TCP: Thread Contention Predictor for Parallel Programs

Abstract— ***With acceptance of chip multicores on desktops and embedded platforms, parallel programs have become ubiquitous. Existence of multiple threads can cause resource contention such as, in on-chip shared caches and interconnects. This may cause a significant degradation in the execution of parallel programs. Hence, we propose a tool which quantifies the average number of threads sharing an object and the pattern in which they share it. We refer this tool as Thread Contention Predictor (TCP). This information can be calculated at the object level or at the cache line level according to its use-case. We believe this information has wide application ranging from predicting a suitable cache configuration, data mapping or predicting conflict in transaction memory. In this paper, we show its use to predict a suitable shared, last level on-chip cache configuration on a chip multicore platform. Our cache configuration predictor is 2.33x faster in simulation speed than the cycle-accurate simulator. We also demonstrate use of TCP to identify data structures in a program which are "hot" and may cause performance degradation. We manually fix layout of such data structures and show up-to 10% and 18% improvement in execution time and energy-delay product of the application.***
Keywords: parallel programs, thread contention, chip multicore caches

I. Introduction

Ever increasing demand for performance, has caused proliferation of chip multicores (CMPs) on desktop, mobile computing and embedded platforms. As a result, parallel programs have become ubiquitous to take advantage of parallelism offered by CMPs. However, writing a parallel program though looks straight-forward, is a very complicated task. This is because of deadlocks, livelocks caused by synchronization variables. Lot of tools are available to debug such problems. But not many tools are available which will help programmers to identify "hot" data structures in their programs or which can quantify the average number of threads sharing an object or contention caused by an object. Such tool will help programmers to identify bottlenecks in their programs and re-write it accordingly. Our thread contention predictor (TCP) tool fills this gap. Following are our contributions in this paper:

1) First, we introduce a term to measure the average number of threads sharing an object. We call it as a "sharing index". However, an object with higher value of sharing index does not mean that, that object will degrade execution of a parallel program. Hence, we also introduce another term called a"contention index" to measure contention that might be caused by threads accessing that object.
2) CMPs use shared caches as the last level cache (LLC) to increase on-chip cache utilization. Kim et al. [1] proposed SNUCA and DNUCA cache access policies to access the large on-chip cache. We demonstrate the application of TCP to predict a suitable cache access policy for a given application using a single time cycle accurate simulation. Our model is 2.33 times faster than the cycle-accurate simulator.
3) We also show the use of sharing and contention indexes of frequently accessed addresses in a program to identify false sharing caused in it. On manually fixing such data structures, program execution showed up-to 10% improvement in the execution time and 18% improvement in energy-delay product of the application on a cycle accurate.

Our paper is organized as follows: Section II describes sharing and contention indexes of an object in TCP. The use of sharing and contention indexes to predict a suitable on-chip cache is described in section III. We refer this tool as cache configuration predictor(CPP). Section IV describes experimental setup and results of our CPP tool. Section V-B elucidates use of TCP in identifying false data sharing in a program. Section VI describes related work. Finally we conclude our paper in Section VII.

II. TCP

TCP can evaluate sharing and contention indexes at the level of cache line addresses or individual data addresses or objects as well. Hence, we will use the term object to emphasize its flexibility. Currently, we have implemented this tool with an in-house cycle accurate simulator but this can easily be ported to a freely available dynamic binary instrumentation tool such as, Pin[1]. For every object, we track the number of times each thread accesses it. This is used to evaluate a sharing index of that object.

A. Sharing Index (SI)

We define the sharing index of an object as the average number of threads accessing it. TCP keeps track of the number of times each thread accesses an object. It determines sharing index using Eq. (1), where, P_i is probability of thread i accessing that object. SI has a resemblance to the term, "entropy", widely used in information theory. Shannon[2] define entropy as in Eq. 1 to denote a

[1]Pin: Building customized program analysis tools with dynamic instrumentation, PLDI, 2005

measure of information present in a message or the rate at which information is produced by various sources$(1, 2.., T)$. $P_1, P_2....P_T$ are probabilities of T sources generating events. We borrow entropy formulation to denote average number of threads accessing a variable.

$$Entropy = -\sum_{0 \le i < T} P_i.\log(P_i) \tag{1}$$

$$\log(SI) = -\sum_{0 \le i < T} P_i.\log(P_i) \tag{2}$$

$$SI = 2^{\log(SI)} \tag{3}$$

An object with higher SI denotes more number of threads access it. For private objects such as local variables declared on the stack, $P_i = 1$ and $\log(P_i) = 0$, for the thread which accesses that object. For rest of the threads, $P_i = 0$. Hence, SI is equal to 1 which tallies with the fact that only one thread accesses that object. On contrary, consider all threads make equal number of accesses to an object. So $P_i = 1/T$ for all threads, where T is the total number of threads present in an application. In this case, $SI = T$. Hence, this formula captures the notion of number of threads accessing an object. However, it does not capture interleaving of accesses made by different threads. This is captured by our next term, "contention index".

