Experience with Improving Distributed Shared Cache Performance on Tilera's Tile Processor

Inseok Choi, Minshu Zhao, Xu Yang, and Donald Yeung
Department of Electrical and Computer Engineering
University of Maryland at College Park
{inseok,mszhao,yangxu,yeung}@umd.edu

Abstract—This paper describes our experience with profiling and optimizing physical locality for the distributed shared cache (DSC) in Tilera's Tile multicore processor. Our approach uses the Tile Processor's hardware performance measurement counters (PMCs) to acquire page-level access pattern profiles. A key problem we address is imprecise PMC interrupts. Our profiling tools use binary analysis to correct for interrupt "skid," thus pinpointing individual memory operations that incur remote DSC slice references and permitting us to sample their access patterns. We use our access pattern profiles to drive *page homing optimizations* for both heap and static data objects. Our experiments show we can improve physical locality for 5 out of 11 SPLASH2 benchmarks running on 32 cores, enabling 32.9%–77.9% of DSC references to target the local DSC slice. To our knowledge, this is the first work to demonstrate page homing optimizations on a real system.

1 Introduction

As core count in multicore chips increases, on-chip cache becomes a key determiner of performance. To keep up with the on-chip parallelism, it is necessary to distribute the cache across the chip and provide independent access to separate cache banks. A multicore in which the shared cache is distributed among the processor's cores is called a distributed shared cache (DSC [1]) architecture. DSC references exhibit non-uniform cost since data placed in a cache bank close to a requesting core can be accessed more quickly than data placed in a distant bank, even when the caches are coherent.

Higher performance can potentially be achieved on DSCs by managing on-chip physical locality so that data are placed in the cache banks closest to their referencing cores. Such bank *homing optimizations*, which have been explored by prior work, can be controlled either in hardware at cache-block granularity [2], [3], [5], [8], [10], [12], [19] or in software at page granularity [6], [9], [13], [14]. Hardware techniques typically map the cache blocks on different banks based on memory block addresses, while software techniques usually rely on the operating system to home individual pages on different banks.

To our knowledge, this prior research was conducted on simulators only. Studies on real processors are valuable because they can highlight real-world issues (*e.g.*, [15]). Such studies have been lacking for homing optimizations because processors did not implement DSCs. But recently Tilera Corporation has shipped many-core CPUs that use a tiled CMP architecture. In these *Tile Processors* [11], the lowest level of cache employs a cache-coherent DSC architecture. A typical Tile processor DSC is composed of 64 independent cache "slices" distributed amongst the cores. Hardware maintains cache coherency, and the operating system controls homing onto DSC slices at page granularity. In this architecture, cache misses incur a variable cache access latency, making homing optimizations relevant.

This paper presents our experience with improving physical locality in the Tile Processor, making several contributions. First, we present a novel technique for acquiring page-based access pattern profiles which can be used to drive homing decisions [9], [13]. The profiles are gathered using the Tile Processor's hardware performance measurement counters (PMCs). In particular, our solution corrects for the Tile Processor's imprecise PMC interrupts (an issue on many CPUs) to permit sampling of individual memory instructions that access the DSC. Second, we develop an optimization library that performs page homing for both heap and static data objects, using our access pattern profiles to place pages on the tile that accesses them the most. Finally, we conduct experiments using programs from the SPLASH2 benchmark suite [18] that quantify the effectiveness of our techniques. Along with our earlier work [7], this study provides the first-ever demonstration of page homing optimizations on an actual commercial CPU.

Manuscript submitted: 15-Apr-2011. Manuscript accepted: 14-Jun-2011. Final manuscript received: 22-Jun-2011

2 Access Pattern Profiles

Software page-based techniques require access pattern information–*i.e.,* the per-page distribution of references performed by cores–to drive page homing decisions. This section describes how access pattern profiles can be acquired using hardware PMCs on Tile Processors.

