

Avoiding Cache Thrashing due to Private Data Placement in Last-level Cache For Manycore Scaling

Jiayuan Meng, Kevin Skadron
Department of Computer Science, University of Virginia
jm6dg, skadron@cs.virginia.edu

Abstract—Without high-bandwidth broadcast, large numbers of cores require a scalable point-to-point interconnect and a directory protocol. In such cases, a shared, inclusive last level cache (LLC) can improve data sharing and avoid three-way communication for shared reads. However, if inclusion encompasses thread-private data, two problems arise with the shared LLC. First, current memory allocators align stack bases on page boundaries, which emerges as a source of severe conflict misses for large numbers of threads on data-parallel applications. Second, correctness does not require the private data to reside in the shared directory or the LLC.

This paper advocates stack-base randomization that eliminates the major source of conflict misses for large numbers of threads. However, when capacity becomes a limitation for the directory or last-level cache, this is not sufficient. We then propose non-inclusive, semi-coherent cache organization (NISC) that removes the requirement for inclusion of private data and reduces capacity misses. Our data-parallel benchmarks show that these limitations prevent scaling beyond 8 cores, while our techniques allow scaling to at least 32 cores for most benchmarks. At 8 cores, stack randomization provides a mean speedup of 1.2X, but stack randomization with 32 cores gives a speedup of 2.7X over the best baseline configuration. Comparing to conventional performance with a 2 MB LLC, our technique achieves similar performance with a 256 KB LLC, suggesting LLCs may be typically overprovisioned. When very limited LLC resources are available, NISC can further improve system performance by 1.8X.

I. INTRODUCTION AND BACKGROUND

Processors optimized for throughput employ many small, multithreaded cores [1] and seem likely to scale up to large core and thread counts. They are sometimes referred to as chip multithreading (CMT). Examples include Sun's Niagara [2] and Intel's Larrabee [3]. At the same time, caches remain important for performance and they reduce off-chip traffic. Hardware cache coherence also remains important because it simplifies the task of writing parallel programs.

As the number of cores increases, a single broadcast medium cannot sustain the inter-processor communication bandwidth. This requires a shift to point-to-point on-chip networks (OCNs) as well as a directory protocol, because the broadcast operations required for snooping are not practical over a point-to-point OCN. The directory coherence, in turn, benefits from a shared, inclusive last level cache (LLC) for management of shared data. This is because exclusive [4] or non-inclusive [5] caching both incur three-way communication (i.e. among data requester, directory, and owner) to locate shared data and they also introduce nontrivial design and verification complexity [6].

Inclusion poses a problem for large-scale, CMT organizations, because each LLC cache set is contended for by all

threads. Multithreaded manycore chips risk excessive conflict misses over the shared, inclusive LLC. Inclusion requires that an eviction in the LLC also evict any copies in the L1s, so conflicts can lead to cache thrashing even for data in active use.

We have observed that one of the most important sources of thrashing is LLC contention by *private* data that need not reside in the LLC at all. In the rest of paper, we refer to data that are only accessed from one core as *private data*, and they are mostly composed of *thread-private* data such as stacks. All other data are referred to as *shared data*, which *may* be accessed from multiple cores. We find that current memory management practice typically distributes private data non-uniformly among the LLC cache sets, with wasteful and unnecessary LLC conflicts. In this paper, we focus on the problem caused by conventional stack allocation mechanisms that tend to align stack bases to page boundaries, with the consequence that thread-private data are likely to concentrate on a few LLC cache sets.

To our knowledge, there has been no prior study that shows to what extent private data contend for the LLC and how they may affect the performance of a large scale CMT connected through a point-to-point OCN. While separate address spaces [7], locality-aware memory allocators [8], [9], [10] and task schedulers [11], [12], [13] can reduce coherence traffic, they do not address capacity or conflict misses in the LLC. Even based upon a locality-aware memory allocator, we show that with 16 eight-way multi-threaded cores in an inclusive organization, 5-10% of cache sets in a 16-way associative LLC are severely contended for by private data, raising conflict misses and unnecessary L1 evictions in an inclusive organization. This may eventually lead to cache thrashing and jeopardize performance.

To reduce LLC conflicts and mitigate cache thrashing, we first propose a simple run-time stack allocation mechanism that randomizes the offset of the stack bases relative to page boundaries. Without any modification to the program nor the hardware, *stack randomization* distributes thread-private data more uniformly, and it alone improves performance by a factor of 2.7X on 32 cores compared to the best baseline configuration. This is different from address space randomization used for security reasons [14], which may randomize stack bases — not necessarily their offsets relative to page boundaries — to lower the chances that stack bases are easily predicted by an attacker.

