Identifying Power-Efficient Multicore Cache Hierarchies via Reuse Distance Analysis

MICHAEL BADAMO, JEFF CASARONA, MINSHU ZHAO, and DONALD YEUNG,
University of Maryland at College Park

To enable performance improvements in a power-efficient manner, computer architects have been building CPUs that exploit greater amounts of thread-level parallelism. A key consideration in such CPUs is properly designing the on-chip cache hierarchy. Unfortunately, this can be hard to do, especially for CPUs with high core counts and large amounts of cache. The enormous design space formed by the combinatorial number of ways in which to organize the cache hierarchy makes it difficult to identify power-efficient configurations. Moreover, the problem is exacerbated by the slow speed of architectural simulation, which is the primary means for conducting such design space studies.

A powerful tool that can help architects optimize CPU cache hierarchies is reuse distance (RD) analysis. Recent work has extended uniprocessor RD techniques-i.e., by introducing concurrent RD and private-stack RD profiling—to enable analysis of different types of caches in multicore CPUs. Once acquired, parallel locality profiles can predict the performance of numerous cache configurations, permitting highly efficient design space exploration. To date, existing work on multicore RD analysis has focused on developing the profiling techniques and assessing their accuracy. Unfortunately, there has been no work on using RD analysis to optimize CPU performance or power consumption.

This article investigates applying multicore RD analysis to identify the most power efficient cache configurations for a multicore CPU. First, we develop analytical models that use the cache-miss counts from parallel locality profiles to estimate CPU performance and power consumption. Although future scalable CPUs will likely employ multithreaded (and even out-of-order) cores, our current study assumes single-threaded in-order cores to simplify the models, allowing us to focus on the cache hierarchy and our RD-based techniques. Second, to demonstrate the utility of our techniques, we apply our models to optimize a large-scale tiled CPU architecture with a two-level cache hierarchy. We show that the most power efficient configuration varies considerably across different benchmarks, and that our locality profiles provide deep insights into why certain configurations are power efficient. We also show that picking the best configuration can provide significant gains, as there is a 2.01x power efficiency spread across our tiled CPU design space. Finally, we validate the accuracy of our techniques using detailed simulation. Among several simulated configurations, our techniques can usually pick the most power efficient configuration, or one that is very close to the best. In addition, across all simulated configurations, we can predict power efficiency with 15.2% error.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; • **Computing methodologies** → *Modeling methodologies*

This research was supported in part by NSF grant CCF-1117042 and in part by DARPA grant HR0011-13-2-0005. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. government.
Authors' addresses: M. Badamo, University of Maryland, 1418 A. V. Williams, College Park, MD 20742; email: mbadamo@umd.edu; J. Casarona, Northrop Grumman Corporation, 9030 Junction Dr, Annapolis Junction, MD 20701; email: Jeff.Casarona@ngc.com; M. Zhao, MathWorks, 3 Apple Hill Dr, Natick, MA 01760; email: Minshu.Zhao@mathworks.com; D. Yeung, University of Maryland, 1323 A. V. Williams, College Park, MD 20742; email: yeung@umd.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2016 ACM 0734-2071/2016/04-ART3 $15.00
DOI: http://dx.doi.org/10.1145/2851503

Additional Key Words and Phrases: Cache performance, reuse distance, design space exploration, chip multiprocessors

ACM Reference Format:
Michael Badamo, Jeff Casarona, Minshu Zhao, and Donald Yeung. 2016. Identifying power-efficient multicore cache hierarchies via reuse distance analysis. ACM Trans. Comput. Syst. 34, 1, Article 3 (April 2016), 30 pages.
DOI: http://dx.doi.org/10.1145/2851503

1. INTRODUCTION

The primary goal of modern CPU design is to achieve high power efficiency. As a result, computer architects have stopped scaling clock rate, which incurs high costs in power consumption, in favor of CPUs that exploit greater amounts of thread-level parallelism. Today, commercial microprocessors with 8 to 12 state-of-the-art multithreaded cores are commonplace. Moreover, CPUs with tens of smaller multithreaded cores, such as Intel's Phi [Intel 2014], are gaining wider acceptance. In the near future, CPUs with hundreds of on-chip cores and/or threading contexts—i.e., large-scale chip multiprocessors (LCMPs) [Hsu et al. 2005; Zhao et al. 2007]—will become possible.

To realize the power-efficiency benefits of multicore CPUs, proper design of the on-chip cache hierarchy will be crucial. This is because the cache hierarchy can have a major impact on performance—for some programs, even more so than the cores themselves. At the same time, the cache hierarchy also significantly affects power consumption since caches make up a large fraction of the total chip area. (And through their performance impact, caches also affect core power as well.) Hence, due to their key role in determining both performance and power, multicore cache hierarchies offer designers enormous leverage to optimize CPU power efficiency.

Although cache hierarchy design will be crucial, at the same time it can be extremely challenging. A significant problem is the combinatorially large number of ways in which caches can be configured. For example, designers must choose the number of caching levels, the capacity employed at each caching level, whether individual caches are private to cores or shared between cores, and so forth. Designers may also need to choose the number of cores and/or threading contexts. The cross-product of all of these choices can lead to thousands of configurations. To identify the most power efficient among them, architects have traditionally relied on detailed simulation to explore the entire design space. Unfortunately, as multicore CPUs continue to scale, their cache design spaces will become even larger, making their exploration via simulation increasingly difficult.

A powerful tool that can potentially help architects optimize multicore caches for power efficiency is reuse distance (RD) analysis. RD analysis measures a program's memory RD histogram, or RD profile, capturing the application-level locality responsible for cache performance. In the past, architects have employed RD analysis extensively to study sequential caches [Ding and Zhong 2003; Zhong et al. 2003, 2009]. Its appeal stems from the fact that uniprocessor RD profiles are architecture independent, so once acquired, a single profile can be used to predict the cache-miss behavior of numerous cache configurations—e.g., every possible cache size—without having to simulate all of them. This can dramatically accelerate design space exploration.

To enable the same analysis benefits for multicore processors, researchers have recently extended traditional uniprocessor RD techniques to handle parallel programs running on multicore CPUs. In particular, parallel profiles have been introduced that can account for interthread interactions occurring within different types of multicore caches. For example, concurrent reuse distance (CRD) profiles [Ding and Chilimbi 2009; Jiang et al. 2010; Schuff et al. 2009, 2010; Wu and Yeung 2011, 2013; Wu et al.

2013] quantify reuse across thread-interleaved memory reference streams and account for thread interactions within shared caches. In addition, private-stack reuse distance (PRD) profiles [Schuff et al. 2009, 2010; Wu and Yeung 2013; Wu et al. 2013] quantify reuse within per-thread memory reference streams under invalidation-based coherence and account for thread interactions within private caches. Other types of parallel profiles have been proposed as well.

One significant issue for CRD and PRD profiles is that they are, in general, architecture dependent because they are derived from parallel memory reference streams whose interleavings are sensitive to timing. However, for homogeneous multithreaded parallel programs, simultaneous threads often execute similar code operating on separate data elements—i.e., from the same parallel loop—in which all threads make progress at roughly the same rate. Such parallel programs produce regular memory reference interleavings that exhibit only minimal perturbations across different cache configurations [Jiang et al. 2010; Wu and Yeung 2013]. Hence, for this important class of parallel programs, CRD and PRD profiles are virtually architecture independent and can be used to accelerate design space exploration just like their uniprocessor counterparts.

Despite all of the prior research on multicore RD analysis, its application to optimizing multicore cache hierarchies has been very limited. One preliminary study investigated choosing between private versus shared caches and how to allocate capacity between caching levels [Wu and Yeung 2012]. Another study considered how to scale cache as core count scales [Wu et al. 2013]. But there has been no research on optimizing power efficiency. Moreover, existing studies have only targeted cache-miss behavior, thus in fact there has been no work on optimizing either performance or power consumption. Instead, most prior work has focused on developing the profiling techniques and validating their accuracy.

This article conducts the first-ever study on optimizing multicore cache hierarchies for power efficiency using RD-based techniques. Our work enables architects to rapidly zero in on the most power efficient cache configurations for a given parallel program. In this article, we look for cache configurations that maximize throughput per watt—i.e., instructions-per-second per Watt—but our techniques will also work for any other power-efficiency metric. Due to the limitations of parallel locality profiles mentioned earlier, our current techniques target homogeneous multithreaded parallel programs only. Specifically, we apply our techniques on programs from high-performance computing (HPC) workloads that tend to have homogeneous threads with good load balance. For this important class of programs, we provide a new evaluation methodology that, when used in concert with simulation, can help architects effectively design multicore CPUs, especially as they continue to scale.

Our work makes several contributions. First, we extend existing multicore RD analysis for homogeneous multithreaded parallel programs with techniques that can predict power efficiency. To do so, we develop analytical models that use the cache-miss counts (CMCs) from our locality profiles to estimate performance and power consumption. Currently, our models target tiled CPUs with single-threaded in-order cores [Zhang and Asanovic 2005]. Although future scalable CPUs will likely employ multithreaded (and even out-of-order) cores, we assume single-threaded in-order cores to simplify the core models, permitting us to focus on the cache hierarchy and our RD-based techniques. To model performance, we derive CPU execution time from estimates of average memory access time. To model power, we employ the McPAT [Li et al. 2009] and DSENT [Sun et al. 2012] tools. We use McPAT for the cores, caches, directories, and memory controllers (MCs), and DSENT for the on-chip network. Although our locality profiles provide all necessary information to drive our memory hierarchy models, they do not yield any information on the cores. We run a single cores-only simulation per benchmark to provide the cores' baseline execution time and dynamic power consumption.

Second, to demonstrate the usefulness of our techniques, we apply them to optimize the design of tiled CPUs. We consider the problem of selecting the best configuration for a two-level multicore cache hierarchy, including choosing the L1 and L2 cache sizes that achieve the highest power efficiency. In addition, although we employ private L1 caches, we also consider selecting the best sharing degree within the L2 cache. In particular, we allow configurable clusters of tiles to aggregate their L2 slices into a single cache that is logically shared among intracluster cores. Hence, by varying cluster size, we can tune the amount of sharing exploited by the architecture. Our study considers designing tiled CPUs with 64, 128, and 256 single-threaded cores.

Our results show that the most power efficient configurations vary considerably across benchmarks. They tend to employ L1 and L2 cache sizes that coincide with large drops in the PRD and CRD profiles (which are application dependent), thus capturing a program's important working sets while simultaneously minimizing the amount of on-chip cache. The most power efficient configurations may also employ either small or large cluster sizes, but the latter is profitable only when significant sharing occurs at the "right" capacities—i.e., across reasonable sizes for the L2 cache. In addition, we find that picking the most power efficient configuration can provide a significant benefit. Across the entire cache hierarchy design space, there is a 2.01x spread in power efficiency. And on average, the best configuration achieves 22.1% higher power efficiency than all other configurations in the design space. We also find the best suite-wide configuration can come close to the absolute best power efficiency achieved per benchmark.

Finally, we conduct a validation of our techniques' accuracy. In particular, we used detailed simulation to measure the actual performance and power consumption of several cache hierarchy configurations randomly sampled across the design space. Then, we compared the simulated results against the predictions made from our RD-based performance and power models. We find that our techniques can usually identify the sampled configuration with the highest power efficiency, or at least a sampled configuration that is within 5% of the best. Additionally, across all sampled configurations, our techniques predict power efficiency with a 15.2% error on average.

The remainder of this article is organized as follows. First, Section 2 reviews previous work on multicore RD analysis. Then, Section 3 presents our analytical models for predicting CPU performance and power from the locality profiles. Next, Section 4.2 presents our design study to identify the most power efficient multicore cache configurations and validates the model-based predictions against detailed simulations. Finally, Sections 5 and 6 cover related work and conclusions, respectively.

2. MULTICORE RD ANALYSIS

In the past, RD (also known as LRU stack distance [Gecsei et al. 1970]) has been used to evaluate uniprocessor cache performance. This evaluation methodology employs RD profiles, which are histograms of RD values for all memory references in a sequential program. (A memory reference's RD value is the number of unique memory blocks referenced since the last memory reference to the same data block.) Once an RD profile has been acquired, it can be used for cache-miss prediction. Because a cache of size C can satisfy references with RD $< C$ (assuming LRU management), the cache's miss count is the sum of all reference counts in an RD profile above the RD value for capacity C. And since this calculation can be performed for any C, a single RD profile can be used to predict the CMC at any cache capacity.

Recently, researchers have extended this basic notion of sequential RD to handle parallel programs running on multicore CPUs. This section reviews the new parallel

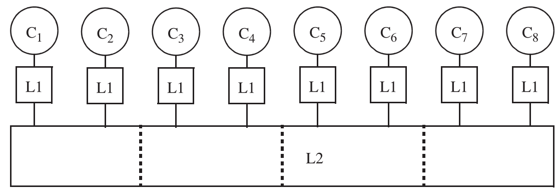


Fig. 1. Example two-level multicore cache hierarchies.

