ARCc: A Case for an Architecturally Redundant Cache-coherence Architecture for Large Multicores

Omer Khan[1,2], Henry Hoffmann[2], Mieszko Lis[2], Farrukh Hijaz[1], Anant Agarwal[2], Srinivas Devadas[2]
[1]University of Massachusetts, Lowell, MA, USA
[2]Massachusetts Institute of Technology, Cambridge, MA, USA

***Abstract*—This paper proposes an architecturally redundant cache-coherence architecture (ARCc) that combines the directory and shared-NUCA based coherence protocols to improve performance, energy and dependability. Both coherence mechanisms co-exist in the hardware and ARCc enables seamless transition between the two protocols. We present an online analytical model implemented in the hardware that predicts performance and triggers a transition between the two coherence protocols at application-level granularity. The ARCc architecture delivers up to 1.6× higher performance and up to 1.5× lower energy consumption compared to the directory-based counterpart. It does so by identifying applications which benefit from the large shared cache capacity of shared-NUCA because of lower off-chip accesses, or where remote-cache word accesses are efficient.**

I. Introduction

Four to eight general-purpose cores on a die are now common across the spectrum of computing machinery [1], and designs with many more cores are not far behind [2], [3]. Pundits confidently predict thousands of cores by the end of the decade [4]. The biggest challenge facing large-scale multicores is convenience of programming. Today the shared-memory abstraction is ubiquitous and some form of cache coherence is required to keep a consistent view of data among cores. Software can provide cache coherence, but at the cost of programming complexity and loss of performance because of limited observability and controllability into the hardware. Therefore, multiprocessors and most recently single-chip multicores support a uniform hardware-coherent address space. In large-scale multicores lacking common buses, this usually takes the form of directory-based coherence hardware.

Traditional directory-based cache coherence faces significant challenges in the era of large-scale multicores; the most significant one being the scalability challenge due to the *off-chip memory wall* [4]. Today's multicores integrate very large on-chip caches to reduce the pressure of off-chip memory accesses and improve data locality [1]. Directory-based coherence requires a logically centralized directory (typically distributed physically) that coordinates sharing among the per-core private caches, and each core-private cache must negotiate shared or exclusive access to each cache line via a complex coherence protocol. On-chip directories must equal a significant portion of the combined size of the per-core caches, as otherwise directory evictions will stress off-chip memory and limit performance [5].

This work was funded by the U.S. Government under the DARPA UHPC program. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

In addition to the complexity of the coherence protocol, the performance of the directory-based coherence architecture is impacted in the following ways: (i) the directory causes an *indirection* leading to an increase in the cache miss access latency for both producer and the consumer, (ii) the *automatic data replication* of shared read data results in one address being stored in many core-local caches, reducing the amount of cache left for other data, thereby adversely impacting cache miss rates, and (iii) a write to shared data or an on-chip directory cache eviction requires *invalidation* of all shared copies of the data, resulting in higher cache miss rates and protocol latencies.

These shortcomings have been partially addressed by shared-NUCA [6]; here, we consider a variant Distributed Shared Cache (DSC) based architecture that ensures cache coherence. (Note that DSC is similar to [7] and Tilera's TILE64 processor, but does not require operating system (OS) or software support for coherence). The DSC architecture unifies the per-core physically distributed caches into one large logically shared cache, in its pure form keeping only one copy of a given cache line on chip and thus steeply reducing off-chip access rates compared to the directory-based coherence.

Under DSC, when a thread needs to access an address cached on another core, the requesting core initiates a remote-cache access to the core where the memory is allowed to be cached. A two-message round trip via the on-chip interconnect ensures any memory load or store can be successfully executed for the data cached on a remote core. This offers a tradeoff: where a directory-based protocol would take advantage of spatial and temporal locality by making a copy of the block containing the data in the local cache, DSC must repeat the round trip for every remote access to ensure sequential consistency; on the other hand, a round trip 2-message protocol is much cheaper (in terms of latency) than a 3-party communication of the directory-based protocol. However, the performance of DSC is primarily constrained by the placement of data; if the requested data is not locally cached, on-chip network latencies add to the cost of accessing remote memory. Various S-NUCA proposals have therefore leveraged data migration and replication techniques previously explored in the NUMA context (e.g., [8]) to move private data to its owner core and replicate read-only shared data among the sharers [9], [10]; but while these schemes improve performance on some kinds of applications, they still do not take full advantage of spatio-temporal locality and may require directory coherence to deliver the desired performance.

In this paper, we propose an architecturally redundant

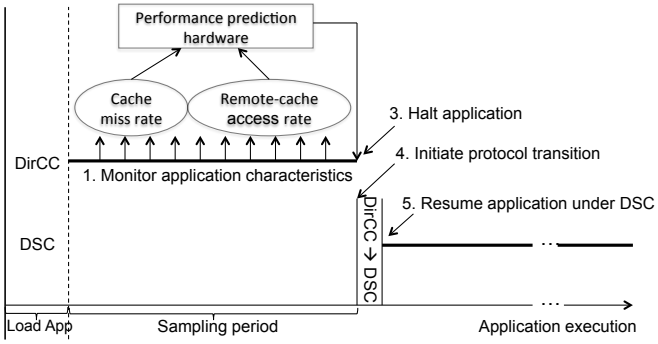

Fig. 1. Key components of the ARCc Architecture.

cache-coherence architecture (ARCc) that combines the performance advantages of both directory and DSC based coherence protocols by enabling these two independent and heterogenous protocols to co-exist in the hardware. We present architectural extensions to seamlessly transition between directory and DSC protocols at application granularity. Figure 1 shows the key components and mechanisms in the proposed ARCc architecture. The x-axis shows the execution of a multithreaded application on a multicore. Initially, the application is initialized under the directory (DirCC) coherence protocol. After initialization, a few application characteristics are monitored at runtime for a short duration of time (the sampling period in Figure 1). ARCc deploys an in-hardware analytical model to concurrently predict the performance of the DirCC and DSC protocols at application-level granularity. The ARCc architecture delivers higher performance than either DirCC or DSC protocols, because it trades off the impact of higher cache miss rate under DirCC with lower cache miss rate coupled with the remote-cache access rate under DSC. When the online analytical model indicates a performance gain of using DSC over DirCC, the hardware automatically transitions from DirCC to DSC and continues execution. This process is repeated whenever the operating system schedules a new application to run on the multicore.