We also compare different LLC replacement policies according to their ability to address the same issue with-

out modifying the application or the run-time system. We observe that the LRU insertion policy (LIP) [15] barely mitigates the penalty of LLC conflicts in such scenario. We also investigate a modified LRU replacement policy, LRSU, that replaces shared data in a conflicting cache set before replacing any private data. This may seem counter-intuitive, but assumes that private data (such as stack variables) are likely to be used actively in the L1 caches. However, like LIP, LRSU also proves to be inferior.

Despite the potential benefit of stack randomization, it does not address the underlying hardware behavior: private data are used by a single core and if threads do not migrate, private data can be excluded from coherence as well as the inclusive hierarchy to make room for shared data in the LLC. Although researchers proposed dedicated storage for thread-private data [16], [17], [18], [19], such techniques run the risk of under- or over-utilization of the private storage. Using the MESI coherence protocol as a baseline design, we exclude private data—including both stacks and private heaps—from the inclusive coherence protocol. We refer to this as a *non-inclusive, semi-coherent (NISC)* cache organization. NISC allows private data to exist only in the L1 caches and private data evicted from LLC need not invalidate their copies in the L1 cache. Results show NISC performs significantly better than cache replacement policies but stack randomization still performs the best for LLCs with sufficient capacity. However, NISC can be employed together with stack randomization, and we show that when the LLC does not have sufficient capacity to support inclusion, NISC’s ability to reduce the number of lines contending for the LLC allows significant performance benefits relative to just stack randomization. The resulting ability to reduce the demand for LLC resources can be used to reduce silicon costs or to increase the number of cores, as well as accommodate more concurrent processes with large working sets.

II. NON-UNIFORM DISTRIBUTION OF PRIVATE DATA

We study a two-level memory hierarchy with private L1 caches and an LLC as shared, distributed L2 caches. Hsu et al. [20] underline the importance of shared LLC, and a great deal of work has explored ways to hide wire delay for distributed L2 caches [21], [22], [23]. However, they do not mitigate LLC contention caused by non-uniform distribution of data, which would arise regardless of whether the LLC is centralized or distributed. In the following discussion we focus on the non-uniform distribution of private data in the LLC, which is mainly caused by conventional stack allocation mechanisms that align stack bases to page boundaries.

As an example, consider an operating system (OS) with a page size of 8 KB and a cache hierarchy with a 1024 KB, 16-way associative LLC with 128 B cache lines. As Figure 1 illustrates, the LLC has 512 cache sets in total each with a 9-bit index, of which 6 bits are part of page offsets and only 3 bits are part of page numbers. Therefore, data located at page boundaries is clustered around cache sets with the indices of “xxx000000”. As a result, private data compete for these cache sets more intensively than others. Specifically, stack bases on page boundaries would compete

for 8 cache sets or 128 cache lines; in other words, thrashing is likely to occur when the LLC is shared by 128 concurrent threads or more. We name this small subset of cache lines that lead to thrashing *critical lines*, and their number can be calculated as

$$\max\left(\frac{LLC_size}{page_size}, LLC_associativity\right) \quad (1)$$

Usually, Equation 1 results in $\frac{LLC_size}{page_size}$ and it indicates that systems with larger page sizes or smaller LLCs have fewer critical lines and data distribution becomes less uniform. Even worse, critical lines also have to accommodate data other than stacks, and they are scattered in different cache sets which may not be evenly contended for. As a result, despite the increased number of critical lines, a larger LLC may only suffer less, but not avoid thrashing caused by private data. We further study this effect in Section V-B.

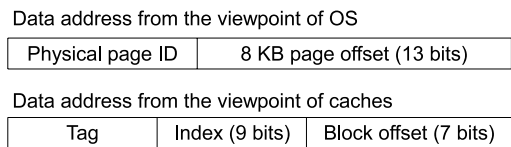


Fig. 1: The OS segments the address space to pages while the hardware segments the address space into cache sets and blocks. Aligning stack bases to page boundaries introduces non-uniform distribution of private data. This is illustrated by OS with 8K pages and a 1024 KB, 16-way associative LLC with 128 B cache lines.

This issue of LLC contention caused by the non-uniform distribution of private data, to the best of our knowledge, has not been studied since the advent of the recent trend toward manycore CMT architectures. We characterize a spectrum of nine benchmarks (shown in Table II in Section III-B) and simulate them on a multithreaded CMP that has 1–32 cores, with each providing 8 hardware thread contexts. For all benchmarks except KMeans, private data consists of stack data only. KMeans also has private heaps that store partial sums during Map-Reduce computation. The effect of fewer critical lines is reflected in Figure 2(a) where a larger page size leads to performance degradation at smaller number of cores or threads.

With a page size of 8 KB, Figure 2(b) shows the occupancy of private data in a 1-2 MB shared, inclusive LLC that accommodates I- and D-caches of 16 KB each. With 16 cores and 128 thread contexts or more, 5-10% of the 16-way associative LLC cache sets have private data occupying more than half of their blocks. These cache sets are intensively contended and victims have to invalidate their D-cache copies as well. Victims in these cache sets are likely to be private data.