Time:	1	2	3	4	5	6	7	8	9	10	11
Core C_1:	A	B	C		D		E			A	C
Core C_2:				F		C (A)		G	H		

Fig. 2. Two interleaved memory reference streams, illustrating different thread interactions.

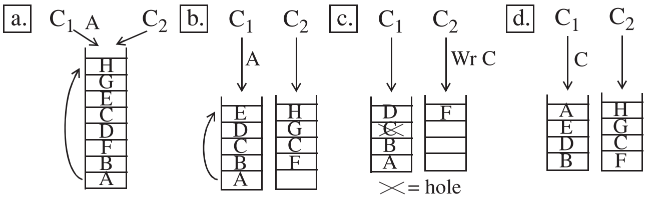


Fig. 3. LRU stacks showing dilation and overlap for CRD (a), scaling, replication (b), and holes (c) for PRD, and forwarding for PRD_f (d).

profiling techniques (Sections 2.1 through 2.3) and explains several characteristics of the locality profiles that are commonly found in parallel programs (Section 2.4).

2.1. CRD Profiles

To handle parallel programs, sequential RD profiling must be extended in two ways. First, profiling must accommodate *multiple* memory reference streams. Thus parallel profiling techniques are needed. Second, the parallel profiling techniques must account for *interthread interactions* that occur in different types of caches. For example, multicore CPUs often contain a combination of shared and private caches, as illustrated in Figure 1. In particular, the solid lines in Figure 1 show an eight-core CPU consisting of a single level of private cache backed by a shared L2 cache. The thread interactions within these private and shared caches differ, so separate profiling techniques are needed to quantify the different interactions, as well as the impact they have on aggregate parallel program locality.

CRD profiles report locality for thread-interleaved memory reference streams, thus capturing interference effects in shared caches. CRD profiles can be measured by applying the interleaved stream on a single (global) LRU stack [Jiang et al. 2010; Schuff et al. 2009, 2010; Wu and Yeung 2011, 2013; Wu et al. 2013]. For example, Figure 2 illustrates the interleaving of two cores' memory references accessing eight memory blocks, A–H. Figure 3(a) shows the state of the global LRU stack when core C_1 re-references A at $t = 10$. C_1's reuse of A exhibits an intrathread $\text{RD} = 4$, but the CRD that accounts for interleaving is 7. In this case, $\text{CRD} > \text{RD}$ because some of C_2's interleaving references (F–H) are distinct from C_1's references, causing *dilation* of intrathread RD.

In many parallel programs, threads share data that offsets dilation in two ways. First, it introduces *overlapping memory references*. For example, in Figure 2, although C_2's reference to C interleaves with C_1's reuse of A, this does not increase A's CRD, because C_1 already references C in the reuse interval. Second, sharing also introduces *intercepts*. For example, if C_2 references A instead of C at $t = 6$ in Figure 2, then C_1's reuse of A has CRD $= 3$, so CRD is actually *less* than RD.

Once acquired, a CRD profile can be used to predict the number of cache misses incurred by a shared cache of any capacity. Moreover, these cache-miss predictions account for the dilation, overlap, and intercept effects described earlier that occur within the shared cache.

2.2. PRD Profiles

PRD profiles report locality across per-thread memory reference streams that access coherent private caches. PRD profiles can be measured by applying threads' references on private LRU stacks that are kept coherent. Without writes, the private stacks do not interact. For example, Figure 3(b) shows the PRD stacks corresponding to Figure 3(a), assuming that all references are reads. For C_1's reuse of A, PRD $=$ RD $= 4$. Note, however, the multiple private stacks still contribute to increased cache capacity. (Here, the capacity needed to satisfy PRD $= 4$ is 10, assuming two caches with five cache blocks each.) To capture this effect, we compute the scaled PRD (sPRD), which equals $P \times PRD$, where P is the total number of cores (or threads) profiled. Hence, for C_1's reuse of A, sPRD $= 8$.

In PRD profiles, read sharing causes *replication*, increasing overall capacity pressure. Figure 3(b) shows duplication of C in the private stacks. Because PRD scaling aggregates private LRU stacks, replication effects are automatically captured in sPRD profiles. In contrast, write sharing causes *invalidation*. For example, if C_2's reference to C is a write instead of a read, then invalidation would occur in C_1's stack, as shown in Figure 3(c). To prevent invalidations from promoting blocks further down the LRU stack, invalidated blocks become *holes* rather than being removed from the stack [Schuff et al. 2009]. Holes are unaffected by references to blocks above the hole, but a reference to a block below the hole moves the hole to where the referenced block was found. For example, if C_1 were to reference B in Figure 3(c), D would be pushed down and the hole would move to depth 3, preserving A's stack depth.

Once acquired, an sPRD profile can be used to predict the number of cache misses incurred by a group of private caches of any capacity. Moreover, these cache-miss predictions account for the scaling, replication, and invalidation effects described earlier that occur within the private caches.

2.2.1. PRD$_f$. PRD (and sPRD) profiles predict cache misses from a core's local private cache hierarchy. If the private cache in question resides at the "sharing point" of the CPU, e.g., at level 2 in Figure 1, then the desired cache block may be found in a remote private cache. In this case, the cache coherence protocol may access the desired block from the remote cache rather than access the next caching level. We refer to these as "forwarding" cache transactions. An additional profile is needed to account for the hits and misses that occur in the remote private caches, which effectively act as another caching level that backs up the private cache in question.

We define PRD$_f$ to account for locality in remote private caches. PRD$_f$ profiles are acquired using the same private LRU stacks for PRD profiles. Moreover, like PRD, each memory reference updates the stacks in the manner illustrated by Figure 3(b) and (c). However, in PRD$_f$, a memory reference's RD value is computed by checking *all* stacks and is set to the *minimum* stack depth found globally. Thus, whereas a memory reference's PRD value represents the minimum cache capacity needed to keep the data block in the local private cache, a memory reference's PRD$_f$ value represents the minimum cache capacity needed to keep the data block in any private cache and hence enable forwarding cache transactions. For example, Figure 3(d) shows the state of the private stacks from Figure 3(c) advanced to $t = 11$ when C_1 re-references block

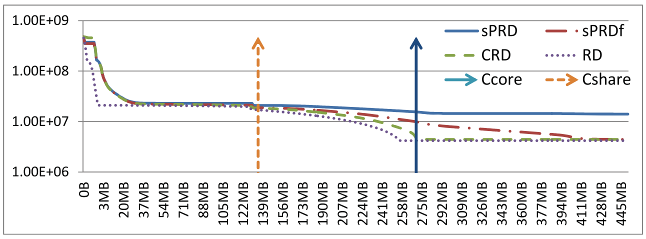


Fig. 4. CMC as a function of cache capacity computed from the sPRD, sPRD$_f$, CRD, and RD profiles for the radix benchmark running on 256 cores. Arrows highlight the C_{share} and C_{core} capacities.

C. Due to the invalidation of C in C$_1$'s stack by C$_2$, C$_1$ suffers a local private cache miss (PRD $= \infty$). But C is found in C$_2$'s stack at depth 2, so PRD$_f$ $= 2$. Similar to sPRD, we also compute scaled PRD$_f$ (sPRD$_f$) as $P \times PRD_f$ to account for the aggregate private cache capacity. In the example in Figure 3(d), sPRD$_f$ $= 4$.

2.3. PRD^{C_l} Profiles

The CRD and PRD profiles described in Sections 2.1 and 2.2 address the basic shared and private caches found in multicore processors. Multicore CPUs may also employ *cluster caches*, especially when scaling to LCMPs. These are hybrid caches in which clusters of cores access a shared cache, but all of the per-cluster caches are treated as private caches requiring coherence. For example, Figure 1's dotted lines suggest dividing the L2 cache into four clustered L2s, each shared by two cores. For simplicity, we always assume that all clusters contain the same number of cores, C_l, which evenly divides the total number of cores, P. (In Figure 1, $C_l = 2$ and $P = 8$.)

Because cluster caches are hybrids between shared and private caches, their locality profiles can be acquired using a combination of the CRD and PRD profiling techniques described earlier. In particular, the locality profile for a group of cluster caches is the PRD profile acquired using $\frac{P}{C_l}$ private stacks in which each private stack's memory reference stream is formed by interleaving the streams from the C_l cores belonging to the cluster. We call the locality profile acquired in this fashion PRD^{C_l} and the forwarding version (using the techniques from Section 2.2.1) PRD$_f^{C_l}$. In addition, we call the scaled PRD^{C_l} and scaled PRD$_f^{C_l}$ profiles that account for aggregated private cache capacity sPRD^{C_l} and sPRD$_f^{C_l}$.

Notice that the C_l parameter in clustered profiles defines a family of caches with different cluster sizes. Moreover, the basic private and shared caches are degenerate cases within this family in which $C_l = 1$ and $C_l = P$, respectively. Thus, the PRD and CRD profiles from Sections 2.1 and 2.2 are equivalent to PRD^1 and PRD^P.

2.4. Locality Characteristics

From our experience, we find that locality profiles exhibit certain characteristics that are common to all homogeneous multithreaded parallel programs. Understanding these locality characteristics can shed light into how to optimize multicore cache hierarchies and why certain configurations are optimal. To illustrate, Figure 4 plots the CMC profiles corresponding to the sPRD, sPRD$_f$, and CRD profiles for radix from SPLASH-2 [Woo et al. 1995] running on a 256-core CPU; the CMC profile for radix's RD profile, i.e., the one-core case, is also plotted. Each CMC profile reports the cache misses predicted by its corresponding locality profile as a function of, e.g., $CMC(CRD, i) = \sum_{j=i}^{\infty} CRD[j]$ for the CRD profile. Along the x-axis, RD is plotted in terms of capacity, and along the y-axis, CMC is plotted on a log scale.

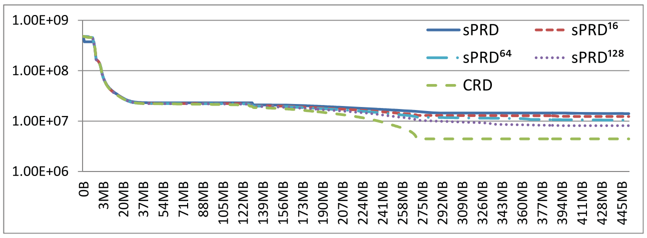


Fig. 5. CMC as a function of cache capacity computed from the sPRD, sPRD^{C_l}, and CRD profiles for the radix benchmark running on 256 cores.

Figure 4 shows that CMC profiles are monotonically decreasing (a direct consequence of the definition of CMC), with the drop being highly nonuniform. In Figure 4, the CMC profiles are relatively flat across certain capacities but sometimes drop steeply. Each significant drop corresponds to a "working set," and the capacity across which the drop occurs indicates the working set's size. The existence of working sets in sequential programs has been known for quite some time [Denning 1968]. Figure 4 shows that the same working sets found in sequential programs (RD profiles) also appear in the parallel versions of the programs (sPRD, sPRD$_f$, and CRD profiles).

Besides locality characteristics inherited from sequential programs, Figure 4 also reveals several new locality characteristics for parallel programs pertaining to the relationship between their sPRD, sPRD$_f$, sPRD^{C_l}, and CRD profiles. First, in Figure 4, we see that sPRD and CRD profiles are coincident at small cache sizes but eventually diverge. Additionally, as cache size increases, the sPRD-CRD gap increases, reaching a maximum at large cache sizes. This is due to *sharing*. Previous work has shown that in the absence of sharing—in particular, no replication or invalidations in PRD stacks—sPRD and CRD profiles are identical [Wu and Yeung 2011, 2013]. However, when threads share data, overlap occurs in shared caches, allowing shared caches to provide a cache-miss reduction over private caches—i.e., CRD $<$ sPRD. The fact that sPRD and CRD profiles diverge only after a certain point in Figure 4 suggests that sharing (and the cache-miss benefit of shared caches) is capacity dependent. We call the cache size at which sPRD and CRD profiles diverge and sharing manifests itself, C_{share}.

Second, Figure 4 also shows that sPRD and sPRD$_f$ profiles exhibit a similar relationship: they are initially coincident, diverge after C_{share}, and exhibit an increasing gap with cache size scaling. As discussed in Section 2.2.1, PRD$_f$ measures temporal reuse across *all* PRD stacks. In the absence of sharing, PRD stacks contain mutually exclusive data, so a core's PRD$_f$ is unaffected by the remote PRD stacks and sPRD$_f$ = sPRD. Beyond C_{share}, sharing occurs so that replicas can exist in remote PRD stacks at potentially shallower stack depths, allowing sPRD$_f$ $<$ sPRD. This sPRD-sPRD$_f$ gap quantifies the number of local private cache misses that can be satisfied from remote private caches—i.e., forwarding cache transactions. Notice that Figure 4 shows that sPRD$_f$ profiles can be significantly lower than sPRD profiles, suggesting that remote private caches can satisfy many local private cache misses. (In fact, the sPRD$_f$ profile in Figure 4 eventually re-merges with the CRD profile, so the sPRD-sPRD$_f$ gap is equal to the sPRD-CRD gap. This occurs when cache capacity is sufficiently large to hold all of the extra replicas that exist in private caches but not in shared caches.)

