Early Experience with Profiling and Optimizing 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.**

I. Introduction

Practically all current high-performance commercial CPUs integrate multiple cores on a single chip. Today, multicore chips with 4-8 cores are commonplace. Several companies have also demonstrated that it is possible to integrate many 10s of cores on-chip [1], while others are shipping manycores [2] that run standard operating systems and are programmable using familiar shared memory models. And since Moore's law scaling will continue at historic rates for the foreseeable future [3], even higher core counts are expected down the road.

A key determiner of multicore performance is the on-chip cache. As the number of cores increases, it becomes necessary to introduce hierarchy or to distribute the cache across the chip and provide independent access to separate cache banks in order to keep up with the on-chip parallelism. A multicore in which the shared cache is distributed among the processor's cores is called a distributed shared cache (DSC [4]) architecture. Memory references to such physically distributed shared caches 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, the cache misses will exhibit a non-uniform cost. This can affect performance each time the distributed cache is accessed–*i.e.*, when a miss occurs from a cache higher up in the on-chip memory hierarchy.

On processors with distributed caches, higher performance can potentially be achieved by managing on-chip physical locality so that data are placed in the cache banks closest to their referencing cores. Such bank *homing optimizations* can be controlled either in hardware at cache-block granularity or in software at page granularity. Hardware techniques, which are implemented within the cache coherence protocol, typically map the cache blocks on different banks based on memory block addresses. Software techniques typically rely on the operating system to provide homing information via the virtual memory layer, thus enabling individual pages to be homed on different banks. Page-based techniques often require profiling to determine per-page access patterns for driving the page homing decisions. Apart from homing optimizations, it is also possible to replicate and/or migrate data at runtime to further improve physical locality (*e.g.*, to track dynamically changing access patterns).

Several researchers have explored homing optimizations in the past, with significant prior work related to both hardware cache block-based [5], [6], [7], [8], [9], [10], [11] as well as software page-based [12], [13], [14], [15] techniques. However, all of this prior research was conducted on simulators. To our knowledge, no study has applied homing optimizations on real processors. Such research is important because it can highlight real-world issues overlooked by simulation studies that must be addressed before possible benefits can be realized.

In the past, processors did not implement distributed caches, so real-system studies were not possible. But this is no longer the case today. For example, Tilera Corporation has recently shipped many-core CPUs that use a tiled CMP architecture. In these *Tile Processors* [2], the lowest level of cache employs a cache-coherent distributed shared cache architecture. A typical Tile processor DSC is composed of 64 independent cache "slices" distributed amongst the cores, hardware maintains cache coherency and operating system provide homing information. When a processor makes a given memory reference and suffers a cache miss, the cache coherency mechanism directs the miss to a *home cache* on another core on the chip, thereby potentially averting a costly off-chip DRAM access. The coherency hardware then moves the referenced data automatically to the referencing core's cache so that subsequent references may be satisfied locally. In this architecture, cache misses incur a variable cache access latency, making homing optimizations relevant.

This paper presents our early experience with improving physical locality in a Tile Processor's DSC. Our work focuses on how to apply page-based homing optimizations on the Tile CPU, making the following contributions. First, we present a novel technique for acquiring fine-grain page-level access pattern information for driving page placement decisions. Although only information about which threads access which pages is needed, determining this requires pinpointing individual memory instructions so that the memory addresses, and hence pages, each thread accesses can be profiled. Our solution leverages the Tile Processor's hardware performance measurement counters (PMCs) to sample the effective addresses of individual memory instructions. PMCs enable low-overhead profiling, but they are typically not designed to provide per-instruction sample resolution. A key part of our solution is to use binary analysis to correct the imprecise hardware samples, thus pinpointing individual memory instructions that reference remote slices and permitting us to profile their access patterns.

Second, we use our access pattern profiles to drive homing decisions, to place the pages on the tile that accesses them the most. Specifically, we try to explicitly home and improve physical locality for pages in the heap and static data memory regions. We currently only optimize pages that are referenced primarily by a single core, placing them on the DSC slice closest to the core with the most references to the page. Our optimizations do not allow page migration. Instead, placement decisions made at memory allocation time are fixed for the duration of the program's run.