B. Contention Index (CI)

Objects with higher SI, may not become performance bottleneck if accesses made by all threads are serialized and not interleaved. Contention index quantifies interleaving of accesses made by different threads. It is measured in terms of a runlength of accesses. A runlength of an object is the number of times the same thread accesses that object consecutively. If the object is accessed by a different thread, runlength counter starts again from zero. This is shown pictorially in Fig. 1. In Fig. 1(A) both the threads make four consecutive accesses. So there are total 8 accesses made by two threads altogether and with two runlengths of size four. Whereas, in Fig. 1(B) accesses made by two threads are interleaved. There are one runlength of 3 accesses, two runlengths of size 2 and one runlength of size 1. *We calculate weighted average of runlengths and its dispersion to determine its contention causing ability.* Smaller average runlength, can cause more contention in shared resources like cache or interconnect and vice-verse.

C. Data Filter

As explained above, SI gives the number of threads sharing an object, whereas, CI gives the pattern in which it is shared by various threads. If CI i.e. runlength is very low then it means that after a few accesses from one thread, it is intercepted by another thread. If CI of that object is improved, it may improve program execution. Improvement

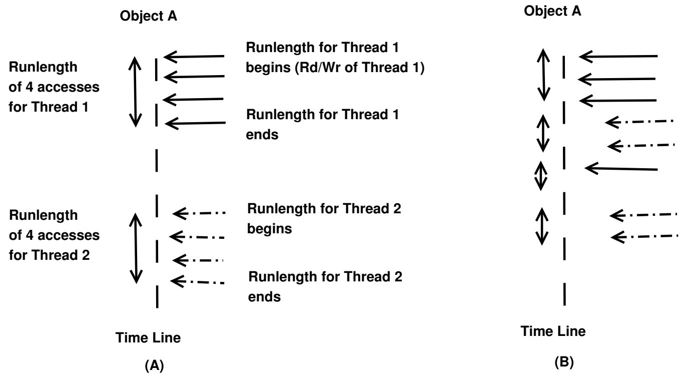

Object A
Runlength of 4 accesses for Thread 1
Runlength for Thread 1 begins (Rd/Wr of Thread 1)
Runlength for Thread 1 ends
Runlength of 4 accesses for Thread 2
Runlength for Thread 2 begins
Runlength for Thread 2 ends
Time Line
(A)
Object A
Time Line
(B)

Figure 1. Both the the threads (t_1 and t_2) make four accesses each. However, in (A), four accesses made by these threads are not interleaved. Hence, these accesses have a runlength of four. Whereas, in (B), accesses made by these threads are interleaved. There are 1 runlength of size 3, 2 runlengths of size 2 and one runlength of size 1.

depends on the actual number of times that object is used. Hence, we introduce another term, which we refer to as "Data Filter (DF)". We define DF as follows:

$$DF = \frac{N * SI}{CI} \tag{4}$$

where, N is the total number of accesses made by different threads, SI and CI are sharing and contention indexes of that object. An object is important in a program execution if it is accessed significant number of times and shared by many threads and it has more contention causing potential.

For every object, TCP also tracks instruction addresses which accessed those objects. These objects can be mapped back to data structures using information found in the executable, instruction addresses and by intercepting malloc calls. Depending on frequency of accesses, "hot" data structures can be determined. This can help programmers to re-organize data structures so that its SI and mainly CI improves. This can also tell whether changes in lock granularity might improve CI. Using this information, programmers can also determine the number of threads created in a data parallel loop. Such decisions can be taken with an offline analysis. In section V-B, we demonstrate how our tool can be used to determine false sharing caused due to various members of a large structure in a multi-threaded application. First, we describe the application of TCP to predict a suitable cache configuration for CMPs.