2.1 Tile Processor

A typical Tile Processor, illustrated in Figure 1, consists of a grid of 64 general-purpose VLIW cores interconnected by multiple 2D mesh on-chip networks. Each core has its own private split L1 cache, and a local L2 cache that acts as one slice of a DSC. The core and its associated cache are connected to the on-chip networks through a switch. The switch, core, and cache form a *tile*. Cores can access their local L2 slice with minimal latency, but incur increasingly higher latencies to access more distant L2 slices due to inter-tile communication across the switched interconnect.

Tile Processors provide several ways in which data can be placed across the DSC caches, including on a per-page basis. Every virtual memory page can be assigned its own home tile. The home tile's L2 cache is where cache blocks from the page are cached on-chip. Hence, this permits flexible OS-controlled distribution of data. Pages are 64KB each; the hardware TLBs can support smaller pages, but the current OS cannot.

To enable measurement of low-level hardware events, the Tile Processor supports 2 32-bit hardware performance measurement counters per tile. Each hardware PMC can observe

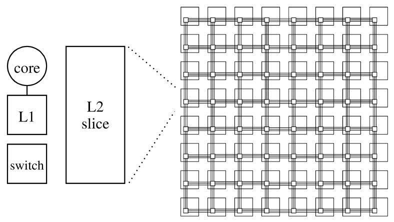

Fig. 1. A typical Tile Processor is composed of 64 tiles

one of 99 pre-defined hardware events at any moment in time. These events monitor instruction execution in the cores, memory operations in the memory hierarchy, as well as traffic across the on-chip network. The Tile Processor runs a Linux operating system which supports OProfile for accessing the hardware PMCs. In addition, we ported PAPI [16] and Perfmon2 [17], two other standard PMC APIs, to the Tile Processor.[1]

(Note, some details about the Tile Processor have been omitted in this section because they are not yet public).

2.2 Using PMCs to Profile Memory References

For every page in memory, we profile the number of references each core makes to the page in the DSC, thus identifying the most frequently referencing core(s). We only profile loads because the Tile Processor does not support monitoring DSC stores. In any case, stores write to a store buffer on a cache miss and do not cause significant performance degradation in the parallel programs we study.

The Tile Processor's PMCs can count remote-read hardware events–*i.e.*, loads that miss in the local L1 cache and hit in a remote L2 slice. Moreover, PMCs can deliver an interrupt after a pre-set number of remote-read events. Each interrupt/sample can identify the core performing the load, as well as the load instruction involved (*i.e.*, its program counter or PC). So, the interrupt handler can probe the register containing the load's effective address and identify the referenced page. For each benchmark, we perform separate profiling runs in which all pages are homed on a spare tile not running any compute threads, thus making all L2 accesses remote and allowing them to be sampled by the remote-read event interrupts. After a large number of samples, we can determine *statistically* the frequency with which all pages in a program are referenced by each core.

One problem is the Tile Processor's PMC interrupts are not precise. After a PMC interrupt is signaled, the core keeps executing. When the interrupt is actually serviced, the core has executed past the event-triggering instruction, so the PC sampled is not the load performing the DSC reference. Such PMC sampling "skid" prevents pinpointing event-triggering loads which is necessary to profile their access patterns.

Fortunately, it is possible to correct for sampling skid on the Tile Processor due to the nature of its pipeline. The Tile CPU employs a register file with presence bits [4] that allow execution past cache-missing loads. Rather than the cache-missing load stalling the pipeline, the first instruction to use the load's target register stalls, as illustrated in Figure 2. We find the delay in signaling a PMC interrupt is larger than the def-use distance for DSC referencing loads (we observe a def-to-use of 1–20 VLIW instruction bundles), but smaller than the latency for the remote L2 slice access. Hence, the PMC interrupt always samples the instruction *dependent* on the event-triggering load. (We verified this manually for a large number of cases).