Finally, we conduct experiments using programs from the SPLASH2 benchmark suite [16] that demonstrate our profiling and optimization techniques. Our results show we can improve physical locality for 5 out of 11 SPLASH2 benchmarks running on 32 cores, enabling 39.3%–77.9% of DSC references to target the local DSC slice. Moreover, 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.

The remainder of the paper is organized as follows. Section II presents our access pattern profiling techniques. Then, Section III describes how we home pages based on the access pattern profiles. Next, Section IV discusses our experiments. Finally, Section V concludes the paper.

II. Access Pattern Profiles

Software page-based techniques require access pattern information to drive page homing decisions. In particular, the distribution of references performed by cores on a per-page basis is needed. In previous work, such access pattern profiles were obtained via simulation which is slow and requires architectural simulators. To enable page-based techniques on real systems, it is crucial to develop more efficient techniques. This section describes how access pattern profiles can be acquired using hardware PMCs. Section II-A begins with an overview of the Tile Processor architecture and its PMC support. Then, Section II-B discusses the problem of profiling individual memory instructions, and describes how we address the problem. Finally, Section II-C presents the profiling system we built.

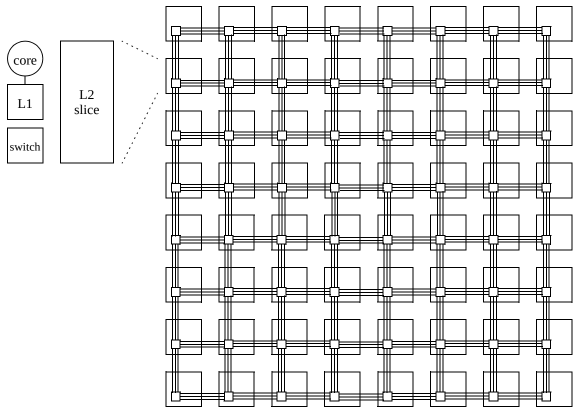

Figure 1. A typical Tile Processor is composed of 64 tiles, each containing a VLIW core + L1 cache, an L2 cache "slice," and an on-chip network switch.

A. Tile Processor

A typical Tile Processor, illustrated in Figure 1, consists of a grid of 64 general-purpose VLIW cores and 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 distributed shared cache. The core and its associated cache are connected to the on-chip networks through a switch. The switch, core, and cache are referred to as 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 allow several ways in which data can be placed across the DSC caches, including on a page-by-page basis in which each page can be homed on any given core. This permits flexible OS-controlled distribution of data. Since our work focuses on page-based homing, we use the Tile Processor's per-page mechanism exclusively.

In the per-page approach, every virtual memory page is assigned its own home tile. The home tile's L2 cache is where cache blocks from the page are cached on-chip. In Section III, we will discuss how software can specify the home for each page, thus controlling data placement on-chip.

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 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, a UNIX system-level utility for accessing the hardware PMCs. In addition, we ported PAPI [17] and Perfmon2 [18] to the Tile Processor.[1] These are standard APIs that export a fuller set of PMC features to users compared to OProfile.

B. *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) on a per-page basis at the DSC level. We only profile read references (loads) since these are the main source of performance degradation. (Stores write to a store buffer on a cache miss. They do stall when a memory fence is performed, but the programs we study, *e.g.* SPLASH2, are coarse grain parallel programs in which fences are very infrequent. Therefore, stores rarely stall in such programs).

The Tile Processor's PMCs can monitor a remote-read hardware event which is useful for acquiring access pattern profiles. A remote-read event occurs each time a core issues a load instruction that misses in the local L1 cache and then hits in a remote L2 slice. This monitors all DSC references except for those issued by the core on the referenced page's home. To get around this problem, during profile runs, we home all pages on a spare tile not running any of the compute threads, thus forcing all DSC references to be non-local and allowing them to be monitored by the remote-read event.

