TCP: Thread Contention Predictor for Parallel Programs

Aparna Mandke Bharadwaj Amrutur Y. N. Srikant
Chiranjib Bhattachryya

IISc-CSA-TR-2011-4
http://archive.csa.iisc.ernet.in/TR/2011/4/

Computer Science and Automation
Indian Institute of Science, India

November 2011

TCP: Thread Contention Predictor for Parallel Programs

Aparna Mandke, Bharadwaj Amrutur,
Y. N. Srikant, Chiranjib Bhattachryya

Abstract

With proliferation of chip multicores (CMPs) on desktops and embedded platforms, multi-threaded programs have become ubiquitous. Existence of multiple threads may cause resource contention, such as, in on-chip shared cache and interconnects, depending upon how they access resources. Hence, we propose a tool - Thread Contention Predictor (TCP) to help quantify the number of threads sharing data and their sharing pattern. We demonstrate its use to predict a more profitable shared, last level on-chip cache (LLC) access policy on CMPs. Our cache configuration predictor is 2.2 times faster compared to the cycle-accurate simulations. We also demonstrate its use for identifying *hot data structures in a program which may cause performance degradation due to false data sharing. We fix layout of such data structures and show up-to 10% and 18% improvement in execution time and energy-delay product (EDP), respectively.*

1 Introduction

Ever increasing demand for performance has caused proliferation of CMPs on desktop and embedded platforms. As a result, multi-threaded programs have become ubiquitous. Writing a multi-threaded program is a very complicated task due to use of synchronization variables. A lot of tools are available to debug data race conditions in multi-threaded programs [1]. But not many tools are available to identify *hot* data structures in a program. Such tool will help programmers identify bottlenecks in their programs. Profiling tools such as "gprof" [2] are limited to instruction profiling, but TCP profiles data accesses for a program. TCP also quantifies the number of threads sharing an object and pattern in which data is shared amongst these threads. We believe, our abstraction of data sharing properties of a program has wide applications such as determining hot data structures and predicting false sharing in a data structure. It can assist in reducing aborts in a transactional memory. This can be achieved by serializing transactions which access data that is highly *popular* among threads. It can also assist in data mapping decisions for CMPs with large distributed shared caches. In this work, we implement a model to predict more profitable cache access policy between Static and Dynamic NonUniform Cache Architectures (SNUCA, DNUCA) [3], as the last level shared distributed cache on CMPs. Researchers have proposed various trace-based[4], analytical[5] or statistical[6, 7] models to predict cache miss rate in a program. However, according to our knowledge, this is the first attempt to predict a suitable cache access policy between SNUCA and DNUCA for CMPs. Apart from this, we also show its use to predict false data sharing in a multi-threaded application. Following are our major contributions:

1. Introducing a metric to measure the number of threads sharing an object. We call it as a “sharing index (SI)”. We also introduce a metric, called “contention index (CI)” to measure contention caused by threads accessing that object.

2. Use of TCP to predict a more profitable cache access policy between SNUCA and DNUCA [3], as the last level shared cache (LLC). Our cache policy predictor is 2.2 times faster compared to cycle-accurate simulations for SNUCA and DNUCA put together.

3. Identifying frequently accessed addresses in a program that might cause false data sharing in cache, using SI and CI. On changing layout of such data structures showed up-to 10% and 18% improvement in execution time and EDP, respectively.

The paper is organized as follows: Section 2 describes prior studies done in the related area. Section 3 explains TCP model. The use of TCP to predict a suitable on-chip NUCA cache on CMP is described in Section 4. Section 5 and Section 6 give details of the experimental setup and results of our cache policy predictor. Section 6.2 demonstrates application of TCP in identifying false data sharing in a program. Finally, Section 7 concludes the paper.

2 Related Work

Harish et al.[1] proposed a tool to debug parallel programs using PIN based dynamic instruction translator. Eggers et al.[8] introduced the notion of writelen to predict suitability between write-invalidate and write-broadcast cache coherence protocol on multiprocessor systems. We extend it to define contention index. Along with a proposed sharing index, we use it to quantify parallelization characteristics of an application. David et al.[6] determine cache contention caused in shared multicore cache when two applications are co-scheduled, using their cache reuse distance determined individually. Chandra et al.[7] proposed three performance models to predict cache misses incurred in a shared L2 cache on scheduling two applications on a dual core CMP. They use stack distance or circular sequence profile to estimate additional cache misses incurred due to cache contention. Both these studies [7, 6] consider two single threaded applications on a dual core CMP. Whereas, in this study we consider multithreaded applications and quantify the sharing characteristics at the level cache line address. We show its use to determine a more profitable cache access policy between SNUCA and DNUCA.

3 TCP

TCP can evaluate SI and CI at the granularity of cache line or individual data addresses. By obtaining additional information from the binary file for an application and trapping malloc/free calls, it can be used at the data structure granularity as well. Hence, we use the term *object* to denote this.

