CHiP: A Profiler to Measure the effect of Cache Contention on Scalability

Bevin Brett
Software and Services Group
Intel Corporation
Nashua, NH, USA
bevin.brett@intel.com

Pranith Kumar, Minjang Kim, Hyesoon Kim
Georgia Institute of Technology
School of Computer Science, College of Computing
Atlanta, GA, USA
{*pranith, minjang, hyesoon.kim*}*@gatech.edu*

***Abstract*—Programmers are looking for ways to exploit the multi-core processors which have become commonplace today. One of the options available is to parallelize the existing serial programs using frameworks like OpenMP etc. However, such parallelization does not always yield the speedup expected by the programmer. This is due to various reasons, one of which is the bottleneck presented by the memory system. Carefully optimized serial algorithms fit into most of the cache available, yet these cache optimized serial algorithms might have the worst speedup when parallelized. We present an efficient method to identify such cases and determine whether a serial algorithm's use of shared memory caches will seriously impact its parallel execution. This can help a programmer adjust their program's cache usage. We demonstrate the effect in isolation with a parallelized synthetic micro-benchmark which uses varying fractions of the shared cache. We also present the results of our analysis of the NAS Parallel Benchmark suite and the Rodinia benchmark suite.**

I. Introduction

The multi-core era is posing a serious challenge to the programmer looking to make efficient use of the hardware resources available. The programmer has to either rewrite his application using parallel algorithms or modify his legacy application to use parallel constructs. In the later case, the programmer has the difficult task of first identifying the regions of code to parallelize and then choosing the appropriate framework (such as Cilk [1], OpenMP [2] etc.,) to parallelize these regions. Identification of regions of code to parallelize is a non-trivial task. A seemingly obvious parallelizable region of code, when parallelized, might not scale at all, instead the performance may degrade due to severe resource contention (cache, network, disk etc.,) or the overhead of parallelization (locking, scheduling etc.,).

It would be helpful for the programmer to know beforehand an estimate of the scalability of his code before investing effort in parallelizing it. Previous work like Intel's Parallel Advisor [3], Kismet [4], and Parallel Prophet [5] aim to help the programmer in making this decision. Although these tools are very useful, we found that they do not sufficiently consider the bottleneck posed by the memory sub-system. Especially, we need to consider the effect of working set size on cache contention because most of the previous methods of measuring working set sizes used high overhead LRU stack distance [6] algorithms. Hence, in this work, we focus on developing a low-overhead memory profiling technique that is specifically designed to understand the effect of cache contention on scalability.

Estimating working set size has been studied extensively starting with the LRU stack distance measurement [6]. Several solutions were proposed to improve the speedup of measurement mechanisms such as [7], [8], [9]. However the speedups obtained were not sufficient for our purposes, because of the high number of instructions executed during their $O(nln(n))$ behavior. Recently a few studies in [10], [11] measured the working set size of parallel applications. Our work differs from this earlier work in that, unlike previous work in measuring working set size of either serial code or parallel code, our work estimates memory usage *when parallelized* from *serial code*. Our proposed solution is a hybrid algorithm that is a slight variation of both the stack reuse distance algorithm and a cache simulator to create an $O(n)$ algorithm with a low number of instructions per item, giving a lower resolution measurement of the working set, but one that is adequate for performance predictions. The profiling algorithm is called *CHiP* (*Cache Hit Profile*), which estimates a program's use of the cache when parallelized by profiling only a sequential code. These estimates can be used to either tune or predict an application's performance [12].

Contributions We make the following contributions in this paper:

- We present a low-overhead memory usage profiler algorithm to efficiently calculate the cache hit/miss ratio for multiple cache sizes
- We present a benchmark which quantifies the effect of the shared cache usage on speedup upon parallelization
- We apply our profiling algorithm to the NPB and Rodinia benchmark suites and present our observations after analyzing the results

II. Related Work

Earlier work to calculate the variation in working set size used either stack distance [13], [8] or LRU based distance measurements [6], [14]. Cache miss equation based measurements [15], [16] for estimating working set size have also been investigated. Trace driven cache simulators for multi-core processors which use Pin are also popular for calculating the cache hit ratios [17]. *CHiP* is different from these works

because the main purpose of profiling here is to predict cache hit/miss ratios when the code is parallelized. All the previous works target to predict cache hit ratio only for either sequential code or parallel code [7], [9].

Recently a few tools have been proposed to predict parallel code performance only from a serial version of code. Parallel Prophet [5] implements an emulator along with a memory performance model to estimate speedup for the annotated regions of code. It assumes that the DRAM accesses do not vary when going from serial to parallel. This assumption is not always true. Our current work can be used to discover and model such cases.

Intel Parallel Advisor [3] is another such tool but it does not include memory interference effects in its current speedup estimates. Kismet [4] implements an extension of hierarchical critical path analysis to calculate the speedup estimates. It implements a memory model in which the memory system is assumed to be perfectly scalable.

III. Background and Motivation

A. Motivation with Parallel Advisor's Results

In this section, we motivate the problem by showing results from the current state-of-the-art commercial tool, Intel's Parallel Advisor [3]. First, we briefly explain the working of Parallel Advisor. Parallel Advisor is a tool to estimate speedup from a sequential code before programmers actually parallelize applications.

Parallel Advisor takes an annotated serial program where the annotations specify potential parallel and protected regions.

```
ANNOTATE_SITE_BEGIN (allloop);
for (int i = 0; i < N; ++i) { // parallel
   ANNOTATE_TASK_BEGIN (p1);  // each-loop
   compute(p1);
   ANNOTATE_TASK_END (p1);
}
ANNOTATE_SITE_END (allloop);
```

Fig. 1. An example of Parallel Advisor annotations. The for-loop is parallelizable. The outcome of Parallel Advisor is an estimated speedup of this code when parallelized.

Parallel Advisor measures execution time between two annotations; `ANNOTATE_SITE_BEGIN` and `ANNOTATE_SITE_END`. Based on the profiled results, it mimics parallel program behavior and predicts performance of parallel code. Finally, speedups are reported against different parallelization parameters such as scheduling policies, threading models and CPU numbers.

Figure 2 shows the estimated speedup from Parallel Advisor and the real speedup for the NPB benchmarks. The detailed experimental methods are described in Section VI. The speedup is measured for each parallel site (i.e., only parallelizable sections), which means that Amdahl's law would predict perfect scalability. The reasons many sites do not show this predicted perfect scalability are as follows; load unbalancing, lock-contention overhead, parallel construct overhead, cache and memory resource contention, and others. Parallel advisor models the first three overheads while estimating the speedup but does not consider the memory resource contention overhead.