Simulations of a 128-core single-chip multicore show that, depending on the application and on-chip cache capacity, the ARCc architecture can significantly improve the overall application performance (up to 1.6×) and energy consumption (up to 1.5×) when compared to DirCC. Even better, ARCc allows *redundant* coherence protocols to co-exist (that share minimum hardware resources), and therefore enables a more dependable architecture that guarantees functional correctness and performance in the presence of alarming failures rates of the future CMOS technologies [11].

II. Background and Motivation

One of the key challenges for large-scale multicores is to preserve a convenient programming model. The shared memory abstraction is now ubiquitous but requires some form of cache coherence. Because energy-performance efficiency is a first-order challenge, we expect a multicore chip to be fully distributed across tiles with a uniform address space shared by all tiles (similar to the Tilera TileGx100 [12]). In our baseline architecture, each tile in the chip communicates with others via an on-chip network. Such physical distribution allows the system layers to manage the hardware resources efficiently.

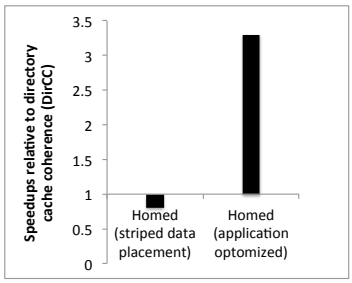

Fig. 2. DSC vs. DirCC performance of 256x256 2D FFT application using the 64-core Tilera's TILEPro64 Processor.

Each tile has an Intel Atom processor-like core with a 2-level L1/L2 instruction and data cache hierarchy. Under private-L1, a directory protocol is responsible for coherence. Although a shared-L2 caters to applications with large shared working sets, many traditional multithreaded applications require larger private working sets, making private-L2 an attractive option [10]. Therefore, our baseline architecture utilizes a private-L1/L2 for the DirCC configuration. On the other hand, a shared-L1/L2 configuration unifies the physically distributed per-core caches into one large logically shared cache. Because only one copy of a cache line can be present on chip, cache coherence is trivially ensured and a directory protocol is not needed. The shared-L1/L2 organization is termed as the Distributed Shared Cache (DSC) architecture. Because multicores are expected to be critically constrained by limited package pin density, we expect a small number of on-chip memory controllers orchestrating data movement in and out of the chip, therefore, limiting off-chip memory bandwidth [4].

To motivate the proposed ARCc architecture that combines the performance benefits of directory and DSC protocols under a unified architecture, we ran a 256x256 2D FFT application using Tilera's TILEPro64 processor. We ran three different setups. First is cache coherence with directories (DirCC), where the home directories are distributed with an OS hash function. Second, homed memory (DSC) where each cache line has a home cache (determined by the OS with a striped hash function based data placement), where everyone can read from the home cache (single-word reads), but the data can only be cached in the home. Third is an optimization for the homed memory DSC architecture (remote store programming (RSP) details in [13]), where homes are determined by the application such that only word-level remote writes are allowed. RSP guarantees all loads to be local and minimum latency.

Figure 2 shows a non-optimized DSC architecture (naive data placement statically assigned using striping, resulting in high remote-cache access rate) that performs 1.23× worse than the directory protocol for the FFT application. After application-level modifications (as in RSP [13]), DSC is shown to outperform DirCC for this application by 3.3×. This shows that DSC is highly dependent on the application and it can yield significant performance gains compared to DirCC. Our proposed ARCc architecture specifically exploits the performance benefits of DSC and DirCC protocols by intelligently choosing the higher performing coherence protocol at runtime.

In what follows, we first present the details of the ARCc architecture, our experimental methodology, and finally simulation-based evaluation of the ARCc architecture.

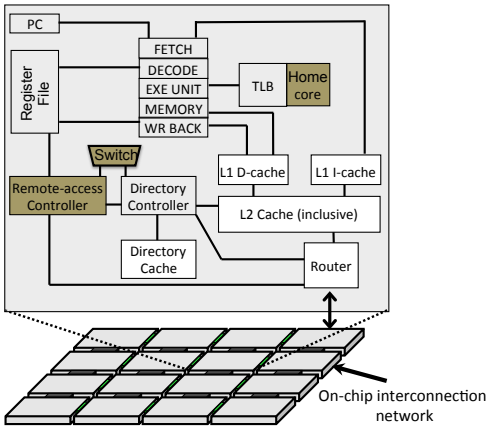

Fig. 3. The architecture of a single ARCc tile with support for directory and DSC cache coherence. The shaded blocks represent the components needed to enable DSC and ARCc in addition to DirCC.