We further characterize our benchmarks and study the breakdown of their memory accesses using the infrastructure described in Section III. For 16 cores with 128 thread contexts, private data account for less than 15% of the applications’ working set, however, 51% of the memory accesses request private data. This reflects that private data

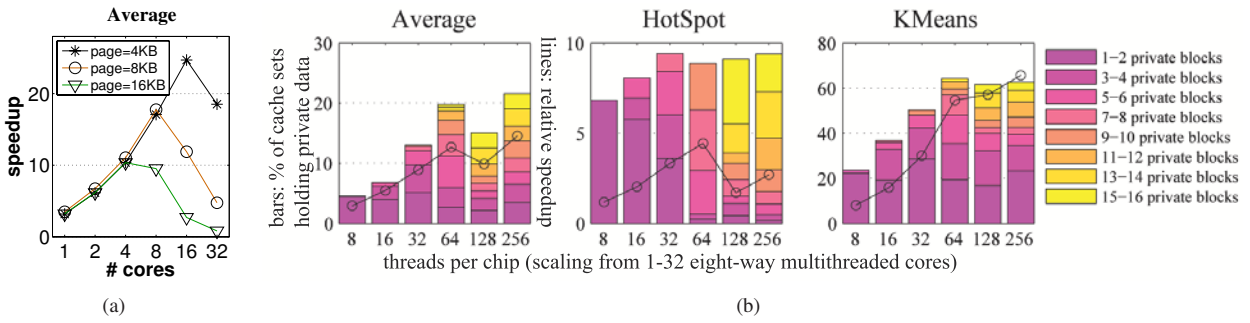


Fig. 2: (a) The average speedup across all benchmarks when we scale the number of cores from 1 to 32, each is 8-way threaded. The LLC is 1 MB and 16-way associative. Speedup is normalized to single-threaded execution. A larger page size leads to more performance degradation at a smaller number of cores. (b) Bars show the distribution of LLC cache sets with regard to private data occupancy, and lines illustrate the correlation with relative performance speedup. The LLC is 16-way associative, with a capacity of 1024 KB with 16 cores or fewer and 2048 KB with 32 cores. Each core is 8-way multi-threaded.

TABLE I: Default System Configuration

Technology Node	65 nm
Cores	Alpha ISA, 2.0 GHz, 0.9V Vdd in order, 8-way multithreaded 2 SIMD thread groups each with 4 threads
L1 Caches	16 KB I-cache and 16 KB D-cache 16-way associative, 32 B line size, write-back physically indexed, physically tagged 4 banks, 16 MSHRs, 3 cycle hit latency
L2 Cache	\geq 1024 KB, distributed 16 banks, up to 8 pending requests each 16-way associative, 128 B line size 32 cycle access latency, write-back physically indexed, physically tagged 64 MSHRs, each can hold up to 8 requests
Interconnect	2-D Mesh, wormhole routing 300 MHz, 57 Gbytes/s 1 cycle routing latency 1 cycle link latency per hop
Memory Bus	266 MHz, 16 GB/s
Main Memory	50 ns access latency

are reused much more often than shared data. On the other hand, 64% of D-cache misses occur to private data and 62% of these misses end up reloading an invalidated block. This implies that evicting private data as victims of LLC conflicts introduces unnecessary invalidations and causes cache thrashing. Note that this phenomenon is only obvious when a large number of threads aggregate a large amount of private data that all compete for the inclusive LLC.

III. EXPERIMENTAL SETUP

A. Simulation

Due to the unavailability of coherent, cache-based many-core products with large numbers of cores and threads, we simulate a future CMT processor using M5 [24], a cycle-accurate, event-driven simulator for networks of processors. We extend the simple, scalar core model to support fine-grained multithreading and add support for a 2D mesh OCN, directory-based coherence, and a shared, banked LLC. We name our large-scale CMT simulator *MV5*.

Table I summarizes the main parameters of the simulated manycore processor. We model throughput-oriented, multi-threaded in-order cores that are loosely inspired by Niagara-style cores [2]. Scalar threads are grouped into SIMD groups

that share a common instruction sequencer as modern GPUs do [25]. Upon memory accesses, the SIMD group cannot proceed until all its threads' requests are fulfilled. In order to hide this memory latency, the core immediately switches to another SIMD group and continues execution with no extra latency by simply indexing into the appropriate register files. We acknowledge the limitation that our core model does not support non-blocking loads or other more aggressive techniques for exploiting memory-level parallelism (MLP), and that the interaction between MLP techniques and the phenomena we study is an interesting area for future work.

The memory system is a two-level coherent cache hierarchy. Each core has a private I-cache and D-cache, which are connected to a shared LLC through a 2D mesh interconnect. The LLC is composed of distributed L2 cache banks, with the number of banks equal to the number of cores. L2 caches and cores are tiled in an interleaved way similar to a checkerboard. The total size of the LLC is 1024 KB with 16 cores or fewer and 2048 KB with 32 cores, note that LLC capacity is no smaller than twice the cumulative L1 cache capacity, as Hsu et al. [20] suggested. The LLC cache sets are statically partitioned across the L2 banks. L1 caches are kept coherent through an inclusive, directory-based MESI protocol.