III. CACHE POLICY PREDICTOR (CPP)

Due to advances in the technology, the number of cores present on the CMP has increased and so has the size of on-chip cache. As a result, large caches are manufactured using smaller banks for power and performance reasons. However, such cache offers variable latency to the cores present on the CMP [1]. Kim et al. [1] proposed two major cache access policies for such dispersed caches, namely, static nonuniform cache access (SNUCA) and dynamic nonuniform cache access (DNUCA). In SNUCA, predetermined bits of the memory address determine the bank in which data is cached. Whereas, in DNUCA, the whole address space is mapped onto a single column and predetermined bits decide the row in which data is cached. Data can be present in any of the banks in that row. These banks form a "bankset". On an L1 miss, L1 first checks data in the nearest L2 bank and if it is a miss, then rest of the L2 banks in that bankset are searched. If data is not present in any of these banks, then it is read in the nearest L2 bank from the offchip DRAM. In DNUCA, as private data is cached in the nearest bank, it offers lesser latency. On the other hand, in SNUCA, data could be in farther bank even if it is private, offering higher latency than that in DNUCA. In case data is shared and is present in the farther bank, then in DNUCA, data migrates gradually towards the accessing core at runtime. However, if data is shared by many threads, it might migrate in conflicting directions, incurring higher latency. As the set spans across multiple banks in a row, traditional replacement logic cannot be applied in DNUCA.

In summary, though DNUCA offers lower access latency, it suffers from a drawback of complex lookup and replacement logic. On the other hand, SNUCA has simple lookup logic but may offer higher cache access latency. Hence, at design time, architects have to make a careful decision between SNUCA and DNUCA policies. Performing cycle-accurate simulation of many workloads is time consuming. Therefore, we solve this problem by making use of data collected by TCP with a one-time cycle-accurate simulation on SNUCA platform. Here, SI, CI are evaluated per cache line address level. We call this approach as cache configuration predictor (CPP). Architects can use CPP to quickly make a choice between DNUCA and SNUCA for a given application.

We consider a scalable tiled architecture in this study which is shown in Fig. 2. In tiled architecture, tiles are replicated and connected through an on-chip switched-network (NoC). Each tile has a core, a private L1 instruction and data cache, L2 cache and a router. In our implementation, L2 cache is distributed across all tiles and it is shared by all cores. To maintain cache coherence between private L1 caches, a directory information is present in each tile. These tiles are connected to one another via 2D-mesh NoC and per-tile router.

In DNUCA, data may migrate in conflicting directions if it is shared by many threads at the same time. Hence, interleaving of accesses made by different threads determines cache access latency. If thread t_1 reads data for the first time then it is read in the nearest L2 slice from the offchip DRAM. Then onwards, it finds data in its nearest L2 slice on making consecutive accesses to that address. Thus achieving lower L2 access latency. If some other thread t_2 accesses the same data, then it will not find data in its

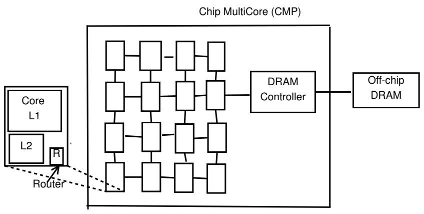


Figure 2. Tiled CMP used for experimentation.

nearest L2 slice. It has to search rest of the L2 slices in a bankset. However, slowly data migrates towards its nearest L2 slice and then it also enjoys lower L2 access latency. Such a scenario is possible in case one thread initializes data and then hands it over to its helper thread for further processing. In this case since both the threads enjoy lower L2 access latency, DNUCA is preferable. However, in case of a synchronization variable or a barrier, where multiple threads access it at the same time, most of the threads have to search all the peer L2 slices in a bankset, incurring a lot of overhead. But instead, if SNUCA policy is used for such addresses, then all of them make a single access to the "home location" of that address, though it is farther than their nearest L2 slice.

To determine penalties incurred by an application with these two policies, we make use of SI and CI estimated by TCP, using a cycle-accurate simulation with SNUCA policy. For every cache line address, we determine total number of accesses made by each thread to that address and maintain runlength statistics per thread.

A. Estimation of Overhead in DNUCA and SNUCA

The meaning of the various terms used in CPP model is explained in Table I.

As an example, in fig. 1(A), threads t_1 and t_2 execute on cores 0 and 1 respectively. There are four L2 slices in a row. Suppose, distance between t_1 and four L2 slices in a bankset is 1, 2, 3 and 4. We make the same assumption for t_2. In SNUCA, data is cached in its "home" location, decided by some predetermined bits from its memory address. Consider distance of home L2 slice from t_1 and t_2 is 3($D_{1\_Home} = 3$) and 4 ($D_{2\_Home} = 4$), respectively. Assuming, data is already read from offchip DRAM, since each thread makes four accesses to L2 home slice in Fig. 1(A), SNUCA cost is 28 (4*3 + 4*4).