While the PMCs can count DSC references, they alone cannot associate the counts to pages and cores. For this, we rely on *sampling*. Hardware PMCs can be configured to deliver an interrupt after a pre-set number of remote-read events have occurred, allowing an interrupt handler to periodically sample load references to the DSC. In particular, each interrupt can identify the core performing the load, as well as the particular load instruction involved (*i.e.*, its program counter or PC). Moreover, given knowledge of the particular load being sampled, the interrupt handler can probe the register containing the load's effective address and identify the referenced page. In this fashion, each interrupt/sample can attribute a single page reference to a particular core. After a large number of such samples, we can determine *statistically* the frequency with which all pages in a program are referenced by each core.

One obstacle to implementing this approach is the Tile Processor's PMCs (as well as those on most other commercial CPUs) does not provide per-instruction sampling resolution. The problem is PMC interrupts are not precise.

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

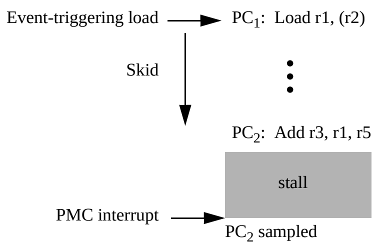

Figure 2. Imprecise handling of PMC interrupts on the Tile Processor results in sampling of the instruction dependent upon the event-triggering load (PC_2) rather than the load itself (PC_1).

When a PMC interrupt is signaled, the core keeps executing. At some later time, the interrupt is actually serviced, but by then the core may have executed past the event-triggering instruction. If so, the PC sampled is not the load performing the DSC reference, but rather some other PC further down the instruction stream. Such PMC sampling "skid" is not a problem when trying to locate the function or thread incurring an event, but it prevents pinpointing individual memory instructions which is necessary to profile their access patterns.

Fortunately, it is possible to correct for sampling skid on the Tile Processor due to certain features of its pipeline. The Tile CPU employs a register file with presence bits [19] that allow execution past cache-missing loads, providing some latency tolerance. 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.[2] In practice, we find the delay in signaling a PMC interrupt is larger than the def-use distance for loads that reference the DSC (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 almost always samples the instruction *dependent* on the event-triggering load.

By performing dependence analysis, we can identify the event-triggering load instruction from the sampled PCs: it is the first load preceding the sampled PC whose destination register matches one of the sampled instruction's source registers. Usually, we encounter 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. To handle these cases, we perform dependence analysis across basic blocks when necessary.

C. *Profiling Tools*

We perform two profiling runs to acquire the access pattern profiles. The first addresses the imprecise PMC interrupt problem described in Section II-B. It collects all of the imprecisely sampled PCs that occur in the profiled program. Then, after this profiling run completes, we perform binary

[2]It is possible that the first use does not stall if the cache-miss latency is completely overlapped, but this is not the common case.

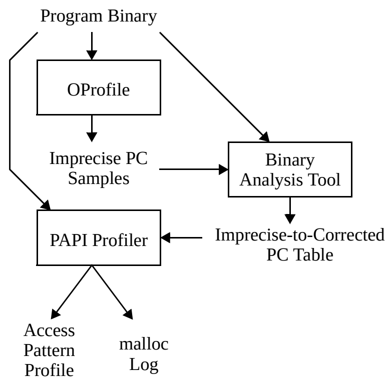

Figure 3. Profiling infrastructure for acquiring access pattern profiles.

analysis to correct the sampling skid and identify the event-triggering loads. From this analysis, we build a table that associates the imprecise PCs with their corresponding corrected PCs, along with the register containing the effective address of the event-triggering load at the corrected PC.

The second profiling run acquires the actual access pattern profiles. During this profiling run, each sampling interrupt consults the imprecise-to-corrected PC table computed from the first profiling run to identify the load responsible for the interrupt as well as its effective address register. As discussed in Section II-B, the interrupt handler probes the register to determine the referenced page, and logs the sample (core ID and page number) in a profile table. At the end of the second profiling run, this profile table–which contains the access pattern profile–is output to the user.

