Thread Scheduling for Chip-Multiprocessors: Final Report

Ravishankar Krishnaswamy (ravishan@cs.cmu.edu)
Harshavardhan Simhadri (harshas@cs.cmu.edu)

December 3, 2009

Abstract

Most parallel programs exhibit more parallelism than is available in processors produced today. While a user may optimize a program to incur few cache misses and run fast, when such a program is cut in to threads and run in parallel, it's run time is heavily dependent on the scheduler used to coordinate these threads. We investigate the case of sceduling for systems with cache hierarchies. We contrast schedulers such as the work stealing scheduler used in programming tools such like Cilk++ with a scheduler we propose. We argue that work stealing may not be optimal under all workloads for such systems.

1 Introduction

Because of the speed limitations imposed by hardware on a single core (threaded) design, much effort has gone into making processors faster by creating many smaller cores and using parallelism in the application to create multiple threads that can use this multiple cores. Since multiple cores are printed on a chip at the expense of complex logic used to accelerate prefetching and prediction logic, it is important for the application to be able to work well with this limited capabilities.

In most parallel applications, it is easy to find which parts can be parallelised. However, it is difficult to figure out how to arrange this parallel computations over the many available cores in such a way that the execution incurs low memory latency due to the distributed nature of the caching system. Even when the algorithm has enormous parallelism, a good thread scheduler is crucial to get good a run time.

1.1 The problem

The problem of schedulers for completely shared [BG04] or completely distributed caches [BL99] is well understood. In fact, the latter work has been successfully implemented as the Cilk++ [BJKLRZ96] run time system. Cilk++ (like OpenMP) enables the programmer to write C programs with simple directives to indicate parallel tasks and get an executable with the scheduler suitable for the given directives built

in. Similar research on implementing schedulers for shared cache machines [N99] has been done. However, there is no consensus on schedulers for hierarchical caches that are neither completely distributed nor shared. The work on provably good schedulers for divide and conquer algorithms [BCGRCK] provides a suitable starting point for this work. Our scheduler is a multi-level generalization of a slightly more aggressive variant of scheduler proposed in this work.

We intend to build a system similar to Cilk++ or OpenMP for such systems which allows the user to specify points of parallelism in a nested parallel program, and uses a smart scheduler to coordinate this parallel pieces of computation. We explore a scheduling algorithm suitable for reasonably parallel programs for systems with such cache hierarchies. Further, we assume that when a new thread is spawned, the programmer is able to provide us with a close upper bound on the number of distinct memory accesses of each threads (including all nested computation spawned by this thread).

While we limit our study to cache hierarchies that look like fat trees, insights learned from such a setting could also be used for cache systems with grid interconnects. For instance, one may view a 9×9 grid as a collection of 9 sets of 3×3 nodes. Such sets may be views as nodes in tree with memory as the root and processors at the leaves.

2 Our approach

[BL99] describes the work stealing scheduler for distributed caches. In a work stealing scheduler, a queue is assigned to each processor. When the thread a processor is running splits in to a set of parallel threads, the processor continues to work on one of these threads and pushes the rest at the end of it's job queue. When a processor runs out of work it takes a job from the front of it's queue. In case there are no jobs in it's queue, it steals a job from the front of another processor's queue. This is a greedy scheduler which can be shown to be reasonably cache friendly for distributed caches. It also works well in practice for such machines.

However, in a cache hierarchy, a processor is not aware of the fact that not all steals are of the same cost. Steals from processors separated by several upper level caches are costly and should be avoided when jobs are available closer in the memory hierarchy. We propose the following non-greedy scheduler *(HR scheduler)* that makes better use of locality in computation:

- Associate a job queue and a bit (called `is_blocked` bit) with each cache.
- Assign the root thread to any processor. If a processor spawns a new thread, check the space requirement of the newly spawned threads. If the space falls between that of a level L_i cache and that of L_{i+1}, push the newly spawned jobs in to the queue of the L_{i+1} cache under which the current processor falls. Also set the `is_blocked` bit of this cache to one.
- If a processor runs out of jobs, it starts searching for a new job starting from the cache closest to it. If it finds a job in it's $L1$ cache, it works on it. If this queue is empty and the $L1$'s `is_blocked` bit is 0, it moves to the cache one level up. It continues this process until it finds a job or it comes across a cache whose `is_blocked` is set to 1. In the latter case, it waits till this bit is reset to 0 or some job shows up on one of the lower level caches between the processor and the blocked cache.

Further motivation for choosing this scheduler is as follows:

- The cache complexity of a computation under this scheduler is almost the same as that under a sequential schedule.
- It minimizes cache invalidations over large distances

This scheduler is non-greedy. The scheduler might wait on a blocked cache when work is available elsewhere. This is not a problem for algorithms where the sequential parts of a computation do not need more than the space available in an L1 cache. This is usually the case with most parallel algorithms - they can be written in a way such that the individual sequential threads are small.

Our implementation approach

Implementing thread scheduling at the OS level would be the most time efficient approach. However, this approach is inflexible in that it does not allow users to choose their scheduling. Further, building a secondary scheduler on top of OS thread scheduler is useless and works against the purpose. Therefore, we choose to implement our scheduler at the user level on top of the OS level basic thread scheduler (which simply leaves the threads in their place in case there are no higher priority threads).