The results are presented based on the order of DRAM bandwidth consumption in the sequential code (ascending order from the left side). For the two-thread case, many sites have almost scalable speedups especially for low bandwidth consumption sites except for a few exceptional cases such as `IS(B).rank1` or `FT(B).compute_indexmap`. When the bandwidth consumption increases, the gap between estimated speedup from Parallel Advisor and actual speedup increases, which is due to memory bandwidth consumptions [5]. For the four-thread case, in almost every cases the parallel advisor predictions are too optimistic. The gap between estimated speedup and actual speedup increases significantly. More benchmarks show a much wider gap even in low-bandwidth sites. The noticeable gaps are `IS(B).rank1`, `SP(B).comput_rhs_3`, and `FT(B).compute_indexmap`. The case of `IS(B).rank1` is particularly interesting. It improves performance when using two threads (1.3) but on using four threads, it shows only 0.89 speedup, which is worse than the sequential version. Figure 3 also shows the predicted speedup from advisor and actual speedup for the Rodina benchmarks. These results also show similar trends. For the 2-thread cases, only high-bandwidth sites show poor speedups but for 4-thread cases, even low-bandwidth sides show poor speedups. Except `nn.main` and `kmeans.kmeans_clustering`, parallel advisor over estimates the performance benefits when parallelized.

Recent work in Parallel Prophet adjusts the estimation of speedup for high bandwidth consumption benchmarks [5]. However, Parallel Prophet also cannot predict slowdown for medium bandwidth consumption sites. This motivates us to investigate the effect of cache contention for the medium bandwidth consumption benchmarks.

B. Analytical Model of Slowdown

To understand the cache contention behavior, we construct a simple analytical model. We model four tasks that, when executed serially, use 75% of the last-level cache (LLC). This fits perfectly in the cache when run in serial. When executed in parallel, there will be more evictions from the LLC resulting in an increase in LLC misses.

This increase may cause the execution time of a task running in parallel to exceed that of the corresponding task when running in serial, since any extra cache misses will now need to be serviced by the DRAM with its longer latencies and lower bandwidth.

In Fig. 4, we plot the slowdown calculated by an analytical model of four tasks using an equal percentage of the shared cache running serially and in parallel. In this model, we vary the percentage of shared cache being used by each of the tasks being run in parallel and calculate the slowdown from a tasks serial run. We can observe that the slowdown starts increasing dramatically when all the tasks do not fit in the shared cache anymore i.e., when each task is using $\approx 25\%$ of the shared cache. We can also observe that when the size of cache being

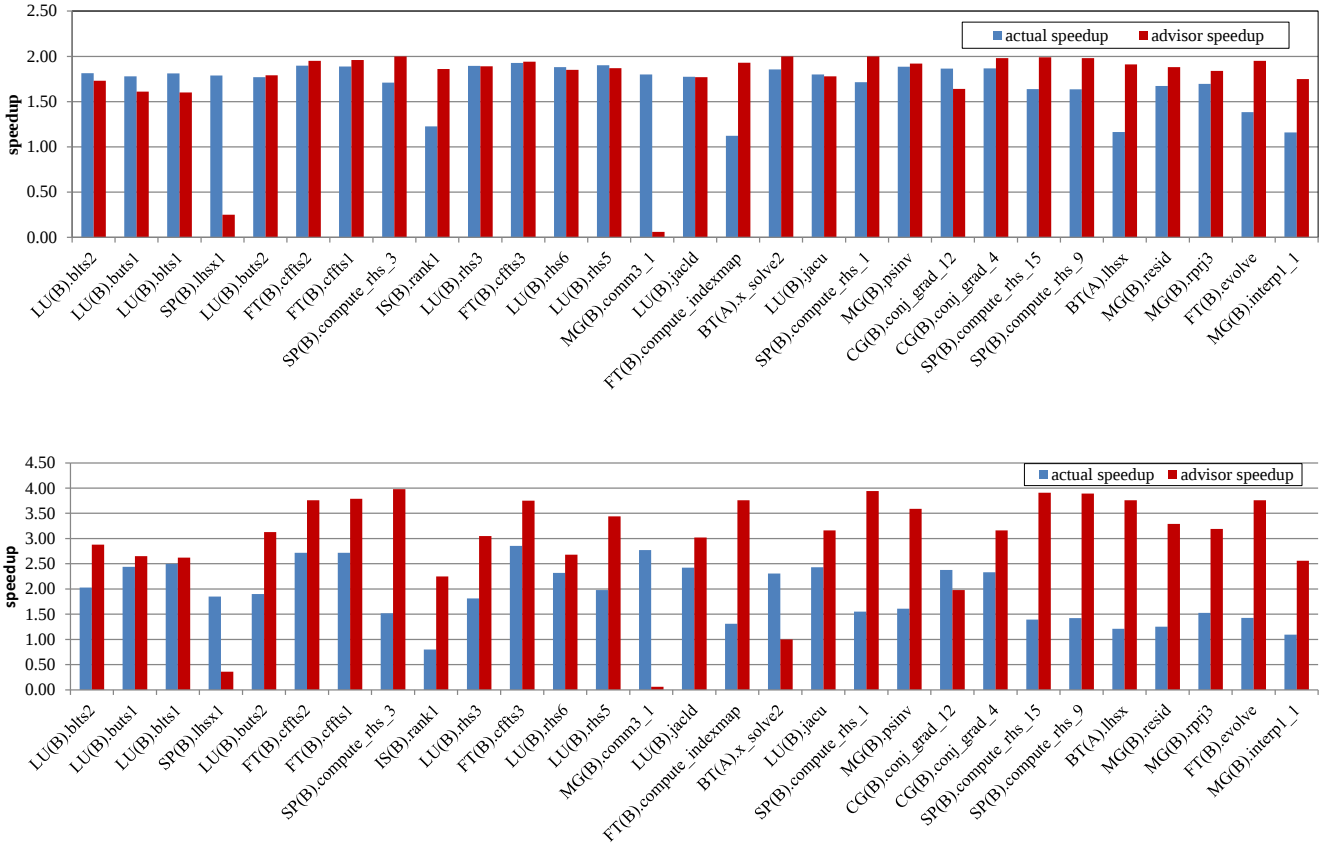

Fig. 2. *Advisor and Actual results comparison for NPB: (top) 2 threads, (bottom) 4 threads. The x-axis shows the benchmark name, input size and the function name where the parallel site is located. e.g., FT(B).cffts2 is FT benchmark with B-input. cffts2 function contains the corresponding parallel site.*

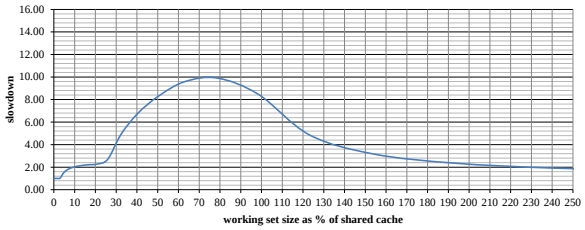

Fig. 4. *The slowdown of each task running in parallel as compared to when run in serial*

used by each task is approximately 75% of the size of the shared cache, that task is the worst performing task when run in parallel. This shows us that *a cache optimized serial algorithm is the most affected when parallelized*. This result also indicates that when an application has a large working set size in sequential code, the performance degradation in the parallel version is less severe because the application is already memory bound.