In DNUCA, as a worst case situation, we assume, for beginning of every runlength, data is not present in the nearest L2 slice for all threads. This is because, since previous access is made by some other thread, data might be present in the nearest tile of that thread. So we assume that for the beginning of every runlength of all threads, they search data in all peer L2 slices in a bankset. If DNUCA cost, calculated using this assumption is less than SNUCA cost, then DNUCA definitely will give better performance

Table I
Table gives the meaning of various terms, used in CPP model

Parameter	Description
t_i	thread executing on core i
T	total # of threads present in an application
A_{ij}	total # of accesses made by thread i to cache line address (CLA) j
N	total # of data cache line addresses
K	runlengths of size $0, 1, ..K$ tracked during one-time simulation on SNUCA. Runlengths of size equal to and greater than K are counted by $(K-1)^{th}$ array entry.
r_{ijk}	# of times thread i exhibits runlengths of size k of an address j
D_{ip}	distance between L1 in tile i and L2 slice in tile p where data can be cached in DNUCA
P	total # of peer L2 slices in a bankset in DNUCA
$D_{i\_Nearest}$	distance between L1 in tile i and its nearest L2 slice where address can be cached in DNUCA
$D_{i\_Average}$	average distance between L1 in tile i and all L2 slices in a bankset where address can be cached in DNUCA
D_{i_Home}	distance between L1 in tile i and "home" tile, where data is cached in SNUCA

for that application. Hence, in the above example, as distance from peer L2 slices in a bankset where data can be cached is 1, 2, 3 and 4, for both the threads, cost required to search in DNUCA for the first access in the runlength, is 10((1+2+3+4)*1) each. To determine distance from the L2 slice where data can be found for rest of the accesses, we evaluate SI and CI of aggregate statistics of all threads. From Eq. (1), SI is 2 in this case, which is obvious since both the threads make equal number of accesses to this address. CI (weighted average runlength) is four. We consider distance between from the nearest L2 slice for rest of the accesses in a runlength, if either SI is 1 or weighted average runlength is greater than 2. In this case, distance between thread and its nearest L2 slice is 1. Hence, DNUCA cost for rest of three accesses of each thread is (3*1=3). Total DNUCA cost = 2*3+2*10 = 26, which is less than that obtained for SNUCA, which is 28. Hence, for this application we conclude that DNUCA policy is better.

Now let us find DNUCA cost for Fig. 1(B). There are 1 runlength of size 1, 2 runlengths of size 2 and 1 runlength of size 3. So total cost required to search data for the first access of every runlength in peer L2s is 40 ((1+2+3+4)*4). In this case, the weighted average runlength (CI) is 2. Hence we consider average distance for rest of the accesses in all runlengths, which is 2.5 ((1+2+3+4)/4). Cost required for rest of the accesses in runlengths is 10 (2.5*(2+2)). Total DNUCA cost is 50 (40+10). Hence, in Fig. 1(B) SNUCA will perform better that DNUCA.

In this application of TCP, we evaluate SI and CI by aggregating statistics of all thread executing on the cores belonging to the same column. This gives us SI in terms

Algorithm 1 Cache Policy Predictor

```
1: Evaluate runlength and total number of accesses made
   by each thread to all data addresses using a cycle
   accurate simulator with SNUCA
2: Evaluate SNUCA cost using Eq. (5)
3: for first accesses of all runlengths of thread and address
   pair do
4:     evaluate peer search cost using Eq. (6)
5: end for
6: for rest of the accesses in runlengths of thread and
   address pair do
7:     Evaluate SI and CI per column for each address
8:
9:     if $SI == 1 || CI \geq 3$ then
10:        Estimate cost using Eq. (7)
11:    else
12:        Estimate cost using Eq. (8)
13:    end if
14: end for
15: Obtain total $DNUCA_{cost}$ using Eq. (9)
16: $CostRatio$ = $DNUCA_{cost}$/$SNUCA_{cost}$
17: if $CostRatio - lt1$ then
18:     DNUCA is a suitable policy for this application
19: else
20:     SNUCA is a suitable policy for this application
21: end if
```

of columns and weighted average runlength of a column. This is because, two threads even if they are executing on different cores, but belonging to the same column (see Fig. 2), have same nearest L2 slice, which is L2 slice present in that column. CPP procedure explained above is given in Algorithm 1.