We model cache latency according to Cacti [26] and work from Kim [27]. Pullini et al. provide the basis for our interconnect latency modeling [28]. For the in-order cores, instructions-per-cycle (IPC) is assumed to be one except for memory references, which are modeled faithfully through the memory hierarchy (although we do not model memory controller effects).

B. Benchmarks

We select a set of emerging data-parallel applications ranging across image processing, scientific computing, physics simulation, machine learning and data mining. Our primary objective is to obtain representatives of different parallel program types. We choose the input size such that our benchmarks exhibit sufficient parallelism for evaluation while maintaining manageable simulation times. Table II summarizes the benchmarks. Benchmarks are cross-compiled

to the Alpha ISA using GCC 4.1.0. Parallel `for` loops are annotated in OpenMP style and are interpreted by an emulated system call that later allocates memory spaces for stacks and spawns threads across cores. Data-parallel tasks are grouped into blocks according to the number of cores. Blocks are assigned to cores in a persistent order for the purpose of cache-affinity.

Our performance study uses a baseline that already employs a scalable locality-aware memory allocator that maintains per-core lists of private pages for private data [9], [10]. By doing so, we isolate our results from previous studies that aim to reduce coherence traffic.

TABLE II: Simulated benchmarks with descriptions and input sizes.

	Benchmark Description
<i>FFT</i>	Fast Fourier Transform (Splash2 [29]) Spectral methods. Butterfly computation Input: a 1-D array of 32,768 (2^{15}) numbers
<i>Filter</i>	Edge Detection of an Input Image Convolution. Gathering a 3-by-3 neighborhood Input: a gray scale image of size 500×500
<i>HotSpot</i>	Thermal Simulation (Rodinia [30]) Iterative partial differential equation solver Input: a 300×300 2-D grid, 100 iterations
<i>LU</i>	LU Decomposition (Splash2 [29]). Dense linear algebra Alternating row-major and column-major computation Input: a 300×300 matrix
<i>Merge</i>	Merge Sort. Element aggregation and reordering Input: a 1-D array of 300,000 integers
<i>N-W</i>	Needleman-Wunsch DNA alignment. (Rodinia [30]) Dynamic programming by updating matrix with a diagonal wavefront Input is a 2-D array of size 512×512
<i>Short</i>	Winning Path Search for Chess. Dynamic programming. Neighborhood calculation based on the the previous row Input: 6 steps each with 150,000 choices
<i>KMeans</i>	Unsupervised Classification (MineBench [31]). Map-Reduce. Distance aggregation. Input: 10,000 points in a 20-D space
<i>SVM</i>	Supervised Learning (MineBench [31]) Support vector machine's kernel computation. Input: 1,024 vectors with a 20-D space

IV. APPROACHES TO REDUCE LLC CONFLICTS

From the aspects of both software and hardware, we propose and compare several solutions to reduce LLC conflicts and cache thrashing caused by non-uniform distribution of private data. We also evaluate existing techniques in their effectiveness to address the same issue.

A. Randomly Offsetting Stack Bases

Since the non-uniform distribution of private data is mostly caused by stack bases that are usually aligned to page boundaries, we modify the thread library so that stack bases are now offset to random locations within a physical page. We still enforce the stack bases to be aligned to cache line boundaries in order to avoid misaligned data. In this way, stack may start with *any* cache lines and private data are likely to span evenly across all LLC cache sets. We do not modify the operating system's mechanism that maps virtual pages to physical pages.

No hardware modification is needed for this approach. However, this may introduce inefficient memory management in the OS since offsetting stack bases introduces fragmentation in the allocated physical pages. In the case

where the stack base is offset to the tail of a page, the fragment can be almost as large as an entire page. The issue of fragmentation warrants further investigation but space limitations preclude their study here.

B. LLC Replacement Policies

Randomizing the stack bases *avoids* conflict misses caused by private data in the LLC. Alternatively, the underlying architecture can be designed to *tolerate* the non-uniform distribution of private data with no need to modify existing stack allocation mechanisms. We extend the LLC's LRU replacement policy to evict shared data first. In this way, actively used private data in the L1s are not likely to be invalidated due to LLC conflicts. This policy is referred to as *LRSU* (least recently used shared data) replacement.

Alternatively, we also experiment LLCs with LIP (LRU Insertion Policy) [15]. Based on our observation that private data are reused more often than shared data (private data evicted from D-caches are likely to be reloaded), we postulate that these more valuable lines are more likely to reside closer to the MRU (most recently used) position. For shared data that are streamed into L1s for one-time access, their LLC cache blocks stay in the LRU position and get evicted during the next replacement, leaving private data intact in the LLC. However, in some circumstances, LIP may increase the chance of private data to thrash the LLC — for those private data that remain in the L1, it is possible that their LLC copies are not reused and remain in the LRU position; these private data are likely to subject to LLC replacement.