C. ShuffledReads benchmark

To observe the slowdown behavior in real hardware, we designed a synthetic micro-benchmark to measure the scalability.

Description: The synthetic parallel micro-benchmark is designed to measure the effect on scalability caused by the variation in fraction of shared cache used by a task. In this benchmark we create homogeneous tasks, each of which accesses two regions of memory. Each task accesses its own exclusive regions of memory. These regions are not shared. One is a smaller, very frequently accessed memory region which acts as the tasks working set and the other is a very large, rarely accessed memory region. Each region has a pointer chain within it. We also have a bit vector with an even number of 1s which encodes when to switch from the smaller region to the larger region. Based on this bit vector we chase the pointer chain in either one of the two regions for several iterations. We run this task stand-alone and in parallel with other similar tasks and measure the time taken to complete the iterations in each case. We then use these execution times to calculate the slowdown of each task when run in parallel compared to when run in serial. Due to running multiple tasks in parallel, there is contention for the shared cache and this contention causes each task to slow down. We vary the fraction of cache being used by the tasks (same for all the tasks) and measure the execution times using which we calculate the slowdown to get a slowdown graph.

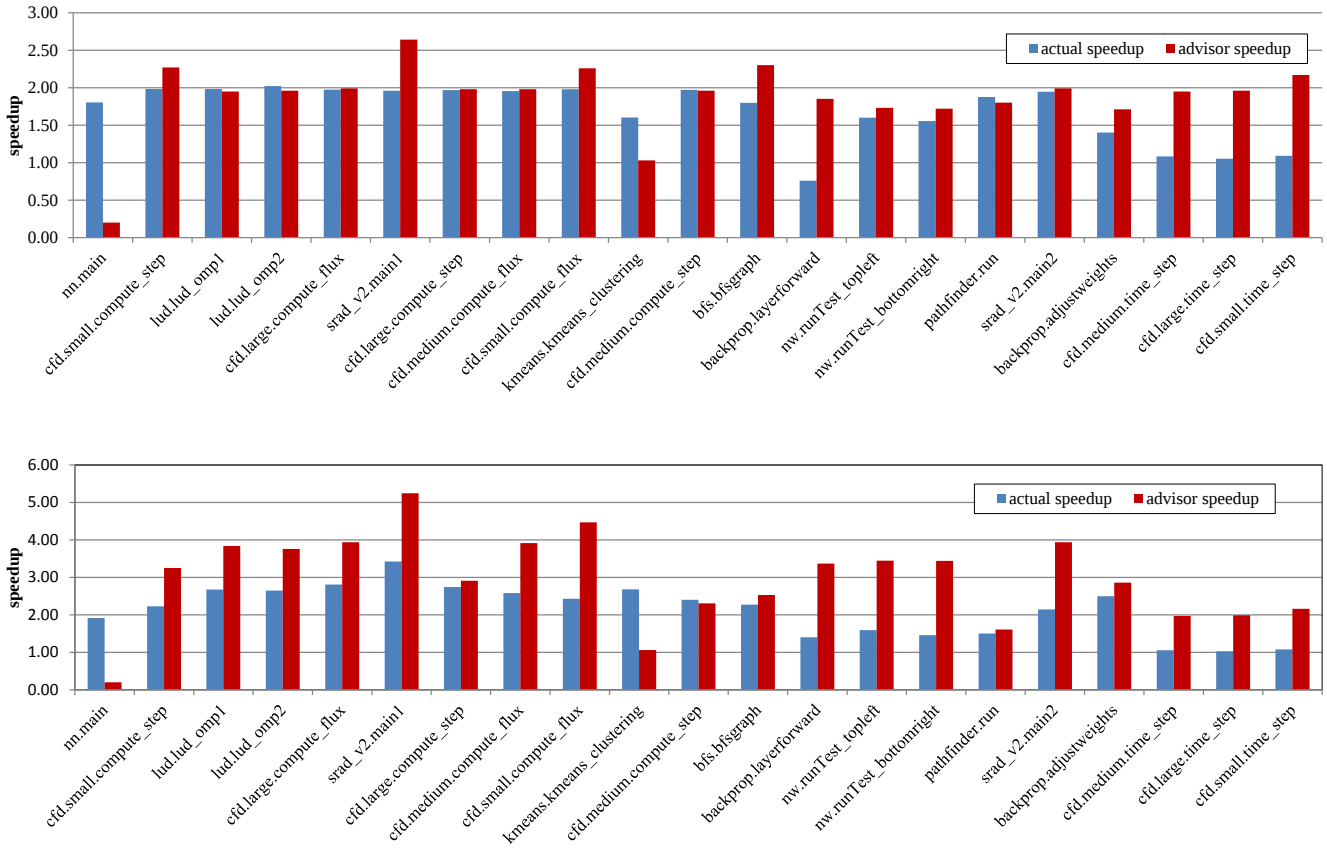

Fig. 3. *Advisor and Actual results comparison for Rodinia: (top) 2 threads, (bottom) 4 threads*

To summarize, we can now vary (a) the ratio of accesses to the two regions (b) the size of the two regions and (c) the number of such tasks to run in parallel, and see the effect these variations have on the slowdown.

Timing Measurement: To get an accurate execution time measurement of a task, we need to reduce the error introduced by (a) rdtsc call overhead and (b) noise due to the OS.

For this purpose we designed and implemented an *Interval Profiling* method. In this method, the execution of the task is divided into unmeasured and measured intervals, with 9 measured intervals spread evenly throughout the execution. We make sure that the length of each measured interval is such that the rdtsc call overhead is not more than 1%. Also, we make sure that the length of the interval is small enough such that any noise introduced due to the OS is minimal. We then time these intervals and choose the median measured times. The shortest are possibly caused by rdtsc noise, and the longest by OS noise.

Calculating Slowdown: We use the execution time measured in calculating the slowdown for each individual task. We are looking at the slowdown of each individual task instead of the speedup of the region because heterogeneous tasks will slow down by varying amounts when run in parallel. Also because a task on the same core as the execution prior to the fork will benefit from the initial cache contents. The tasks which are run on a different core will have cache warm-up costs associated with them. Looking at overall slowdown instead of slowdown of each individual task will miss this important factor.