We use cycle-accurate simulator to obtain runlength and total accesses made by each thread. We consider data addresses missed in L1 alone, to obtain time spent in SNUCA and DNUCA. This is because, instruction addresses usually show very good spatial and temporal locality. So a very few instructions misses are served by lower level unified cache. Same is true for data addresses showing good locality. Time spent in transit in SNUCA by thread i while accessing an address j is estimated using Eq. 5.

$$SNUCA_{cost} = \sum_{0 \leq i < T} \sum_{0 \leq j < N} A_{ij}.D_{i\_Home} \qquad (5)$$

For DNUCA, as explained above, we evaluate cost separately for first accesses in every runlength and remaining accesses in every runlength. Eq. (6) estimates cost for a thread in tile i, accessing address j, when all peer L2 slices have to be searched, which is done for the first access of every runlength of all threads.

$$PeerSearchCt_{ij} = \sum_{0 \leq p < P} \sum_{0 \leq k < K} r_{ijk} * D_{ip} \qquad (6)$$

For rest of the references made by each thread, which are not the beginning of a runlength, we first determine SI and CI by aggregating statistics of all threads belonging to the same column. If SI of an address is 1 which is true when all threads accessing that address belong to the same column, then we use distance between the thread and nearest L2 slice ($D_{i\_Nearest}$). We also use $D_{i\_Nearest}$, if CI is greater than or equal to 3. Clearly, if accesses are mostly private or done through a single column, then runlengths are longer is size. Threads in that column will find data in its nearest L2 slice. Eq. (7) estimates cost of the remaining accesses made by thread i to an address j, which lesser contention causing potential.

$$NearSearchCt_{ij} = (A_{ij} - \sum_{0 \leq k < K} r_{ijk}) * D_{i\_Nearest} \quad (7)$$

However, if SI is greater than 1 or CI is less than 3, then most of the threads will have to search data in all L2 slices for the remaining accesses in a runlength. Hence, we use average of distance for all L2 slices in a bankset ($D_{i\_Average}$). If average runlength is less than 3, then the term $(A_{ij} - \sum_{0 \leq k < K} r_{ijk})$ in Eq. (8) is negligible. $PeerSearchCt_{ij}$ in Eq. (7) contributes majority of penalty in this case. Eq. (8) evaluates time spent in transit by rest of the accesses in a runlength, for addresses causing more contention.

$$AvgDistanceSearchCt_{ij} = (A_{ij} - \sum_{0 \leq k < K} r_{ijk}) * D_{i\_Average} \quad (8)$$

Total time spent in accessing data by DNUCA policy is given by Eq. (9).

$$DNUCA_{cost} = \sum_{0 \leq i < T} \sum_{0 \leq j < N} (PeerSearchCt_{ij} + NearSearchCt_{ij} + AvgDistanceSearchCt_{ij}) \quad (9)$$

If $DNUCA_{cost}$ is lesser than $SNUCA_{cost}$ for an application then DNUCA is more suitable policy for that application and vice verse.

IV. Experimental Configuration

A. Applications used in Experiments

We evaluate multi-threaded workloads with one-to-one mapping between threads and cores (Table II)[2]. We have skipped initial serial portion and simulate only parallel section in all the test cases. We test all workloads with 16 threads and execute 1B instructions.

[2]Rest of the PARSEC benchmarks either use OpenMP APIs or libraries which are not supported by SESC compiler. Hence remaining benchmarks cannot be compiled using SESC compiler.

Name	Description, WSS(L/M/S)
Alpbench Benchmark [3]	
mpegenc	Encodes 15 Frames of size 640x336, M
mpegdec	Decodes 15 Frames of size 640x336, M
Splash2 Benchmark[4]	
cholesky	blocked sparse matrix factorization on tk29, L
fft	FFT on 1M points, M
lu (noncontinuous)	1024x1024 LU matrix factorization, S
radix	Radix sort on 1M keys, M
fmm	simulate interaction of 16K bodies system, M
water_spatial	simulation of 512 water molecules, M
water_nsquared	simulation of 512 water molecules, M
barnes	Barnes-Hut method on 16K bodies, M
ocean (continuous)	512x512 grid points, L
PARSEC Benchmark[5]	
blackscholes	SimLarge i/p, Financial Domain, S
swaptions	SimLarge i/p, Financial Domain, M
fluidanimate	SimMedium i/p, Animation, M
x.264 Encoder	SimLarge i/p, Media Domain, M

Table II
TABLE SHOWS APPLICATIONS USED FOR STUDY AND THEIR WSS INFORMATION(L:LARGE, M:MEDIUM, S:SMALL).

