Thread Migration Prediction for Distributed Shared Caches

Keun Sup Shim*, Mieszko Lis*, Omer Khan‡, Srinivas Devadas*
*Massachusetts Institute of Technology, Cambridge, MA
‡University of Connecticut, Storrs, CT

Abstract—Chip-multiprocessors (CMPs) have become the mainstream parallel architecture in recent years; for scalability reasons, designs with high core counts tend towards tiled CMPs with physically distributed shared caches. This naturally leads to a Non-Uniform Cache Access (NUCA) design, where on-chip access latencies depend on the physical distances between requesting cores and home cores where the data is cached. Improving data locality is thus key to performance, and several studies have addressed this problem using data replication and data migration.

In this paper, we consider another mechanism, hardware-level thread migration. This approach, we argue, can better exploit shared data locality for NUCA designs by effectively replacing multiple round-trip remote cache accesses with a smaller number of migrations. High migration costs, however, make it crucial to use thread migrations judiciously; we therefore propose a novel, on-line prediction scheme which decides whether to perform a remote access (as in traditional NUCA designs) or to perform a thread migration at the instruction level. For a set of parallel benchmarks, our thread migration predictor improves the performance by 24% on average over the shared-NUCA design that only uses remote accesses.

Index Terms—Parallel Architecture, Distributed Caches, Cache Coherence, Data Locality

1 Background

In recent years, transistor density has continued to grow and Chip Multiprocessors (CMPs) with four or more cores on a single chip have become common. To efficiently use the available transistors, architects are resorting to large-scale multicores both in academia (e.g., TRIPS [11]) and industry (e.g., Tilera [1]); pundits predict 1000+ cores in a few years [2]. For such massive multicores, a tiled architecture where each core has its own cache slice has become a popular design. These physically distributed cache slices can form one logically shared cache, known as Non-Uniform Cache Access (NUCA) architecture [7], [5]. In the "pure" form of NUCA where per-core caches are fully shared, each cache line corresponds to a unique core where it can be kept on chip, which maximizes effective on-chip cache capacity and reducing off-chip access rates. Furthermore, because only a single core can have a copy, there is no need for a cache coherence protocol. Private caches must rely on a coherence protocol to be coherent; these mechanisms not only incur large area costs but may also degrade performance when repeated cache evictions and invalidations are required for replicated shared data.

The downside of NUCA is high on-chip access latency, since every access to an address cached on a remote core must travel there. Various NUCA and hybrid NUCA/directory-coherence designs have therefore attempted to improve data locality, leveraging data migration and replication techniques previously explored in the NUMA context (e.g., [12]). These techniques assign private data to its owner core and replicate shared data among the sharers at OS level [6] or with hardware aid [3]. While these schemes improve performance on some kinds of data, they still do not take full advantage of spatio-temporal locality and rely on two-message round trips to access read/write shared data cached on remote cores.

To address this limitation and take advantage of available data locality in a NUCA memory hierarchy, we turn to *fine-grained hardware-level thread migration* [4], [8]. In this approach, accesses to data cached at a remote core cause the executing thread to migrate to that core and continue execution there. When several consecutive accesses are made to data at the same core, migration allows those accesses to be local to that core, potentially significantly improving performance. Figure 1 shows the breakdown of non-local memory accesses by the consecutive access count at the same non-local core before accessing another core in our set of benchmarks: in some cases (e.g., *radix*), memory accesses comprise long stretches at the same remote core, suggesting opportunities for a performance boost via thread migration. Due to the high cost of thread migration, however, it is crucial to migrate only when *multiple* remote word round-trip accesses would be replaced to make the cost "worth it." In this paper, we present a novel, program counter-based migration prediction scheme which decides at instruction granularity whether to perform a remote access or a thread migration; through simulations, we show that migrations can complement remote accesses to improve performance of baseline NUCA designs with our migration predictor.