C. Non-Inclusive, Semi-Coherent Cache Hierarchy

Cache replacement policies, however, do not address the fundamental issue: private data compete for LLC space due to the enforcement of inclusion property, however they require no coherence and therefore do not benefit from inclusion property's savings of coherence overhead. By excluding private data from the inclusive coherence protocol, private data can be well accommodated in D-caches and the LLC can host more shared data. We call this a Non-Inclusive, Semi-Coherent cache organization (NISC). As Figure 3 illustrates, NISC allows private data to exist only in the L1 cache—replaced private data in LLC do not have to invalidate their copies in the L1 cache. As a result, private data contend less for LLC cache sets, reducing LLC conflict misses. Moreover, the overall on-chip storage is utilized better because of the reduced data replicates, and this reduces capacity misses as well.

NISC is compatible with *any* inclusive coherence protocols, and we demonstrate how NISC works with a simple MESI coherence protocol [32] for a two-level write-back cache hierarchy. An additional *coherence toggle bit (CTB)* is added to each cache line's state. The CTB marks whether the cache line stores shared or private data so that coherence is toggled on and off for that cache line, respectively. The CTB can be determined in several ways:

- Option 1 (OS approach): Utilizing run-time scalable memory allocators which maintain a list of per-core private pages and shared pages [9], [10]. Private pages can

mark their TLB entries as private using an additional bit per TLB entry, whose value can be inherited by the corresponding cache line's CTB.

- Option 2 (Compiler approach with extended ISA): Using implicit static analysis or explicit code annotations to identify private data. The compiler then uses a different set of memory access instructions specifically for accessing private data. For example, the ATI instruction set [33] and NVIDIA's parallel thread extension (PTX) [34] already use a different set of memory instructions for private data. These private-data oriented memory instructions set the corresponding cache lines' CTBs. For example, stack data can be assumed to be thread-private in most cases (although not guaranteed). On the other hand, private data may be easily identified and annotated by programmers, as demonstrated in several parallel programming languages and APIs such as OpenMP [35] and CUDA [36].

Any access to those cache lines with their CTBs set by-passes the MESI coherence protocol entirely and is managed through a non-inclusive, non-coherent protocol. Accesses to other cache lines proceed with the original MESI protocol. In this way, NISC manages private data and shared data distinctly while balancing them in the same storage hierarchy.

Care has to be taken for NISC upon thread migration. Since coherence is disabled for cache lines with CTBs set, these lines have to be flushed to the shared cache and reloaded to the destination L1 cache that will host the thread after its migration. Since we are dealing with data-parallel workloads with abundant parallelism, we assume workload can be mostly balanced and thread migration need not take place frequently. Nevertheless, context switches may still occur occasionally due to page faults or interrupts, and flushing the cache serves as a solution in such circumstances.

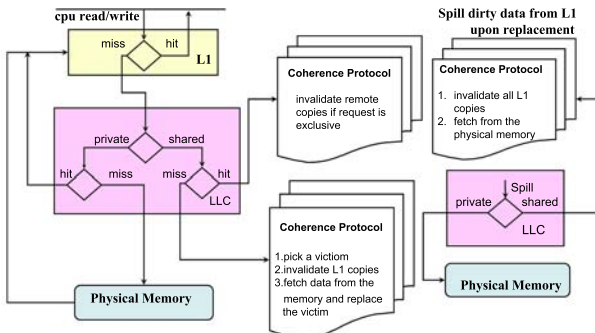


Fig. 3: The NISC protocol based on MESI. Except for requests and spillings for private data, all other data movement exhibits the same behavior as in MESI. Note that blocks holding thread-private data should not have copies in remote L1s and therefore are always writable.

V. EVALUATION

The following configurations are compared and their sensitivity to the last level cache design is studied as well.

- *conv*: The conventional system with stack bases aligned to page boundaries and the memory system is maintained by an inclusive MESI coherence protocol with LRU replacement policy.
- *randStack*: The OS offsets the stack base to some random distance away from the page boundary. The underlying hardware is unmodified.
- *LRSU*: The OS is unmodified. The LLC employs LRSU replacement policy that attempts to hold as much private data as possible.
- *LIP*: The OS is unmodified. The LLC employs LIP that first replaces lines that are not reused.
- *NISC*: The OS is unmodified except it signals private data to the hardware which sets the CTB for corresponding cache lines and excludes them from the inclusive coherence.
- *NISC+randStack*: The OS randomizes stack bases as in *randStack* and it operates over an architecture with NISC cache organization.