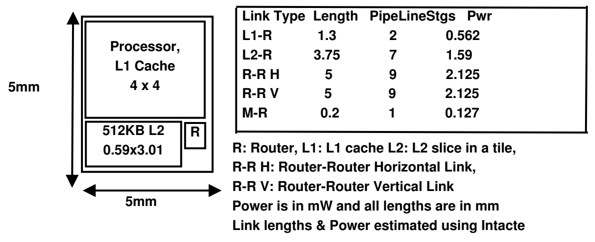
B. Experimental Setup And Methodology

We model all the system components with reasonable accuracy in our framework. We use SESC [6] to simulate a core, Ruby component from GEMS [7] to simulate the cache hierarchy and interconnects. DRAMSim is used to model the off-chip DRAM. DRAMSim uses MICRON power model to estimate power consumed in DRAM accesses. Intacte [8] is used to estimate low level parameters of the interconnect such as the number of repeaters, wire width, wire length, degree of pipelining and power consumed by the interconnect. Power consumed by the cache components is estimated using CACTI 6.0 [9].

In order to estimate the latency (in cycles) of a certain wire, we estimate area of all components in a tile and then create the floorplan which is shown in Fig. 3. We make following assumptions to determine area of various components at 32nm technology and 3GHz frequency:

- core : This is estimated based on the area of Intel Nehalem core [10]
- cache : The L1 cache is of size 32KB whose area is very small and is included in the processor area. The area occupied by the L2 cache is obtained using CACTI 6.0. We assume directory information is stored along with each L2 slice. We conservatively assume area of per-tile directory to be negligible. If directory area is considered then interconnect lengths will increase which is more beneficial for the remap policy.
- router : The area of the router is assumed to be quite negligible at 32nm.

Fig. 3 also shows wire lengths and their power consumption. The latency of a link in clock cycles is equal to the number of its pipeline stages. To obtain power consumption of NoC, we compute the link activity and coupling factors of all links, caused due to the messages sent over NoC.



Link Type	Length	PipeLineStgs	Pwr
L1-R	1.3	2	0.562
L2-R	3.75	7	1.59
R-R H	5	9	2.125
R-R V	5	9	2.125
M-R	0.2	1	0.127

R: Router, L1: L1 cache L2: L2 slice in a tile,
R-R H: Router-Router Horizontal Link,
R-R V: Router-Router Vertical Link
Power is in mW and all lengths are in mm
Link lengths & Power estimated using Intacte

Figure 3. Floorplan of a tile with 512KB L2 slice.

Core	out-of-order execution, 3GHz frequency, issue/fetch/retire width of 4
L1 Cache	32KB, 2 way, 64 bytes cache line size, access latency of 2 cycles (estimated using CACTI), private, cache coherence using MOESI protocol
L2 Cache	512KB/tile, 16 way, 64B line size, 4 subbanks per slice, 3 cy. latency (estimated using CACTI), noninclusive, shared and distributed across all tiles
Directory	Tag bits of L2 cache line include full bitmap for L1 sharers A separate table of 3000 entries maintains dir info. for cache lines not cached in L2 but only in L1s.
Interconnect	16 bits flit size, 4x4 2D MESH, deterministic routing, 4 virtual channels/port, credit based flow control, router queues with length of 10 buffers
DRAM	offchip, 4GB, DDR2, 667MHz freq, 2 channels of 8B in width, 8 banks 16K rows, 1K columns, close page row management policy

Table III
SYSTEM CONFIGURATION USED IN EXPERIMENTS

C. Simulation Procedure

Table III gives the system configuration used in our experiments. The simulation procedure includes computing the area of tile components, computing link lengths and low level link parameters using Intacte and then performing simulation. Our simulator estimates the activity and coupling factors of all the links. Intacte determines power dissipated in NoC using these activity factors. Power consumed by the off-chip DRAM and on-chip cache is estimated using DRAMSim (MICRON) and CACTI power models, respectively.

V. RESULTS

A. Evaluation of CPP Model

We evaluate time spent in transit if SNUCA or DNUCA is used for applications using Algorithm (1). Fig. 4 plots DNUCA cost normalized with respect to SNUCA cost. Fig. 4 also shows normalized time spent in transit obtained using simulation. Our model evaluates higher DNUCA cost for applications like mpegenc, mpegdec and raytrace. These applications show 6%, 8% and 25% degradation in their execution time, respectively. For applications like ocean, blackscholes and X.264, cost ratio predicted by TCP is less than 1, which also tallies with our experimental results. These applications show 8%, 4% and 2% improvement in execution time with DNUCA over SNUCA. Most of the accesses in these applications are private. X.264 has very poor thread scalability, as a result, with DNUCA, it can cache data in nearer L2 slice, giving lower L2 access latency. Fig. 4 also shows access latency for these applications. For other applications like, lu, fmm etc. TCP estimates higher cost for DNUCA. L2 access latency for these application is higher in DNUCA than SNUCA. However, execution time does not show large degradation. This is because, degradation in execution time depends on the percentage of load/store instructions compared to other type of instructions. We simulate out-of-order type of execution, hence, L2 access latency gets overlapped with execution of other instructions. CPP predictions are correct for all applications which we have considered for experimentation. This tool can be used by system architects to decide among DNUCA and SNUCA cache policy for their workloads while designing a system. Fig. 5 plots the simulation speed-up obtained with CPP over cycle-accurate simulation of SNUCA and DNUCA policies. On an average, CPP is 2.33x faster than the cycle accurate simulation.