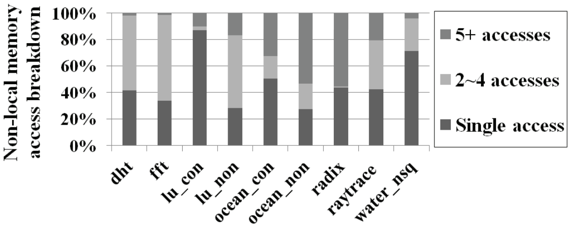

Fig. 1. Non-local memory access breakdown in the remote-access-only NUCA baseline

2 Memory Access Framework

NUCA architectures divide the address space among the cores in such a way that each address is assigned to a unique *home core* where the corresponding data can be cached [7], [5]. To read and write data cached in a remote core, the NUCA architectures proposed so far use a *remote access* mechanism where a request is sent to the home core and the resulting data (or acknowledgement) is sent back to the requesting core. In what follows, we describe this remote access protocol, as well as a protocol based on *hardware-level thread migration*. We then compare the two mechanisms and present a framework that combines both.

2.1 Remote Cache Access

Under the remote-access framework of standard NUCA designs [7], [5], all non-local memory accesses cause a request to be transmitted over the interconnect network, the access to be performed in the remote core, and the data (for loads) or acknowledgement (for writes) to be sent back to the requesting core. When a core C executes a memory access for address A, it must first find the *home* core H for A (e.g., by consulting a mapping table or masking some address bits). If $H = C$ (a *core hit*), the request is served locally at C. If $H \neq C$ (a *core miss*), on the other hand, a remote access request needs to be forwarded to core H, which will send a response back to C upon its completion. Note that, unlike a private cache organization where a coherence protocol (e.g., directory-based protocol) takes advantage

- *Manuscript submitted: 14-May-2012. Manuscript accepted: 24-Aug-2012. Final manuscript received: 3-Sep-2012.*

Posted to the IEEE & CSDL on 9/25/2012
DOI 10.1109/L-CA.2012.30

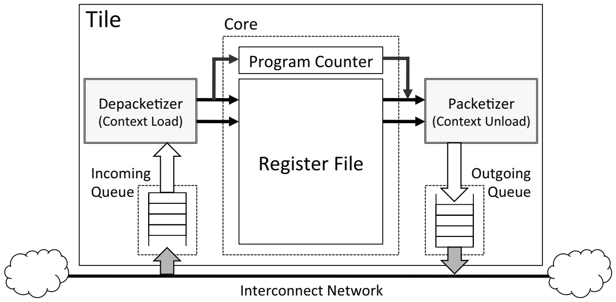

Fig. 2. Hardware-level thread migration via the on-chip interconnect

of spatial and temporal locality by making a copy of the block containing the data in the local cache, this protocol incurs round-trip costs *for every remote word access*.

2.2 Thread Migration

Fine-grained, hardware-level thread migration has been proposed to exploit data locality for NUCA architectures [8]. This mechanism brings the *thread* to the locus of the data instead of the other way around. When a core C running thread T executes a memory access for address A, it must first find the *home* core H for A. If $H = C$ (a *core hit*), the request is served locally at C. If $H \neq C$ (a *core miss*), the hardware interrupts the execution of the thread on C, packs the thread's execution context (microarchitectural state) to a network packet (as shown in Figure 2), and sends it to H via the on-chip interconnect where the packet is loaded to the context and an execution of T is resumed. This provides faster migrations than other approaches (such as OS-level migration or Thread Motion [10], which leverages the existing cache coherence protocol to migrate threads) since it migrates threads directly over the interconnect.

If a thread is already executing at the destination core, it must be evicted and moved to a core where it can continue running. To reduce the need for evictions, cores duplicate the architectural context (register file, etc.) and allow a core to multiplex execution among two (or more) concurrent threads. To prevent deadlock, one context is marked as the *native context* and the other as the *guest context*: a core's native context may only hold the thread that started execution there (called the thread's *native core*), and evicted threads must return to their native cores to ensure deadlock freedom [4].