III. Architecturally Redundant Cache-coherence

ARCc enables runtime transition between the two cache coherence protocols at application granularity, thereby enabling coarse-grain redundancy with the goal of optimizing user-level performance. In case one of the coherence mechanisms is expected to yield degraded performance (cf. Figure 1), it is disabled and the system transitioned to the alternate mechanism. Figure 3 shows a tile-level view of the proposed ARCc architecture. In addition to a directory controller and directory cache orchestrating coherence at cache line granularity, the DSC protocol requires a remote-access controller (RAC) in each tile to allow accessing cache resources in another tile via the interconnection network.

As shown in Figure 4, the system is composed of directories for tracking actively shared cache lines, translation lookaside buffers (TLB) with additional entry bits to track the *home core* for memory addresses (note that DSC operates on the OS-page granularity and each memory access looks up the TLB to extract the core where the cache line is allowed to be cached), and system registers to differentiate whether an application is running under DirCC or DSC. By default, each application is initialized to run under DirCC. However, regardless of the coherence protocol, the system always enables data placement for assigning a home core to each active page. During runtime, an application's progress is time-sliced into two phases: a sampling period followed by a steady-state period that lasts until the application executes. During each sampling period, the system uses in-hardware profiling to collect runtime parameters that are used to predict the performance of the DirCC and DSC protocols. At the end of the sampling period, if the application will substantially benefit from transitioning the coherence protocol, the system halts and initiates the transition mechanism and updates the system registers. It is important to note that the application executes normally during the sampling period and only halts if the coherence protocol transition is triggered.

For each memory instruction, the core first consults the system registers to discover the coherence protocol mode of operation. If *DSC enable* is set, the TLB access stage of the pipeline extracts the *home core* for that address. If the memory address' home core is on another tile, it is a *core miss*

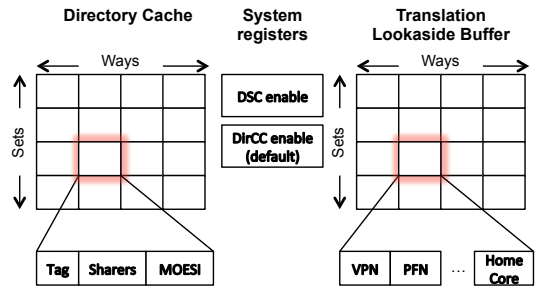

Fig. 4. The ARCc Architecture.

and a 2-message memory transaction is initiated to access the remote cache hierarchy, otherwise, the local cache hierarchy is accessed. If *DirCC enable* is set, the core simply initiates a cache lookup in the local cache hierarchy and possibly consults the directory to extract the requested data.

A. Protocol Transition Trigger Mechanism

The proposed framework for transitioning between DirCC and DSC consists of two components: the online profiler and the performance predictor. The online profiler concurrently and non-invasively profiles certain aspects of the execution characteristics during each sampling period. The performance predictor collects the profiled application characteristics at the end of the sampling period, and predicts whether to transition from DirCC to DSC for that application.

By default, the application is loaded and initialized under DirCC. The time axis is sub-divided into a sampling interval followed by a longer steady-state interval (cf. Figure 1). The online profilers are implemented in hardware to collect runtime statistics during the sampling interval to concurrently estimate the performance of the application under DirCC and DSC. At the end of each sampling period, the system hardware uses the following procedure to decide when/how to transition between the coherence protocols.

if (DSC performance > DirCC performance)
  Halt execution of the application code
  Transition coherence protocol from DirCC to DSC
  Update system registers to indicate "DSC enable"
  Continue execution of the application under DSC
else Continue execution of the application under DirCC

1) Performance Predictor: Estimating an application's performance on different cache coherence mechanisms based on the observed application characteristics is a key step to avoid expensive trial runs. To gain some intuition for where a coherence mechanism can win on performance, we consider the average memory latency (AML), a metric that dominates program execution times with today's fast cores and relatively slow memories.

Under the DSC remote-cache access architecture, AML has three components: cache access (for cache hits and misses), off-chip memory access (for cache misses), and a 2-message cost to access remote data (for core misses):

$$\begin{aligned} AML_{DSC} = {} & cost_{\$access,DSC} + \\ & rate_{\$miss,DSC} \times cost_{\$miss,DSC} + \\ & rate_{core_miss} \times cost_{remote_cache_access} \end{aligned} \quad (1)$$

While $cost_{\$access,DSC}$ mostly depends on the cache technology

itself, DSC improves performance by optimizing the other variables: DSC is expected to significantly lower $rate_{\$miss,DSC}$ when compared to DirCC, and therefore, its AML primarily depends on $rate_{core_miss}$. $cost_{remote_cache_access}$ depends on the distance of the remote core from the requesting core; we estimate an average under a particular interconnect technology. $rate_{core_miss}$ is application dependent and must be measured using an online profiler.

Under the directory-based architecture, AML has two key components: cache access (for cache hits and misses), and cache misses (including on-chip protocol costs, cache-to-cache transfers and/or off-chip accesses).

$$AML_{DirCC} = cost_{\$access,DirCC} + rate_{\$miss,DirCC} \times cost_{\$miss,DirCC} \quad (2)$$

While $cost_{\$access,DirCC}$ mostly depends on the cache technology itself, DirCC improves performance by optimizing the $rate_{\$miss,DirCC}$. Both $cost_{\$miss,DirCC}$ and $rate_{\$miss,DirCC}$ are application dependent and must be measured using an online profiler.

We propose to use relative comparisons between equations 1 and 2 to predict when it is beneficial for an application to switch from DirCC to DSC. Transitioning between the two coherence protocols can be expensive, therefore, we only allow a transition when our proposed online predictor shows a performance advantage that amortizes the cost of protocol transition. For this paper, we empirically fixed the transition criterion to allow a protocol transition when predicted performance advantage is in excess of 5%.