3.1 Sharing Index (SI)

TCP keeps track of the number of times each thread accesses an object. For example, $P_1, P_2....P_T$ are probabilities of T threads accessing a single object such that $\sum_{i=1}^{T} P_i = 1$. We define the sharing index, *SI*, of an object by Eq. (2). This definition is inspired from *entropy* as used in information theory [9].

$$H(P) = -\sum_{1 \le i \le T} P_i . \log(P_i) \quad (1)$$

$$SI = 2^{H(P)} \quad (2)$$

As per this definition, $1 \le SI \le T$. The case $SI = 1$ arises when only one of the $P_i = 1$ and rest of all probabilities are 0. This happens when objects such as local variables declared on the stack, for whom $P_i = 1$ and $\log(P_i) = 0$, for the accessing thread. For rest of the threads, $P_i = 0$. Hence, $SI = 1$ tallies with the fact that only one thread accesses that object. On the contrary, for an object where all threads make equal number of accesses, $P_i = 1/T$ for all threads, where T is the total number of threads present in an application. In this case, $SI = T$. Hence, Eq. (2) captures the notion of number of threads accessing an object.

3.2 Contention Index (CI)

Objects with higher SI, may not cause performance bottleneck if accesses made by all threads are serialized and not interleaved. Hence, there is a need to quantify interleaving of accesses done by different threads. This is done by CI. We define a runlength as the number of consecutive accesses made by the same thread to an object. Runlength statistics are collected for each object. *The weighted average of runlength and its dispersion is defined as contention index.* The average runlength is smaller if accesses done by different threads are more interleaved. Fig. 1 shows serialized and interleaved accesses done by two threads, executing on separate cores. In Fig. 1(a), both the threads make four consecutive accesses. So there are two runlengths of size four. Whereas, in Fig. 1(b) due to interleaved accesses, there is one runlength of size 1 and 3 each and two runlengths of size 2. Access to objects with smaller value of CI (average runlength) can cause higher traffic in shared components like cache or interconnect.

While evaluating CI, we do not consider the number of clock cycles between the accesses. This is because, typically programs have good temporal locality. Hence, if average runlength is smaller, then accesses made by different threads occur within a short span of time and mostly would cause contention by increasing cache coherence messages. We also treat reads and writes equally in our use-cases. The contention index definition can be adapted appropriately with respect to the number of clock cycles between the accesses and read/write distinction, depending on the use-case.

3.3 Popularity Index (PI)

As explained above, SI gives the number of threads sharing an object, whereas, CI indicates the sharing pattern. On improving CI, program execution may improve. However, improvement depends on the actual number of times an object is accessed. Hence, we define another term, called *Popularity Index*

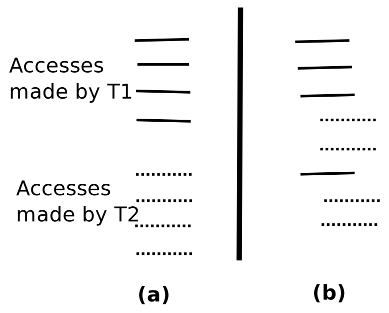


Figure 1: Threads t_1 and t_2 make four accesses each. However, in (a), four accesses made by these threads are not interleaved. Whereas, in (b), accesses made by these threads are interleaved.

(PI) as follows:

$$PI = \frac{N * SI}{CI} \tag{3}$$

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

For every object, TCP also tracks instruction address when an object is accessed. Objects can be mapped back to data structures using information found in the application binary, instruction addresses and by intercepting malloc/free calls. Depending on PI, hot data structures can be determined. This can help programmers to re-organize data structures so that their SI and mainly CI improves. Next section describes the application of TCP to predict a preferable cache access policy for CMPs.

4 Cache Policy Predictor(CPP)

Due to advances in technology, the number of cores and on-chip cache present on CMPs has increased. Hence, we study a scalable tiled architecture (Fig. 2). In a tiled architecture, tiles are replicated and connected through an on-chip switched-network (NoC). Each tile has a core, a private split L1 cache, a portion of L2 cache (L2 slice) and a router. The L2 cache is distributed across all tiles and shared by all cores. To maintain cache coherence between private L1 caches, directory information is present in each tile. These tiles are interconnected via 2D-mesh NoC and per-tile router.