Third, the sPRD-CRD and sPRD$_f$-CRD gaps are "bridged" by sPRD^{C_l} and sPRD$_f^{C_l}$ profiles, respectively. As discussed in Section 2.3, cluster caches are hybrids, with private and shared caches comprising the degenerate cases. Hence, sPRD^{C_l} profiles lie in between sPRD and CRD profiles, starting from sPRD and moving toward CRD as C_l increases from 1 to P. Figure 5 illustrates this behavior. In this figure, we include the same curves labeled "sPRD" and "CRD" from Figure 4, but we also plot the sPRD^{C_l} curves for $C_l = 16$, 64, and 128, showing how they bridge the sPRD-CRD gap. sPRD$_f^{C_l}$ profiles behave similarly, except they lie in between sPRD$_f$ and CRD profiles. The family of sPRD^{C_l} and sPRD$_f^{C_l}$ profiles quantify the benefits of increasing cluster size to reduce intercluster replicas and invalidations. Since sPRD, sPRD$_f$, and CRD profiles are coincident below C_{share}, these clustering benefits only occur for cache capacities larger than C_{share}.

Finally, scaling core count degrades locality and increases cache misses, shifting parallel locality profiles (sPRD, CRD, and everything in between) to larger cache capacities. For example, the gap between the CRD and RD profiles in Figure 4 quantifies the cache-miss increase for shared caches due to scaling core count from 1 to 256 cores. This CRD-RD gap quantifies the impact of destructive interference in shared caches caused by interleaving threads' memory references to nonoverlapping data. Notice that Figure 4 shows that shared cache locality degradation only occurs up to a certain cache size after which CRD and RD profiles converge. This is because parallelization scope in programs is limited, e.g., within a parallel loop. Because parallel regions typically access less data than a program's total memory footprint, the amount of destructive interference is also limited. We call the cache size at which CRD and RD profiles converge, C_{core}. Essentially, C_{core} quantifies a program's "parallel working set size."

3. PERFORMANCE AND POWER MODELS

Having discussed previous multicore RD analysis techniques, we now develop models that take as input the cache-miss predictions from our locality profiles and compute CPU execution time and power consumption. First, we describe the tiled CPU architecture that our work targets (Section 3.1). Then, we present our techniques for modeling performance (Section 3.2) and power (Section 3.3).

3.1. Tiled CPUs

The locality profiles discussed in Section 2 can analyze any multicore cache hierarchy. But when considering performance and power consumption, we must model all aspects of the CPU, not just its caches. Thus, we must settle upon a specific CPU architecture. In this work, we assume *tiled CPUs* [Zhang and Asanovic 2005]. A tiled CPU consists of several identical replicated tiles, each with its own compute, memory, and communication resources. Tiled CPUs are highly scalable [Zhang and Asanovic 2005; Hardavellas et al. 2009], so they are good candidates for future LCMPs, which is the focus of our work.

Figure 6 shows a small 16-core example of a tiled CPU (later, we consider CPUs with up to 256 cores). Each tile contains a core, an L1 cache, an L2 cache slice, and a switch for a two-dimensional on-chip point-to-point mesh network. We assume simple in-order cores that issue a single instruction per cycle. Moreover, our cores implement a relaxed memory consistency model, permitting store buffers to mitigate write-miss latencies. In addition to the cores, we assume an on-chip cache hierarchy that includes different types of caches, as depicted in Figure 1. In particular, we assume that the L1 caches are always private, but the L2 cache slices can be configured as private, shared, or cluster caches. This is done by clustering tiles and treating all of the L2 slices from the same cluster as a single logically shared cache. By varying the logical cluster size from $C_l = 1$ to $C_l = P$, the L2s can span the entire family of caches between private and fully shared (see Section 2.3). Hence, cluster size serves as a "knob" for controlling the amount of sharing exploited in our tiled CPU.

Besides cores and caches, our CPU also integrates multiple MCs on-chip, each of which is connected to a separate DRAM channel. As illustrated in Figure 6, we assume that every fourth compute tile is also a "memory tile" with an integrated MC. As we

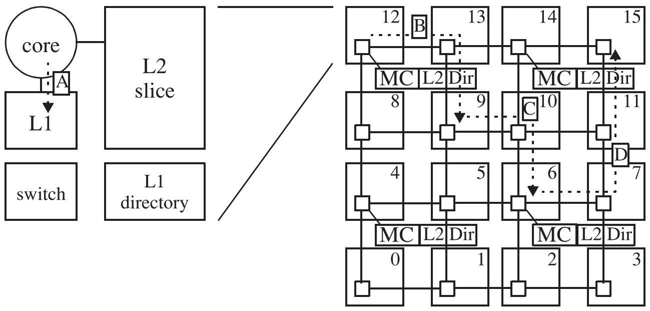


Fig. 6. Tiled CPU assumed in our study. Each tile contains a core, a private L1 cache, an L2 cache slice, an L1 directory slice, and an on-chip network switch. “Memory tiles” also have an MC and an L2 directory slice. The coherence protocol is hierarchical, with per-tile L1 directory slices tracking L1 cache blocks and per-memory tile L2 directory slices tracking L2 cache blocks. Dotted arrows labeled “A” through “D” indicate different memory transactions.

scale the CPU, this creates a large number of off-chip I/Os. We assume DRAM dies can be stacked on top of the CPU die [Loh 2008], providing the I/Os and memory bandwidth needed to support LCMP-sized CPUs. Physical memory is interleaved across the MCs at cache-block granularity such that consecutive cache blocks are “homed” on different MCs.

3.1.1. Cache Coherence. To enable scaling to a large number of cores, we employ a directory-based cache coherence protocol. The type of protocol depends on the L2 cache’s organization. For private L2 caches, we use a single MOSI protocol and a single L2 directory that tracks sharers across the private L2s. We maintain inclusion in the L2 caches, so we can piggy-back L1 coherence on top of L2 coherence to avoid extra bookkeeping for the L1 caches. As shown in Figure 6, the L2 directory is distributed across the MCs. We place each L2 directory entry at the home MC for its associated cache block.

For clustered L2 caches, we use a hierarchical protocol [Guo et al. 2010; Wallach 1993] to maintain coherence for the L1 and L2 caches separately. Coherence at the L2 level employs the same MOSI protocol described earlier for private L2s, except the L2 directory tracks sharers across cluster caches instead of private caches. To maintain coherence at the L1 level, we couple an MSI protocol and an L1 directory per cluster to the L2 protocol. Similar to the approach taken at the L2 level, each L1 directory is distributed within its cluster based on address mapping. In particular, physical addresses are interleaved across intracluster tiles at cache-block granularity such that consecutive cache blocks are homed on different tiles. We place each L1 directory entry at the home tile for its associated cache block.

As discussed in Section 2.3, the private cache is a degenerate case of the cluster cache, with $C_l = 1$. At the opposite end of the spectrum, the fully shared cache is another degenerate case, with $C_l = P$. In our tiled CPU, we aggregate all L2 cache slices in the CPU together to implement a single fully shared L2 cache. As with the private L2s, the hierarchical protocol collapses into a single protocol. But instead of using the L2 directory, the fully shared L2 uses a single L1 directory to maintain coherence across the L1 caches.

3.2. Performance

To model performance of the processor described in Section 3.1, we use a version of the CPU execution time equation for parallel programs:

$$CPUTime = \frac{IC}{P} \times [CPI_{execute} + (MemReadsPerInst \times AMAT)]. \quad (1)$$

In this equation, IC is the total instruction count, P is the number of cores, $CPI_{execute}$ is the average cycles per instruction (excluding memory stalls), $MemReadsPerInst$ is the average read references per instruction, and $AMAT$ is the average memory access time for reads. This equation assumes that instruction execution is load balanced across the cores—hence the IC/P term—which typically occurs in the homogeneous multithreaded parallel programs that we study. It also assumes that memory stalls are due to read latency, and that write latency is largely hidden by the cores' store buffers (see Section 3.1). As we will discuss in Section 4.1, IC, $MemReadsPerInst$, and $CPI_{execute}$ are directly measured by our tools (in particular, we run a single simulation to obtain $CPI_{execute}$). In this section, we focus on $AMAT$, which is computed using our locality profiles.

3.2.1. AMAT Calculation. To derive $AMAT$, we use our profiles to compute the miss rates at different caching levels within the multicore cache hierarchy. For the tiled CPU in Section 3.1, there are three levels of cache. First, we compute the miss rates for the private L1 and cluster L2 caches, $L1_{m.r.}$ and $L2_{m.r.}$:

$$L1_{m.r.} = \frac{CMC_{read}(sPRD, P \times L1_{size})}{N_{read}}; \quad L2_{m.r.} = \frac{CMC_{read}(sPRD^{C_l}, \frac{P}{C_l} \times L2_{size})}{N_{read}}. \tag{2}$$

Then, we compute the miss rate for *all* L2 cluster caches (after considering forwarding), which collectively form the last-level cache. We call this miss rate, $L2_{a.m.r}$:

$$L2_{a.m.r.} = \frac{CMC_{read}\left(sPRD_f^{C_l}, \frac{P}{C_l} \times L2_{size}\right)}{N_{read}}. \tag{3}$$

In these equations, the number of cache misses is derived by summing the CMCs from the corresponding profiles: sPRD for the private L1, sPRD^{C_l} for the cluster L2, and sPRD$_f^{C_l}$ for the cluster L2 with forwarding. Because Equation (1) considers stalls from read references, we sum memory references in the profiles associated with reads only, denoted as "CMC_{read}." (Our profiler keeps track of reads and writes separately for this purpose.) We also divide by the number of reads, N_{read}, when computing the miss rates. Additionally, because sPRD profiles aggregate the capacities of all profiled caches, we multiply by the number of caches when indexing the profiles—i.e., $P \times L1_{size}$ for the private L1s and $\frac{P}{C_l} \times L2_{size}$ for the cluster L2s.

The miss rates computed in Equations (2) and (3) only reflect cache misses due to limited capacity. In a real CPU, the private L1 and cluster L2 caches would also suffer conflict misses. We use Qasem and Kennedy's conflict model [Qasem and Kennedy 2005] to predict these additional misses. This model takes as input a locality profile and uses a binomial distribution to predict the number of conflict misses for a given capacity and associativity. We apply this model to the sPRD, sPRD^{C_l}, or sPRD$_f^{C_l}$ profiles to predict the number of conflict misses that occur in the private L1 cache, cluster L2 cache, and cluster L2 cache with forwarding, respectively. These predicted conflict misses are added to the CMC$_{read}$ terms in Equations (2) and (3) to derive the final miss rates. (We have omitted these extra conflict terms from Equations (2) and (3).)

Notice that the miss-rate equations do not depend on performance—i.e., there is no feedback from Equation (1) into Equations (2) and (3) (nor into the AMAT equations presented later). This is due to the homogeneous threads assumption made in our work. Although multicore cache-miss rates are generally sensitive to per-thread performance due to perturbations in how threads' memory references interleave, homogeneous threads tend to speed up or slow down by the same amount as miss rates change, thus largely preserving memory interleaving order. This is in contrast to cache-miss models for multiprogrammed workloads [Craeynest and Eeckhout 2011; Eklov et al. 2011] in which cache-miss rates depend upon performance, requiring the different sets of equations to be solved in a mutually consistent fashion.

Using these cache-miss rates, we prorate the latencies across the multicore cache hierarchy to compute the AMAT components for different memory transactions. In total, we model five different AMAT components for the tiled CPU in Section 3.1:

$$AMAT_{L1} = L1_{lat} \tag{4}$$

$$AMAT_{L2} = [(2 \times LocalComm_{lat}) + L2_{lat}] \times L1_{m.r.} \tag{5}$$

$$AMAT_{Dir} = [(2 \times GlobalComm_{lat}) + Dir_{lat}] \times L2_{m.r.} \tag{6}$$

$$AMAT_{remote} = [(2 \times GlobalComm_{lat}) + L2_{lat}] \times (L2_{m.r.} - L2_{a.m.r.}) \tag{7}$$

$$AMAT_{DRAM} = Mem_{lat} \times L2_{a.m.r.} \tag{8}$$

To illustrate, the dotted arrows in Figure 6, labeled “A” through “D,” show the different memory transactions assuming $C_l = 4$—i.e., there are four 2×2 clusters in the 16-tile CPU. First, a read reference, performed on tile 12 in our example, may hit in the L1 cache (arrow “A” in Figure 6). This memory transaction simply incurs the L1 access latency, $L1_{lat}$ (Equation (4)). Second, the same read reference may instead miss in the L1 cache, causing tile 12 to send a message across the on-chip network to the referenced cache block’s home tile within the local 2×2 cluster, tile 9 in our example (arrow “B”). There, the request may hit in the L2 cluster cache, causing the data to be sent back to tile 12. This memory transaction incurs the intracluster communication latency, $LocalComm_{lat}$ ($\times 2$ for the round trip), and the L2 access latency, $L2_{lat}$, prorated by the L1 miss rate (Equation (5)). Third, the memory reference may miss at the home tile, causing another message to be sent to the referenced cache block’s home MC, tile 6 in our example (arrow “C”), and an L2 directory lookup to be performed. This memory transaction incurs the intercluster communication latency, $GlobalComm_{lat}$, and the L2 directory lookup latency, Dir_{lat}, prorated by the L2 miss rate (Equation (6)). Note that both $LocalComm_{lat}$ and $GlobalComm_{lat}$ are constants and represent the average intra- and intercluster communication latencies, respectively. Section 4.1 will discuss the values for these model parameters.