Four critical parameters need to be profiled at runtime or otherwise estimated to accurately make a prediction about switching coherence protocols at the application level: (a) average round trip latency of remote-cache access, (b) $rate_{core_miss}$, (c) average cache miss latency under DirCC, and (d) $rate_{\$miss,DirCC}$.

2) Online Profilers: Because the ARCc architecture always starts an application under DirCC and may transition the coherence protocol based on the performance predictor, the average cache miss latency under DirCC, and $rate_{\$miss,DirCC}$ can be accurately profiled at runtime. The TLB contains the home core information for each memory access, therefore for DSC $rate_{core_miss}$ can also be precisely measured for every memory access. Since the round trip latency of a remote-cache access cannot be profiled in hardware, we estimate it using a 12×12 mesh network (we assume a 128 core multicore for this study) with three-cycle-per-hop 128-bit flit pipelined routers, and an average distance of 8 hops per transit; making the round trip network transit cost 48 cycles. We deploy hardware support to efficiently profile the average cache miss latency under DirCC, $rate_{\$miss,DirCC}$, and $rate_{core_miss}$.

Each core implements hardware support to instrument the following parameters during the sampling period: (a) number of memory accesses, (b) number of core misses (although core misses are actually not initiated unless the DSC protocol is activated, this parameter is profiled to predict the performance of DSC during the sampling period), and (c) accumulated latency of L2 misses. (a) is implemented using a 64-bit counter that is incremented on every memory access. Core misses are tracked when a memory access from a thread is to a home core that is different from the core this thread is running on. (b) is implemented using a 64-bit counter that is incremented on each core miss. Finally, (c) is implemented using a 64-bit register that is updated by accumulating the latency of each L2 miss. Because the core is stalled and waiting for the L2 miss data, this register is updated using the core's ALU. These profiling registers are reset every time a new thread or a new application is mapped to a core.

The computation cost of the performance prediction for DirCC and DSC is mainly attributed to converting the profiled data into parameters used to predict performance. We summarize the performance prediction calculation in the following equations:

$$\mathrm{DirCC}_{Perf} = \frac{\sum_{i=1}^{num_cores} accumulated\ latency\ of\ L2\ misses}{\sum_{i=0}^{num_cores} num\ of\ memory\ accesses}$$

$$\mathrm{DSC}_{Perf} = \frac{48\ cycles\ network\ transit \times \sum_{i=1}^{num_cores} num\ of\ core\ misses}{\sum_{i=0}^{num_cores} num\ of\ memory\ accesses}$$

Assuming a 128 core processor, this requires approximately 384 accumulate, 1 multiply, and 2 divide operations. Since the prediction is made only once every sampling interval, these operations can be performed on the functional units already present on the chip by stealing their idle slots. By starting the process of estimating performance several thousands of cycles before the end of the sampling period, the computation for performance prediction can be completely hidden and will not incur performance penalties.

B. Protocol Transition: DirCC → DSC

The final required component of the ARCc architecture is to enable coherence protocol transition from DirCC to DSC when the performance predictor indicates substantial performance gains of using DSC over DirCC. Note that this transition is initiated only once at the end of the sampling period for each application. During the protocol transition phase, the execution of the application is halted and the following procedure is initiated by the hardware to transition the cache coherence protocol.

A system wide message is sent to each core to flush all modified data to the *home core* for that data. Flushing the data to the home core has the advantage of avoiding unnecessary off-chip memory accesses. Each L2 cache initiates a "cache walk" procedure to scan all cache lines and check whether a cache line is in *modified* state. If a cache line is in modified state, a TLB lookup extracts the home core for that data and a cache-to-cache transfer is initiated to move the cache line to its home core. Note that on arrival at the home core's cache, if the cache line being replaced is itself in modified state, then depending on the home core for that cache line, it is either evicted to its home core or stored back to main memory. Evictions to the home core may cause a cascade effect, but the system will stabilize when all modified cache lines reach their home cores. Because this procedure is implemented in

hardware, it is transparent to the operating system and the application. At the end of the protocol transition, the system registers are updated to disable DirCC and enable DSC.

The protocol transition phase is the only performance penalty observed by the application and may need efficient implementation. Our estimates for a 128-core processor with 256KB L2 cache per core show that a cache walk with 10%-20% of the cache lines in modified state requires 50K to 100K cycles overhead for transitioning coherence protocols.

IV. Methods

We use Pin [14] and Graphite [15] to model the proposed ARCc architecture as well as the DirCC and DSC baselines. We implemented a tile-based multicore similar to Tilera's TILE-GX with 128 cores; various processor parameters are summarized in Table I. We swept the per-tile L2 cache sizes to characterize the performance and energy tradeoffs between the proposed ARCc, DirCC, and DSC architectures. On-chip directory caches (not needed for DSC) were set to sizes recommended by Graphite on basis of the total L2 cache capacity in the simulated system. For all experiments using ARCc, we fixed the sampling period to 5 million cycles. As discussed in Section III-B, we modeled the overheads associated with transitioning the coherence protocol from DirCC to DSC.

Our experiments used a set of SPLASH-2 benchmarks: FFT, LU_CONTIGUOUS, OCEAN_CONTIGUOUS, RADIX, RAYTRACE, and WATER-N^2. For the benchmarks for which versions optimized for directory coherence exist (LU and OCEAN [16]), we chose the versions that were most optimized for DirCC. Each application has 128 threads and was run to completion using the recommended input set for the number of cores used. For each simulation run, we tracked the completion time and cycles per instruction for each thread, the percentage of memory accesses causing cache hierarchy misses, and the percentage of memory accesses causing remote-cache accesses.

