this category, resulting in the power law that characterizes average performance. For individual benchmarks, performance as a function of cache capacity is in fact a step function. Using undersubscription, the point at which this step occurs can be postponed: on architectures with more cores and less cache, these benchmarks can be undersubscribed, leading to, if not optimal, at least acceptable performance levels. This makes aggressive architectures (i.e., more cores, less cache) more interesting: as long as there are benchmarks that benefit from the increased number of cores, and thus achieve higher theoretical maximum chip performance, cache-limited benchmarks can use undersubscription to retain reasonable performance.

To quantify this effect, we set up the following experiment in which we project our architecture from Section 5 forward to a 14 nm technology node. Keeping in line with expected technological developments, off-chip bandwidth is kept constant at 64 GB/s, but we add 1 GB of on-package DRAM cache connected through a 512 GB/s interface. We

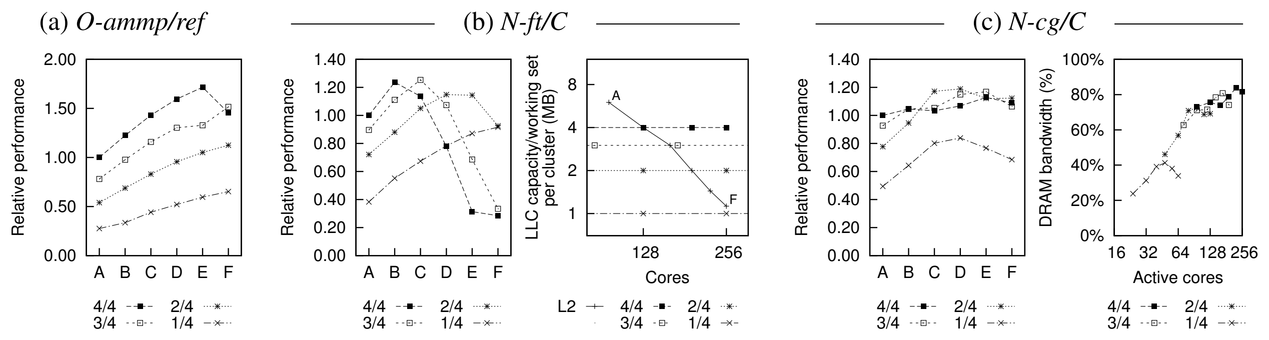

Figure 8. Performance of each architecture at different undersubscription levels.

Design	A	B	C	D	E	F
Cores	96	128	160	192	224	256
L2 size (MB per core)	1.5	1.0	0.8	0.5	0.4	0.3
Core fraction	25%	33%	40%	50%	58%	64%

Table 4. Cores-vs-cache area trade-off points.

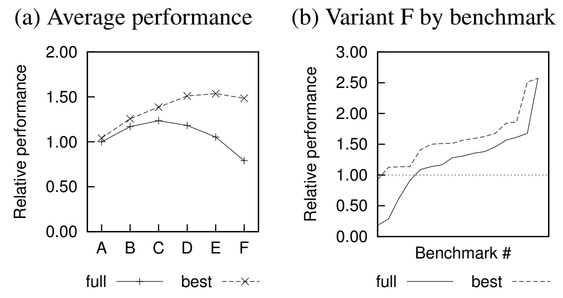

Figure 9. Average and per-benchmark performance for varying cache area tradeoff points under full and optimum subscription.

use Hill and Marty's notion of base core equivalent area (BCE) [9], and assign an area of 1 BCE per core and 2 BCEs per MB of last-level cache. We assume 1.5 mm^2 per BCE for a 14 nm process, as obtained through analysis of a number of publicly available die shots and accounting for less than ideal process scaling of a 35% area reduction per technology node. Six design options are summarized in Table 4. Each design option represents a different trade-off in core versus cache area, but adheres to a maximum area of 400 BCEs (equivalent to a 600 mm^2 die), which is typical for large HPC chips such as Intel's Xeon Phi [5] and high-end GPUs such as NVIDIA's Tesla [14].