Results: We plot the results in Fig. 5. The system has an 8MB shared cache, and 3 cores were used to parallelize the accesses. In Fig. 5 the peak slowdown is observed at about 3-4MB, where it would take 3x3MB to hold all the frequently accessed memory regions and the reads from the much larger, rarely accessed memory are using about 10% of the cache. The combined 10MB is exceeding the available 8MB of shared cache, causing extra cache misses. The slowdown in this case may even be caused by the contention for the memory accessing fabric itself. These experimental results are consistent with the observations of the analytical model in Section III-B.

IV. Definitions and Measurements

As we describe in Section III-A, we annotate the parallelizable code sections in a serial program and generate a binary executable, with all optimizations. This brings us closer to the real life execution of the program. Our profiler is implemented as a Pin[18] tool which instruments these annotated regions in the executable. We instrument the memory accesses within the

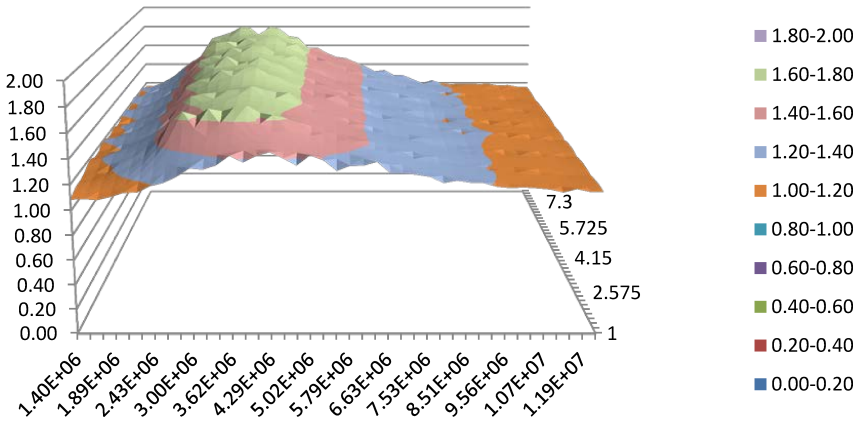

Fig. 5. *The graph shows the slowdown of each task for various working set sizes. x-axis: working set size(bytes), y-axis: slowdown, z-axis: number of times all the working set is read*

annotated region to extract the accessed memory addresses. The collected addresses are used to generate a *Cache Hit Profile*.

A. Cache Hit Profile

Each *Cache Hit Profile* (*CHiP*) consists of:

1) an array of CacheSets of width w. Each CacheSet is a LRU array with d entries. Each entry in a CacheSet represents a recently accessed cache line. The most recently accessed cache lines are at the front of the LRU array. The entire array of CacheSets describes a $(w * d * cacheLineSize)$ byte cache.
2) an array of counters of length $(d + 1)$ which store the total number of hits at the corresponding depth into the CacheSets. The counter at depth $(d+1)$ stores the total misses. The first n counters of the CacheSets describe a $(n * w * cacheLineSize)$ byte cache.

B. The number of CacheSet entries d and CacheSets width w

The number of entries d in the CacheSet controls the accuracy of our generated profile. A 1-wide, infinite entry CacheSet is what is used in the Stack Distance algorithm [13]. Increasing the width, and decreasing the depth will decrease accuracy and overhead. So choosing depth d is a trade-off between accuracy and overhead.

We set d to 16, because knowing the fraction of the cache used to within 1/16th is sufficiently accurate for our purposes, and because the CacheSet then fits in one or two cache lines, it can be very quickly updated.

The width w of the array of CacheSets is determined by the maximum cache size for which we want to profile our program.

$$w = \frac{max\_cache\_size}{d \times cache\_line\_size}$$

Example: Consider a profile which tries to map 16 MB of shared cache with a 64 byte cache line. If the number of entries in each CacheSet is 16, then the width of the array of CacheSets is 16 MB / (64 x 16) = 16K. 16K CacheSets of 16 entries deep are required to map such a shared cache.

C. Measuring the CHiP

We extract the cache line addresses for each memory address accessed using a Pin tool. The address is then mapped to its corresponding CacheSet using a hash. We then lookup the mapped CacheSet for the presence of this cache line by updating from the front until the address is found or until the end of the LRU array of the CacheSet is reached. If this cache line was accessed recently enough, it will be present in the CacheSet. If the cache line is being accessed for the first time or if the cache line was not accessed recently enough, it will not be present in the CacheSet. In either case the scanning search updates the CacheSet to bring this cache line to the front of the LRU array, pushing the other entries back by an entry. We increment the array of counters entry for the depth at which the cache line was found, otherwise we increment the counter at depth $(d + 1)$ signifying a miss. We also present pseudo code in Listing 1.

D. Overhead

1) Memory Overhead: In our implementation each cache line address is 8 bytes and cache line size is 64 bytes. Each counter is an unsigned 8 byte integer. We profiled for a cache size of 16 MB and the number of entries in each CacheSet is 16 giving us a granularity of 1 MB. The memory required for the cache data structure in this case is:

$$= \frac{max\_cache\_size \times sizeof(address)}{cacheLineSize} + (d+1) \times sizeof(counter)$$

which is = (16 MB * 8 / 64) + (17 * 8) = 2 MB.

If necessary a second *CHiP* can be maintained simultaneously with a smaller max cache size, and it can get finer resolution for determining effects when there are 8 or more threads contending for the cache causing a

```
CacheSetArray is the array of all CacheSets
CountsArray is the array which stores the hits at each depth
cacheSet_depth is the depth of a CacheSet

UPDATE_CHIP(address):

  cacheLine = address >> log2(CACHE_LINE_SIZE);
  hashedCacheLine = hash_function(address);
  cacheSet_index = hashedCacheLine & mask; /* 0 − width */
  current_cacheSet = CacheSetArray[cacheSet_index];

  depth = find(cacheLine, current_cacheSet);

  CountsArray[depth]++;

/* This searches for cacheLine in the cacheSet
 * and returns the depth at which the cacheLine is found
 * else, return cacheSet_depth
 */
find(cacheLine, cacheSet):
  temp_cacheLine = cacheLine;

  idx = 0;
  while(idx < cacheSet_depth)
    oldCacheLine = cacheSet[idx];
    cacheSet[idx] = temp_cacheLine;
    if (oldCacheLine == cacheLine)
        break;
    temp_cacheLine = oldCacheLine;
    idx++;

  return idx;
```

Listing 1. Pseudo code for *CHiP*

need to know the usage to greater precision.

2) Profiling Overhead: Our profiling overhead comes from instrumenting the code regions using Pin and then processing the extracted addresses through our *CHiP* profiler. We found that the combined overhead to be $\approx$ 10x the non-instrumented execution time. In the profiler, for each cache line address we perform at most d comparison operations. So the processing cost is $O(n)$ where n is the number of memory addresses extracted.