A. Performance Scaling

We increase the number of cores from 1 to 32 and the total number of thread contexts from 8 to 256, and speedup is calculated based on single threaded execution. These results are shown in Figure 4. Performances of *conv*, *LIP* and *LRSU* start to degrade beyond 8 cores. Note that we double the LLC size to 2 MB with 32 cores to ensure the LLC size is no smaller than twice the aggregate size of L1 caches. However, performance with 32 cores is still much worse than 8 cores with *conv*. In order to eliminate partial accesses to LLC lines which may reduce the effectiveness of *LIP*, we also experiment an LLC with a line size equal to that of the L1 line size, however, performance scaling for *LIP* remains similar. *LIP* and *LRSU* fail to improve performance mainly because they do not reduce the amount of data that contend for the same LLC cache sets. We also observe that applications with more private data competing for the LLC degrades more severely, as we illustrate in Figure 2(b).

On the other hand, performance of *randStack*, *NISC* and *NISC+randStack* continues to scale. Comparing to the peak performance of *conv* at 8 cores, the middleware approach, *randStack*, achieves average speedups of 1.2X at 8 cores, 1.8X at 16 cores, and 2.7X at 32 cores. The average speedups for the hardware approach, *NISC*, are 1.0X, 1.5X and 2.0X, respectively. Their combination, *NISC+randStack*, performs similarly to *randStack*. We also evaluate the combinations of *randStack* with *LIP* and *LRSU*, and they perform similarly to *randStack* as well. Therefore, provided with stack randomization and sufficient LLC capacity, there is no need to further employ hardware approaches to address the non-uniform distribution of private data.

B. LLC Sensitivity

If the LLC is simply too small, techniques such as stack randomization that only address conflicts will be insufficient. The LLC may not have sufficient capacity for programs with large working sets, or when trading off LLC area to include more cores within a given die budget. NISC is attractive here

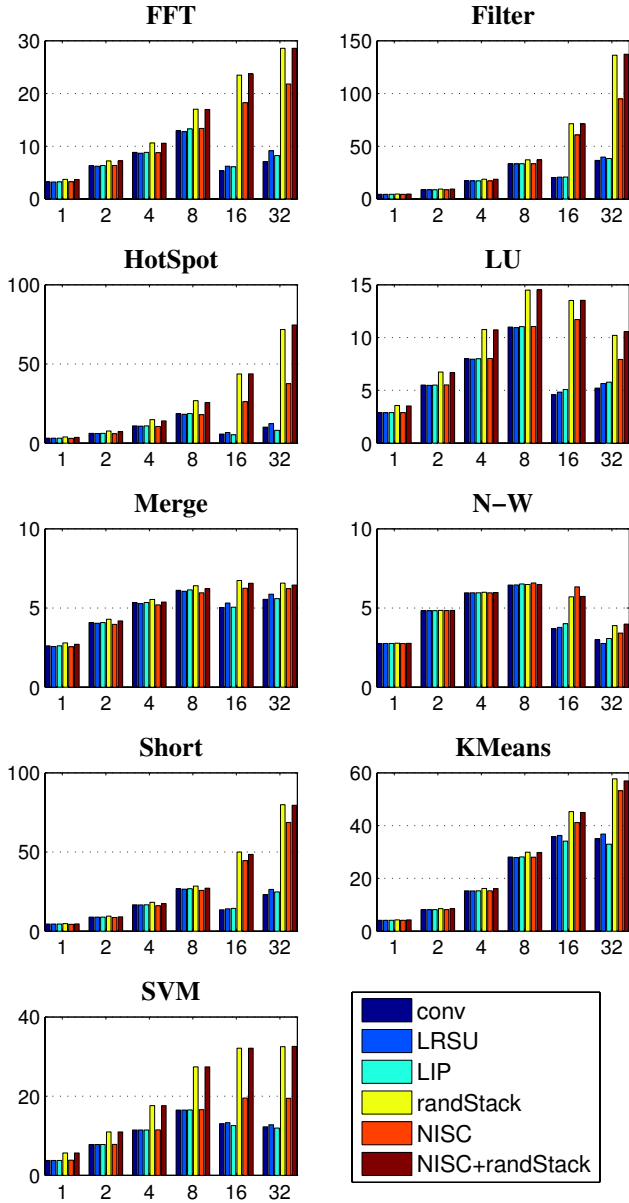


Fig. 4: Speedup vs. Number of eight-way multithreaded cores. Speedup is normalized to single-threaded execution. LLC size is no smaller than twice the aggregate size of L1 caches, therefore it is 2048 KB in the case of 32 cores and 1024 KB otherwise.

because it reduces the capacity requirement of the LLC by relaxing inclusion for private data. On the other hand, if this is merely a capacity issue, then *randStack* is not expected to benefit large enough LLC.