Focusing results on specific applications first, Figure 8(a) shows that `O-ammp/ref` exhibits close to linear scaling when the number of cores is increased, except for the F variant where cache capacity finally falls below the application's working set. For this application, undersubscription always reduces performance. A benchmark amenable to undersubscription is `N-ft/C`, for which, according to Figure 8(b), full subscription (the 4/4 line) works well only on the A and B variants while performance quickly degrades from C onwards. Plotting `N-ft/C`'s working set versus the available last-level cache capacity of each architecture variant (Figure 8(b), right) makes clear why: the combined working set of four `N-ft/C` threads equals B's cache capacity, while for architectures with a larger fraction of area dedicated to cores ever fewer threads are needed to fill up all available cache capacity. This has a corresponding effect on the thread count needed to obtain optimum performance for `N-ft/C` on that architecture. Finally, `N-cg/C`, shown in Figure 8(c), is a bandwidth-bound application. Performance is more or less constant as long as at least around 64 cores are actively making DRAM requests, which is enough to saturate off-chip bandwidth.

When looking at average performance across all benchmarks,[7] Figure 9(a) shows that without taking undersubscription into account, a more conservative option, design C, would yield the highest average performance. However, while C may be a good compromise design able to provide acceptable performance across all benchmarks, undersubscription changes this balance. CRUST can improve average performance for all architectures, but those architectures with smaller cache sizes will benefit more, as their average performance was impacted more by the applications that did not fit in the cache. Now, the best average performance is no longer obtained by design C, but by design E which has 40% more cores, yielding 15% higher performance on average. Undersubscription allows the architecture to be more adaptive to differences across application's data access requirements. Figure 9(b) plots results per application for the most extreme design F (normalized to each benchmark's performance on A): applications at the right side of the graph are able to exploit the extra available cores, while those towards the left suffer from F's reduced cache capacity but can be 'reclaimed' through undersubscription.

[7]These experiments use static undersubscription and include all benchmark and input combinations listed in Table 1.

8 Related Work

Undersubscription has been proposed earlier in a number of different contexts, obtaining energy savings or performance improvements through a variety of mechanisms.

Hardware analysis. Guz et al. [7] provide a uniform way to describe how thread count affects performance. They identify a performance valley where thread count is too high for the available cache size (amongst other performance mechanisms), but not high enough to sustain sufficient outstanding off-chip requests to overlap latency and obtain the machine's peak arithmetic performance. Yet, when limited off-chip bandwidth is taken into account, most real architectures do not recover from this valley; moreover, trying to keep data on-chip can yield significant energy savings over massive multithreading by avoiding expensive (in both latency and energy) off-chip accesses.

Software techniques. Chen et al. [4] explore thread coarsening and propose how data partitioning changes at the algorithmic level can be used to improve data reuse across threads that share a (fully shared or clustered) cache. Volkov [22] focuses on GPU architectures, and shows how reduced occupancy can improve the use of registers (a resource shared between threads in GPUs) and avoid excessive queuing delays when accessing off-chip memory. Since these techniques involve application or algorithmic changes, they can achieve large benefits but require more manual work, making them less practical than approaches that can be integrated into the runtime or operating system.

Dynamic threading techniques. Bandwidth-Aware Threading (BAT), proposed by Suleman et al. [21], is a technique that uses undersubscription at the core level, and runs multi-threaded applications with just enough threads to saturate off-chip bandwidth. Unused cores are power-gated to save energy and power. As shown in Section 6, BAT's linear extrapolation of per-thread off-chip bandwidth requirements is suboptimal in the context of data sharing (sub-linear growth) or competitive cache capacity allocation (super-linear increase when transitioning from cache-fitting to bandwidth-bound operation).

Fedorova et al. [6] study the optimum number of SMT threads to be used on a Sun Niagara system. In addition to the SMT-specific issue of contention for execution resources, they identify L2 *thrashing* (i.e., capacity conflicts between threads active on a single SMT core) as a contributor to processor stalls. They then propose a non-work-conserving operating system scheduler that selects the number of active threads depending on the workload mix. The implementation of this scheduler consists of highly processor-specific modeling, however, and in addition requires miss rate profiles as a function of cache capacity to be collected upfront or provided by the compiler for each application. This scheduler is evaluated in the context of optimizing throughput for multiprogrammed workloads.