For power and performance reasons, large caches are implemented using smaller banks. However, such cache offers variable latency to cores present

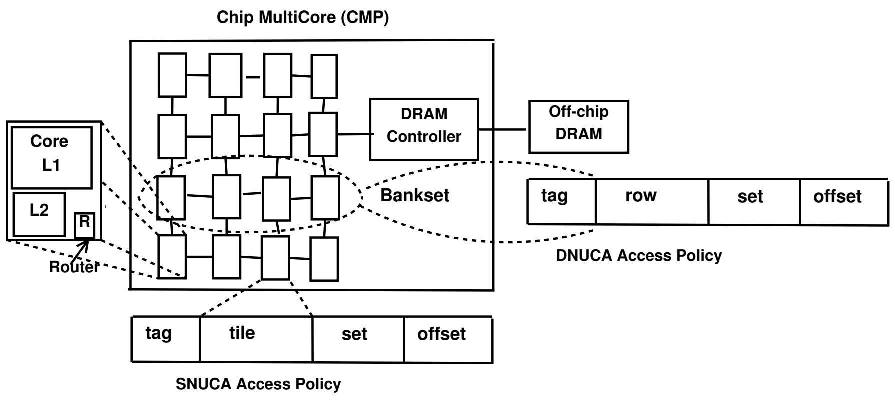


Figure 2: Tiled CMP used for experimentation.

on CMP [3]. Kim et al. [3] proposed two major access policies for dispersed caches, viz. static nonuniform cache architecture (SNUCA) and dynamic nonuniform cache architecture (DNUCA). In SNUCA, predetermined bits of memory address determine the bank in which data is cached (Fig. 2). Whereas, in DNUCA, the whole address space is mapped onto a single column and predetermined bits decide the row in which data is cached (Fig. 2). Data can be cached in any of the banks in a row, which form a “bankset”. On an L1 miss, L1 first checks data in the nearest L2 bank and then rest of L2 banks in that bankset are searched. The data is read in the nearest L2 bank from memory if it is not present in any of these banks. In DNUCA, private data is cached in the nearest bank, offering lesser latency. On the contrary, in SNUCA, private data could be in a farther bank, incurring higher latency than that in DNUCA. In DNUCA, consecutive accesses cause data to migrate in nearer bank at run-time. However, data shared by many threads, might migrate in conflicting directions, incurring higher latency. Since the cache set spans across multiple banks in a row, traditional replacement logic cannot be applied in DNUCA. We assume data is sent offchip on replacement in an L2 slice.

In summary, though DNUCA offers lower access latency, it suffers from a drawback of complex lookup and replacement logic. On the contrary, SNUCA has simple lookup logic but may suffer from higher cache access latency. Hence, at design time, architects have to make a careful choice between SNUCA and DNUCA policies. Performing cycle-accurate simulation of many workloads is time consuming. Therefore, we solve this problem by using data collected by TCP with a one-time cycle-accurate simulation on SNUCA platform. Here, SI, CI are evaluated per cache line address (CLA). We call this approach cache policy predictor (CPP).

Table 1: 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 CLAs
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 runlength of size k for 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" L2 slice, where data is cached in SNUCA
$SNUCA_{cost}$	Time spent in transit for SNUCA
$DNUCA_{cost}$	Time spent in transit for DNUCA

4.1 Estimation of Overhead in DNUCA and SNUCA

While determining penalties incurred by DNUCA and SNUCA, we assume that the data is already present in on-chip cache. This assumption is valid since the number of DRAM accesses for an application depends on its working set size and is independent of cache access policy. The meaning of various terms used in CPP model is explained in Table 1.

In DNUCA, data might 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. Consecutive accesses made by the same thread, as in Fig. 1(a), cause data to migrate gradually in nearer L2 slice, offering lower access latency. However, in SNUCA all threads 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, for every CLA, we track the total number of accesses made by each thread and maintain runlength statistics per thread, using a cycle-accurate simulation with SNUCA. 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 bankset (see Fig. 2). Suppose, distance between t_1 and four L2 slices is 1, 2, 3 and 4. We make similar assumption for t_2. Suppose, distance of home L2 slice from t_1 and t_2 is 3 ($D_{1\_Home} = 3$) and 4 ($D_{2\_Home} = 4$), respectively. Assume that data is already present in home L2 location. As each thread makes four accesses to home L2 slice in Fig. 1(a), SNUCA cost is 28 (4*3 + 4*4).

Let us assume a worst case scenario in DNUCA. At the beginning of every runlength, the data is not present in the nearest L2 slice for all threads. This is because, as previous access is made by some other thread, data might be present in the nearest tile of that thread. Hence, due to our assumption, all threads search data in all peer L2 slices in a bankset at the first access of every runlength. If $DNUCA_{cost}$, calculated using this assumption is less than $SNUCA_{cost}$, then DNUCA will definitely give better performance for that application.

As per our earlier assumption, distance from peer L2 slices in a bankset where data can be cached is 1, 2, 3 and 4. In Fig. 1(a), for both the threads, cost incurred to search for the first access in the runlength, is 10 ((1+2+3+4)*1) each. To determine distance between core and L2 slice where data will be found for rest of the accesses in a runlength, we evaluate SI and CI of a CLA. From Eq. (2), SI is 2 in this case as both threads make equal number of accesses to the CLA. CI (weighted average runlength) is four. For rest of the accesses in a runlength, we assume that data is present in the nearest L2 slice, if either SI is 1 or CI is greater than 2 (as there are 4 L2 slices in a row). In this case, distance between thread and its nearest L2 slice is 1. Hence, the cost for remaining three accesses of each thread is (3*1=3) and total $DNUCA_{cost}$=2*10+2*3=26, which is less than that of SNUCA (28). Hence, in case of Fig. 1(a) we conclude that DNUCA is preferable over SNUCA.