As we observe from Figure 5 when scaling the LLC size up to 16 MB over 16 cores, *the benefit of randStack is not limited to small LLC sizes*. With smaller LLCs, *conv*, *LIP* and *LRSU* degrade drastically, and *randStack* degrades more gracefully. With an LLC larger than 256 KB, *randStack*, as a technique that *avoids* LLC conflicts caused by private data, out-performs *NISC*, which merely attempts to *tolerate* the

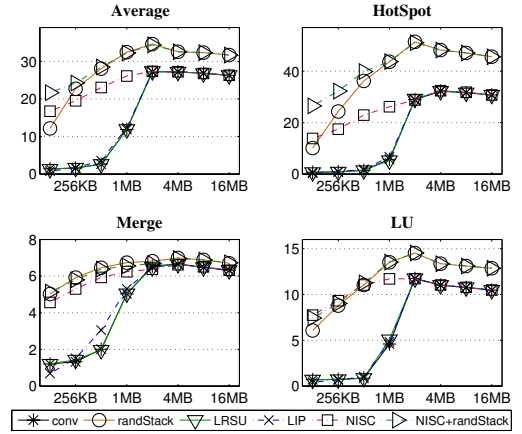


Fig. 5: Speedup over single-threaded execution vs. LLC sizes. Speedup is measured on a system with 16 cores and 128 thread contexts, and is calculated relative to the performance of a single-threaded execution. LLC associativity is 16.

LLC conflicts. However, when the LLC is smaller than 256 KB, capacity misses dominate and this is only addressed by *NISC*. The best performance with a small LLC is achieved by combining these two approaches — in *NISC+randStack*, *NISC* further improves the performance of *randStack* by a factor of 1.8 with a small LLC capacity of 128 KB. This shows that the software approach (stack randomization) and the hardware approach (*NISC*) are complementary. We also experiment with *LIP* and *LRSU* when combined with stack randomization. However, they do not further improve performance over *randStack*.

Similar benefit of *NISC+randStack* is observed with smaller LLC associativity, as shown in Figure 6. Besides, Equation 1 indicates that larger associativity is not likely to increase the number of critical lines, but only helps distributing private data more evenly across cache sets that host critical lines. Therefore, we observe that the conventional performance improves significantly, but still cannot match that of *randStack* with an LLC associativity of 512.

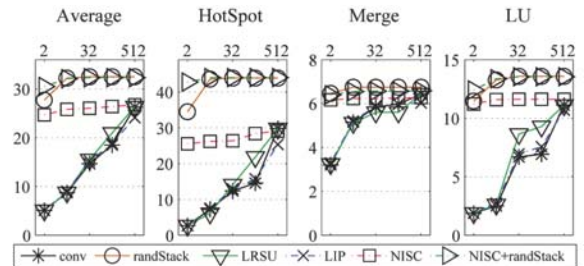


Fig. 6: Speedup over single-threaded execution vs. LLC associativity. Speedup is measured on a system with 16 cores and 128 thread contexts, and is calculated relative to the performance of a single-threaded execution. The size of the LLC is 1024 KB.

VI. CONCLUSIONS AND FUTURE WORK

In the presence of a large number of concurrent threads, even a small amount of private data per core can swamp a shared inclusive LLC. Worse yet, LLC evictions due to this contention then evict private data in the L1s. However, a shared, *inclusive* LLC is valuable in directory-coherence.

We study the performance impact of non-uniform distribution of private data caused by conventional stack allocation mechanisms. Several solutions are proposed. Our data-parallel benchmarks show that these limitations prevent scaling beyond 8 cores, while our techniques allow scaling to at least 32 cores for most benchmarks. At 8 cores, stack randomization provides a mean speedup of 1.2X, and stack randomization with 32 cores gives a speedup of 2.7X over the best baseline configuration. However, stack randomization may introduce fragmentation in the physical pages. We therefore investigate several hardware approaches including different LLC replacement policies and a non-inclusive, semi-coherent (NISC) cache organization that excludes private data from cache coherence. While LLC replacement policies fail to improve performance significantly, NISC alone scales performance up to at least 32 cores in most cases. Comparing to the best baseline configuration at 8 cores, NISC provides a mean speedup of 1.5X at 16 cores and 2.0X at 32 cores.

In cases where the LLC may have insufficient capacity for programs with large working sets, or when trading off LLC area to include more cores within a given die budget, combining NISC with stack randomization yields the best performance. With a limited LLC capacity of 128KB, NISC further improves the performance of *randStack* by a factor of 1.8. NISC results with small LLC suggest an interesting avenue for further research in reducing LLC area and allowing that to be used for other purposes.

VII. ACKNOWLEDGEMENTS

This work was supported in part by SRC grant No. 1607, NSF grant nos. IIS-0612049 and CNS-0615277, a grant from Intel Research, and a professor partnership award from NVIDIA Research.