2.3 Performance Overhead of Thread Migration

Since the thread context is directly sent across the network, the performance overhead of thread migration is directly affected by the context size. The relevant architectural state that must be migrated in a 64-bit x86 processor amounts to about 3.1Kbits (sixteen 64-bit general-purpose registers, sixteen 128-bit floating-point registers and special purpose registers), which is what we use in this paper[1]. This introduces a *serialization* latency since the full context needs to be loaded (unloaded) into (from) the network: with 128-bit flit network and 3.1Kbits context size, this becomes $\left\lceil \frac{pkt\,size}{flit\,size} \right\rceil = 26$ cycles[2].

Another overhead is the *pipeline insertion* latency. Since a memory address is computed at the end of the execute stage, if a thread ends up migrating to another core and re-executes from the beginning of the pipeline, it needs to *refill* the pipeline. In case of a five-stage pipeline core (cf. Figure 3), this results in an overhead of three cycles.

To make fair performance comparisons, all these migration overheads are included as part of memory latency for architectures that use thread migrations, and their values are specified in Table 4.1.

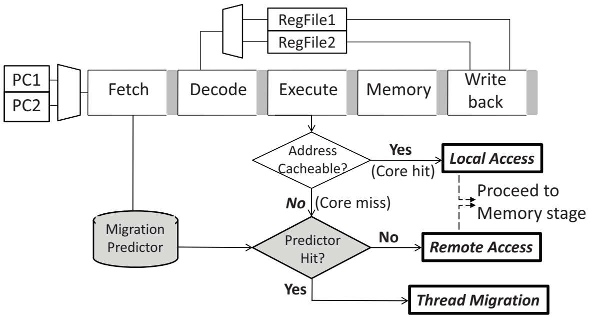

Fig. 3. Hybrid memory access architecture on a 5-stage pipeline core. The architectural context (*RegFile2* and *PC2*) is duplicated to support deadlock-free thread migration (cf. Section 2.2). The shaded modules are the components of migration predictor.

2.4 Hybrid Framework

While migrating the thread context can potentially better exploit spatiotemporal locality, for "one-off" remote accesses thread migration costs more than remote-access-only NUCA due to the large thread context size. We therefore propose a hybrid memory access framework for NUCA architectures which combines the two mechanisms described: each core-miss memory access may either perform the access via a remote access or migrate the current execution thread. Figure 3 illustrates the hybrid architecture: for each access to memory cached on a remote core, a decision algorithm determines whether the thread should migrate to the target core or execute a remote access. Considering the thread migration cost, when a thread is migrated to another core, it must make several *local* memory accesses there to make the migration "worth it"; our approach is to predict such long sequences of accesses to the same core and migrate only for those.

3 Thread Migration Prediction

As described in Section 2, it is crucial for the hybrid memory access architecture to make a careful decision whether to migrate the thread or perform a remote access. To this end, we will describe a per-core *migration predictor*—a PC-indexed direct-mapped data structure where each entry simply stores a PC. The predictor is based on the observation that sequences of consecutive memory accesses to the same home core are highly correlated with the program flow, and that these patterns are fairly consistent and repetitive across program execution. Our baseline configuration uses 128 entries; with a 64-bit PC, this amounts to about 1KB total per core. If the home core is not the core where the thread is currently running (a *core miss*), the predictor must decide between a remote access and a thread migration: if the PC hits in the predictor, it instructs a thread to migrate; if it misses, a remote access is performed.

In the next section, we describe how a certain instruction (or PC) is detected as migratory and thus inserted into the migration predictor.

3.1 Detection of Migratory Instructions

At a high level, the prediction mechanism operates as follows:

1) when a program first starts execution, it runs with a standard NUCA organization which only uses remote accesses;
2) as it continues execution, it monitors the home core information for each memory access, and remembers the first instruction of every multiple access sequence to the same home core;
3) depending on the length of the sequence, the instruction address is either inserted into the predictor (*migratory*) or is removed from the predictor (*remote-access*), if it exists;
4) the next time a thread executes the instruction, it migrates to the home core if it is a migratory instruction (a "hit" in the predictor), and performs a remote access if it is a remote-access instruction (a "miss" in the predictor).

[1] The context size will vary depending on the architecture; in the TILEPro64 [1], for example, it amounts to about 2.2Kbits (64 32-bit registers and a few special registers).