While event-triggering loads cannot be directly profiled on Tile Processors, they can be inferred from the sampled PCs via

1. The latest versions of PAPI are implemented on top of Perfmon2.

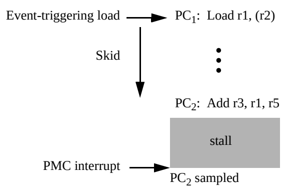

Fig. 2. Imprecise handling of PMC interrupts

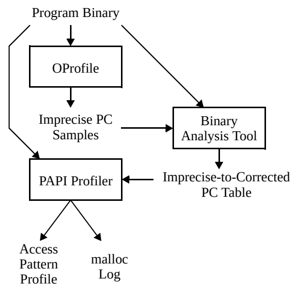

Fig. 3. Profiling infrastructure

dependences: an event-triggering load is the first load preceding the sampled PC whose destination register matches one of the sampled instruction's source registers. We perform such dependence analysis on the static program binary. Usually, dependence analysis encounters the event-triggering load in the same basic block as the sampled instruction; however, in some cases, the event-triggering load resides in the basic block preceding the block containing the sampled instruction.

2.3 Profiling Tools

Figure 3 illustrates the tools involved in profiling. We perform two profiling runs to acquire the access pattern profiles. In the first run, we use OProfile to collect the imprecisely sampled PCs, and then use our own binary analysis tool to perform the sampling skid correction analysis. From this analysis, we build a table that associates the imprecise PCs with their corresponding corrected PCs and the register containing the effective address of the event-triggering load.

We use a modified version of PAPI to perform a second profile run that acquires the access pattern profiles. On each sampling interrupt, PAPI consults the table to get the effective address register, probes the register to determine the referenced page, and logs the sample (core ID and page number) in a separate profile table. At the end of the second profiling run, this profile table is output to the user.

In addition to profiling access patterns, we also log all calls to malloc, the heap memory allocator, to associate pages in the access pattern profiles to individual heap objects.

3 Page Homing Optimization

Once the access pattern profile and malloc log have been acquired for a given program, subsequent executions of the program can use them to drive page homing optimizations. This section presents our optimizations.

3.1 Optimization Opportunities

Our page homing optimization tries to home heap and static data memory region pages that are referenced primarily by a single core on the tiles where they are referenced most frequently. Figure 4 illustrates opportunities for doing this. In Figure 4, we graph the access pattern profile for a 16-core execution of Ocean from the SPLASH2 benchmarks [18]. Pages are plotted along the X-axis while cores are plotted along the Y-axis. The graph plots the normalized number of samples acquired for each page from each core along the "Z-axis" (extending out of the paper). Samples that are particularly large are highlighted by the shaded peaks. In Figure 4, the pages numbered 106 to 700 are referenced primarily by a single core. These are the pages our optimization tries to explicitly home.

For the SPLASH2 benchmarks, we find there are two major types of objects that can be optimized. The first type, *distributed arrays*, is illustrated in Figure 4 by pages 148–274 and 274–442.

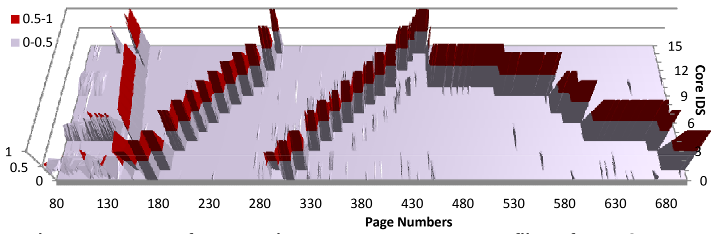

Fig. 4. Part of example access pattern profile of a 16-core execution of Ocean from the SPLASH2 benchmark suite.

Each of these is a single object (*i.e.*, allocated by a single malloc call), accessed by all the cores. But most of the per-core accesses are destined to mutually exclusive and contiguous pages in the object. They can be optimized by distributing pages in chunks across neighboring tiles to match their diagonal access patterns.

The second type, *privately accessed objects*, is illustrated in Figure 4 by pages 106–127 and 442–700. Here, each set of pages that are referenced by the same core belongs to a separate object. These objects can be optimized by homing all of their pages on the tile where most of the memory references occur.