After the L2 directory lookup, the memory transaction can finally complete in one of two ways. The requested cache block may be absent from all L2 cache slices, in which case main memory is accessed. This memory transaction incurs the DRAM access latency, Mem_{lat}, prorated by the LLC miss rate, $L2_{a.m.r.}$ (Equation (8)). (The data is then sent back to tile 9 from tile 6, hence the factor $2\times$ in intercluster communication latency in Equation (6).) Or, the requested cache block may reside in a remote L2 cluster cache on-chip (the forwarding case), causing a message to be sent to the home tile in the remote cluster, tile 15 in our example (arrow “D”), followed by the data being returned to the L2 directory. This memory transaction incurs another inter-cluster round-trip and L2 access latency (Equation (7)). Because the forwarding and DRAM cases are mutually exclusive, the forwarding memory transaction is prorated by ($L2_{m.r.} - L2_{a.m.r}$). This corresponds to the sPRD-sPRD$_f$ gap discussed in Section 2.4.

The final AMAT value in Equation (1) is computed by simply summing the AMAT components in Equations (4) through (8). Notice that since our locality profiles can be used to compute CMCs at any cache capacity (see Section 2), we can use Equation (1) to estimate performance for any tiled CPU cache hierarchy.

3.2.2. Degenerate Cases. As discussed earlier, the L2 cluster cache becomes a private cache or fully shared cache when $C_l = 1$ or $C_l = P$, respectively. The basic models from

Section 3.2.1 can be used to handle these cache hierarchies, but a couple of modifications are needed. First, CMC calculations should use the nonclustered versions of our locality profiles. In particular, for private L2 caches, $L2_{m.r.}$ and $L2_{a.m.r}$ should sum CMCs using sPRD and sPRD_f profiles (instead of sPRD^{C_l} and $\text{sPRD}_f^{C_l}$ profiles). For fully shared L2 caches, $L2_{m.r.}$ should sum CMCs using CRD profiles. ($L2_{a.m.r.}$ is not needed for shared caches, as described in the following.)

Second, certain memory transactions described in Section 3.2.1 cannot occur, simplifying AMAT calculation. For private L2 caches, the local L2 cache slice is accessed on every L1 miss, so there is no intracluster communication. As a result, Equation (5) simplifies to $AMAT_{L2} = L2_{lat} \times L1_{m.r.}$. (Equations (4) and (6) through (8) stay the same for private L2 caches.)

For fully shared L2 caches, each L1 cache miss can access any L2 cache slice, making intracluster communication global across the entire chip. In fact, there is no intercluster communication (there is only one cluster). Thus, there are no forwarding memory transactions, i.e., no coherence, and the three-level hierarchy collapses to two levels. To model the global nature of intracluster communication, we set $LocalComm_{lat} = GlobalComm_{lat}$ in Equation (5). To model the lack of forwarding and coherence, we completely remove Equation (7) ($AMAT_{remote} = 0$) and eliminate the directory access latency in Equation (6) ($Dir_{lat} = 0$). Last, because the cache hierarchy becomes two levels, there is no need for $L2_{a.m.r.}$, and main memory accesses in Equation (8) are simply prorated by $L2_{m.r.}$.

3.2.3. Model Imprecision. Due to the complexity of multicore processors, it is impossible to model every aspect of the CPU. Therefore, we made many simplifying assumptions to keep the models tractable. One example is that we were unable to account for every type of memory transaction in the hierarchical cache coherence protocol. A significant omission is forwarding transactions within clusters through the L1 directory. Like the L2 cache, a miss in the L1 cache may find a copy of the data in a remote L1 cache within the same cluster. Whether forwarding happens or not depends on the state of the cache block. If the cache block is in the *modified* state, then the remote L1 cache has the only up-to-date copy, and forwarding from the remote L1 cache is necessary (just like forwarding at the L2 level). However, if the cache block is in the *shared* state, then the copy in the L2 cluster cache is up-to-date and can be returned to the requesting L1 cache immediately, without going to the remote L1 cache. Because the shared state tends to be more common than the modified state, we assume that there is no forwarding at the L1 level. As a result, we underpredict stalls for read misses to modified cache blocks within the same cluster, i.e., intracluster migratory sharing.

Another omission is write stalls. As discussed in Section 3.2, our CPU execution time equation only considers read stalls. Although relaxed memory consistency models overlap a significant portion of write stalls, they do not overlap all of it. For example, write misses that occur just before a memory fence operation will likely stall the core performing the write. Hence, we underpredict CPU stalls when memory fences are frequent.

Finally, our models do not account for any contention in the memory hierarchy. In particular, we do not model contention in the on-chip network, nor do we model queueing at the MCs. Thus, we underpredict stalls when communication or memory bandwidth saturates.

3.3. Power Consumption

We model both the static and dynamic power consumption components of the processor described in Section 3.1. To do so, we extract the static power for all of the different hardware components within the CPU. For dynamic power, we handle the memory hierarchy and the cores separately. In the case of the memory hierarchy, we extract the per-access dynamic energy for the different hardware components and use our locality profiles to provide the activity counts to compute dynamic power. In the case of the cores, since our locality profiles do not provide core-specific activity counts, we rely on our tools to directly measure the dynamic power (see Section 4.1 for more details).

More specifically, we use the McPAT [Li et al. 2009] and DSENT [Sun et al. 2012] tools to provide the static power and per-access dynamic energy values. We use McPAT's model for ATOM-based cores, which is a good match to the simple in-order cores discussed in Section 3.1. We also use McPAT's models for the private L1 caches, the L2 cache slices, the L2 directories, and the MCs. Additionally, we use DSENT's models for the on-chip network. Last, we use published energy values for die-stacked DRAM to model power consumption in the memory subsystem. In all of these models, we assume the 32nm technology node. Although we account for most of the hardware modules in Figure 6, we do not model power consumption in the L1 directories. These are small hardware structures due to the limited number of cache blocks in the L1 caches, so they do not make up a large portion of the overall power consumption. As we will see in Section 4.2, our omission does not introduce significant error.

To provide the activity counts for computing dynamic power, we sum the cache misses that are incident at different caching levels using our locality profiles (in a manner similar to Section 3.2.1). However, instead of summing only the read references as was done for AMAT computation, we sum both reads and writes, denoted as *CMC*, to provide the total activity for computing dynamic power:

$$Activity_{L1} = CMC(sPRD, 0) \tag{9}$$

$$\begin{aligned} Activity_{L2} \;=\;& CMC(sPRD, P \times L1_{size}) \\ &+ \left[CMC\left(sPRD^{C_l}, \frac{P}{C_l} \times L2_{size}\right) - CMC\left(sPRD_f^{C_l}, \frac{P}{C_l} \times L2_{size}\right)\right] \end{aligned} \tag{10}$$

$$Activity_{Dir} = CMC\left(sPRD^{C_l}, \frac{P}{C_l} \times L2_{size}\right) \tag{11}$$

$$Activity_{DRAM} = CMC\left(sPRD_f^{C_l}, \frac{P}{C_l} \times L2_{size}\right) \tag{12}$$

$$Activity_{NOC} \approx Activity_{L2}. \tag{13}$$

Equation (9) provides the private L1 cache activity. This is simply the total number of memory references (or equivalently, the CMC for a cache with zero capacity, i.e., 100% miss rate). Next, Equation (10) provides the L2 cluster cache activity. This consists of two components: the downstream traffic from the private L1 caches and the upstream traffic from forwarding requests through the L2 directory. The former is simply the CMC for the sPRD profile at the L1 capacity. The latter is the CMC "gap" between the traffic from the L2 caches (or equivalently, sPRD^{C_l} at the L2 capacity) and the traffic to DRAM (or equivalently, $\text{sPRD}_f^{C_l}$ at the L2 capacity). (This is the same CMC gap mentioned when explaining Equation (7) in Section 3.2.1.) Finally, the two values used in this CMC gap computation correspond to the L2 directory activity (Equation (11)) and the DRAM activity (Equation (12)). The on-chip network activity (Equation (13)) is related to the L2 cache activity, and will be explained in Section 4.1.

4. CACHE HIERARCHY OPTIMIZATION STUDY

In this section, we use the RD-based performance and power models from Section 3 to optimize tiled CPUs. Specifically, we identify the cache hierarchy configurations that

Table I. Parallel Benchmarks, Their Problem Sizes, and Resulting Instruction Counts (Labeled “Inst”) Assuming 256 Cores

Benchmark	Problem Size	Inst	Benchmark	Problem Size	Inst
fft (kernel)	2^{22} elements	2.46	kmeans	2^{22} objects, 18 features	10.7
lu (kernel)	2048^2 elements	25.3	blackscholes	2^{22} options	2.44
radix (kernel)	2^{24} keys	3.85	canneal	2500000.net	1.04
barnes	2^{19} particles	16.3	fluidanimate	in_500k.fluid	3.80
water	40^3 molecules	2.31			

Note: All instruction counts are reported in billions.

allow a CPU to achieve the highest power efficiency possible for a given parallel program. First, we discuss our experimental methodology (Section 4.1). Then, we present our model-based results (Sections 4.2 and 4.3). Finally, we validate the model results against detailed architectural simulations (Section 4.4).

4.1. Experimental Methodology

We built a profiler using the Intel PIN tool [Luk et al. 2005] that implements the multicore RD analyses described in Section 2. Our PIN-based profiler can acquire CRD, PRD, PRD$_f$, and PRD^{C_l} profiles via the parallel stack profiling techniques presented in Sections 2.1 through 2.3. Using this tool, we profiled a subset of the SPLASH-2 [Woo et al. 1995] and PARSEC [Bienia et al. 2008] benchmarks chosen as follows. Among the 24 benchmarks that comprise both benchmark suites, we eliminated 9 benchmarks because they did not exhibit homogeneous threads or because we could not run them on our tools.[1] To keep the number of profile runs (as well as detailed simulation runs, which we will describe later) at a manageable level, we further selected 8 out of the remaining 15 benchmarks to include a mix of longer- and shorter-running benchmarks. Specifically, we profiled fft, lu, radix, barnes, and water from the SPLASH-2 suite, and blackscholes, canneal, and fluidanimate from the PARSEC suite. We also ran our profiler on kmeans from the MineBench suite [Narayanan et al. 2006], which is another benchmark that we have used in previous research. Table I lists our benchmarks.

For the benchmarks that are kernels (fft, lu, and radix), we profiled the entire benchmark. For the remaining benchmarks, we profiled a single iteration of the benchmark's parallel region. The second and fifth columns in Table I report the problem sizes used, and the third and last columns in Table I report the number of instructions profiled assuming 256 cores. In addition to 256 cores, we ran profiles for 128 and 64 cores as well. (See the following for a detailed discussion of the machine configurations that we study)

Next, for each benchmark, we used the acquired profiles at each core count to identify the CPU that achieves the highest power efficiency (performance per watt). This was done by using the performance and power models from Section 3 (along with our profiles) to evaluate the power efficiency for many different tiled CPUs and then choosing the best one. The different tiled CPUs evaluated were defined by varying the capacity of the private L1 D-caches, the capacity of the cluster L2 caches, and the cluster size. In particular, at each core count, P, we allow the private L1 D-caches to take on 4 different capacities. In addition, we vary the L2 cluster caches, allowing the cluster size, C_l, to take on values between 1 and P in powers of 2, and the size of each L2 slice within a cluster to take on 19 different capacities. Table II specifies the different private L1 D-cache sizes and cluster L2 slice sizes allowed in our study.

The different L1 and L2 sizes that we evaluate are implemented by starting from a baseline cache design and then varying the number of cache ways and/or sets. Specifically, the private L1 D-cache is based on a 32KB four-way cache and implements

[1]Some of the benchmarks could not be ported to our architectural simulator, described later in this section.