In Fig. 1(b), there is one runlength of size 1 and 3 each, and 2 runlengths of size 2. In this case, $SNUCA_{cost}$ remains same as the total number of accesses made by both the threads is same as in Fig. 1(a). However, for DNUCA, total cost required to search data for the first access of every runlength in peer L2s is 40 ((1+2+3+4)*4). As CI is 2 in this case, we consider average distance for rest of the accesses in all runlengths. This is because consecutive accesses gradually migrate data towards the nearest L2 slice. As the average distance is 2.5 ((1+2+3+4)/4), cost required for rest of the accesses in runlengths is 10 (2.5*(2+1+1)). Hence, $DNUCA_{cost}$ is 50 (40+10). Therefore, for Fig. 1(b) we conclude that SNUCA will perform better that DNUCA. *It should be noted that when average runlength is small, cost incurred for first accesses in runlengths is major part of the DNUCA cost.*

In this application of TCP, we evaluate SI and CI by aggregating statistics

Algorithm 1 Cache Policy Predictor

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

for all threads executing on the cores which belong to the same column. Hence, SI and CI are evaluated on a per column per object basis. This is because, even if two threads are executing on different cores, but belong to the same column (see Fig. 2), they have same nearest L2 slice, which is the L2 slice present in that column. CPP procedure explained above is described 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 evaluate SNUCA and DNUCA costs. This is because, instruction addresses show very good spatial and temporal locality. So a very few instruction misses are served by unified LLC. Same holds true for data addresses having good locality.

Total time spent in transit in SNUCA by thread i while accessing an address j is estimated using Eq. 4.

$$SNUCA_{cost} = \sum_{0 \leq i < T} \sum_{0 \leq j < N} A_{ij}.D_{i\_Home} \tag{4}$$

For DNUCA, as explained above, we evaluate costs separately for the first access and remaining accesses in every runlength. Eq. (5) 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.

$$PeerSearchCt_{ij} = \sum_{0 \le p < P} \sum_{0 \le k < K} r_{ijk} * D_{ip} \tag{5}$$

For rest of the references made by each thread, we determine SI and CI by aggregating statistics of all threads belonging to the same column. Depending on values of SI and CI, there are two cases as listed in *if-else* conditions on lines 9-12 in Algorithm 1. If SI is 1 (which is true when all threads accessing that CLA belong to the same column), then we use distance between the thread and its nearest L2 slice ($D_{i\_Nearest}$). We also use $D_{i\_Nearest}$, if CI is greater than or equal to 3 ($CI_{Threshold} = 3$ in Algorithm 1). This is because, in our experimental setup (Fig. 2), there are 4 L2 slices in a bankset. If the core makes on an average 3 or more consecutive accesses to CLA then it will find data in its nearest L2 slice. In summary, if accesses are private or done through a single column, then runlengths are longer in size and threads in that column will find data in their nearest L2 slice. Eq. (6) estimates cost for the remaining accesses made by thread i to an address j, with lesser contention causing potential.

$$NearSearchCt_{ij} = (A_{ij} - \sum_{0 \le k < K} r_{ijk}) * D_{i\_Nearest} \tag{6}$$

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 distance of all L2 slices in a bankset ($D_{i\_Average}$). If average runlength is less than 3, then ($A_{ij} - \sum_{0 \le k < K} r_{ijk}$) in Eq. (7) is negligible and $PeerSearchCt_{ij}$ in Eq. (6) contributes majority of DNUCA cost (Please refer to Table 4 for validation). Eq. (7) evaluates time spent in transit for rest of the accesses in a runlength, for addresses causing more contention.

$$AvgDistanceSearchCt_{ij} = (A_{ij} - \sum_{0 \le k < K} r_{ijk}) * D_{i\_Average} \tag{7}$$

Total time spent in accessing data with DNUCA is given by Eq. (8).

$$\begin{split} DNUCA_{cost} = \sum_{0 \le i < T} \sum_{0 \le j < N} (PeerSearchCt_{ij} + \\ NearSearchCt_{ij} + AvgDistanceSearchCt_{ij}) \end{split} \tag{8}$$

Name	Description, WSS(L/M/S)
Alpbench Benchmark [10]	
MPGEnc	Encodes 15 Frames of size 640x336, M
MPGDec	Decodes 15 Frames of size 640x336, M
Splash2 Benchmark[11]	
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[12]	
Blackscholes	SimLarge i/p, Financial Domain, S
Swaptions	SimLarge i/p, Financial Domain, M
Fluidanimate	SimMedium i/p, Animation, L
H.264 Encoder	SimLarge i/p, Media Domain, M

Table 2: Table shows applications used for study and their WSS information(L:Large, M:Medium, S:Small).