V. *ANALYSIS*

A. Analysis

In this section we present a method to analyze the increased DRAM traffic using the the generated *CHiP* profile. The *CHiP* profile gives us the cache hit percentage at various depths which correspond to various cache sizes. Using these hit percentages we can estimate the shared cache usage for different sizes of the shared cache. When a parallel section of code is being executed, each thread will effectively get a part of the cache (ignoring the conflicts and true sharing in this shared cache). The *CHiP* profile will give us an estimate of the cache hit ratio in this parallelized scenario, using which we can estimate the increase in memory traffic to the DRAMs.

B. Assumptions

- In our analysis, we assume that only the last level cache (LLC) is being shared among all the cores. Our analysis can be easily extended to cases where the LLC is not shared by all the cores(i.e., multi-socket configurations).
- We also assume that the LLC will be equally divided among the multiple cores, i.e., the contention in the shared cache is purely due to capacity misses and not due to either conflict or coherency misses because of sharing of the same cache lines between multiple cores. In [19] the authors examine PARSEC and conclude that positive or negative interference effects of cache sharing between threads are not significant on scalability. We assume similar behavior in the NPB and Rodina benchmarks. Also, although conflicts and coherency issues affect the speedup in the shared cache, since we perform our analysis on a serial program, it is not straight forward to estimate these effects without incurring a large overhead.
- We did not consider the super scalar behavior that might arise when the LLC misses are reduced due to parallelization.

C. Analytical Model

Given the *CHiP* profile and the native execution time t_{ser} for a serial region of code, we calculate the approximate DRAM traffic generated using the following equations. The total number of memory accesses is the sum of all the counts over the depth $d+1$.

$$Total_{accesses} = \sum_{1}^{d+1}(Counts) \tag{1}$$

The number of cache hits until depth d' is given by:

$$Hits_{d'} = \sum_{1}^{d'}(Counts) \tag{2}$$

The $HitRatio_{d'}$ is the ratio of all the memory accesses which are hit in the cache which is mapped until depth d':

$$HitRatio_{d'} = \frac{Hits_{d'}}{Total_{accesses}} \tag{3}$$

Using these equations, we can now calculate the approximate DRAM traffic as follows:

$$Bandwidth_{ser} = Total_{accesses} \times (1 - HitRatio_{d}) \times \frac{cache\_line\_size}{t_{ser}} \tag{4}$$

where *d* is the depth in the *CHiP* profile which corresponds to the size of the shared cache in the system.

When parallelized, the new depth d' will correspond to the effective shared cache available to each thread. We approximate this as

$$d' \approx \frac{d}{\#threads} \tag{5}$$

The ideal execution time for this region of code will be

$$t_{par} = \frac{t_{ser}}{\#threads} \tag{6}$$

Using the above equations, the bandwidth required when parallelized is:

$$Bandwidth_{par} = Total_{accesses} \times (1 - HitRatio_{d'}) \times \frac{cache\_line\_size}{t_{par}} \quad (7)$$

An ideal memory system which can supply the required bandwidth should not have any adverse effect on the scalability of this parallel region of code.

Consider that the individual hit % for this sample profile is given and Figure 6 shows the % hits expected with a given cache size (MB). So for a 16 MB cache size, the hit ratio is almost 1, for a 8 MB cache, the hit ratio is 0.79.

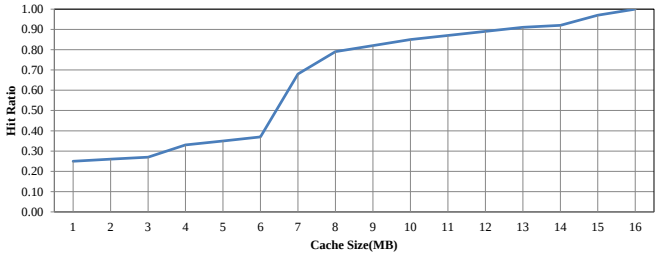

Fig. 6. Sample *CHiP* profile (cache hit ratio)

Assuming that the total accesses were 100 million, when executed in a system with a 4 MB cache, the DRAM access count will be 67 million. Similarly if the system has a 12 MB cache the DRAM access count will be just 11 million. So on a 12 MB shared cache machine the serial version would cause 11 million reads from the DRAM, but a 3 core parallel version (only 4 MB shared cache available for each core) would produce approximately 67 million reads (6x as many) in one third the time, increasing the needed DRAM bandwidth from 11 million/unit time to 201 million/unit time assuming ideal parallel execution time. If the memory subsystem bandwidth limit is about a hundred million accesses per unit time, it will be totally flooded because of the huge spike in memory traffic caused by the extra cache misses, which would drastically slowdown the parallel execution.

VI. Results

A. Comparisons of CHiP and a Cache Simulator

In Figure 7, we compare the hit ratio result from the *CHiP* profile and a cache simulator for the NPB *bfs* benchmark. This graph shows that the hit ratio from a *CHiP* profile is sufficiently accurate and can be used to estimate hit ratios for multiple cache sizes. The overhead of *CHiP* is almost 1/16th of the cache simulator for this case since *CHiP* requires only one run of the cache simulation whereas a naive cache simulator needs to simulate the same address trace multiple times.

B. Analysis on Benchmarks

We use a quad-core Intel Core i7 system which has an 8 MB 16-way shared cache for our experimentation [20]. In our results we plot the bandwidth required so as not to affect the speedup of the benchmarks analyzed. In the bandwidth graphs,

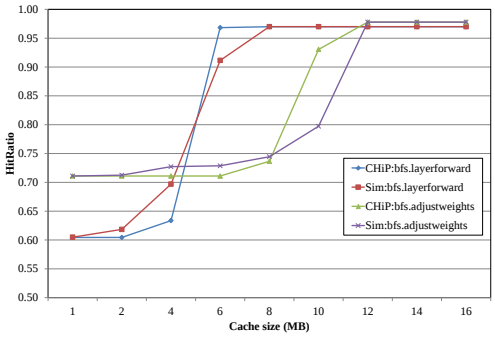

Fig. 7. *Comparison of hit ratio from CHiP profile and a Cache Simulator(sim)*

we show overlapped bars for serial, serial*(2 or 4), and the estimated bandwidth for 2/4 threads from *CHiP*. The delta between the outcome of *CHiP* and serial*(2 or 4) shows the additional required bandwidth because of the cache contention in parallel programs. (This delta is presented as green color bar in the figures). So, a higher green color bar means the benchmark would have a severe slowdown because of the cache contention when the code is parallelized.