The remaining pages in Figure 4 are primarily accessed by multiple cores. Our optimization does not try to improve physical locality for such shared pages. Instead, we simply distribute shared pages in round-robin fashion across tiles. Note, Figure 4 only shows references in the parallel region. Most pages are initialized by tile 0 at program startup (this is true for all SPLASH2 benchmarks). Hence, a first-touch policy would place all pages on tile 0, resulting in poor performance.

3.2 Homing Heap Pages

Page homing in the heap can be controlled via the Tile Processor's "mspace" abstraction. A Tilera mspace is essentially a segment, with a particular homing policy for all pages in the segment. We create multiple mspaces with different homing policies tailored to the different heap objects and access patterns described in Section 3.1. An optimization library is provided to place heap objects in the appropriate mspace according to each object's profiled access pattern.

For privately accessed heap objects, our optimization library creates one mspace per tile, with each mspace homing its pages on a unique tile. For heap-based distributed arrays, our optimization library creates an mspace that distributes pages across tiles so that each portion of the distributed array resides in its referencing core's local tile. We set the *chunking factor* (the number of contiguous pages to place on one tile before moving onto the next tile) to be the ratio of the distributed array size and the number of tiles in the machine times the page size.

In order to select the appropriate mspace for each allocated heap object, our custom malloc function first matches the call to its corresponding call of malloc in the malloc log, then allocates the heap object on the corresponding mspace according to its access pattern. If the object is not of type privately accessed object or distributed array, the custom malloc function allocates the object onto a default mspace that distributes the object's pages across tiles in round-robin fashion.

Note, the malloc log's access pattern information is not tied to a specific machine or data object size. For example, malloc calls for privately accessed heap objects allocate "locally" rather than to a hard-wired core ID. So, local allocation still occurs even if core count changes. Moreover, malloc calls for distributed arrays re-compute the chunking factor in each execution. So, array distribution still occurs uniformly across all the tiles even if core count and/or allocated array size change. While not perfect, this approach increases the likelihood that pages are correctly placed for optimized runs employing a different machine and/or problem size than the profile runs.

TABLE 1
SPLASH2 benchmarks with their input problem sizes.

Benchmark	Input	Benchmark	Input
FFT	2^{20} points	Ocean	1026 grid
Barnes	16384 bodies	Water-NS	1000 molecules
Cholesky	tk17.O	Water-SP	1000 molecules
Radix	2097152 keys	Radiosity	7832 objects
LU	1024 matrix	Raytrace	ball4
FMM	input.2048		

3.3 Homing Static Data Pages

Unlike heap objects, static data objects are allocated at compile time, and are bound to a particular mspace. We use memory mapping and unmapping to change the homing policy. We first identify all pages in the static data region from the access pattern profile that are referenced primarily by a single core. Next, we copy the contents of these identified pages to an external file. Then, we unmap the copied pages from the program's address space, and map into their place the copied data from the external file using the `mmap_mbind()` system call, which permits specifying a home tile. Hence, this permits per-page homing control in the static data region.

4 Experimental Results

This section demonstrates the profiling and optimization techniques discussed in Sections 2 and 3, and studies their benefits.

4.1 Experimental Methodology

We conduct experiments on a Tile Processor running the Linux operating system from the Tilera MDE version 2.1. To drive our study, we use the entire SPLASH2 benchmark suite [18] except for volrend. We use `tile-cc` (the Tile Processor's C compiler) to compile the benchmarks with the highest level of optimization. Table 1 lists the benchmarks and the input problems we use in the experiments.

To quantify improvements, we compare the optimized and unoptimized benchmarks. To obtain optimized benchmarks, we first acquire access pattern profiles and malloc logs using our profiling tools on 32-core executions. Then, we instrument the benchmarks to call our optimization library routines and to perform the homing optimizations for the static data region. Lastly, we re-compile the benchmarks, linking them against our optimization library. For the unoptimized benchmarks, we use our optimization library to distribute all heap and static data pages across tiles in round-robin fashion.