B. TCP to determine false sharing

For complex multi-threaded applications, where the number of lines in a program is very large, it is difficult to determine false sharing between threads statically. Static analysis is conservative, hence it might pad too many dummy fields in a large data structure to avoid false sharing. We observed experimentally, that due to conservative extra padding, instead of improving execution time of an application, in many cases, it increases the number of offchip DRAM accesses and also working set size of an application. DRAM access latency is much higher than time might be spent in cache coherence messages, induced due to false data sharing. Hence, accurate estimation of contention causing potential of false sharing is important to improve performance. We use SI and CI evaluated by TCP to determine data addresses causing too many cache coherence overhead. Higher SI indicates more data sharing. Such addresses might degrade performance if CI for them is also low. Lower CI means lower average runlength of accesses made by different threads. This means that accesses made by one thread are intercepted by another thread. We automatically filter addresses with SI greater than 8 and CI less than 2 and also with considerable value (in ten thousands) of data filter (see section II-C. Our tool also gives information of instruction addresses which accessed these data addresses. Considering information obtained in an executable and trapping malloc calls of these data addresses, we could determine the culprit data structures.

Raytrace application in Splash2 [4] benchmark suite allocates a global gmem structure which has many locks and a barrier as its members as shown in Fig. 6(a). In this case, barrier start, pidlock and ridlock get allocated in the

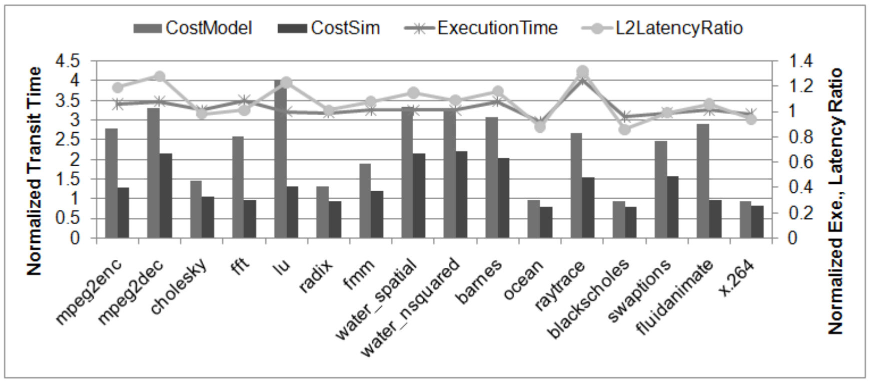


Figure 4. **Accuracy of CPP Model**

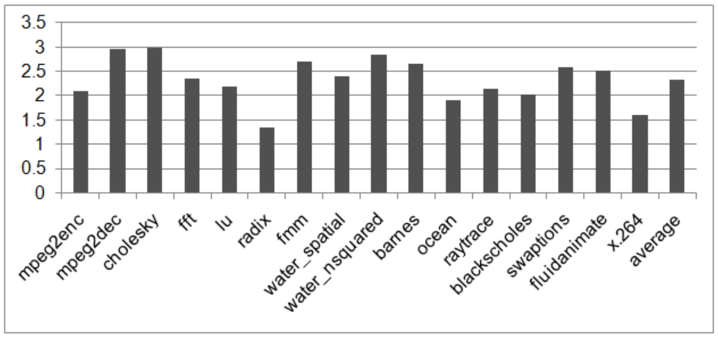


Figure 5. **Graph shows simulation speed-up obtained by CPP over cycle-accurate simulation of DNUCA and SNUCA combined.**

same cache line of size 64B. Hence, we changed the gmem structure to as shown in Fig. 6(b).

We collocated pidlock and ridlock with members pid and rid, respectively, which they protect from a concurrent use. We also allocated barrier start in a separate cache line since it is heavily used in raytrace by adding dummy variables. With these changes, we could obtain a significant improvement in performance of an application. We made similar improvements in barnes application and obtained performance improvement. Table IV summarizes % energy-delay product and execution time improvement. Considering Splash2 is a very well studied benchmark suite, still we could achieve execution time improvement upto 10.7% in these application with TCP.