Table II. Tiled CPU Configuration Parameters Used in the Experiments

Number of Cores/Tiles	64, 128, 256
Core Type	Single issue, in-order, CPI = 1, clock speed = 2GHz
Private L1 I-Cache	32KB
	64B blocks, 4-way, 1 CPU cycle
Private L1 D-Cache	8KB, 16KB, 24KB, or 32KB
	64B blocks, 4-way, 1 CPU cycle ($L1_{lat}$)
Cluster L2 Slice	32KB, 48KB, 64KB, 80KB, 96KB, 112KB, 128KB, 160KB, 192KB, 224KB
	256KB, 320KB, 384KB, 448KB, 512KB, 640KB, 768KB, 896KB, or 1MB
	64B blocks, 8-way, 8 CPU cycles ($L2_{lat}$)
L2 Directory	Full-map directory entries, 4 CPU cycles (Dir_{lat})
2D Mesh	2 cycles per hop, 64-bit-wide links
Memory Channels	Latency: 100 CPU cycles (Mem_{lat}), bandwidth: 5 GB/s per MC

the 8KB, 16KB, 24KB, and 32KB capacities from Table II by using one to four ways of the baseline cache. In contrast, the L2 cache slices are based on 1MB eight-way caches. The four largest capacities in Table II– - 640KB, 768KB, 896KB, and 1MB—are implemented by using five to eight ways of the baseline cache. Smaller L2 slices are implemented by successively reducing the number of sets from the baseline. For example, the 320KB, 384KB, 448KB, and 512KB L2 slices use half of the sets from the baseline cache and five to eight ways; the 160KB, 192KB, 224KB, and 256KB L2 slices use a quarter of the sets from the baseline cache and five to eight ways; and the seven smallest L2 slices—32KB, 48KB, 64KB, 80KB, 96KB, 112KB, and 128KB—use an eighth of the sets from the baseline cache and two to eight ways. When evaluating different cache hierarchies, we permit all combinations of L1 and L2 sizes as long as their ratio does not exceed 1:4, e.g., a 16KB L1 and a 32KB L2 is not permitted. Because we maintain inclusion, the L2 effectiveness becomes limited when its size approaches the L1 size.

When evaluating power efficiency for all of these configurations, the main inputs to Equations (1) through (13) are the locality profiles mentioned earlier, but other inputs are also required. These include each benchmark's instruction count (IC) and total memory read references (N_{read}), as well as memory reads per instruction ($MemReadsPerInst$). We acquire these values in our profiler along with the locality profiles. In addition, our models require the cache hierarchy latencies, i.e., $L1_{lat}$, $L2_{lat}$, Dir_{lat}, and Mem_{lat}. Table II specifies the values that we used for these latencies. In addition, our models require the average latencies to traverse the on-chip network within a cluster ($LocalComm_{lat}$) and across clusters ($GlobalComm_{lat}$). These latencies depend on both the per-hop latency, which we assume to be two cycles, as well as the average number of hops. We assume that the destination for intra- and intercluster messages is uniformly distributed across the tiles within and between clusters, respectively. Thus, for intracluster communication, we assume $\sqrt{C_l}$ hops on average, whereas for intercluster communication, we assume $\sqrt{P}$ hops on average. Last, our models require each benchmark's $CPI_{execute}$ and dynamic power consumption in the cores. These values are directly measured, which we discuss next.

Besides our PIN-based profiler and the performance and power models that it drives, our study also employs detailed architectural simulation. We use the Graphite multicore simulator [Miller et al. 2010] for all of our simulation experiments.[2] In particular, we configured Graphite to model the tiled CPU discussed in Section 3.1. The main change needed for this was the addition of a cluster L2 cache, as well as the

[2]This simulator has been continuously updated since its introduction. Because of collaborations with the original developers, we were able to use the very latest versions of the simulator before being made available via public releases.

Table III. Configurations with Highest Power Efficiency Predicted by our Profile-Based Models

	256 Cores			128 Cores			64 Cores		
Benchmark	L1	L2	Cluster	L1	L2	Cluster	L1	L2	Cluster
fft	8KB	160KB	1	16KB	192KB	1	16KB	192KB	1
lu	8KB	32KB	1	8KB	32KB	1	16KB	64KB	4
radix	24KB	112KB	2	24KB	128KB	1	16KB	128KB	1
barnes	16KB	64KB	2	16KB	96KB	2	16KB	128KB	4
water	16KB	64KB	16	16KB	64KB	8	16KB	64KB	4
kmeans	8KB	48KB	32	16KB	96KB	4	16KB	64KB	4
blackscholes	16KB	64KB	2	16KB	64KB	2	16KB	64KB	2
canneal	24KB	896KB	256	24KB	512KB	128	32KB	128KB	64
fluidanimate	8KB	32KB	1	16KB	64KB	1	16KB	64KB	1

hierarchical cache coherence protocol described in Section 3.1.1 to enable clustering. Our Graphite simulator also uses McPAT to compute power in the cores, caches, directories, and MCs, and DSENT to compute power in the on-chip network. These are identical to the McPAT and DSENT models discussed in Section 3.3 for our locality profiles. But instead of using activity counts from our profiles, Graphite uses detailed simulation events to drive its power models.

In our study, we use Graphite for two different purposes. First, Graphite provides the baseline cycles per instruction excluding memory stalls, $CPI_{execute}$, and the dynamic power consumption in the cores needed by our performance and power models from Section 3. To obtain these values, we perform a single simulation per benchmark with an “ideal memory system,” i.e., all memory operations complete in a single cycle. Once acquired, these values are used to estimate the power efficiency of all tiled CPUs defined by the parameters in Table II. Second, we also use Graphite to simulate many of the configurations from Table II evaluated by our profile-driven models to validate the models’ accuracy. In these simulations, we turn on all performance and power models within Graphite, including those for the memory hierarchy, to measure the actual power efficiency achieved.

4.2. Power-Efficient Configurations

Table III reports the configurations that exhibit the highest power efficiency as predicted by our locality profiles and performance/power models from Section 3. In particular, for each of the core counts that we consider—256, 128, and 64—Table III reports the private L1 size, the L2 slice size, and the cluster size for each benchmark that achieves the highest throughput per unit power (GIPs/W). These configurations are the result of evaluating a total of 16,896 configurations using our models. (There are 576 unique configurations for the 256-core CPU, 512 configurations for the 128-core CPU, and 448 configurations for the 64-core CPU, yielding 1,536 configurations per benchmark: 1,536 $\times$ 9 benchmarks = 13,824 configurations in total.)

This data shows that the best configurations vary considerably, especially across different benchmarks. As we can see in Table III, there is no configuration that is best in more than a few cases. (The most common configuration in Table III is a 16KB L1, a 64KB L2, and a cluster size of 2, which occurs three times out of 27 cases.) In particular, most of the private L1 sizes and cluster sizes from Table II appear at least once in Table III. (Many of the L2 slice sizes are, however, on the lower end of the spectrum—i.e., 64KB. We will discuss this later on.) Using our techniques, we can identify these highly power efficient configurations without having to simulate all of them.

Not only can our techniques help to identify power-efficient configurations, but they can also explain why certain configurations are optimal. As discussed in Section 2.4,

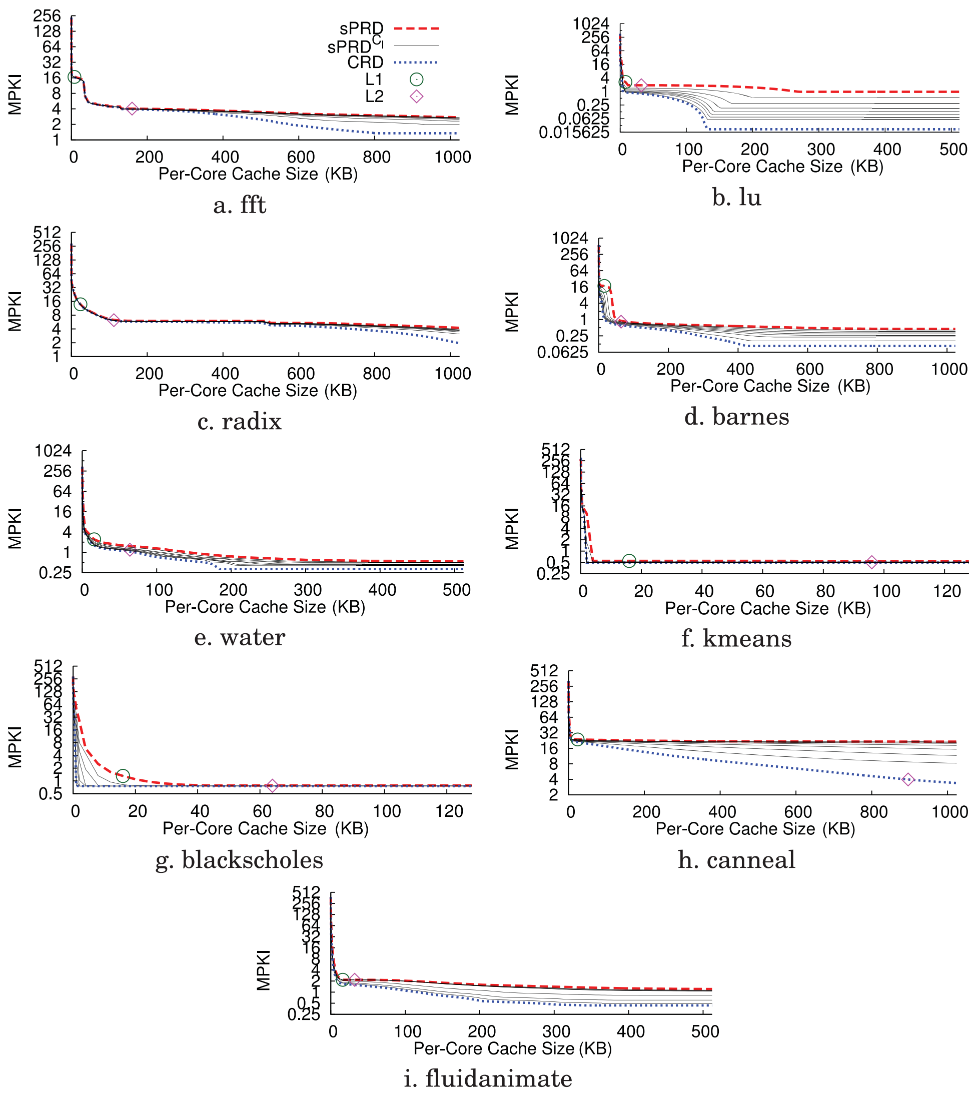


Fig. 7. sPRD, sPRD^{C_l}, and CRD profiles for our benchmarks running on 256 cores. The most power efficient configurations listed in Table III are indicated on the profiles.

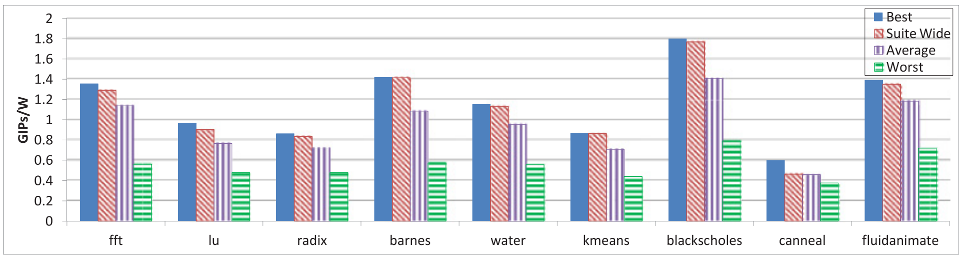
our locality profiles can provide such insights. To illustrate, Figure 7 shows the locality profiles for all of our benchmarks running on 256 cores. Similar to Figures 4 and 5, each graph in Figure 7 plots the CMC profiles associated with a single benchmark. But instead of plotting raw CMC values along the y-axis as is done in Figures 4 and 5, Figure 7 normalizes CMC values by each benchmark's instruction count (IC) and multiplies by 1,000, so the y-axes plot cache misses per 1,000 instructions (MPKI), which is a better indicator of performance than raw CMC. Additionally, similar to Figure 5, Figure 7 plots the sPRD^{C_l} profiles, including the full range $C_l = 2$ through 128, thus showing the impact of cluster size variation. (Each benchmark's sPRD and CRD profiles are also plotted in Figure 7, just like in Figures 4 and 5.) Last, the circle and diamond symbols in Figure 7 indicate the private L1 cache and cluster L2 slice selections, respectively, that correspond to the most power efficient configurations from Table II for 256 cores. In particular, the x-axis value for each circle/diamond indicates the capacity of each cache/slice, and the profile upon which each diamond is plotted indicates the L2's cluster size.

The graphs in Figure 7 reveal several important insights. First, the best L1 and L2 configurations tend to occur after large drops in the locality profiles. As discussed in Section 2.4, parallel locality profiles exhibit working sets, just like uniprocessor locality profiles, that are marked by large drops at each working set's size. Tuning a cache's capacity to capture different working sets can yield large reductions in cache misses, simultaneously boosting performance and reducing power consumption. However, increasing cache size along a "plateau" of the locality profiles is not profitable, as doing so only increases power consumption without providing a performance gain. Thus, sizing caches just beyond a drop but not much further can provide high power efficiency. (In some cases, e.g., blackscholes in Figure 7(g), it appears that the cluster L2 cache is sized down a plateau, but this is done to meet the 1:4 ratio requirement between the L1 and L2 capacities, as discussed in Section 4.1. The L2 for kmeans in Figure 7(f) also seems to be sized along a plateau, but there is actually a small benefit that is hard to see in the graph, which justifies the larger cache size.)