[2] With a 64-bit register file with two read ports and two write ports, one 128-bit flit can be read/written in one cycle and thus, we assume no additional serialization latency due to the lack of ports from/to the thread context.

Memory Instruction		Present State			Next State			Action
PC	Home	Home	Depth	Start PC	Home	Depth	Start PC	
I_1 : PC_1	A	-	-	-	A	1	PC_1	Reset the entry for a new sequence starting from PC_1
I_2 : PC_2	B	A	1	PC_1	B	1	PC_2	Reset the entry for a new sequence starting from PC_2 (evict PC_1 from the predictor, if exists)
I_3 : PC_3	C	B	1	PC_2	C	1	PC_3	Reset the entry for a new sequence starting from PC_3 (evict PC_2 from the predictor, if exists)
I_4 : PC_4	C	C	1	PC_3	C	2	PC_3	Increment the depth by one ***Insert PC_3 into the migration predictor***
I_5 : PC_5	C	C	2	PC_3	C	2	PC_3	Do nothing (PC_3 is already inserted)
I_6 : PC_6	C	C	2	PC_3	C	2	PC_3	Do nothing (PC_3 is already inserted)
I_7 : PC_7	A	C	2	PC_3	A	1	PC_7	Reset the entry for a new sequence starting from PC_7

Fig. 4. An example of how instructions (or PC's) which are followed by consecutive accesses to the same home location, i.e., *migratory instructions* are detected in the case of the depth threshold $\theta = 2$.

The detection of *migratory instructions* can be easily done by tracking how many consecutive accesses to the same remote core have been made, and if this exceeds a threshold, inserting the PC into the predictor to trigger migration. If it does not exceed the threshold, the instruction is classified as a remote-access instruction, which is the default state. Each thread tracks (1) *Home*, which maintains the home core ID for the current requested memory address, (2) *Depth*, which indicates how many times so far a thread has contiguously accessed the current home location (i.e., the *Home* field), and (3) *Start PC*, which tracks the PC of the very first instruction among memory sequences that accessed the home location in the *Home* field. We separately define the depth threshold θ, which indicates the depth at which we determine the instruction as migratory.

The detection mechanism is as follows: when a thread T executes a memory instruction for address A whose PC = P, it must first find the home core H for A; then,

1) if $Home = H$ (i.e., memory access to the same home core as that of the previous memory access),
    a) if $Depth < \theta$,
        i) increment $Depth$ by one; then if $Depth = \theta$, $StartPC$ is regarded as a migratory instruction and thus, is inserted into the migration predictor;.
2) if $Home \neq H$ (i.e., a new sequence starts with a new home core),
    a) if $Depth < \theta$,
        i) $StartPC$ is regarded as a remote-access instruction;
    b) reset the entry (i.e., $Home = H$, $PC = P$, $Depth = 1$).

Figure 4 shows an example of the detection mechanism when $\theta = 2$. Suppose a thread executes a sequence of memory instructions, $I_1 \sim I_7$ (non-memory instructions are ignored in this example because they do not change the entry content nor affect the mechanism). The PC of each instruction from I_1 to I_7 is PC_1, PC_2, ... PC_7, respectively, and the home core for the memory address that each instruction accesses is specified next to each PC. When I_1 is first executed, the entry {Home, Depth, Start PC} will hold the value of {A, 1, PC_1}. Then, when I_2 is executed, since the home core of I_2 (B) is different from *Home* which maintains the home core of the previous instruction I_1 (A), the entry is reset with the information of I_2. Since the *Depth* to core A has not reached the depth threshold, PC_1 is regarded as a remote-access instruction (default). The same thing happens for I_3. When I_4 is executed, it accesses the same home core C and thus the *Depth* field is incremented by one; since the *Depth* to core C has reached the threshold $\theta = 2$, PC_3 in *Start PC*, which represents the first instruction (I_3) that accessed C, is classified as a migratory instruction and thus is added to the migration predictor. For I_5 and I_6 which keep accessing the same home core C, we need not update the entry because the *Start PC* has already been added to the predictor. Lastly, when I_7 is executed, the predictor resets the entry and starts a new sequence starting from PC_7 for the home core A.