Figure 3 illustrates the tools involved in profiling. We use the OProfile utility (see Section II-A) in its unmodified form to collect the imprecise PC samples. We built our own binary analysis tool to construct the imprecise-to-corrected PC table. This binary analyzer extracts a control flow graph from the program binary to permit inter-basic block analysis when searching for the corrected PCs. Finally, we use PAPI to acquire the access pattern profiles. We modified PAPI to download the imprecise-to-corrected PC table into the kernel. We also modified PAPI's kernel-level PMC counter overflow handler to perform the PC sample correction and load effective address identification.

In addition to profiling access patterns, we also log all calls to malloc, the heap memory allocator. During each malloc call, we record the call site as well as the dynamic instance for that call site (in case it is executed multiple times). When each malloc call returns, we record the starting address and size of the allocated object. This information allows us to associate pages in the access pattern profiles back to individual heap objects, and to identify where (call site and dynamic call instance) those objects were created. As the next section will show, this information can be used for optimizing heap objects.

III. 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. First, Section III-A describes the access patterns that we target. Then, Sections III-B and III-C explain how we drive page homing for the heap and static data regions, respectively.

A. Optimization Opportunities

Our page homing optimization tries to home pages residing in the heap and static data memory regions on the tiles where they are referenced most frequently. Currently, our optimization targets pages in the access pattern profiles that are referenced primarily by a *single core*. Figure 4 shows an example access pattern profile, illustrating the different access patterns and objects we optimize.

In Figure 4, we graph the access pattern profile for a 16-core execution of Ocean, a program from the SPLASH2 benchmark suite [16]. 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. As Figure 4 shows, the pages numbered 106 to 883 are referenced primarily by a single core (*i.e.*, at each X-axis point in this range, there is always a single Y-axis point with a dominant peak). These are the pages our optimization tries to explicitly home.

In addition to identifying the pages to optimize, we must also identify which program-level objects the pages belong to. This is particularly important for heap objects because it determines which malloc calls must be instrumented to control homing (see Section III-B). In practice, we find there are two different types of objects. The first is illustrated in Figure 4 by pages 148–274 and 274–442 which form diagonal access patterns that increase in core ID with increasing page number. Each of these two memory regions is a single object (in this example, they are both on the heap and each is allocated by a single malloc call). Due to their diagonal access pattern, each object is accessed by all the cores, but most of the per-core accesses are destined to mutually exclusive and contiguous pages in the object. These two memory regions are examples of *distributed arrays*. They can be optimized by distributing their pages in chunks across neighboring tiles to match their diagonal access patterns.

The second type of object is illustrated in Figure 4 by pages 106–127 and 442–883 which form diagonal access patterns that decrease in core ID with increasing page number. Again, most of the per-core accesses in these two memory regions are destined to mutually exclusive and contiguous pages. But instead of one object containing all of the pages on the diagonal, each set of pages that are referenced by the same core is a separate object (*i.e.*, on the

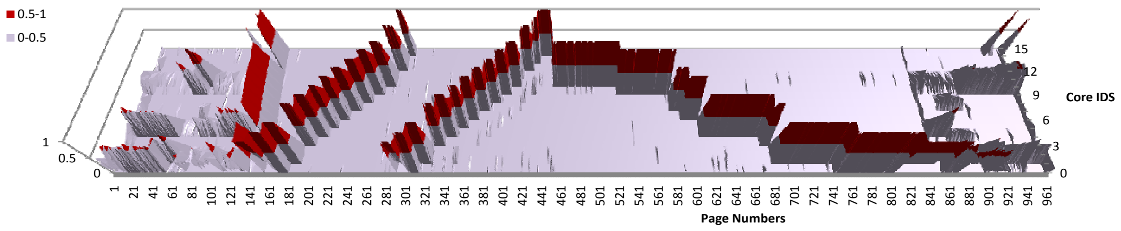

Figure 4. Example access pattern profile of a 16-core execution of Ocean from the SPLASH2 benchmark suite. Page numbers are plotted along the X-axis while core IDs are plotted along the Y-axis. Normalized sample count per core/page is plotted along the "Z-axis" (extending out of the paper).

heap, each would be allocated by a separate malloc call). These memory regions are examples of *privately accessed objects*. They 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 in the ranges 1–106 and 883–950 are primarily accessed by multiple cores, usually 2 or 3. Although not shown in Figure 4, another common case is pages that are accessed equally by all the 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.