We perform our analysis on the *NAS Parallel Benchmark* (NPB) suite of benchmarks [21] and on the *Rodinia* [22] benchmark suite. We identify the interesting regions of code which are parallelized using OpenMP pragmas. We present the results of our analysis on regions of code which have significant execution time (>4% of total execution time). We chose the B input parameter to the NPB suite except for the benchmark BT for which we used the A set of inputs. These inputs were chosen so that the benchmarks are sufficiently memory intensive. The region names are based on the functions in which the parallelized code regions were found.

Due to space limitation, we present detailed results from a subset of NPB and Rodina benchmarks. We select benchmarks that show non-scalable behavior and also the benchmarks that show significant deviation between Parallel Advisor's speedup prediction and the observed speedup.

1) NPB - FT.B: Figure 8 shows three graphs of which Figure 8.(a) is a graph of per-region *CHiP* profiles for the FT.B benchmark. Figure 8.(b) shows the serial bandwidth, linearly scaled bandwidth for two threads and the required bandwidth estimated using the per-site *CHiP* profile for two threads. Figure 8.(c) shows a similar graph for the four thread case. We can clearly identify regions in which bandwidth does not linearly scale.

In Figure 8.(b), of the five significant parallelizable regions in FT.B, only one region, *evolve*, generates significant bandwidth in the serial version. When this region is parallelized by creating two threads, the bandwidth required scales linearly. But when this region is parallelized by creating 4 threads, the increase in required bandwidth is no longer linear. This non-linear increase can be explained by looking at the variation in the cache hit ratio from the *CHiP* profile generated for this region. There is an increase in the cache misses when the

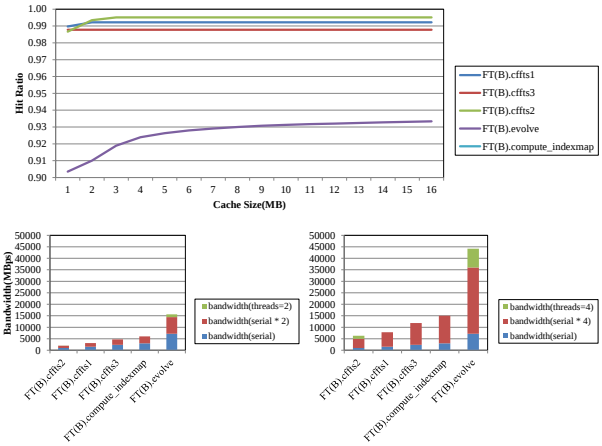

Fig. 8. **FT(B)**: *(a) CHiP (b) 2 threads (c) 4 threads*

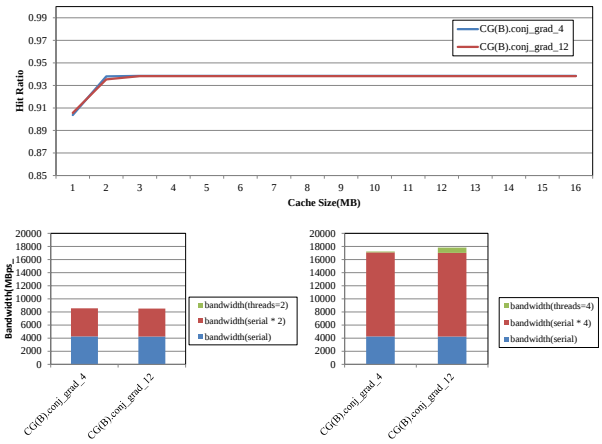

Fig. 10. **CG(B)**: *(a) CHiP (b) 2 threads (c) 4 threads*

effective cache size is reduced from 8 MB to 2 MB which is causing this increase in the required bandwidth.

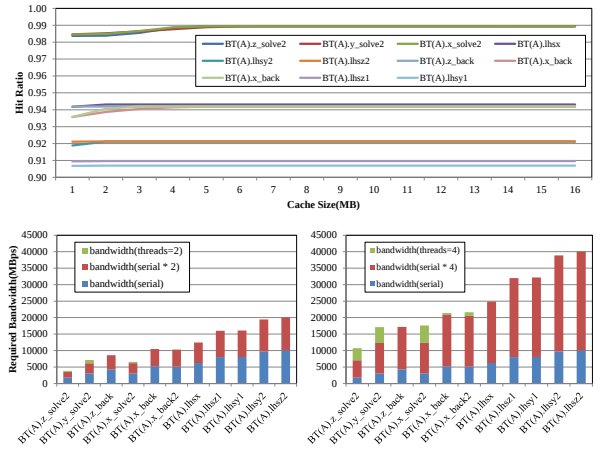

Fig. 9. **BT.A**: *(a) CHiP (b) 2 threads (c) 4 threads*

2) NPB - BT.A: Figure 9 shows BT.A. The parallelizable regions with high cache miss ratio or equivalently, high required bandwidth are *lhsx, lhsy2, lhsz2, lhsz1, lhsy1*.

On the other hand the low bandwidth required parallelizable regions *z_solve2, y_solve2, x_solve2* perform much better when parallelized due to not being limited by bandwidth. The other medium bandwidth required regions *z_back, x_back, x_back2* gain moderately when parallelized to two threads, but do not gain anything when parallelized to four threads.

3) NPB - CG.B: In Figure 10, the two significant parallelizable regions of code *conj_grad_4, conj_grad_12* generate moderate bandwidth in serial and do not require any significant bandwidth when parallelized to two and four threads. The speedups in Figure 2 for these two regions shows good speedup when using two threads and a moderate speedup when using four threads.

4) NPB - IS.B: The *rank1* region of the NPB benchmark **IS.B** is significant in that it accounts for $\approx$ 30% of total serial execution time. The bandwidth required by this region

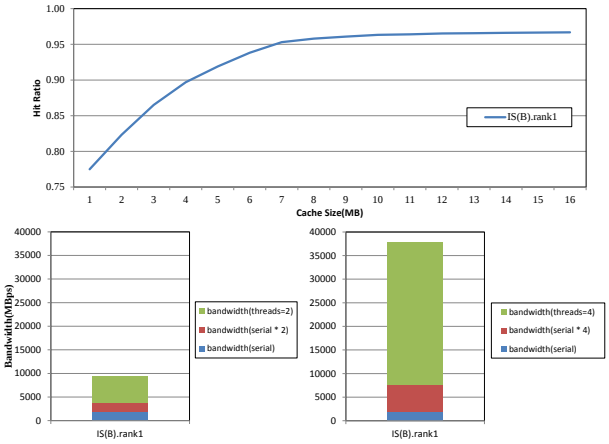

Fig. 11. **IS(B)**: *(a) CHiP (b) 2 threads (c) 4 threads*

in serial execution as seen in Figure 11 is not high. When this is parallelized using four threads, the required bandwidth increase is actually much greater than 20E+03 MBps. This has a direct bearing on the speedup that it actually regresses when four threads are created due to the bandwidth bottleneck in Figure 2. In this case, we are better off creating two threads instead of four for this particular region as the bandwidth required for two threads can still be provided for moderate gain in speedup. The bandwidth required for four threads is more than 10x the region's required bandwidth for the serial version.