Yet, no existing solution is able to properly account for data sharing, or for the interaction between cache capacity effects and off-chip bandwidth usage in multi-threaded applications running on many-core processors. Both cache capacity and bandwidth are interrelated, i.e., a change in last-level cache miss rates has an immediate effect on off-chip bandwidth requirements. The CRUST approach promotes data sharing and cache capacity to first-class effects, but eschews modeling and profile input as much as possible to make the proposed methodology easily applicable. It exploits phase information that is readily available in the form of parallel section boundaries, and we show this technique to be able to improve both performance and energy efficiency when running multi-threaded applications.

9 Conclusions

We explore undersubscription in the context of multi-threaded applications running on many-core processor architectures. Reducing the number of concurrently executing threads can significantly improve cache hit rates, or can reduce pressure on saturated off-chip memory links. These effects result in significant increases in both application performance and energy efficiency. Whereas prior work could only realize limited gains in practice, we propose ClusteR-aware Undersubscribed Scheduling of Threads (CRUST), an approach that takes data sharing into account and is therefore able to achieve improvements in both performance and energy efficiency of 15% on average, and up to 50%, for a set of NPB and SPEC OMP benchmarks.

Based on these insights, we propose that future many-core architectures use clustered caches to exploit data sharing in parallel applications, and employ undersubscription to be adaptive to differences in application data requirements. This way, architectures can be designed that provide both higher average performance and consume less power and energy across diverse workload behaviors, while being relatively easier to implement than fully shared last-level caches.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. This work is supported by Intel and the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT). Additional support is provided by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295. Experiments were run on computing infrastructure at the ExaScience Lab, Leuven, Belgium; the Intel HPC Lab, Swindon, UK; and the VSC Flemish Supercomputer Center.

References

[1] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. Jones, and B. Parady. SPEComp: A new benchmark suite for measuring parallel computer performance. In R. Eigenmann and M. Voss, editors, *OpenMP Shared Memory Parallel Programming*, volume 2104, pages 1–10. July 2001.

[2] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011.

[3] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 264–276, June 2006.

[4] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 105–115, June 2007.

[5] G. Chrysos. Intel® xeon phi coprocessor (codename knights corner). In *Proceedings of the 24th Hot Chips Symposium*, 2012.

[6] A. Fedorova, M. Seltzer, and M. D. Smith. A non-work-conserving operating system scheduler for SMT processors. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, pages 10–17, June 2006.

[7] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser. Many-core vs. many-thread machines: Stay away from the valley. *Computer Architecture Letters*, 8(1):25–28, Jan. 2009.

[8] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 184–195, June 2009.

[9] M. Hill and M. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7):33–38, July 2008.

[10] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. Keckler. A NUCA substrate for flexible CMP cache sharing. *IEEE Transactions on Parallel and Distributed Systems*, 18(8):1028–1040, Aug. 2007.

[11] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS Parallel Benchmarks and its performance. Technical report, NASA Ames Research Center, Oct. 1999.

[12] A. Krishna, A. Samih, and Y. Solihin. Data sharing in multi-threaded applications and its impact on chip design. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 125–134, Apr. 2012.

[13] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. The McPAT framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(1):5, Apr. 2013.

[14] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, Mar. 2008.

[15] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi. Scale-out processors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 500–511, June 2012.

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, June 2005.

[17] M. Qureshi. Adaptive spill-receive for robust high-performance caching in CMPs. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 45–54, Feb. 2009.

[18] C. Ramey. Tile-gx100 manycore processor: Acceleration interfaces and architecture. In *Proceedings of the 23th Hot Chips Symposium*, 2011. Tilera Corporation.

[19] B. Rogers, A. Krishnaz, G. Bellz, K. Vuz, X. Jiangy, and Y. Solihin. Scaling the bandwidth wall: Challenges in and avenues for CMP scaling. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 371–382, June 2009.

[20] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. *SIGMETRICS Performance Evaluation Review*, 21(1):24–35, June 1993.

[21] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 277–286, 2008.

[22] V. Volkov. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference (GTC)*, volume 10, Sept. 2010.