B. *Homing Heap Pages*

Page homing in the heap can be controlled via the Tile Processor's mspace abstraction. A standard Linux parameter, mspace is a segment, with a particular homing policy for all pages in the segment. By default, the heap resides in a single mspace that homes its pages on the tile performing the first malloc to each page. For programs in which core 0 allocates all of the heap objects (*i.e.*, most of the SPLASH2 programs), this default policy places the entire heap on tile 0.

To improve physical locality for the different heap objects and access patterns described in Section III-A, we create multiple mspaces with different homing policies. We also provide a custom malloc function in a separate optimization library that can select between these different mspaces, thus binding different homing policies to heap objects as they are allocated at runtime. Users need only link their program against our optimization library, and provide the access pattern profile and malloc log for their program to enable our heap optimizations.

For privately accessed heap objects, our optimization library creates one mspace per tile, with each mspace homing its pages on a unique tile. At allocation time, the custom malloc function selects one of these mspaces according to the access pattern profile, thus homing the entire object on the tile where most of its references occur.

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. To achieve the desired physical locality, we set the distribution *chunking factor*–*i.e.*, 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.[3] Since each mspace can only support a single chunking factor, we must create one mspace for every unique chunking factor across all of the distributed arrays in the program.

In order to select the appropriate mspace for each allocated heap object, our custom malloc function consults the malloc log and access pattern profile acquired during the profiling runs. In particular, as the custom malloc function is called at runtime, it matches the call to its corresponding call of malloc in the malloc log. (The custom malloc function keeps track of the same call site and dynamic call instance information logged during profiling, as described in Section II-C, to enable matching). Once the corresponding malloc call from the profiling run is identified, the heap object being allocated can be determined along with its access pattern. If the heap object is a distributed array or a privately accessed object, then the custom malloc function allocates the object onto the mspace that supports the object's access pattern. Otherwise, the custom malloc function allocates the object onto a default mspace that distributes the object's pages across tiles in round-robin fashion.

Since our optimizations are profile-driven, their effectiveness is sensitive to discrepancies in access patterns between the profiling and optimized runs. The chunk size that each thread accesses will be different if the input data size changes. In particular, it may be desirable for optimized runs to use a different input problem or core count compared to the profile runs. Our optimization library tries to compensate for changes to these two parameters. For example, our custom malloc function adjusts the chunking factor for distributed arrays if array size and/or machine size changes from profiling run to optimized run. However, aside from problem input and core count variation, we do not compensate for any other factors that may alter access patterns at runtime, for example dynamic work distribution (*e.g.*, using work queues).

Our current page homing optimizations for the heap are mostly (though not fully) automatic. As mentioned above, users must link their programs against our optimization

[3]The chunking factor may not be an integral number of pages. Mspaces permit specifying a separate chunking factor per tile in the distribution. This allows placement of the majority of a distributed array's elements on the optimal tile.

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		

Table I
SPLASH2 BENCHMARKS USED IN OUR STUDY ALONG WITH THEIR INPUT PROBLEM SIZES.

library. In addition, they must call our library initialization routines which requires adding 4 lines of code to their program. Aside from this, there are no additional source code changes needed to apply our heap optimizations.

C. *Homing Static Data Pages*

Unlike heap objects, static data objects are allocated at compile time, and are bound to a particular mspace. Hence, they are already assigned a home by the time a program begins execution. Similar to the heap, the default policy is to home all pages from the static data region on tile 0.

To control page homing in the static data region, we use memory mapping and unmapping to change the homing policy from the default policy. In particular, we 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. Similar to mspaces, the mmap_mbind() system call permits specifying a home tile for the mapped pages. Hence, this permits per-page homing control.