5) NPB - LU.B: In Figure 12, the regions *buts1, blts1* require very low bandwidth in the serial run. Even when parallelized for two threads, the increase in the required bandwidth is not significant. But when we parallelize using four threads, the required bandwidth increases dramatically.

In Figure 12(a), from the *CHiP* profile, we can observe that different parallel regions are optimized for different cache sizes. For example, if we have a 10 MB LLC, from the *CHiP* profile we can see that most of the accesses will be hit. But with a 4 MB cache, only two regions *buts1, blts1* will have a

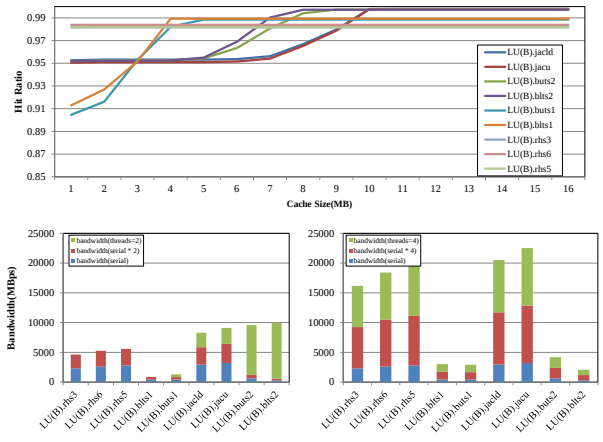

Fig. 12. **LU(B)**: *(a) CHiP (b) 2 threads (c) 4 threads*

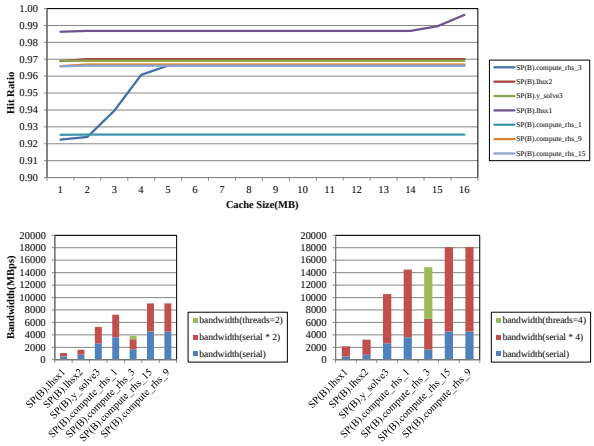

Fig. 14. **SP(B)**: *(a) CHiP (b) 2 threads (c) 4 threads*

high cache hit ratio.

For these two regions, when two threads are created, most of the accesses are still being hit in the shared cache. But once we create four threads, each thread gets only 2 MB of the shared cache which increases the cache misses to 8%. This is an 8x increases in the number of cache misses causing a dramatic increase in the required bandwidth.

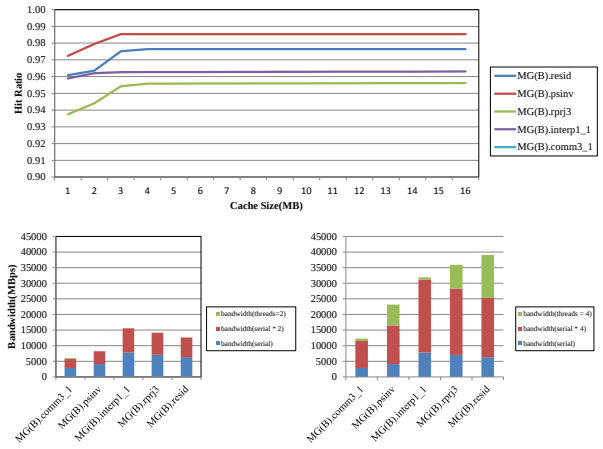

Fig. 13. **MG(B)**: *(a) CHiP (b) 2 threads (c) 4 threads*

6) NPB - MG.B: In Figure 13.(b) the regions *resid, psinv, rprj3* regress when four threads are created due to the significant increase in the bandwidth required. In these regions, no bandwidth increase is measured when going from one thread to two threads but significant bandwidth is required once you create four threads.

The region *comm3_1* has reasonable speedup because of the low bandwidth generated in serial case and has a minor increase in bandwidth required going from serial to two and four threads which can be readily supplied.

7) NPB - SP.B: In Figure 14, most of the regions are low bandwidth intensive regions and hence scale well when parallelized. In the region *compute_rhs_3*, the required bandwidth for four threads is significantly higher, causing the slowdown when going from two to four threads.

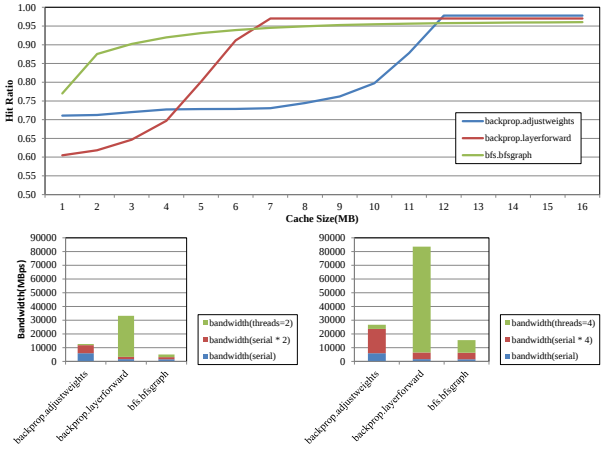

Fig. 15. **Rodinia: backprop, bfs**: *(a) CHiP (b) 2 threads (c) 4 threads*

8) Rodinia - backprop, bfs: In Figure 15.(a), the region *backprop.layerforward* has an interesting *CHiP* profile as the region has a high hit ratio when given a cache of 8 MB. When run in parallel, we can observe that there is an increase in required bandwidth because of the increased LLC misses in Figure 15.(a),(b). In Figure 3, we can see the effect of the increased bandwidth requirement on this regions speedup.

9) Rodinia - lud: The *lud* benchmark has two significant parallel regions which have a smooth *CHiP* profile as shown in Figure 16.(a). From this profile we can observe that a cache size of 16 MB will give us a very high hit ratio in the serial version. Also, when run in parallel, there will be a significant increase in the miss ratio which will cause a spike in the required bandwidth as shown in Figure 16.(b),(c).