REFERENCES

- [1] J. D. Davis, J. Laudon, and K. Olukotun, "Maximizing CMP throughput with mediocre cores," in *PACT*, 2005, pp. 51–62.
- [2] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: a 32-way multithreaded Sparc processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.
- [3] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM TOG*, vol. 27, no. 3, pp. 1–15, 2008.
- [4] N. P. Jouppi and S. J. E. Wilton, "Tradeoffs in two-level on-chip caching," in *ISCA*, Apr. 1994, pp. 34–45.
- [5] M. Zahran, K. Albayraktaroglu, and M. Franklin, "Non-inclusion property in multi-level caches revisited," in *Int'l J. Computers and their Applications*, no. 2, 2007, pp. 99–108.
- [6] X. Chen, Y. Yang, G. Gopalakrishnan, and C.-T. Chou, "Reducing verification complexity of a multicore coherence protocol using assume/guarantee," in *FMCAD*, 2006, pp. 81–88.
- [7] S. B. Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: An operating system for many cores," in *OSDI*, December 2008.
- [8] "Intel threading building blocks," *Intel Corporation*.
- [9] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos, "Scalable locality-conscious multithreaded memory allocation," in *ISMM*, 2006, pp. 84–94.
- [10] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: a scalable memory allocator for multithreaded applications," *SIGPLAN Not.*, vol. 35, no. 11, pp. 117–128, 2000.
- [11] G. E. Suh, S. Devadas, and L. Rudolph, "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *HPCA*, 2002, p. 117.
- [12] J. Torrellas, A. Tucker, and A. Gupta, "Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors," *JPDC*, vol. 24, no. 2, pp. 139–151, 1995.
- [13] S. Subramaniam and D. L. Eager, "Affinity scheduling of unbalanced workloads," in *SC*, 1994, pp. 214–226.
- [14] H. Shacham, E. jin Goh, N. Modadugu, B. Pfaff, and D. Boneh, "On the effectiveness of address-space randomization," in *CCS*, 2004, pp. 298–307.
- [15] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *ISCA*, 2007, pp. 381–391.
- [16] H. S. Lee and G. S. Tyson, "Region-based caching: an energy-delay efficient memory architecture for embedded processors," in *CASES*, 2000, pp. 120–127.
- [17] S. Park, H. woo Park, and S. Ha, "A novel technique to use scratch-pad memory for stack management," in *DATE*, 2007, pp. 1478–1483.
- [18] P. R. Panda, N. D. Dutt, and A. Nicolau, "On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems," *ACM TODAES*, vol. 5, no. 3, pp. 682–704, July 2000.
- [19] C. McCurdy and C. Fischer, "A localizing directory coherence protocol," in *WMPI*, 2004, pp. 23–29.
- [20] L. Hsu, R. Iyer, S. Makineni, S. Reinhardt, and D. Newell, "Exploring the cache design space for large scale CMPs," *dasCMP*, vol. 33, no. 4, pp. 24–33, 2005.
- [21] J. Chang and G. S. Sohi, "Cooperative caching for Chip Multiprocessors," in *ISCA*, 2006, pp. 264–276.
- [22] M. Zhang and K. Asanovic, "Victim replication: Maximizing capacity while hiding wire delay in tiled Chip Multiprocessors," in *ISCA*, 2005, pp. 336–345.
- [23] —, "Victim migration: Dynamically adapting between private and shared CMP caches," in *MIT Technical Report MIT-CSAIL-TR-2005-064, MIT-LCS-TR-1006*, 2005.
- [24] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, no. 4, 2006.
- [25] N. Corporation, "GeForce GTX 280 Specifications," 2008.
- [26] D. Tarjan, S. Thoziyoor, and N. P. Jouppi, "Cacti 4.0," HP Laboratories Palo Alto, Tech. Rep. HPL-2006-86, 2006.
- [27] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," *ASPLOS*, vol. 36, no. 5, 2002.
- [28] A. Pullini, F. Angiolini, S. Murali, D. Atienza, G. D. Micheli, and L. Benini, "Bringing NoCs to 65 nm," *IEEE Micro*, vol. 27, no. 5, 2007.
- [29] J. P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford parallel applications for shared memory," *ISCA*, vol. 20, no. 1, pp. 5–44, Mar. 1995.
- [30] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general purpose applications on graphics processors using CUDA," *JPDC*, 2008.
- [31] R. Narayanan, B. Ozisikylmaz, J. Zambreno, G. Memik, and A. Choudhary, "Minebench: A benchmark suite for data mining workloads," *WC*, pp. 182–188, Oct. 2006.
- [32] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach*, 3rd ed. Morgan Kaufmann, 2003.
- [33] "Reference guide: R700 family instruction set architecture," http://developer.amd.com/gpu_assets/R700-Family_Instruction_Set_Architecture.pdf, 2009.
- [34] "NVIDIA compute PTX: Parallel thread execution," *NVIDIA Corporation*, 2007.
- [35] L. Dagum, "OpenMP: A proposed industry standard API for shared memory programming," citeseer.ist.psu.edu/476450.html, October 1997.
- [36] "NVIDIA CUDA compute unified device architecture programming guide," *NVIDIA Corporation*, 2007.