In our current implementation, we determine the pages to optimize in the static data region manually, and insert the unmapping and mapping calls manually into the program source code. However, due to the systematic nature of these analyses and source code instrumentation, we believe it is possible to automate them in the future.

IV. Experimental Results

This section demonstrates the profiling and optimization techniques discussed in Sections II and III, and studies the potential benefits they can provide. In particular, our experiments quantify the number of remote L2 slice references that are converted into local L2 slice references by the page homing optimizations. We begin by discussing experimental methodology in Section IV-A. Then, Section IV-B presents our results.

A. *Experimental Methodology*

We conducted all 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 [16] except for volrend. We used `tile-cc` (the Tile Processor's C compiler) to compile the benchmarks with the highest level of optimization. Table I lists the benchmarks and the input problems we used in the experiments.

Unfortunately, we encountered some bugs in our page homing code that prevented us from running with a large number of cores. At the time of writing this paper, we were unable to perform profiling and optimized runs on more than 32 cores for a number of SPLASH2 benchmarks. So, we only report experiments on at most 32 cores.

For each benchmark binary, we acquire access pattern profiles and malloc logs using the profiling tools described in Section II-C. All of our profiles are acquired on 32-core executions of the benchmarks. Then, we instrument the benchmark source codes to call our optimization library initialization routines and to perform the homing optimizations for the static data region, as discussed in Sections III-B and III-C. Lastly, we re-compile the benchmarks, linking them against our optimization library, and run them to measure optimized performance. These optimized runs use the same configurations as the profiling runs.

To quantify improvements, we compare the optimized and unoptimized benchmarks. As discussed in Sections III-B and III-C, the default homing policy places all pages in the original unoptimized benchmarks on tile 0. To provide a better baseline against which to compare our techniques, we link the unoptimized benchmarks against our optimization library, but configure the system to distribute all heap and static data pages across tiles with a chunking factor of 1. This utilizes the DSC capacity fully, but randomly distributes pages across the on-chip L2 slices.

In our results, we report sampled page references at the DSC level. Since we use a sampling frequency of 7000, sampling counts can be converted into page reference counts (at least approximately) by multiplying by 7000. (The selection of a proper sampling frequency is important. If the sampling frequency is too small, profiling will incur large overhead; but if the sampling frequency is too large, less frequent events may not be sampled. Our choice of 7000 for the sampling frequency was determined experimentally, and works well for SPLASH2 benchmarks. It may be necessary to tune this sampling frequency parameter for other benchmarks.) Lastly, we only report measurements in the parallel region of each benchmark. We exclude program initialization, which is performed at the beginning of each SPLASH2 benchmark on a single core.

B. *Physical Locality Results*

Table II reports our page reference count results. In particular, the 2^{nd} and 3^{rd} columns of Table II (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, respectively. This data shows that across our benchmarks, heap objects receive more memory references than objects in the static data region,

	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

Table II
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.

but both types of objects are important. (One case with anomalous behavior is LU which we will discuss shortly).

The 4th and 5th columns of Table II (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 6th column of Table II reports the percentage of the total sampled references that these baseline local references represent–*i.e.* $(\%\ Total)_{Baseline} = \frac{(Heap+Static)_{Baseline}}{(Heap+Static)_{Total}} \times 100$. This data shows the unoptimized benchmarks exhibit poor physical locality. Only 5%–8% of all DSC references are to local L2 slices. In other words, more than 90% of DSC references must traverse the on-chip network to communicate with a remote L2 slice. This makes sense because page homing in the unoptimized benchmarks is essentially randomized across the Tile Processor's DSC.