A. Energy estimation

For energy, we assume a 32 nm process technology and use CACTI [17] to estimate the dynamic energy consumption of the caches, routers, register files, and DRAM. The dynamic energy numbers used in this paper are summarized in Table II. We implemented several energy counters (for example the number of DRAM reads and writes) in our simulation framework to estimate the total energy consumption of running SPLASH-2 benchmarks for DirCC, DSC and ARCc. Note that DRAM only models the energy consumption of the RAM, and the I/O pads and pins will only add to the energy cost of going off-chip.

B. Data Placement for DSC

In standard S-NUCA architectures and our DSC variant, data placement is key, as it determines the frequency and distance of remote-cache accesses. Data placement has been studied extensively in the context of NUMA architectures (e.g., [8]) as well as more recently in the S-NUCA context (e.g., [10]), the operating system controls memory-to-core mapping via the existing virtual memory mechanism: when a virtual address is first mapped to a physical page, the OS chooses where the relevant page should be cached by mapping the virtual page to a physical address range assigned to a specific core. Since the OS knows which thread causes a page fault, more sophisticated heuristics are possible: for example, in a first-touch-style scheme, the OS can map the page to the thread's originating core, taking advantage of data access locality to reduce the remote-access rate while keeping the threads spread among cores. It is plausible to combine a first-touch data placement policy [18], which maps each page to the first core to access it, with judicious profiling-based placement and replication of read-only shared data [10], making DSC an attractive alternative for cache coherence. In this paper, we consider a first-touch style data placement scheme, and defer optimizations for data placement to future work.

Parameter	Settings
Cores	128 in-order, 5-stage pipeline, single-issue cores
L1 instruction/data cache per core	32/16 KB, 4/2-way set associative
L2 cache per core	{16, 32, 64, 128, 256} KB, 4-way set associative Cache line size = 64 bytes
Electrical network	2D Mesh, XY routing, 128b flits 2 cycles per hop (+ contention)
Data placement scheme	First-touch, 4 KB page size
Directory protocol	MOESI, Full-map physically distributed directories Entries per directory = {256, 512, 1K, 2K, 4K} 32-way set associative
Memory	30 GB/s bandwidth, 75ns latency

TABLE I
SYSTEM CONFIGURATIONS USED.

Component	#	Read energy (nJ/instance)	Write energy (nJ/instance)	Details
Register File	128	0.005	0.002	4-Rd, 4-Wr ports; 64x24 bits
Router	128	0.011	0.004	5-Rd, 5-Wr ports; 128x20 bits
Directory cache	128	{0.17, 0.18, 0.36, 0.73, 0.74}	{0.17, 0.18, 0.4, 0.88, 0.9}	{8, 16, 24, 40, 64} KB cache, 32-way associative
L2 cache	128	{0.03, 0.04, 0.08, 0.14, 0.28}	{0.02, 0.03, 0.07, 0.12, 0.28}	{16, 32, 64, 128, 256} KB cache, 4-way associative
L1 data cache	128	0.034	0.017	16 KB (2-way associative)
Off-chip DRAM	8	6.333	6.322	1 GB RAM

TABLE II
AREA AND ENERGY ESTIMATES.

C. Directory vs. Distributed Shared Cache (DSC) Coherence

Our baseline directory-based hardware cache coherence architecture (DirCC) configures the L1/L2 caches private to each tile and enables a sequentially consistent memory system. A coherence controller utilizes a directory-based MOESI protocol and all possible cache-to-cache transfers to manage coherence for the associated address space regions. The DirCC architecture implements a full-map physically distributed directory [19]. To keep track of data sharers and to minimize expensive off-chip accesses, we deploy an on-chip directory cache [5]. The directory cache is sized appropriately as a function of the number of L2 cache entries tracked by the coherence protocol. On a directory cache eviction, the entry with the lowest number of sharers is chosen and all sharers for that entry are invalidated.

The most performance sensitive aspect of DirCC is the sizing and organization of the *data and directory caches*. A directory cache conflict can result in evicting active cache lines, causing an increase in off-chip memory accesses due to data re-fetch. A key feature of DirCC is automatic replication

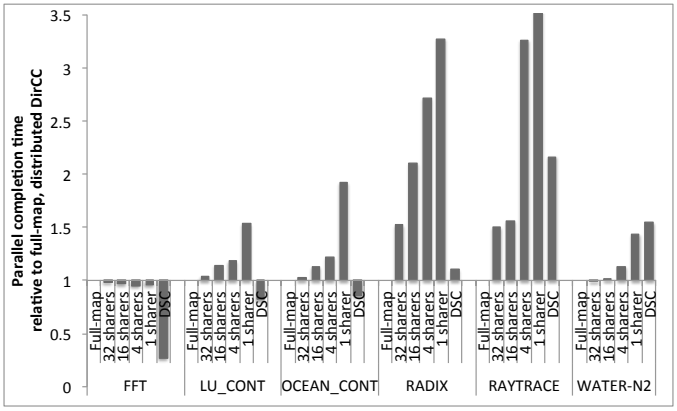

Fig. 5. Performance of full-map directory coherence is optimal compared to the limited hardware sharer variants; as maximum allowed hardware sharers per cache line are reduced the performance of DirCC drops because limiting the *S* state only spreads sharing and therefore cache pollution over time. On the other hand, DSC only allows a cache line to be cached in a single tile, therefore increasing the total available cache capacity and effectively dramatically reducing off-chip accesses.

of shared data that exploits temporal and spacial locality. On the flip side, replication decreases the effective total on-chip data cache size because, as the core counts grow, a lot of cache space is taken by replicas and fewer lines in total can be cached, which in turn can lead to an increase in off-chip memory access rates and lower performance.