3.2 Possible Thrashing in the Migration Predictor

Since we use a fixed size data structure for our migration predictor, collisions between different migratory PCs can result in suboptimal performance; the size of the predictor can be increased to mitigate such collisions. Another subtlety is that mispredictions may occur if memory access patterns for the same PC differ across two threads (one native thread and one guest thread) running on the same core simultaneously because they share the same per-core predictor and may override each other's decisions. This interference can be resolved by implementing two predictors instead of one per core — one for the native context and the other for the guest context.

In our set of benchmarks, we rarely observed performance degradation due to these collisions and mispredictions with a fairly small predictor (about 1KB per core) shared by both native and guest context. This is because each worker thread executes very similar instructions (although on different data) and thus, the detected migratory instructions for threads are very similar. While such application behavior may keep the predictor simple, however, our migration predictor is not restricted to such applications and can be extended if necessary as described above. It is important to note that even if a rare misprediction occurs due to either predictor eviction or interference between threads, the memory access will still be carried out correctly, and the functional correctness of the program is still maintained.

4 Evaluation

4.1 Simulation Framework

We use Graphite [9] to model the proposed NUCA architecture that supports both remote-access and thread migration. The default system parameters are summarized in Table 4.1. Our experiments use a distributed hash table benchmark (*dht*) and a set of *Splash-2* [13] benchmarks. For each simulation run, we measured the *core miss rate*, the number of *core-miss* memory accesses divided by the total number of memory accesses. Since each core-miss memory access must be handled either by remote access or by thread migration, the core miss rate can further be broken down into *remote access rate* and *migration rate*. For the baseline, remote-access-only NUCA, the core miss rate equals to the remote access rate (i.e., no migrations); for the hybrid NUCA, the core miss rate is the *sum* of the remote access rate and the migration rate. For performance, we measured the parallel completion time, i.e., the longest completion time in the parallel region; this includes migration overheads (cf. Section 2.3) for our hybrid NUCA architecture.

4.2 Performance

We first compare the core miss rates for a NUCA system without and with thread migration: the results are shown in Figure 5. The depth

TABLE 1
System configurations used

Parameter	Settings
Cores	64 in-order, 5-stage pipeline, single-issue cores, 2-way fine-grain multithreading
L1/L2 cache per core	32/128 KB, 2/4-way set associative, 64B cache block
Electrical network	2D Mesh, XY routing, 3 cycles per hop, 128b flits
Migration Overhead	3.1 Kbits full execution context size, Full context load/unload latency: $\left\lceil \frac{pkt\,size}{flit\,size} \right\rceil$ = 26 cycles Pipeline insertion latency for context load = 3 cycles
Data Placement	First-touch after initialization, 4 KB page size

threshold θ is set to 3 for our hybrid NUCA, which aims to perform remote accesses for memory sequences with one or two accesses and migrations for those with ≥ 3 accesses to the same core. While 38% of total memory accesses result in *core misses* for remote-access-only NUCA on average, NUCA with our migration predictor results in a core miss rate of 25%, a 35% improvement in data locality.

Figure 5 also shows the fraction of *core miss* accesses handled by remote accesses and thread migrations in our hybrid NUCA scheme. We observe that a large fraction of remote accesses are successfully replaced with a much smaller number of migrations. *Ocean_non_contiguous*, for example, originally showed a 86% remote access rate under a remote-access-only NUCA; with a small number of migrations, however, core miss rates drop to 45%. Across all benchmarks, the average migration rate is only 3% resulting in 35% fewer core misses overall. This improvement of data locality directly relates to better performance for NUCA with thread migration as shown in Figure 6. For our set of benchmarks, our hybrid NUCA shows 24% better performance on average (geometric mean) across all benchmarks; it performs no worse than the baseline NUCA, except for *raytrace* where migrations did not reduce the core miss rate while introducing the migration overhead.