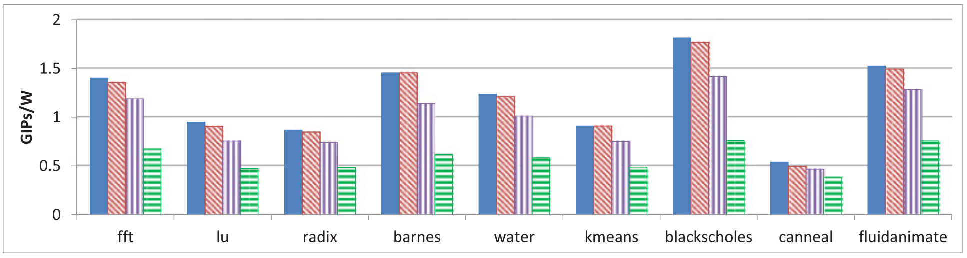
Second, some drops in the locality profiles, especially at large capacities, are not captured by any on-chip cache. This is due to two reasons. One reason is that MPKI may already be small, so further drops do not provide sufficient gains to justify the increase in cache size. (Keep in mind that the y-axes in Figure 7 use a log scale, so some of the drops occur across very small MPKI values.) Generally, we find that once a cache achieves 1.0 MPKI or less, the benefit of further capacity scaling is minimal. The water benchmark in Figure 7(e) is an example of this. Another reason is that further drops may occur at too large a capacity, so the power consumption of the cache needed to capture the drop is not justified by the improvement the cache provides. The radix benchmark in Figure 7(c) is an example of this.

And third, large cluster sizes in the L2 only occur when there is a gap between the sPRD and CRD profiles—i.e., when sPRD^{C_l} and CRD profiles can noticeably reduce cache misses relative to the sPRD profile. As discussed in Section 2.4, this sPRD-CRD gap is due to sharing. Figure 7 shows that some benchmarks exhibit very little sharing, e.g., radix and kmeans in Figures 7(c) and (f). For these benchmarks, private caches—or cluster caches with small C_l—are the best choices. Even if sharing occurs, it may not manifest itself at the "right" capacities. As discussed in Section 2.4, sharing is capacity dependent and tends to occur in larger caches, i.e., beyond a program's C_{share}. If C_{share} occurs at too large a capacity, e.g., fft in Figure 7(a), or in a region of a program's locality profiles where MPKI is already small, e.g., barnes in Figure 7(d), then private caches or cluster caches with small C_l may still be the best choices. Only when sharing provides a noticeable benefit at reasonable cache sizes are cluster or shared caches best. Canneal in Figure 7(h) is an example of this.

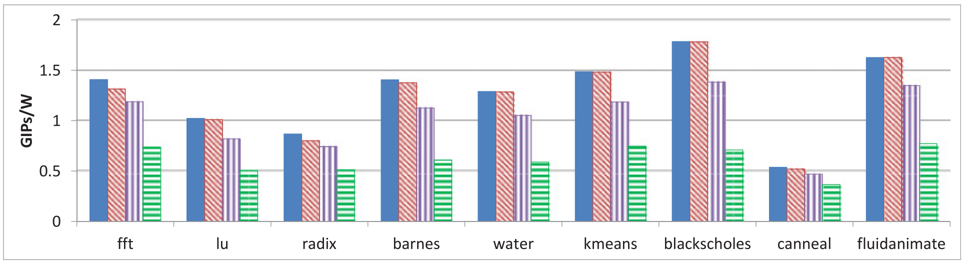
Finally, although Figure 7 only plots the profiles for 256 cores, our locality profiles can also explain the behavior observed across core count. As Table III shows, core count scaling changes the best configuration for each benchmark only slightly. Usually, private L1 cache and cluster L2 slice sizes stay the same, or decrease a bit, as core count increases. This makes sense given what is known about locality profiles under scaling [Wu and Yeung 2013; Wu et al. 2013]. As discussed in Section 2.4, core count scaling causes locality profiles to shift to larger cache capacities. But this shift is nonuniform. It is larger (usually linear) at smaller cache sizes and slows down at larger cache sizes. Since the capacities reported in Table III are *per core*, no change as core count increases means that the total cache capacity is increasing linearly with core count. In these cases, the locality profiles are shifting linearly, causing the total



a. 256 Cores



b. 128 Cores



c. 64 Cores

Fig. 8. Power efficiency (in GIPs/W) achieved by the best, average, worst, and best suite-wide configurations for each benchmark running on 256 cores (a), 128 cores (b), and 64 cores (c).

cache capacity to increase linearly to maintain the same MPKI. On the other hand, there are also cases in Table III in which per-core cache size decreases with increasing core count. In these cases, the locality profiles are shifting sublinearly, so total cache capacity can increase slowly and still maintain the same MPKI.

4.3. Optimization Benefits

Although identifying the most power efficient configurations is our main goal, it is also instructive to use our techniques to study how much benefit memory hierarchy optimization can provide. Because our locality profiles and performance/power models from Section 3 can evaluate *all* memory hierarchy configurations, we can determine the range of power efficiencies exhibited across the CPU's entire design space. Hence, we can quantify the degree to which the best configuration can improve upon all of the other configurations.

Figure 8 presents our optimization benefit results. In particular, the bars labeled "Best" in Figure 8 report the power efficiency in GIPs/W achieved by the most power-efficient configuration in the design space for each benchmark and core count. These correspond to the configurations listed in Table III. In contrast, the bars labeled "Worst" in Figure 8 report the configurations that achieve the lowest power efficiency in the design space. The bars labeled "Average" report the power efficiency achieved on average.

These bars give a sense for the range and distribution of power efficiencies that occur when varying the memory hierarchy configuration. Results are presented separately for 256, 128, and 64 cores in Figures 8(a), (b), and (c), respectively.

Comparing the “Best” and “Worst” bars in Figure 8, we see that there is a large range of power efficiencies across the design space. For all of the benchmarks and core counts, the ratio of the best and worst configurations’ power efficiencies varies between 1.4x (for canneal at 128 cores) to 2.5x (for blackscholes at 64 cores). On average, this best-to-worst ratio for 256, 128, and 64 cores is 2.04x, 1.98x, and 2.0x, respectively. Averaged across all benchmarks and core counts, the best-to-worst ratio is 2.01x. These results show that changing the memory hierarchy configuration can significantly affect a multicore CPU’s power efficiency.

Although the range of power efficiencies across the design space is very large, the “Average” bars in Figure 8 show that it is nonuniform. In particular, the power efficiencies achieved on average are visibly closer to the best configurations than the worst configurations, suggesting that really bad configurations make up a disproportionately smaller fraction of the overall design space. Comparing the “Best” and “Average” bars, we see the ratio varies between 1.15x (for canneal at 64 cores) to 1.3x (for barnes at 256 cores). For 256, 128, and 64 cores, the best configuration improves upon the average power efficiency by 23.7%, 21.2%, and 21.3%, respectively. Across all benchmarks and core counts, the improvement is 22.1%. These results show identifying the best configuration can still provide significant gains over an arbitrary or randomly chosen configuration, although not as big as the “Best” and “Worst” bars in Figure 8 would suggest.

Although the “Best” bars show the benefits from optimizing the memory hierarchy for individual benchmarks, our techniques can also be used to optimize for entire benchmark suites. In addition to the best configuration per benchmark, we also identified the configuration that achieves the highest power efficiency on average across all of our benchmarks—i.e., the best suite-wide configuration. (This analysis was performed separately for 256, 128, and 64 cores, thus identifying three different best suite-wide configurations.) In Figures 8(a), (b), and (c), the bars labeled “Suite Wide” report the power efficiency achieved per benchmark by this best suite-wide configuration for each core count.

Interestingly, Figure 8 shows that the best suite-wide configuration comes very close to the absolute best configuration for each benchmark. In almost all benchmarks, the best suite-wide configuration is within 5% or 6% of the absolute best configuration. For canneal at 256 cores, optimizing specifically for the benchmark provides a more noticeable gain over the best suite-wide configuration by 28.7%. However, overall, the best suite-wide configuration does pretty well for our benchmarks. This is because many of the best configurations in Table II employ smaller cluster L2 caches. In general, the input problem sizes that we used are on the smaller side, especially for CPUs with 64 to 256 cores. (This was done to enable the detailed simulations that we ran for our validation study in Section 4.4.) Hence, many of the working sets visible in Figure 7 occur at smaller cache capacities. Were we to scale the input problems, there would likely be greater variability in the best choice of cluster L2 cache, which would result in more differentiation between the bars for “Best” and “Suite Wide” in Figure 8. Nevertheless, these results demonstrate that our techniques can identify the best configuration for entire benchmark suites.

4.4. Validation

To validate the accuracy of our techniques, we compared the power efficiency predictions made from our locality profiles and analytical models against the power efficiency simulated on the Graphite simulator described in Section 4.1. Our study considers two aspects of accuracy. The first is how often our techniques pick the most power efficient configurations—i.e., the relative accuracy across our predictions. The second is how close our predicted power efficiencies are to actual simulation—i.e., the absolute accuracy of our predictions.

Due to the large design spaces associated with our tiled CPU architecture, our validation only considers a subset of all of the configurations studied in Sections 4.2 and 4.3. In particular, we only look at CPUs with 256 cores. Additionally, in our initial experiments, we validate 10 configurations per benchmark sampled from the 576 unique cache hierarchy configurations in our 256-core CPU. (Later, we will consider additional validation points on top of these 10.)

The 10 initial sampled configurations were chosen in the following manner. We sorted all 576 configurations from highest predicted power efficiency to lowest predicted power efficiency. (As discussed in Section 4.2, we already predict the entire design space to identify the best configuration, so we have power efficiencies for every configuration.) Then, we divided the configurations into 10 groups by grouping sets of 57 (or 58) configurations that appear contiguously in the sorted list. In the top group (group 1), we sampled the very top configuration. These are the best configurations predicted by our models already reported in Table III. In the remaining 9 groups (groups 2 through 10), we sampled a configuration chosen at random from each group. Table IV reports the L1 size, L2 size, and cluster size for the 9 configurations randomly sampled from groups 2 through 9 for each of our benchmarks. (The columns labeled "Group" indicate from which group each sample comes.) This method tends to sample configurations that are differentiated with respect to each other in terms of power efficiency. Thus, the samples test whether our techniques can distinguish configurations from dissimilar points in the design space. In total, there are 90 sampled configurations across Tables III and IV. All of these configurations were simulated on Graphite.

Figure 10 presents our initial validation results. In the figure, the bars labeled "Model" report the power efficiencies predicted by our techniques, whereas the bars labeled "Graphite" report the power efficiencies measured on the Graphite simulator. Predictions and simulations are presented for all 90 sampled configurations from Tables III and IV.

These results show that our locality profiles and analytical models are effective at identifying power-efficient cache hierarchy configurations. In four of the benchmarks—fft, radix, barnes, and water—the simulated power efficiency for the configuration from group 1 does indeed achieve the highest power efficiency among all simulated configurations. In another four benchmarks—lu, kmeans, blackscholes, and canneal—the configuration with the highest simulated power efficiency is in either groups 2 or 3 instead of group 1, but the configuration from group 1 is not very far behind. For lu, kmeans, and canneal, the group 1 configuration is within 5% of the best simulated configuration (either in group 2 or 3). And in blackscholes, the configuration from group 1 is a bit worse, within 10.7% of the group 2 (or 3) configuration. Our techniques exhibit the greatest error for fluidanimate. Once again, the configuration with the highest simulated power efficiency is in group 2, but the group 1 configuration is 19.6% worse than the group 2 configuration.