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

5 Experimental Configuration

5.1 Applications used in Experiments

We evaluate multi-threaded workloads with one-to-one mapping between threads and cores (Table 2)[1]. This assumption is in line with other work done for CMP platforms. We have skipped initial serial portion and simulate only parallel section in all the test cases. We execute 1B instructions, unless otherwise mentioned. We test all workloads with 16 threads.

[1]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.

5.2 Experimental Setup And Methodology

We model all the system components with reasonable accuracy in our framework. We use SESC[13] to simulate a core, Ruby component from GEMS[14] to simulate the cache hierarchy and interconnects. DRAMSim is used to model the offchip DRAM. DRAMSim[15] uses MICRON [16] power model to estimate power consumed in DRAM accesses. Intacte [17] 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.

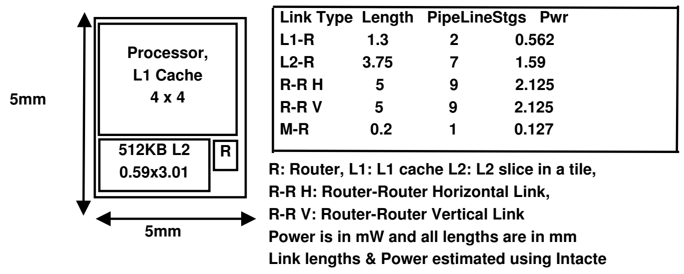
In order to estimate the latency (in cycles) of a certain wire, we estimate area of all components in a tile and then create the floorplan which is shown in Fig. 3. We make following assumptions to determine area of various components at 32nm technology and 3GHz frequency:

- core : This is estimated based on the area of Intel Nehalem core[18]
- 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.

5.3 Simulation Procedure

Table 3 gives the system configuration used in our experiments. The flow chart in Fig. 4 shows the experimental procedure. It 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



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.

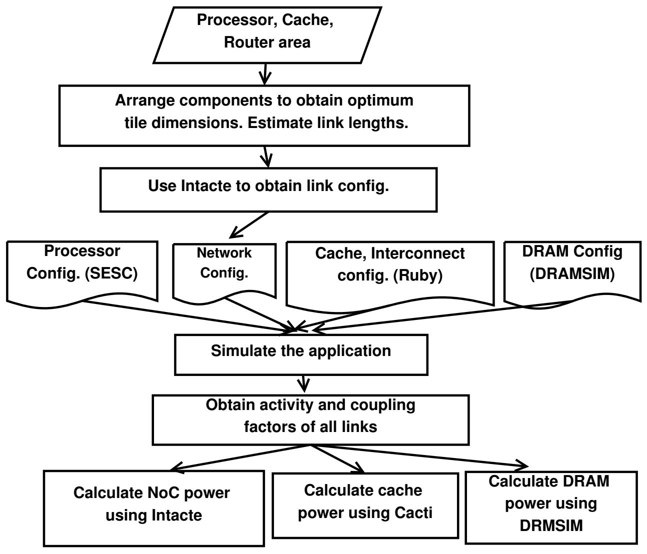

Processor, Cache, Router area
Arrange components to obtain optimum tile dimensions. Estimate link lengths.
Use Intacte to obtain link config.
Processor Config. (SESC)
Network Config.
Cache, Interconnect config. (Ruby)
DRAM Config (DRAMSIM)
Simulate the application
Obtain activity and coupling factors of all links
Calculate NoC power using Intacte
Calculate cache power using Cacti
Calculate DRAM power using DRMSIM

Figure 4: **Experimental Procedure**

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[19]), 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
Off-chip DRAM	4GB, DDR2, 667MHz freq, 2 channels of 8B in width, 8 banks 16K rows, 1K columns, close page row management policy

Table 3: System configuration used in experiments

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.

6 Results

6.1 CPP Model Validation

Table 4 shows the contribution of the individual components[2] towards the total DNUCA cost. This is obtained by normalizing these components against the total DNUCA cost (Eq. (8)). We compare costs obtained by the CPP against those obtained using simulator, which is also shown in Table 4. Cost distribution determined by CPP closely matches that obtained using simulator. For applications like, ocean and blackscholes most of the accesses are private ($SI = 1$) or are done from cores belonging to the same column, hence, $CI > 2$ for them. As a result, $NearSearchCt$ is a major component. X.264 has very poor thread scalability with uneven load balance. It creates only 3 concurrent threads even if executed with 16 threads as a command line

[2] $PeerSearchCt$ in Eq. (5), $NearSearchCt$ in Eq. (6), $AvgDistanceSearchCt$ in Eq. (7)

parameter. Hence, DNUCA allocates data near to these threads. For these applications, CPP estimates $DNUCA_{cost}$ to be lesser than $SNUCA_{cost}$.