10) Rodinia - srad - v1, v2: The *v2* version of the *srad* benchmark has parallel regions which show a linear *CHiP* profile. Having bigger caches for such a benchmark will not result in significant performance improvement. The *CHiP* profile for the *v1* version of this benchmark shows us

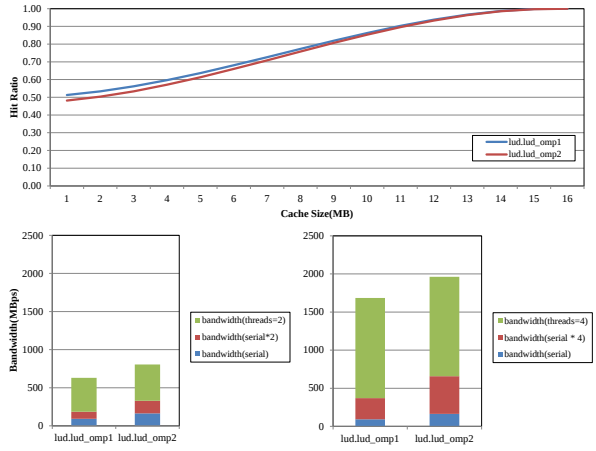

Fig. 16. **Rodinia: lud**: *(a) CHiP (b) 2 threads (c) 4 threads*

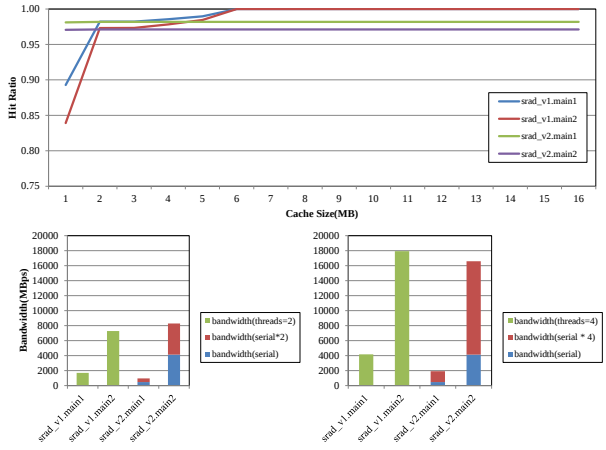

Fig. 17. **Rodinia: srad**: *(a) CHiP (b) 2 threads (c) 4 threads*

that having an 8 MB cache will give significant benefit for hit ratio in the serial version. But when parallelized, there is a 2.5% drop in the hit ratio which causes a significant increase in the required bandwidth as shown in Figure 17.(b),(c).

VII. Conclusion and Future Work

We presented a method of constructing a Cache Hit Profile of a to-be parallelized region of code in the serial program, which can be used to predict the possibility of poor parallel scalability. We show that *CHiP* profiler can identify code sections that could cause cache contention. In our future work, we will apply our *CHiP* profile algorithm along with other parallel speedup prediction algorithms to estimate possible speedup. Tools of this kind will be very useful in aiding the programmer to identify potential problematic regions of code and in fixing identified problems. The *CHiP* profiler can be also used in many other performance analysis tools to estimate cache hit/miss ratio when cache size varies.

VIII. Acknowledgements

We would like to thank John Pieper and the members of the Discovery Analyzers group at Intel, the Georgia Tech HPArch members and the anonymous reviewers for their valuable suggestions and feedback. Most of this work was done while Pranith Kumar was on an internship at Intel. We gratefully acknowledge the support of Intel and NSF CAREER award 1139083 for Hyesoon Kim.

References

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," in *PPOPP '95: Proc. of the fifth ACM SIGPLAN Symp. on Principles and practice of parallel programming*, 1995.

[2] "OpenMP," http://openmp.org/wp/, The OpenMP Architecture Review Board.

[3] *Intel Parallel Advisor*, Intel Corporation, http://software.intel.com/en-us/articles/intel-parallel-advisor/.

[4] C. L. Donghwan Jeon, Saturnino Garcia and M. B. Taylor, "Kismet: Parallel speedup estimates for serial programs," in *OOPSLA'11*, 2011.

[5] M. Kim, P. Kumar, H. Kim, and B. Brett, "Predicting potential speedup of serial code via lightweight profiling and emulations with memory performance model," in *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[6] B. T. Bennet and V. J. Kruskal, "Lru stack processing," *IBM Journal of Research and Development*, vol. 19, no. 4, pp. 353–357, 1975.

[7] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, ser. PLDI '03, 2003.

[8] G. Almási, C. Caşcaval, and D. A. Padua, "Calculating stack distances efficiently," in *Proceedings of the 2002 workshop on Memory system performance*, ser. MSP '02, 2002.

[9] D. L. Schuff, M. Kulkarni, and V. S. Pai, "Accelerating multicore reuse distance analysis with sampling and parallelization," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT '10, 2010, pp. 53–64.

[10] Y. Zhong, S. G. Dropsho, X. Shen, A. Studer, and C. Ding, "Miss rate prediction across program inputs and cache configurations," *IEEE Transactions on Computers*, vol. 56, no. 3, p. 2007, 2007.

[11] Y. Jiang, E. Zhang, K. Tian, and X. Shen, "Is reuse distance applicable to data locality analysis on chip multiprocessors?" in *Compiler Construction*, ser. Lecture Notes in Computer Science, 2010, vol. 6011, pp. 264–282.

[12] C. Ding and K. Kennedy, "Bandwidth-based performance tuning and prediction," in *IASTED Conference on Parallel and Distributed Computing and Systems*, 1999.

[13] C. Caşcaval and D. A. Padua, "Estimating cache misses and locality using stack distances," in *Proceedings of the 17th annual international conference on Supercomputing*, ser. ICS '03, 2003.

[14] Y. Liu and W. Zhang, "Exploiting stack distance to estimate worst-case data cache performance," in *Proceedings of the 2009 ACM symposium on Applied Computing*, ser. SAC '09, 2009, pp. 1979–1983.

[15] S. Ghosh, M. Martonosi, and S. Malik, "Cache miss equations: a compiler framework for analyzing and tuning memory behavior," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 4, pp. 703–746, Jul. 1999.

[16] N. Bermudo, X. Vera, A. Gonzalez, and J. Llosa, "An efficient solver for cache miss equations," in *Performance Analysis of Systems and Software, 2000. ISPASS. 2000 IEEE International Symposium on*, 2000, pp. 139 –145.

[17] A. Jaleel, R. S. Cohn, C. keung Luk, and B. Jacob, "Cmp$im: A binary instrumentation approach to modeling memory behavior of workloads on cmps," Tech. Rep., 2006.

[18] C.-K. L. et al., "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.

[19] E. Z. Zhang, Y. Jiang, and X. Shen, "The significance of cmp cache sharing on contemporary multithreaded applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, pp. 367–374, 2012.

[20] "Intel Core i7 2600 Quad core Processor," http://ark.intel.com/products/52213, Intel.

[21] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance," NASA, Tech. Rep., 1998.

[22] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC'09*, 2009.