In addition to how well our techniques identify power-efficient configurations, Figure 10 also shows how closely our techniques can predict the simulated power efficiency. The absolute prediction error is quite small in six benchmarks (fft, radix, water, kmeans, blackscholes, and canneal) but more noticeable in three benchmarks (lu, barnes, and fluidanimate). Across all 90 sampled configurations, the average prediction error is 15.2%. Figure 9 illustrates the absolute error for all 90 sampled configurations sorted from smallest to largest, thus showing the error distribution. In Figure 9, we can see that 90% of the samples exhibit 32% or less error. The remaining 10% of the

Table IV. Nine Additional Configurations Per Benchmark for 256 Cores (on Top of Table III) Sampled Across the Cache Hierarchy Design Space That Represents Disparate Points

Group	L1	L2	Cluster	Group	L1	L2	Cluster	Group	L1	L2	Cluster
					fft						
2.	16KB	224KB	8	3.	24KB	192KB	8	4.	24KB	160KB	16
5.	24KB	640KB	2	6.	16KB	112KB	128	7.	32KB	320KB	128
8.	24KB	896KB	64	9.	24KB	1,024KB	256	10.	8KB	32KB	128
					lu						
2.	16KB	128KB	16	3.	8KB	160KB	8	4.	32KB	192KB	2
5.	24KB	256KB	4	6.	24KB	384KB	4	7.	24KB	448KB	64
8.	8KB	640KB	32	9.	8KB	896KB	32	10.	32KB	1,024KB	1
					radix						
2.	8KB	80KB	2	3.	16KB	256KB	2	4.	16KB	224KB	128
5.	16KB	384KB	1	6.	32KB	448KB	32	7.	8KB	48KB	256
8.	16KB	640KB	1	9.	8KB	896KB	64	10.	8KB	1,024KB	256
					barnes						
2.	32KB	128KB	8	3.	8KB	256KB	2	4.	16KB	320KB	1
5.	8KB	256KB	16	6.	32KB	384KB	32	7.	24KB	512KB	32
8.	32KB	1,024KB	2	9.	8KB	768KB	64	10.	24KB	1,024KB	256
					water						
2.	16KB	160KB	8	3.	8KB	192KB	4	4.	24KB	256KB	16
5.	24KB	320KB	1	6.	16KB	384KB	16	7.	32KB	448KB	32
8.	8KB	512KB	64	9.	8KB	1,024KB	128	10.	8KB	1,024KB	256
					kmeans						
2.	16KB	96KB	4	3.	8KB	112KB	2	4.	16KB	160KB	4
5.	8KB	256KB	1	6.	8KB	224KB	128	7.	32KB	640KB	2
8.	8KB	768KB	8	9.	8KB	896KB	256	10.	32KB	1,024KB	128
					blackscholes						
2.	24KB	128KB	32	3.	32KB	160KB	8	4.	24KB	224KB	1
5.	8KB	80KB	64	6.	8KB	320KB	2	7.	32KB	512KB	16
8.	24KB	640KB	128	9.	8KB	896KB	1	10.	32KB	1,024KB	128
					canneal						
2.	24KB	768KB	256	3.	32KB	224KB	256	4.	8KB	1,024KB	256
5.	32KB	192KB	8	6.	24KB	96KB	8	7.	32KB	160KB	128
8.	24KB	640KB	1	9.	8KB	768KB	2	10.	8KB	1,024KB	16
					fluidanimate						
2.	24KB	112KB	8	3.	32KB	160KB	4	4.	24KB	320KB	128
5.	16KB	384KB	1	6.	24KB	448KB	256	7.	8KB	384KB	8
8.	32KB	768KB	1	9.	32KB	1,024KB	1	10.	8KB	1,024KB	256

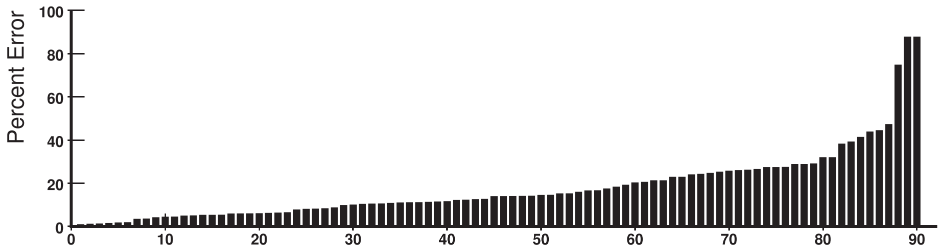


Fig. 9. Error distribution results. Percent error for 90 sampled configurations sorted from smallest to largest error.

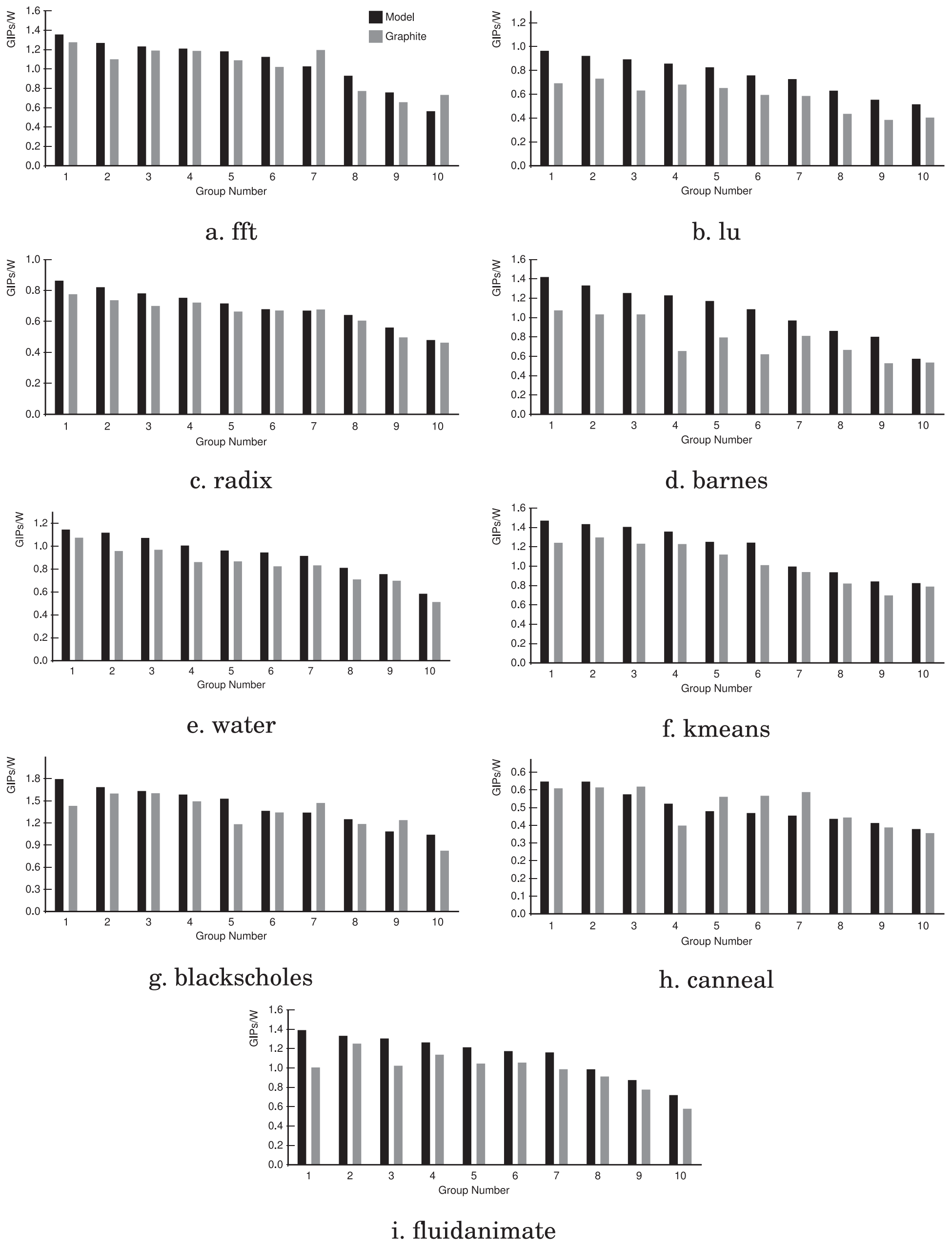


Fig. 10. Power efficiencies predicted by our techniques (bars labeled "Model") and measured on Graphite (bars labeled "Graphite") for the sampled configurations in groups 1 through 10 from Tables III and IV.

samples have more elevated error, with the three worst samples exhibiting between 74% and 88% error.

For the most part, our techniques *overpredict* the actual power efficiency. Almost every bar labeled "Model" in Figure 10 is higher than its corresponding bar labeled "Graphite." As discussed in Section 3.2.3, there are several events that our models

Table V. Three Additional Configurations Per Benchmark for 256 Cores Sampled Across the Configurations in Group 5 from Table IV That Represent Similar Design Points

Group	L1	L2	Cluster	Group	L1	L2	Cluster	Group	L1	L2	Cluster
					fft						
5a.	32KB	768KB	1	5b.	24KB	320KB	8	5c.	32KB	192KB	64
					lu						
5a.	24KB	224KB	1	5b.	32KB	192KB	1	5c.	32KB	160KB	1
					radix						
5a.	32KB	320KB	128	5b.	8KB	192KB	128	5c.	8KB	256KB	32
					barnes						
5a.	24KB	320KB	4	5b.	16KB	64KB	128	5c.	8KB	128KB	64
					water						
5a.	8KB	256KB	8	5b.	8KB	192KB	64	5c.	8KB	224KB	64
					kmeans						
5a.	8KB	224KB	64	5b.	8KB	192KB	128	5c.	16KB	256KB	32
					blackscholes						
5a.	8KB	128KB	32	5b.	8KB	160KB	16	5c.	8KB	224KB	4
					canneal						
5a.	16KB	160KB	1	5b.	24KB	320KB	4	5c.	32KB	224KB	16
					fluidanimate						
5a.	8KB	256KB	32	5b.	8KB	320KB	4	5c.	8KB	128KB	128

do not take into consideration. These omissions tend to underpredict the execution time and power consumption, which inflate the predicted power efficiency. Another source of error is cache conflicts prediction. As discussed in Section 3.2, we use Qasem and Kennedy's conflict model for this purpose. Although this model works well on average, it underpredicts conflicts when they become more severe, hence leading to further underprediction of execution time and power consumption. Underprediction of conflicts is a significant source of error for the barnes benchmark in Figure 10(d). Last, as mentioned in Section 3.2, our performance model (Equation (1)) assumes load-balanced threads. We find that the lu benchmark exhibits some load imbalance, which is responsible for much of its error in Figure 10(b). But as Figure 10 shows overall, these sources of error do not significantly degrade our relative prediction accuracy, so we can still rank order the configurations fairly effectively.

In addition to validating whether our techniques can identify the best configurations across the design space, we also validated whether our techniques can identify groups of configurations that are similar. To facilitate these experiments, we went back to group 5 in each benchmark and randomly sampled another three configurations. Table V reports the L1 size, L2 size, and cluster size for the three additional configurations randomly sampled from group 5 for each of our benchmarks. These three samples (labeled "5a," "5b," and "5c" in Table V), along with the original sample from group 5 in Table IV, should be similar to each other in terms of power efficiency. In total, there are 27 group-5 samples. All of these configurations were simulated on Graphite.

Figure 11 presents our additional validation results. Like Figure 10, the bars in Figure 11 labeled "Model" report the power efficiencies predicted by our techniques, and the bars labeled "Graphite" report the power efficiencies measured on the Graphite simulator. Predictions and simulations are presented for all 27 sampled configurations from Table V in pairs of bars labeled "5a" through "5c." The original group-5 predictions and simulations from Figure 10 are also presented in pairs of bars labeled "5."

These results show that our locality profiles and analytical models are effective at identifying similar configurations across the design space. For four benchmarks (lu, radix, canneal, and fluidanimate), the Graphite simulations for the four sampled

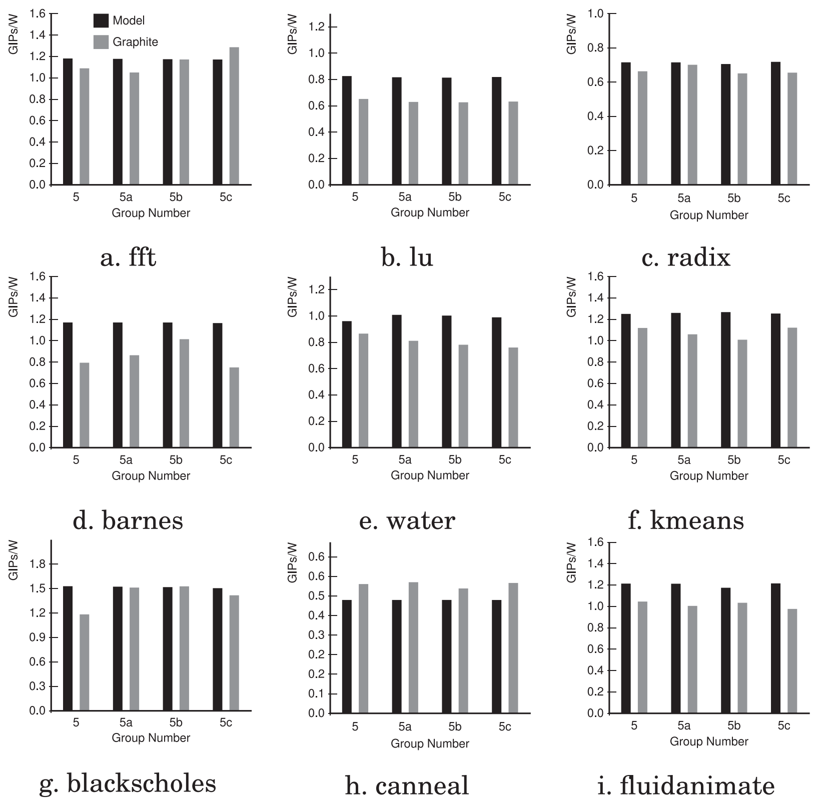


Fig. 11. Power efficiencies predicted by our techniques (bars labeled "Model") and measured on Graphite (bars labeled "Graphite") for the sampled configurations in group 5 from Table IV and in groups 5a through 5c from Table V.

configurations exhibit power efficiencies that are very close to one another—within 7.2%. For two additional benchmarks (water and kmeans), the sampled configurations are a bit farther apart, but still reasonable—within 12.2% of one another. For FFT, the spread is 18.4%, and for blackscholes and barnes, the spread is 22.5% and 26.0%, respectively.