For the rest of the applications, more data is shared by more than one threads and hence our model predicts a higher cost for DNUCA than SNUCA. For such applications, $PeerSearchCt$ contributes a majority of $DNUCA_{cost}$. Interestingly, in case of LU, even though 99% accesses are done by a single thread, CPP predicts SNUCA as profitable over DNUCA. In case of LU, all data addresses have CI less than 3. This is due to low L1 miss rate. Hence, once data is read into L1, it is rarely accessed from L2.

Name	CLA Distribution				CPP Costs			simulator Costs	
	$SI = 1$	$SI > 1$	$CI < 3$	$CI \geq 3$	PeerSCt	NearSCt	AvgSCt	PeerSCt	NearSCt
mpegenc	0.47	0.53	0.63	0.39	0.92	0.06	0.02	0.85	0.15
mpegdec	0.56	0.44	0.32	0.68	0.96	0.02	0.02	0.93	0.07
cholesky	0.71	0.29	0.47	0.53	0.66	0.31	0.03	0.76	0.25
fft	0.5	0.49	0.66	0.34	0.9	0.09	0	0.81	0.19
lu (noncontinuous)	0.99	0.01	1	0	1	0	0	0.87	0.13
radix	0.24	0.76	0.27	0.73	0.6	0.38	0.02	0.7	0.29
fmm	0.65	0.35	0.69	0.31	0.78	0.19	0.03	0.83	0.17
water_spatial	0.75	0.25	0.34	0.66	0.94	0.04	0.02	0.92	0.08
water_nsquared	0.84	0.16	0.18	0.82	0.95	0.05	0	0.91	0.09
barnes	0.09	0.91	0.36	0.64	0.93	0.05	0.02	0.93	0.07
ocean (continuous)	**0.95**	0.05	0.19	**0.81**	0.39	**0.6**	0.01	0.47	0.53
raytrace	0.64	0.36	0.89	0.1	0.88	0.07	0.05	0.89	0.11
blackscholes	**0.95**	0.05	0.07	**0.93**	0.11	**0.89**	0	0.01	0.99
swaptions	0.68	0.32	0.42	0.58	0.89	0.09	0.02	0.84	0.16
fluidanimate	0.97	0.03	0.87	0.13	0.92	0.08	0	0.78	0.22
x.264 Encoder	**0.97**	0.03	0.75	0.25	0.32	**0.68**	0	0.27	0.73

Table 4: shows PeerSearchCt, NearSearchCt and AvgDistanceSearchCt cost components, normalized with respect to (w.r.t.) DNUCA cost. It also shows distribution of normalized CLAs with $SI = 1$, $SI > 1$, $CI < 3$ and $CI \geq 3$.

Fig. 5 shows ratio of DNUCA to SNUCA costs predicted by CPP and that obtained using a cycle accurate simulator. The ratio of execution time with DNUCA to that with SNUCA, both obtained using cycle accurate simulator is also shown in Fig. 5. 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, with DNUCA. For applications like ocean, blackscholes and X.264, CPP predicts lesser $DNUCA_{cost}$ than $SNUCA_{cost}$, 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. Fig. 5 also shows L2 access latency obtained using DNUCA and normalized with that obtained with SNUCA for these applications. Normalized L2 latency seen in DNUCA does not show large degradation as seen in cost ratios. This is because, while calculating average L2 latency in simulator, offchip DRAM access latency is also considered. DRAM latency is

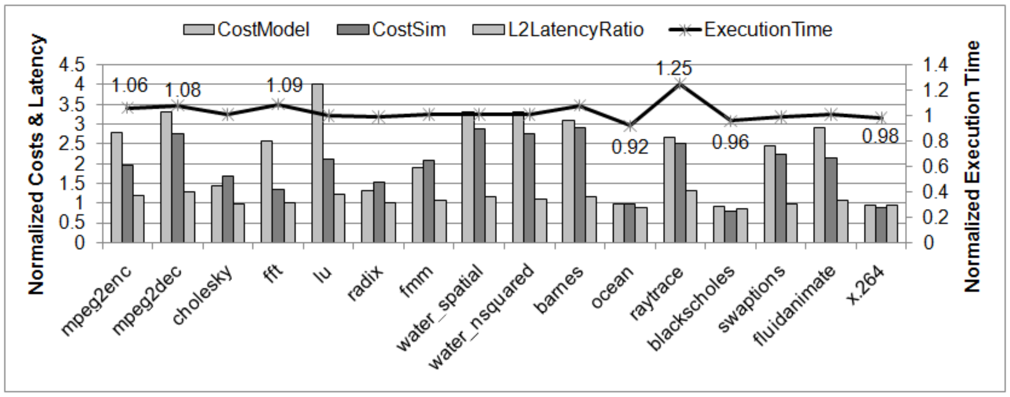
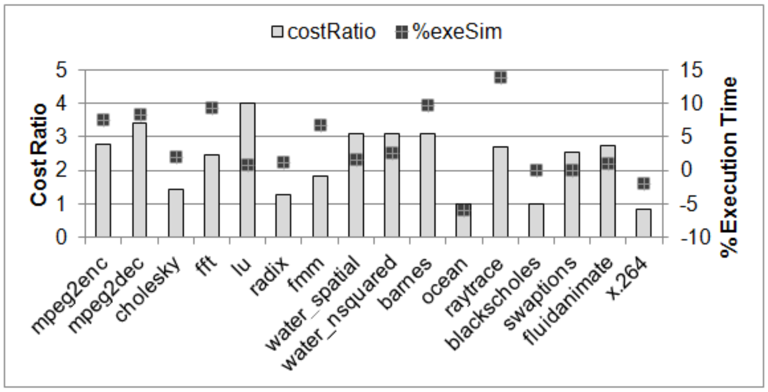