The 7th and 8th columns of Table II (labeled "Optimized") report the number of sampled page references in the optimized benchmarks that are destined to local L2 slices broken down into heap and static data references, respectively. The 9th column of Table II reports the percentage of the total sampled references that these optimized local references represent–*i.e.* $(\%\ Total)_{Optimized} = \frac{(Heap+Static)_{Optimized}}{(Heap+Static)_{Total}} \times 100$. 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 columns in Table II provide insight into how much of the potential physical locality in our benchmarks we actually exploit. Since our homing optimization must place each page on a specific tile, it is only effective for pages that are referenced by a small number of cores. In particular, pages that are shared by most/all of the cores in the machine are unlikely to yield any benefit. The 10th and 11th columns of Table II (labeled "Potential") report the number of samples destined to pages in the heap and static data regions, respectively, that are referenced by *no more than half the cores* (*i.e.*, 16 cores) in the profiling runs. Although some of these pages can still be "widely shared," we believe these sampled reference counts are a good estimate for the potential physical locality improvement.

Comparing the "Potential" and "Optimized" results in Table II, 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 heap and static data counts. (In some cases, the optimized counts are actually larger than the potential counts. These are due to references destined to local L2 slices for pages shared by more than half the machine.) The greatest missed potential is in FMM where there are a large number of heap references none of which are optimized. There is also some missed potential in Water-SP. But overall, our homing optimizations are fairly comprehensive.

These results suggest that for our optimizations to do substantially better, we must create more opportunities for homing. Comparing the last two columns against the 2nd and 3rd columns of Table II, we see there is a significant discrepancy between the potential and total sampled reference counts, especially for pages in the static data region. This implies there are a large number of references to pages shared by most of the machine. Upon closer examination, we found a major reason for this is false sharing induced by the Tile Processor's large page size, 64 KB. We believe our optimizations can become more effective if page size is reduced.[4] For pages with false sharing, a smaller page size can create more pages with low-degree sharing that our optimizations can exploit.

Finally, Table II shows LU cannot be optimized because it does not exhibit any sampled references. This is due to the fact that LU performs function calls very frequently. The calls are so frequent that the interrupt handler skid after a remote-read event almost always straddles a function call (*i.e.*, all interrupts sample the called code). Unfortunately, our current binary analysis tool cannot analyze across func-

[4]In fact, the Tile Processor's TLBs can support smaller pages, but the current OS doesn't exploit this hardware feature.

tions, so we fail to identify any of the event-triggering loads in LU. We verified by hand that LU does indeed present significant opportunities for our homing optimizations. In the future, we plan to support inter-procedure analysis in our binary analysis to handle cases like LU.

V. 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] Y. Hoskote, S. Vangal, N. Borkar, and S. Borkar, "Teraflop Prototype Processor with 80 Cores," in *Proc. of the Symp. on High Performance Chips*, 2007.

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

[3] "Silicon Industry Association Technology Roadmap," 2009.

[4] 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.

[5] B. M. Beckman and D. A. Wood, "Managing Wire Delay in Large Chip-Multiprocessor Caches," in *Proc. of the 37th Int'l Symp. on Microarchitecture*, Portland, OR, December 2004, pp. 319–330.

[6] J. Chang and G. S. Sohi, "Cooperative Caching for Chip Multiprocessors," in *Proc. of the 33rd Int'l Symp. on Comp. Arch.*, June 2006.

[7] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Optimizing Replication, Communication, and Capacity Allocation in CMPs," in *Proc. of the 32nd Int'l Symp. on Comp. Arch.*, Madison, WI, June 2005.

[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] 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, October 2008.

[10] 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.

[11] M. Zhang and K. Asanovic, "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors," in *Proc. of the 32nd Int'l Symp. on Comp. Arch.*, Madison, WI, June 2005.

[12] S. Cho and L. Jin, "Managing Distributed, Shared L2 Caches through OS-Level Page Allocation," in *Proc. of the 39th Int'l Symp. on Microarchitecture*, December 2006.

[13] 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, pp. 184–195.

[14] L. Jin and S. Cho, "SOS: A Software-Oriented Distributed Shared Cache Management Approach for Chip Multiprocessors," in *Proc. of the 18th Int'l Conf. on Parallel Arch. and Compilation Techniques*, Raleigh, NC, September 2009.

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

[16] 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 22nd Int'l Symp. on Comp. Arch.*, Santa Margherita Ligure, Italy, June 1995.

[17] "Performance Application Programming Interface."

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

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