Figure 5 characterizes the performance of our full-map directory coherence implementation for 64KB L2 cache per core. The data presented here is an average across all benchmarks. Although at first blush it may seem plausible that reducing the maximum number of allowed hardware sharers per cache line may yield similar performance as the DSC counterpart, in reality our results show that reducing the number of allowed hardware sharers only spreads the sharing and therefore cache pollution over time. On the other hand, DSC only allows a cache line to be cached in a single tile, therefore trading off cache capacity and policy with a communication protocol to access remote data, effectively dramatically reducing off-chip accesses. We observe that full-map DirCC and DSC enable the most interesting tradeoffs in application performance. In the next section, we show that our ARCc architecture detects and exploits the performance advantage of transitioning between DirCC to DSC protocols at application-level granularity.

V. Evaluation

A. Cache Hierarchy Misses vs. Remote-cache Accesses

Figure 6 illustrates how the DSC and DirCC coherence protocols differ in cache miss rates. Under DSC, which unifies the per-core caches into one large logically shared cache and does not replicate data in caches, cache miss rates not only start lower to begin with, but also deteriorate much more slowly as the cache capacity drops.

Although cache miss rates are a primary factor in determining overall application performance, non-uniform latency remote-cache accesses in DSC mean that DSC performance critically depends on how often and how far the remote access messages must travel. This factor is highly dependent on the application and the placement of its data in the per-core cache slices: core miss rate (i.e., the fraction of memory references

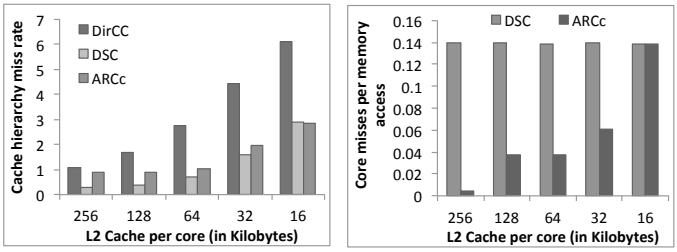

Fig. 6. On the one hand, DirCC is highly sensitive to cache miss rates, while DSC markedly lowers cache misses; on the other, DSC is highly dependent on data placement, which dictates remote-cache access rates (a.k.a core misses). The data presented in this figure is an average across all benchmarks.

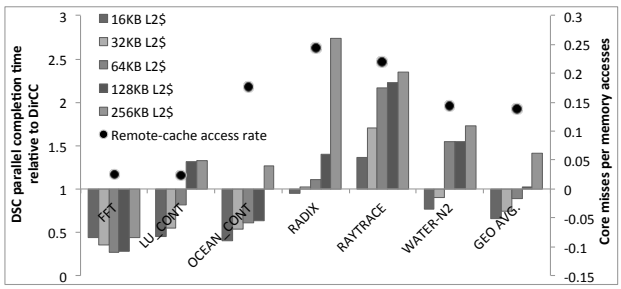

Fig. 7. DSC performance compared to DirCC for various L2 cache sizes.

that result in remote-cache accesses) for DSC varies from less than 3% in FFT to about 25% in RADIX, and we observe an average of 14% remote-cache accesses (see Figure 7).

The ARCc architecture intelligently chooses the DirCC or DSC coherence protocol to tradeoff the latency of the directory 3-party communication (of every cache hierarchy miss) with the DSC round trip latency (of each remote-cache access). Figure 6 shows that a 256KB L2 cache lowers the cache hierarchy miss rates of DirCC to under 1% and the cost of (on average 14%) remote-cache accesses for DSC exceeds the latencies associated with cache misses under DirCC. Therefore, ARCc only transitions from DirCC to DSC for a few select applications. As the per-core L2 cache size is reduced, the cache miss rates of DirCC increase sharply compared to DSC, but the remote-cache accesses under DSC remain relatively constant. This results in more applications choosing DSC as the coherence protocol under ARCc. Eventually, at 16KB L2 cache per core, we observe that most of the applications transition to DSC under ARCc as the benefits of reducing cache miss rates (and associated communication latencies) outweigh the latencies associated with the remote-cache accesses.

B. Performance Advantage of ARCc

The key contribution of the proposed ARCc architecture is its performance advantage over DirCC and DSC protocols. To understand where ARCc wins in performance, we evaluate the parallel completion time (PCT) of DSC relative to DirCC. Figure 7 shows the PCT for various per-core L2 cache configurations. Because DSC does not allow data replication and utilizes the available cache capacity more efficiently, the performance for DSC is on average superior to DirCC at smaller per-core L2 cache configurations. The remote-cache accesses (a.k.a core misses) primarily dictate the memory latency and performance under DSC. Results show that this factor is highly dependent on the application and the data placement in the per-core caches; applications with higher

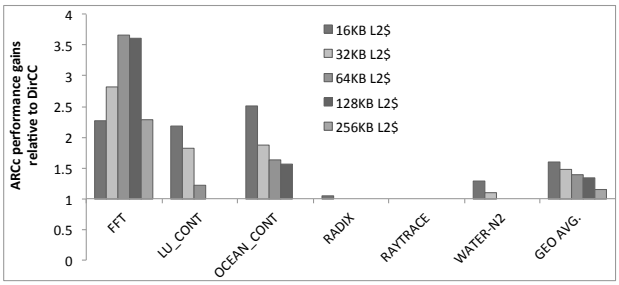

Fig. 8. ARCc performance compared to DirCC for various L2 cache sizes.