In our results, we report sampled page references at the DSC level in the parallel region of each benchmark. The sampling counts can be converted into page reference counts (at least approximately) by multiplying by the sample frequency, 7000. (This sampling frequency was determined experimentally for SPLASH2. It may be necessary to tune it for other benchmarks.)

4.2 Physical Locality Results

Table 2 reports our page reference count results. In particular, the 2^{nd} and 3^{rd} columns (labeled "Total") report the number of sampled page references in each benchmark's profiling run that are destined to the heap and static data memory regions.

The 4^{th} and 5^{th} columns (labeled "Baseline") report the number of sampled page references in the unoptimized benchmarks that are destined to local L2 slices broken down into heap and static data references, respectively. The 6^{th} column reports the percentage of the total sampled references that these baseline local references represent–*i.e.* $(\%\ Total)_{Baseline}$ =

TABLE 2
Number of sampled page references to the heap and static data regions in total, that are destined to local L2 slices in the baseline and optimized benchmarks, and that can be potentially optimized.

	Total		Baseline			Optimized			Potential	
	Heap	Static	Heap	Static	% Total	Heap	Static	% Total	Heap	Static
FFT	5376	8387	289	536	6.0%	5372	536	42.9%	5376	0
Barnes	8197	11324	521	711	6.3%	521	7152	39.3%	246	6920
Cholesky	37361	6735	1890	389	5.2%	1907	389	5.2%	113	2
Radix	4299	79	276	5	6.4%	3404	5	77.9%	3425	15
LU	0	2	0	0	0%	0	0	0%	0	2
FMM	19667	123	1583	9	8.0%	1583	9	8.0%	19667	1
Ocean	90703	26030	5400	1387	5.8%	87857	1387	76.5%	88783	0
Water-NS	543	3211	22	190	5.6%	438	190	16.7%	543	0
Water-SP	415	0	1	0	0.2%	1	0	0.2%	415	0
Radiosity	4741	1824	430	68	7.6%	430	68	7.6%	192	259
Raytrace	30750	14580	1796	964	6.1%	1796	964	6.1%	5	0

$\frac{(Heap+Static)_{Baseline}}{(Heap+Static)_{Total}} \times 100$. This data shows the unoptimized benchmarks exhibit poor physical locality. Only 5%–8% of all DSC references are to local L2 slices.

Similarly, the 7th and 8th columns (labeled "Optimized") report the number of sampled page references in the optimized benchmarks, and the 9th column reports the percentage of these optimized local references. As this data shows, our page homing optimizations improve physical locality for 5 benchmarks: FFT, Barnes, Radix, Ocean, and Water-NS. In these benchmarks, 39.3%–77.9% of DSC references are to local L2 slices, a 6–12X increase over the baseline. For the remaining 6 benchmarks, our homing optimizations do not find many pages to optimize–*i.e.*, that are referenced primarily by a single core–so the number of localized DSC references does not change compared to the baseline.

The remaining (10th and 11th) columns report the number of samples destined to heap and static data pages that are referenced by *no more than half the cores* (16) in the profiling runs. Since our homing optimization must place each page on a specific tile, it is only effective for pages referenced by a small number of cores. Hence, these sampled reference counts are a good estimate for the potential physical locality improvement.

Comparing the "Potential" and "Optimized" results, we see our optimizations capture most of the physical locality in the SPLASH2 benchmarks–*i.e.*, many of the optimized heap and static data counts are close to the corresponding potential counts. This suggests that for us to do substantially better, we must create more opportunities. We notice many pages remain unoptimized because they are shared by many cores. False sharing is a major reason for this since the Tile Processor's page size is rather large, 64 KB. Our optimizations could potentially become more effective if the page size were reduced.