Table IV
Table gives % improvement achieved in various metrics with our source level changes in an application

10.7

Application	Execution Time	L2 Latency	EDP
Raytrace	10.7	23.7	18.9
Barnes	2	4.2	3.77

VI. Related Work

J. Eggers [11] introduced the notion of writelen to predict suitability between write-invalidate and write-broadcast cache coherence protocol on multiprocessor systems. Since, we treat read or write accesses equally, we rephrase the term as runlength. We determine weighted average of runlength and its dispersion to filter out addresses which do not cause contention. Addresses with lower runlength have more potential to cause resource level contention and hence users studying applications at the source level, can make use of contention index to focus their attention for improving application performance.

VII. Conclusion

In this paper, we first introduce metrics to quantify sharing pattern in a multi-threaded application. We borrow "entropy" formulation for sharing index to denote the number of threads accessing an object. Higher sharing index does not necessarily denote higher contention of that object. Hence, we next introduce contention index expressed in terms of runlength. Higher average runlength size denotes lower contention for that object. We use sharing index and contention index to estimate transit time spent with DNUCA and SNUCA cache policies. This model can

```
typedef struct gmem {
   INT nprocs; /* Number of processes. */
   INT pid; /* Global process id counter. */
   INT rid; /* Global ray id counter. */
   |
   |
   sbarrier_t start; /* Barrier for startup sync. */
   slock_t pidlock; /* Lock to increment pid. */
   slock_t ridlock; /* Lock to increment rid. */
   slock_t memlock; /* Lock for memory manager. */
   slock_t (wplock)[MAX_PROCS];
   |
} GMEM;
```

(a) Original structure definition

```
typedef struct gmem {
   INT nprocs; /* Number of processes. */
    INT pid; /* Global process id counter. */
   slock_t pidlock; /* Lock to increment pid. */
   char PAD1[40];

   INT rid; /* Global ray id counter. */
   slock_t ridlock; /* Lock to increment rid. */
   char PAD2[48];

   |
   |
   sbarrier_t start; /* Barrier for startup sync. */
   char PAD4[60];

   slock_t memlock; /* Lock for memory manager. */
   char PAD5[60];

   slock_t (wplock)[MAX_PROCS];
   |
} GMEM;
```

(b) Changed structure definition

Figure 6. **False sharing detected by TCP**

accurately predict a suitable cache configuration for an application. Results predicted by our model tallies with the experimental results and has simulation speed-up of an average 2.33x over the cycle-accurate simulator.

We also demonstrate use of TCP to determine bottlenecks in the code and find false sharing. With our changes in the code, we show up-to 10% improvement in execution time and 18.9% improvement in energy-delay product of the application. We believe this tool can also be used to solve other problems such determination of “hot” data structures, changing lock granularity, data mapping to L2 slices on a tiled architecture and evaluating parallelism bottlenecks.

REFERENCES

[1] C. Kim, D. Burger, and S. W. Keckler, “An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches,” in *ASPLOS*, 2002.

[2] C. E. Shannon, “A mathematical theory of communication,” in *Bell Systems Technical Journal*, 1948.

[3] M. lap Li, R. Sasanka, S. V. Adve, Y. kuang Chen, and E. Debes, “The ALPBench benchmark suite for complex multimedia applications,” in *IEEE ISWC*, 2005.

[4] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The SPLASH-2 programs: Characterization and methodological considerations,” in *ISCA*, 1995.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC benchmark suite: Characterization and architectural implications,” in *PACT*, 2008.

[6] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, “SESC simulator,” 2005, http://sesc.sourceforge.net.

[7] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, “Multifacets general execution-driven multiprocessor simulator (gems) toolset,” 2005.

[8] R. Nagpal, A. Madan, A. Bhardwaj, and Y. N. Srikant, “Intacte: an interconnect area, delay, and energy estimation tool for microarchitectural explorations,” in *CASES*, 2007.

[9] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, “CACTI 6.0: A tool to model large caches,” 2009. [Online]. Available: http://www.hpl.hp.com/techreports/2009/HPL-2009-85.html

[10] A. Mandke, Y. N. Srikant, and A. Bharadwaj, “Adaptive power optimization of onchip snuca cache on tiled chip multicore platform using remap policy,” no. IISc-CSA-TR-2011-02. [Online]. Available: http://www.csa.iisc.ernet.in/TR/2011/2/

[11] S. J. Eggers and R. H. Katz, “A characterization of sharing in parallel programs and its application to coherency protocol evaluation,” in *Proceedings of the 15th Annual International Symposium on Computer architecture*, ser. ISCA ’88.