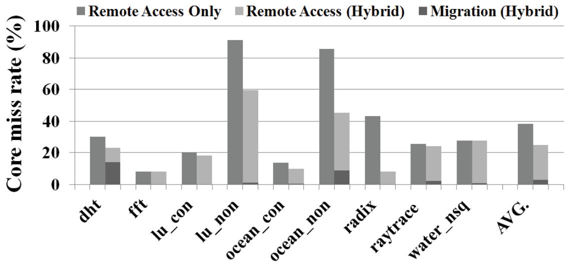

Fig. 5. Core miss rate and its breakdown into remote access rate and migration rate

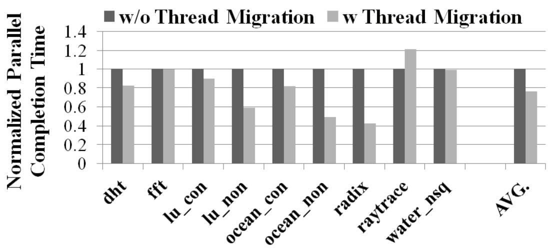

Fig. 6. Parallel completion time under our hybrid NUCA with θ = 3 normalized to the baseline remote-access-only NUCA.

4.3 Effects of the Depth Threshold

We change the value of the depth threshold θ = 2, 3 and 5 and explore how the fraction of core-miss accesses being handled by remote-accesses and migrations changes. As shown in Figure 7, the ratio of remote-accesses to migrations increases with larger θ. The average performance improvement over the remote-access-only NUCA is 15%, 24% and 20% for the case of θ = 2, 3 and 5, respectively (not shown in the paper). The reason why θ = 2 performs worse than θ = 3 with almost the same core miss rate is because of its higher migration rate; due to the large thread context size, the cost of a single thread migration is much higher than that of a single remote access and needs, on average, a higher depth to achieve better performance.

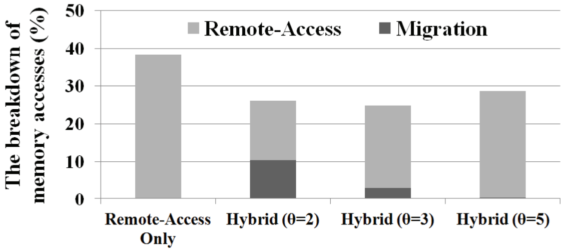

Fig. 7. The fraction of remote-accesses and migrations for the standard NUCA and hybrid NUCAs with the different depth thresholds.

5 Conclusions

In this manuscript, we have presented an on-line, PC-based thread migration predictor for memory access in distributed shared caches. Our results show that migrating threads for sequences of multiple accesses to the same core can improve data locality in NUCA architectures, and with our predictor, it can result in better overall performance compared to the baseline NUCA designs which only rely on remote-accesses.

References

[1] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - processor: A 64-Core SoC with mesh interconnect. In *ISSCC*, pages 88–598, 2008.

[2] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of DAC*, pages 746–749, 2007.

[3] M. Chaudhuri. PageNUCA: Selected policies for page-grain locality management in large shared chip-multiprocessor caches. In *HPCA*, 2009.

[4] Myong Hyon Cho, Keun Sup Shim, Mieszko Lis, Omer Khan, and Srinivas Devadas. Deadlock-free fine-grained thread migration. In *Proceedings of NOCS*, 2011.

[5] C. Fensch and M. Cintra. An OS-based alternative to full hardware coherence on tiled CMPs. In *HPCA*, 2008.

[6] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *ISCA*, 2009.

[7] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *ASPLOS*, 2002.

[8] Mieszko Lis, Keun Sup Shim, Myong Hyon Cho, Omer Khan, and Srinivas Devadas. Directoryless shared memory coherence using execution migration. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing*, 2011.

[9] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *Proceedings of HPCA*, pages 1–12, 2010.

[10] Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. Thread motion: Fine-grained power management for multi-core systems. In *Proceedings of ISCA*, 2009.

[11] K. Sankaralingam, R. Nagarajan, H. Liu, J. Huh, C. K. Kim, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP using polymorphism in the TRIPS architecture. In *ISCA*, June 2003.

[12] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. *SIGPLAN Not.*, 31:279–289, 1996.

[13] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of ISCA*, 1995.