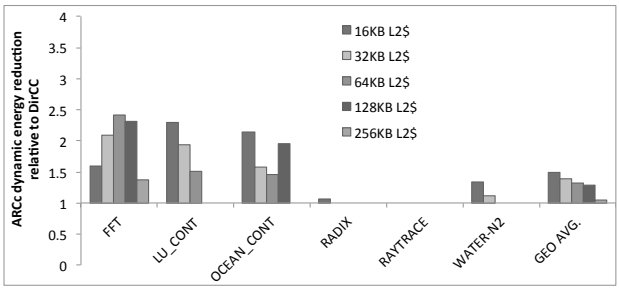

Fig. 9. ARCc energy compared to DirCC for various L2 cache sizes.

core miss rates tend to perform worse under DSC, specifically when the corresponding cache hierarchy miss rates are low under DirCC. For example, the average cache hierarchy miss rate for DirCC under 16KB per-core L2 cache is 6% and most applications tend to perform better under DSC where the latency cost of round trip remote-cache accesses results in lower memory access latency compared to the 3-party directory protocol for the (6%) cache misses under DirCC.

The proposed ARCc architecture intelligently selects (cf. Section III-A) the appropriate coherence protocol at per-application granularity and as a result delivers higher performance relative to either DSC or DirCC protocols. Figure 8 shows the performance advantage of ARCc relative to DirCC for several per-core L2 cache configurations. The point where DSC outperforms DirCC is different for each application and also depends on the available cache capacity; this motivates our ARCc architecture and automatic DirCC to DSC coherence protocol transition.

FFT has the lowest core miss rate for DSC, and a high cache miss rate for DirCC, resulting in DSC outperforming DirCC by a wide margin. Under ARCc, our performance model switches to DSC for all L2 cache configurations in this application. On the other hand, WATER-N^2 exhibits a mixed behavior, where the cache miss rate under DirCC ranges from 4% (16KB L2) to less than 1% (256KB L2). Because DSC has 15% remote-cache accesses, the small cache miss rate of DirCC (less than 1% under 256KB per-core L2) allows it to outperform DSC. On the other hand, the 4% cache hierarchy miss rate of 16KB per-core L2 results in several round trip messages due to 3-party communication under DirCC. The latency of 15% remote-cache access round trips is offset by much lower (under 1%) cache hierarchy misses resulting in superior performance for DSC relative to DirCC.

In summary, our results show that ARCc always predicts the better performing coherence protocol correctly under the proposed online analytical model. As a result ARCc delivers 1.6× (for 16KB L2) to 1.15× (for 256KB L2) performance advantage over the directory-based DirCC protocol.

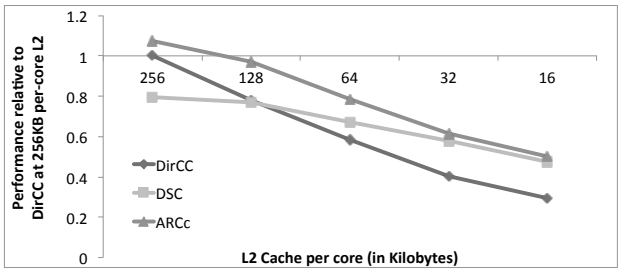

Fig. 10. Performance averaged over all benchmarks as cache sizes decrease for 128-core multicore.

C. Dynamic Energy Advantage of ARCc

Since energy consumption is a critical factor in future single-chip processors, we employed an energy model (cf. Section IV-A) to estimate the dynamic energy consumed by the ARCc, DSC and DirCC coherence protocols. On the one hand, remote-cache accesses incur dynamic energy costs due to increased traffic in the on-chip network; on the other hand, dramatic reductions in off-chip accesses equate to very significant reductions in DRAM access energy.

Figure 9 shows that energy consumption depends on each application's access patterns. For FFT, for example, which incurs crippling rates of eviction invalidations under DirCC, the energy expended by the coherence protocol messages and DRAM references far outweigh the cost of energy used by remote-cache accesses. ARCc chooses to switch to the DSC protocol and exploits energy benefits in addition to the performance gains. On the other extreme, the mostly on-chip data usage and read-only paradigm of RAYTRACE allows DirCC to efficiently keep data in the per-core caches and consume far less energy (in addition to performance gains) compared to DSC.

In summary, our results show that in addition to performance, ARCc also exploits the energy advantage of DSC. For all applications and per-core cache sizes, where ARCc chooses to switch from DirCC to DSC, dynamic energy benefits materialize. Overall, ARCc delivers 1.5× (for 16KB L2) to 1.05× (for 256KB L2) dynamic energy advantage over the directory-based DirCC protocol.

D. Overall Performance and Dependability

Figure 10 shows, for a 128-core processor, how the ARCc architecture performs on average as the capacity of caches is reduced. Although the DSC protocol performs better at lower cache sizes and DirCC performs better at higher cache sizes, the combined ARCc architecture responds to system conditions to select the best combination on a per-application granularity, and outperforms both baselines. We observe that ARCc becomes more advantageous when the system cache sizes are no longer sized exactly to deliver maximum performance for a particular protocol.

Even under our default 256KB per-core L2 cache, ARCc picks the best performing protocol and delivers an average of 8% performance gain over DirCC protocol. At 128KB and 64KB per-core L2 cache sizes, ARCc incurs 3% and 27% performance loss respectively, whereas, the DirCC counterpart would have experienced in excess of 30% and 50% performance loss compared to the default 256KB L2 cache per core.