In terms of absolute prediction error, the results in Figure 11 are consistent with those in Figure 10. Averaged across all 27 sampled configurations, the error between the Model and Graphite bars is 14.5%. In terms of error distribution, the results in Figure 11 are a bit better than those in Figure 10, with 25 out of 27 of the samples exhibiting less than 30% error and the worst-case sample exhibiting 35.6% error. As in Figure 10, the results in Figure 11 show that our techniques usually overpredict the actual power efficiency for the same reasons given in the discussion for Figure 10.

5. RELATED WORK

Many previous studies have tried to characterize multicore performance and/or power efficiency [Davis et al. 2005; Hsu et al. 2005; Huh et al. 2001; Li et al. 2006, 2009; Li and Martinez 2005; Rogers et al. 2009; Zhao et al. 2007]. These studies often vary core count and cache capacity to quantify how different designs behave. But all of them employ detailed architectural simulation to evaluate different design configurations. In our work, we propose using RD techniques instead of architectural simulators, permitting more efficient—as well as exhaustive—exploration of cache hierarchy design spaces.

To enable our approach to design space exploration, researchers have recently developed RD analysis techniques for multicore cache hierarchies. In particular, Ding and Chilimbi [2009] proposed CRD profiling for analyzing shared caches, and Schuff et al. [2009, 2010] proposed PRD profiling for analyzing private caches. In addition, our own prior work proposed PRD_f profiling [Wu and Yeung 2012] to analyze forwarding requests in private caches and PRD^{C_l} profiling [Wu et al. 2013] to analyze cluster caches. The work described in this article is heavily based on these techniques. However, instead of development and assessment of the techniques (which is the primary focus of these prior studies), the research reported in this article employs the RD techniques to actually optimize multicore cache hierarchies.

That said, this is not the first research to apply multicore RD analysis. Our own earlier paper [Wu and Yeung 2012] was the first to use locality profiles for design space exploration and optimization purposes. But this was a very preliminary study that only optimized for cache-miss behavior. Another one of our previous papers [Wu et al. 2013] used locality profiles to determine the best rate at which to scale cache capacity as core count and problem size scale. But again, this work only considered cache-miss behavior. There has been no prior work on using multicore RD analysis to identify power-efficient cache hierarchies.

Researchers have also tried to perform architectural optimization using machine learning and control theory [İpek et al. 2006; Lee and Brooks 2006; Maggio et al. 2011]. These techniques try to model how system features impact performance, power, or both. They learn very little per sample (a single IPC value or power number) but are very general and can optimize any architectural feature. In comparison, our approach learns more per sample (an entire locality profile), reducing the number of needed samples. However, our approach can only optimize memory-centric features—i.e., cache capacity and cluster size.

Finally, some researchers have employed simpler notions of RD to enable more efficient RD analysis. In our work, we compute RD based on LRU stacks, which achieves good accuracy, yet this approach comes at the cost of having to maintain the global LRU ordering of memory blocks. Alternatively, one can compute RD by simply counting the total number of memory references performed in between two references to the same memory block [Berg and Hagersten 2005, 2004; Eklov et al. 2011; Eklov and Hagersten 2010]. This approach is much cheaper, as it does not require LRU stacks; however, it sacrifices some accuracy because it can only *estimate* the number of memory blocks associated with a given RD value. Our techniques are used for offline analysis, and thus we employ the more accurate LRU-based method of computing RD. But one advantage of the alternate approach is that it opens up the possibility for online analysis and optimization.

6. CONCLUSION

As multicore CPUs continue to scale, the complexity of their design spaces will also continue to grow. Moreover, due to combinatorial growth, the rate at which design space complexity increases will outpace the speed of traditional simulation-based evaluation methodologies. Unless addressed, this problem will make it more and more difficult for architects to explore multicore design spaces and to optimize CPU performance and power consumption in the future. Hence, developing new methodologies to help architects cope with increased design complexity is crucial.

This article investigates using multicore RD analysis to accelerate design space exploration. Our goal is to help architects rapidly identify the most power efficient configurations for a multicore's cache hierarchy given one or more parallel programs. In particular, we developed several analytical models that use the CMCs from locality profiles of parallel programs to estimate performance and power consumption. Due to the architecture independence characteristics of our locality profiles, we can perform predictions across entire cache hierarchy design spaces once a small number of profiles have been acquired. Next, we applied our techniques to optimize the design of a tiled CPU with up to 256 cores. We showed the power efficiency of the best configuration in the design space, provided insight into why it is the best, and quantified how much better it is compared to other points in the design space. Finally, we also validated the accuracy of our predictions against detailed architectural simulation.

Our results show that the most power efficient configurations vary considerably across benchmarks. They tend to employ L1 and L2 cache sizes that coincide with large drops in programs' PRD and CRD profiles, using large cluster sizes only when there is significant sharing at feasible L2 capacities. In addition, our results show that picking the best configuration can provide significant benefits, as there is a 2.01x spread in power efficiency across the entire design space. And compared to the average, the best configuration achieves a 22.1% gain in power efficiency. Our results also show that our techniques can be quite accurate. Among several sampled configurations that were simulated on Graphite, we usually correctly identify the most power efficient configuration for each benchmark, or a configuration within 5% of the best. Additionally, across all sampled configurations, we predict the power efficiency with a 15.2% error on average.

In the future, we hope to improve the techniques presented in this article to achieve better relative and absolute prediction accuracy. Specifically, we will expand our models to handle some of the omitted memory transactions mentioned in Section 3.2.3 and to predict the effects of write stalls. (Our locality profiles already track read and write misses separately, so we should be able to detect when write misses are very frequent, and hence write stalls are likely to occur.) In addition, inaccurate prediction of conflict misses is a significant source of error, as discussed in Section 4.4. Because we almost always underpredict conflicts, there are likely systematic effects that we may be able to identify and model. In addition, at the moment, our profiling runs do not record the difference in number of instructions executed across profiled threads. If we add this to our profiler, we could potentially update Equation (1) to account for load imbalance.

Another major direction for future work is to generalize our techniques to other architectures and applications. On the architecture side, we have limited ourselves to tiled CPUs with single-threaded in-order cores for this article. We hope to apply our techniques to more complex CPUs—e.g., with multithreaded and/or out-of-order cores and nonmesh cache topologies. (Out-of-order core models already exist [Eyerman et al. 2009], so we can try to incorporate these models into our own work.) Another architectural extension is to apply our techniques for GPUs. Modern GPUs employ caches in addition to software-managed memories to mitigate off-chip bandwidth requirements. Our techniques could provide valuable insights into the best GPU cache configurations. Finally, on the application side, we have limited ourselves to analyzing homogeneous multithreaded parallel programs for this article. We hope to develop techniques for programs with nonhomogeneous threads as well. This is challenging given that such programs exhibit architecture dependent locality profiles. (Nonhomogeneous threads may speed up or slow down at different rates as cache configurations change, perturbing their memory reference interleaving.) Nevertheless, it is a worthwhile research direction, as it would enable our techniques for more irregular, e.g., task-level, parallelism.

REFERENCES

Erik Berg and Erik Hagersten. 2004. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*.

Erik Berg and Erik Hagersten. 2005. Fast data-locality profiling of native execution. In *Proceedings of the ACM SIGMETRICS Conference*.

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*.

Kenzo Van Craeynest and Lieven Eeckhout. 2011. The multi-program performance model: Debunking current practice in multi-core simulation. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*.

John Davis, James Laudon, and Kunle Olukotun. 2005. Maximizing CMP throughput with mediocre cores. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*.

Peter J. Denning. 1968. The working set model for program behavior. *Communications of the ACM* 11, 5, 323–333.

Chen Ding and Trishul Chilimbi. 2009. *A Composable Model for Analyzing Locality of Multi-Threaded Programs*. Technical Report MSR-TR-2009-107. Microsoft Research.

Chen Ding and Yutao Zhong. 2003. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*.

David Eklov, David Black-Schaffer, and Erik Hagersten. 2011. Fast modeling of shared caches in multicore systems. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*.

David Eklov and Erik Hagersten. 2010. Statstack: Efficient modeling of LRU caches. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software*.

Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and Jim Smith. 2009. A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems* 27, 2, Article No. 3.

J. Gecsei, D. R. Slutz, and I. L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM Systems Journal* 9, 2, 78–117.

Song-Liu Guo, Hai-Xia Wang, Yi-Bo Xue, Chong-Min Li, and Dong-Sheng Wang. 2010. Hierarchical cache directory for CMP. *Journal of Computer Science and Technology* 25, 2, 246–256.

Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2009. Reactive NUCA: Near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th International Symposium on Computer Architecture*.

Lisa Hsu, Ravi Iyer, Srihari Makineni, Steve Reinhardt, and Donald Newell. 2005. Exploring the cache design space for large scale CMPs. *ACM SIGARCH Computer Architecture News*, 4, 24–33.

Jaehyuk Huh, Stephen W. Keckler, and Doug Burger. 2001. Exploring the design space of future CMPs. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*.

Intel. 2014. Intel Xeon Phi Product Family. Available at http://www.intel.com/XeonPhi.

Engin İpek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. 2006. Efficiently exploring architectural design spaces via predictive modeling. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*. DOI:http://dx.doi.org/10.1145/1168857.1168882

Yunlian Jiang, Eddy Z. Zhang, Kai Tian, and Xipeng Shen. 2010. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *Proceeding of the International Conference on Compiler Construction*.

Benjamin C. Lee and David M. Brooks. 2006. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*. DOI:http://dx.doi.org/10.1145/1168857.1168881

Jian Li and Jose F. Martinez. 2005. Power-performance implications of thread-level parallelism on chip multiprocessors. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*.

Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the International Symposium on Microarchitecture*.

Yingmin Li, Benjamin Lee, David Brooks, Zhigang Hu, and Kevin Skadron. 2006. CMP design space exploration subject to physical constraints. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*.

Gabriel H. Loh. 2008. 3D-stacked memory architectures for multi-core processors. In *Proceedings of the 35th International Symposium on Computer Architecture*.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Martina Maggio, Henry Hoffman, Anant Agarwal, and Alberto Leva. 2011. Control-theoretical CPU allocation: Design and implementation with feedback control. In *Proceedings of the 6th International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks*.

Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald III, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. 2010. Graphite: A distributed parallel simulator for multicores. In *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture*.

Ramanathan Narayanan, Berkin Ozisikyilmaz, Joseph Zambreno, Gokham Memik, and Alok Choudhary. 2006. MineBench: A benchmark suite for data mining workloads. In *Proceedings of the International Symposium on Workload Characterization*.

Apan Qasem and Ken Kennedy. 2005. *Evaluating a Model for Cache Conflict Miss Prediction*. Technical Report CS-TR05-457. Rice University.

Brian Rogers, Anil Krishna, Gordon Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. 2009. Scaling the bandwidth wall: Challenges in and avenues for CMP scaling. In *Proceedings of the 36th International Symposium on Computer Architecture*.

Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai. 2010. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*.

Derek L. Schuff, Benjamin S. Parsons, and Vijay S. Pai. 2009. *Multicore-Aware Reuse Distance Analysis*. Technical Report TR-ECE-09-08. Purdue University.

Chen Sun, Chia-Hsin Owen Chen, George Kurian, Lan Wei, Jason Miller, Anant Agarwal, Li-Shiuan Peh, and Vladimir Stojanovic. 2012. DSENT—a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In *Proceedings of the 6th International Symposium on Networks-on-Chip*.

Deborah A. Wallach. 1993. PHD: A Hierarchical Cache Coherent Protocol. Master's Thesis. Massachusetts Institute of Technology.

Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture*.

Meng-Ju Wu and Donald Yeung. 2011. Coherent profiles: Enabling efficient reuse distance analysis of multicore scaling for loop-based parallel programs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*.

Meng-Ju Wu and Donald Yeung. 2012. Identifying optimal multicore cache hierarchies for loop-based parallel programs via reuse distance analysis. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*.

Meng-Ju Wu and Donald Yeung. 2013. Efficient reuse distance analysis of multicore scaling for loop-based parallel programs. *ACM Transactions on Computer Systems* 31, 1, Article No. 1.

Meng-Ju Wu, Minshu Zhao, and Donald Yeung. 2013. Studying multicore processor scaling via reuse distance analysis. In *Proceedings of the 40th International Symposium on Computer Architecture*.

Michael Zhang and Krste Asanovic. 2005. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture*.

Li Zhao, Ravi Iyer, Srihari Makineni, Jaideep Moses, Ramesh Illikkal, and Donald Newell. 2007. Performance, area and bandwidth implications on large-scale CMP cache design. In *Proceedings of the Workshop on Chip Multiprocessor Memory Systems and Interconnect*.

Yutao Zhong, Steven G. Dropsho, and Chen Ding. 2003. Miss rate prediction across all program inputs. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*.

Yutao Zhong, Xipeng Shen, and Chen Ding. 2009. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems* 31, 6, Article No. 20. DOI:http://dx.doi.org/10.1145/1552309.1552310

Received October 2014; revised October 2015; accepted November 2015