Figure 5: Accuracy of CPP Model: CostModel-Ratio of $DNUCA_{cost}/SNUCA_{cost}$ estimated by CPP. CostSim - Ratio of time spent in transit in DNUCA to that in SNUCA, both obtained with simulation

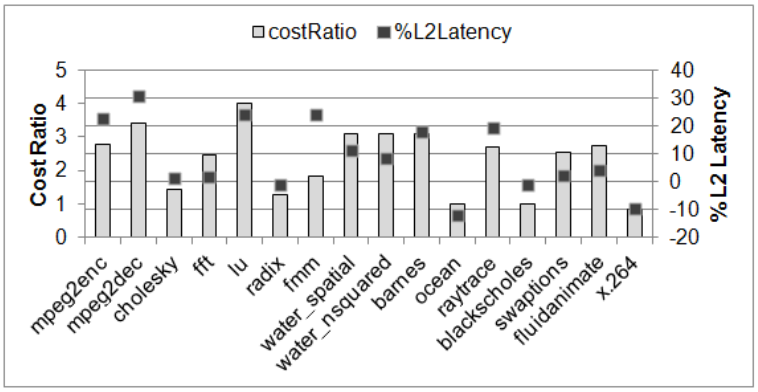
much larger than time spent in NoC. Our cost model considers time spent in accessing data in L2 banks alone. It assumes data is already present in L2 cache. The number of DRAM accesses depends on working set size of an application, and does not depend heavily on DNUCA/SNUCA cache access policy. Costs determined by the model do not reflect exact costs obtained using simulation. Their relative ratio indicates whether DNUCA would be a better choice over SNUCA or vice verse.

We also took readings for L2 slice of size 256KB. Our predictions match with the simulation results. Graph in Fig. 6(a) compares ratio of costs determined by CPP to percentage execution time difference. The percentage execution time difference is the difference between execution time obtained with DNUCA to that obtained with SNUCA (obtained using simulator) and normalized w.r.t execution time of SNUCA. Similarly, Fig. 6(b) compares cost ratio to percentage L2 latency difference obtained using simulator. When L2 slice of 256KB is used, all applications give a better performance with SNUCA, except ocean, blackscholes and x.264. This behaviour is same as with L2 slice of 512KB size. This shows that the CPP model is independent of size of L2 slice.

CPP predicts costs for DNUCA and SNUCA policies with one-time simulation using SNUCA. Hence, by saving one simulation with DNUCA, it achieves an average simulation speed-up of 2.18 times compared to SNUCA/DNUCA cycle accurate simulations put together(Fig. 7). Fig. 7 also shows overhead over SNUCA simulations. It can be seen that overhead to track CLA access pattern is very negligible. This is because we only track cache accesses which



(a) Execution Time



(b) L2 Latency

Figure 6: CPP model predicts accurately for L2 slice of size 256K

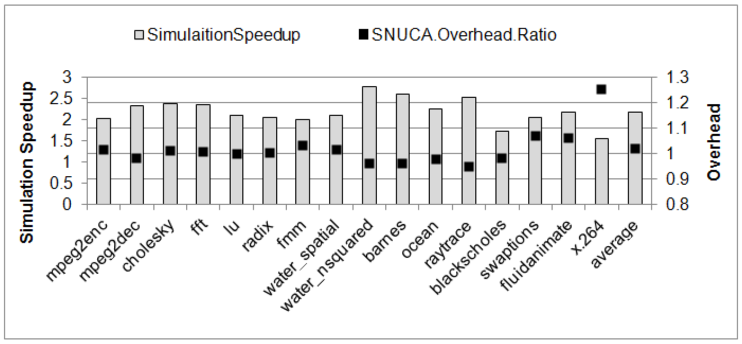


Figure 7: Graph shows simulation speed-up obtained by CPP over cycle-accurate simulations of DNUCA and SNUCA

miss in L1s which are much lesser than the total number of accesses done to the cache subsystem. Due slight variation in the number of instructions executed by each core, in some cases simulation time of CPP is slightly lesser than SNUCA simulation. Moreover, this can be done by another thread without affecting SNUCA simulation time. Hence, CPP can be used by system architects to make a choice between DNUCA and SNUCA cache policies while designing a system.