Finally, when the L2 cache is sized at 32KB or 16KB per core, DirCC performs significantly worse than DSC, whereas ARCc matches or performs slightly better than DSC. Thus, the ARCc architecture exploits application-level asymmetrical behaviors to boost system performance and consistently outperforms both DSC and DirCC coherence mechanisms. The fact that ARCc allows two redundant and independent coherence protocols to co-exist in hardware can be exploited to improve dependability; we will explore this in future work.

VI. Related Work

Although directory-based coherence protocols have become the de facto standard to keep on-chip caches coherent, they have certain drawbacks such as frequent indirections due to the directory storage overhead and protocol complexity. Therefore, researchers have recently proposed several alternative architectures to simplify the hardware requirements for cache coherence. The COHESION architecture [20] combines hardware-managed and software-managed coherence domains at fine-grained granularity. COHESION offers reduced message traffic and does not require an on-chip directory when software coherence is used. Our ARCc architecture is superior to COHESION in that it utilizes a more efficient hardware alternative to ensure architectural redundancy. Because ARCc is all-hardware, it also does not require any software changes.

Pugsley et al. [21] proposed SWEL, a directory-less coherence protocol. The SWEL protocol replaces the directory with a much smaller bookkeeping structure that tracks private, read-only and shared read/write cache lines. The read-only data is allowed to be replicated in the L1 caches and the read/write data is only allowed to be present in the L2 cache that is shared by all cores. SWEL greatly reduces the number of coherence operations, but it also requires a fallback broadcast-based snooping protocol when infrequent coherence is needed. Their results show that SWEL can improve performance over the directory-based counterpart when an application has frequent read-write data sharing.

The ARCc architecture proposes distributed shared cache (DSC) based coherence as an alternative, architecturally redundant mechanism to directory-based coherence. The DSC protocol is similar to the shared-memory mechanism for coherence proposed by Fensch and Cintra [7]. They argue that directory-based hardware cache coherence is not needed and that the OS can efficiently manage the caches and keep them coherent. The L1s are kept coherent by only allowing one L1 to have a copy of any given page of memory at a time.

VII. Future Work

In this paper, we have presented the ARCc architecture that allows a one-way transition from directory to DSC based coherence at application-level granularity. Hence, only one application can run at a time. In the future we plan to evaluate the ARCc architecture at the granularity of phases within application, OS-pages, as well as cache lines. Novel static and dynamic methods will be explored that correctly predict when DSC is preferable to the directory protocol. After prediction, efficient transitions need to be made between the two protocols.

VIII. Conclusion

In this paper we have identified the need for architecturally redundant cache-coherence in large-scale multicore processors. We have proposed a novel cache coherence architecture (ARCc) that provides architectural redundancy for maintaining coherence across on-chip caches. ARCc combines traditional directory-based coherence with a remote-cache-access based coherence architecture (DSC) to ensure significant performance and energy gains. ARCc allows these two independent and heterogeneous coherence protocols to co-exist in hardware and enables a more dependable architecture.

References

[1] S. Rusu, S. Tam, H. Muljono, D. Ayers, J. Chang, R. Varada, M. Ratta, and S. Vora, "A 45nm 8-core enterprise Xeon® processor," in *A-SSCC*, 2009, pp. 9–12.

[2] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown *et al.*, "TILE64 - processor: A 64-Core SoC with mesh interconnect," in *ISSCC*, 2008, pp. 88–598.

[3] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain *et al.*, "An 80-Tile Sub-100-W TeraFLOPS processor in 65-nm CMOS," *IEEE J. Solid-State Circuits*, vol. 43, no. 1, pp. 29–41, 2008.

[4] S. Borkar, "Thousand core chips: a technology perspective," in *DAC*, 2007, pp. 746–749.

[5] A. Gupta, W. Weber, and T. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *International Conference on Parallel Processing*, 1990.

[6] C. Kim, D. Burger, and S. W. Keckler, "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches," in *ASPLOS*, 2002.

[7] C. Fensch and M. Cintra, "An os-based alternative to full hardware coherence on tiled cmps," in *International Conference on High Performance Computer Architecture*, 2008.

[8] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Operating system support for improving data locality on cc-numa compute servers," *SIGPLAN Not.*, vol. 31, no. 9, pp. 279–289, 1996.

[9] M. Zhang and K. Asanović, "Victim replication: maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *ISCA*, 2005.

[10] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: near-optimal block placement and replication in distributed caches," in *ISCA*, 2009.

[11] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," in *IEEE Micro*, 2005.

[12] http://www.tilera.com, "Tile-gx processor family: Product brief," 2011.

[13] H. Hoffmann, D. Wentzlaff, and A. Agarwal, "Remote Store Programming A Memory Model for Embedded Multicore," in *HiPEAC*, 2010.

[14] M. M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C. Luk, G. Lyons, H. Patil *et al.*, "Analyzing parallel programs with pin," *Computer*, vol. 43, pp. 34–41, 2010.

[15] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *HPCA*, 2010, pp. 1–12.

[16] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *ISCA*, 1995.

[17] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi, "A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies," in *ISCA*, 2008, pp. 51–62.

[18] M. Marchetti, L. Kontothanassis, R. Bianchini, and M. Scott, "Using simple page placement policies to reduce the cost of cache fills in coherent shared-memory systems," in *IPPS*, 1995.

[19] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, "Directory-based cache coherence in large-scale multiprocessors," in *COMPUTER*, 1990.

[20] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel, "Cohesion: A hybrid memory model for accelerators," in *International Conference on Computer Architectures*, 2010.

[21] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian, "Swel: Hardware cache coherence protocols to map shared data onto shared caches," in *International Conference on Parallel Architectures and Compilation Techniques*, 2010.