5 Conclusions

This paper describes our experience with page-level homing optimizations on a real system, Tilera's Tile Processor running a Linux OS. We show hardware PMCs can be used to acquire page-level access pattern profiles. Moreover, we show that binary analysis can be used to correct for interrupt skid–due to imprecise PMC interrupts–to pinpoint individual memory operations incurring remote-core references and sample their access patterns. We find our page homing optimizations driven by our access pattern profiles can improve physical locality for 5 out of 11 SPLASH2 benchmarks, enabling 39.3%–77.9% of DSC references to target the local L2 slice. In addition, we find our homing optimizations already exploit most of the potential physical locality in the SPLASH2 benchmarks. Significant improvements can only come by creating more opportunities for homing, perhaps by addressing false sharing via smaller virtual memory pages.

References

[1] A. Agarwal, "Tiled Multicore Processors: The Four Stages of Reality," http://groups.csail.mit.edu/cag/raw/documents/tiled-processors-ieee-micro-keynote-2007.pdf 2007.

[2] B. M. Beckman and D. A. Wood, "Managing Wire Delay in Large Chip-Multiprocessor Caches," in *Proc. of the Int'l Symp. on Microarchitecture*, Portland, OR, Dec. 2004.

[3] J. Chang and G. S. Sohi, "Cooperative Caching for Chip Multiprocessors," in *Proc. of the Int'l Symp. on Comp. Arch.*, June 2006.

[4] T.-F. Chen and J.-L. Baer, "Reducing Memory Latency via Non-blocking and Prefetching Caches," University of Washington, 92-06 03, June 1992.

[5] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Optimizing Replication, Communication, and Capacity Allocation in CMPs," in *Proc. of the Int'l Symp. on Comp. Arch.*, Madison, WI, June 2005.

[6] S. Cho and L. Jin, "Managing Distributed, Shared L2 Caches through OS-Level Page Allocation," in *Proc. of the Int'l Symp. on Microarchitecture*, Dec. 2006.

[7] I. Choi, M. Zhao, X. Yang, and D. Yeung, "Early Experience with Profiling and Optimizing Distributed Shared Cache Performance on Tilera's Tile Processors," in *Proc. of the Int'l Workshop on Unique Chips and Systems*, Atlanta, GA, Dec. 2010.

[8] Z. Guz, I. Keidar, A. Kolodny, and U. C. Weiser, "Utilizing Shared Data in Chip Multiprocessors with the Nahalal Arch." in *Proc. of the Int'l Symp. on Parallelism in Algorithms and Arch.*, Munich, Germany, June 2008.

[9] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," in *Proc. of the Int'l Symp. on Comp. Arch.*, Austin, TX, June 2009.

[10] E. Herrero, J. Gonzalez, and R. Canal, "Distributed Cooperative Caching," in *Proc. of the Int'l Conf. on Parallel Arch. and Compilation Techniques*, Toronto, Canada, Oct. 2008.

[11] http://tilera.com/products/processors, "Processors from Tilera Corporation."

[12] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, "A NUCA Substrate for Flexible CMP Cache Sharing," in *Proc. of the Int'l Conf. on Supercomputing*, Boston, MA, June 2005.

[13] L. Jin and S. Cho, "SOS: A Software-Oriented Distributed Shared Cache Management Approach for Chip Multiprocessors," in *Proc. of the Int'l Conf. on Parallel Arch. and Compilation Techniques*, Raleigh, NC, Sept. 2009.

[14] L. Jin, H. Lee, and S. Cho, "A Flexible Data to L2 Cache Mapping Approach for Future Multicore Processors," in *Proc. of the ACM SIGPLAN Workshop on Memory System Performance and Correctness*, Oct. 2006.

[15] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems," in *Proc. of the Int'l Symp. on High Perf. Comp. Arch.*, Salt Lake City, UT, Feb. 2008.

[16] "Performance Application Programming Interface."

[17] "The Hardware-Based Performance Monitoring Interface for Linux."

[18] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proc. of the Int'l Symp. on Comp. Arch.*, Santa Margherita Ligure, Italy, June 1995.

[19] M. Zhang and K. Asanovic, "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors," in *Proc. of the Int'l Symp. on Comp. Arch.*, Madison, WI, June 2005.