6.2 TCP to determine false data sharing

```
typedef struct gmem {
   INT nprocs;
   INT pid;
   INT rid;
   |
   barrier_t start;
   lock_t pidlock;
   lock_t ridlock;
   lock_t memlock;
   lock_t (wplock)[MAX_PROCS];
   |
} GMEM;
```

(a) Original structure definition

```
typedef struct gmem {
   INT nprocs;
   INT pid;
   lock_t pidlock;
   char PAD1[40];
   INT rid;
   lock_t ridlock;
   char PAD2[48];
   |
   barrier_t start;
   char PAD4[60];
   lock_t memlock;
   char PAD5[60];
   lock_t (wplock)[MAX_PROCS];
   |
} GMEM;
```

(b) Changed structure definition

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

(c) Table gives % improvement achieved in various metrics with our source level changes in an application

Figure 8: False data sharing detected by TCP

For large multi-threaded applications, it is difficult to determine false data sharing between threads statically. Conservative static analysis might pad too many dummy variables in a large data structure. We observed that in many cases, conservative padding degrades execution time of an application, due to increase in the number of offchip DRAM accesses. DRAM access latency is much higher than time spent by an application in cache coherence messages, induced by false data sharing. Hence, accurate estimation of effective false sharing is required to improve performance. We use SI, CI and PI (Section 3.3) values evaluated by TCP to determine data addresses causing excessive cache coherence messages. We consider addresses with SI greater than 8, CI less than 2 and PI greater than ten thousand. We determined these values empirically. Our tool also gives information about instruction addresses accessing these CLAs. By using information obtained from the application binary and trapping malloc calls, we determined the culprit data structures.

Raytrace in Splash2[11] allocates a global gmem structure which has multiple locks and a barrier as its members (Fig. 8(a)). In this case, barrier and locks are allocated in the same cache line of 64B in size. Hence, we changed the gmem structure as shown in Fig. 8(b). We co-allocated pidlock and ridlock with members which they protect from concurrent use. We also allocated barrier in a separate cache line. With these changes, we could obtain a significant improvement in performance of an application. We made similar changes in barnes. Table 8(c) summarizes % energy-delay product and execution time improvement. We could achieve execution time improvement up-to 10.7% for this well studied benchmark applications with TCP.

7 Conclusions and Future Work

We present TCP, a tool to profile data accessed by a multi-threaded application. TCP quantifies the number of threads sharing data and their sharing pattern. We propose a sharing index (SI), inspired from *entropy* in information theory. SI quantifies the number of threads sharing an object. Higher SI does not necessarily imply higher contention for an object. Hence, we define contention index (CI) expressed in terms of the average number of consecutive accesses made by the same thread. Higher average runlength size denotes lower contention for an object. We use SI and CI to accurately predict a more profitable cache access policy between SNUCA and DNUCA for an application. This model achieves an average simulation speed-up of 2.2 times compared to SNUCA/DNUCA cycle-accurate simulations put together.

We also demonstrate use of TCP to predict contention bottleneck in the code by finding false data sharing. With our changes in the program, we achieve up-to 10% and 18.9% improvement in execution time and energy-delay product of the application. Currently, we manually detect and change data structure layout. This can be partially automated which we plan to do in future. One way to do this is to ask a programmer to specify boundaries of different tasks in the application code. If a cache line *popular* with many threads receives accesses from instructions belonging to different tasks then it could be due to false data sharing and a hint could be given to the programmer to reorganize that data structure. TCP can also be used to solve other problems such as changing lock granularity, and evaluating thread scalability.

References

[1] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, “Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs,” in *CGO'2010*.

[2] “Gnu prof.” [Online]. Available: http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html

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

[4] M. D. H. Jan Edler, “Dinero trace-driven uniprocessor cache simulator,” http://www.cs.wisc.edu/ markhill/DineroIV.

[5] S. Ghosh, M. Martonosi, and S. Malik, “Cache miss equations: an analytical representation of cache misses,” ser. ICS '97.

[6] D. Eklov, D. Black-Schaffer, and E. Hagersten, “Fast modeling of shared caches in multicore systems,” in *ACM HiPEAC,2011*.

[7] D. Chandra, F. Guo, S. Kim, and Y. Solihin, “Predicting inter-thread cache contention on a chip multi-processor architecture,” in *HPCA*, 2005.

[8] S. J. Eggers and R. H. Katz, “A characterization of sharing in parallel programs and its application to coherency protocol evaluation,” in *ISCA '88*.

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

[10] 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.

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

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

[13] 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.

[14] 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.

[15] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, “DRAMsim: a memory system simulator,” 2005.

[16] “Micron DRAM power data sheet.” [Online]. Available: http://www.micron.com/products/partdetail?part=MT47H128M8HQ-3E

[17] 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.

[18] “Intel Nehalem.” [Online]. Available: http://www.3dnow.net/phpBB2/viewtopic.php?f=1\&t=1474\&p=6720

[19] 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