An alternate approach would have have been to collect program trace (on the fly), and feed this trace in to memory unit simulators such as Ruby, a module in the GEMS simulator which can handle cache coherence on chip multiprocessors with various different interconnects. However, such an approach while providing great experimental control would not be useful for programmers directly. Moreover, getting per access timing information from Ruby's interface to compute program run times seems to be a round about way.

The most suitable approach for both experimental verification and as a programmer's tool seems to be to implement our own software-level *thread-pool*, and scheduling tasks of a parallel program on these threads. The programmer specifies points of parallelism and our tool takes such a program, and runs them using a thread pool. Any spawn called inside our parallel program would invoke a call to our thread pool, which will then look for idle threads and assign the spawned task to such an appropriate idle thread (or queue the job in a suitable queue for later execution). The main exploratory part of the project is to tackle the problem of choosing the ideal job for an idle thread at a given time.

2.1 Our contributions

- We have built a framework to allow write programmers right parallel programs with out thinking about actual system threads. The programmer just uses spawn and sync procedures provided by the interface to write the program.
- We have described a candidate scheduler suitable for hierarchical memories and studied it experimentally.

2.2 Other Related Work

[CGKLABFFHMW07] has an experimental study of work stealing and PDF schedulers. In [LJWZ08], Li et al. consider the setting of a shared memory multiprocessor, and argue

for the case of a scheduler which determines the bandwidth utilization of the two threads of a fork operation, and test if the combined utilization exceeds some threshold value. If so, the threads would be scheduled on different processor clusters that do not have the same paths between the common memory and the processors. If not, then the threads may be allocated on the same processor cluster that shares cache among processors. In some sense, their idea is also a means of distributing load of a multi-threaded program across different processors in a non-trivial manner.

In [CPKCPRSPM05], Chen et al. propose an architecture for chip multiprocessors where even the scheduling decisions are done at the hardware level. They contend that by building even a simple scheduling policy at the hardware level, they could improve the performance of the multi-threaded task several fold, simply because of the speed (and simplicity) of the logic.

On an orthogonal front, in [HFFA09], Hardavellas et al. focus on the equally important problem of non-trivial *data placement* on a chip multiprocessor. **R-NUCA** is an architecture which places data blocks on the chip by classifying these into distinct classes, and observing that each class would lend itself to a different policy.

3 Design Overview

In our design, the core of the program is a **Thread Pool Manager**, which *spawns* N worker threads and one scheduler thread up front, at the beginning of the program. The threads are set up so that each thread is preferentially assigned to an unique processor by the OS. All these threads are initially idle and asleep. The parallel program communicates with the thread-pool manager through an interface we call **Barriers**.

For an illustration, suppose a program wants to fork into two sub-components, divide the work into these two parts, and then finally synchronize to clean up after these two parts complete. Such a program would look like the following snippet using our barriers.

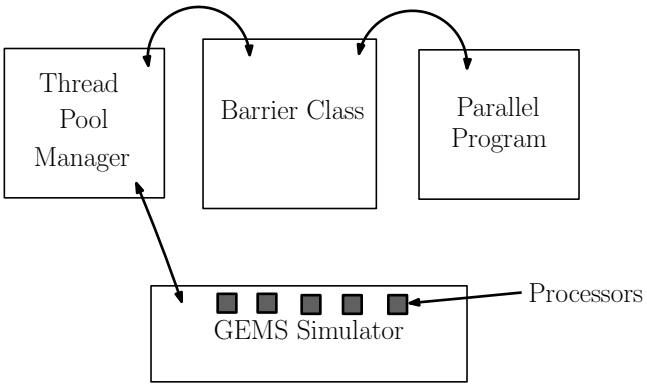

Figure 1: High Level Design

```
Barrier *newBarrier = new Barrier (2, (&fn_recursion_close),  params_rec_close, space_rec_c
newBarrier->spawn (0, (&fn_recursion), params1, space_requirement1);
newBarrier->spawn (1, (&fn_recursion), params2, space_requirement2);
```

This means that we associate this level of the recursion tree with a barrier, and link the two spawned threads to this barrier. Here, `fn_recursion` is the function called for both the new threads, and the parameters are passed as `params1` and `params2` – also, the `params1` and `params2` pointers contain a link back to the parent barrier for synchronization. The space requirement of each thread is also passed. The barrier created includes the function to compute after the barrier is reached, it's parameters and space requirement. The barrier includes space for the spawned threads to communicate any values with the function after the barrier. When the spawned one of the threads completes, it writes the return arguments onto this particular barrier, using the following sync operation:

```
(params.parentBarrier)->sync (my_local_tid, return_structure);
```

Once all spawned threads complete, the barrier automatically spawns the `addRecursionClose` function with the return values of each thread. This structure mimics the "divide and conquer" nature of parallelizable program. The function interface specified above is slightly simplified for presentation, the actual interface involves passing around few thread ids.

It is the `barrier` structure which acts as interface between the thread pool manager and the parallel program. When a spawn is called, it creates a new task (using the `pthreads spawn`) and adds it to the job queue. Immediately after adding it, the scheduler thread is woken up. The scheduler then looks at the job queue, and also looks at the list of idle processors, and sends this task to one of the idle threads — here is where the different scheduling algorithms take differing actions; an ad-hoc one might arbitrarily assign idle threads to tasks, whereas some other kind of scheduler might assign tasks to threads based on locality of data access, etc. Once the assigned task finishes, the thread becomes idle, and the scheduler is again woken up. Notice that the scheduler is always asleep, except when a thread becomes idle, or when a new task arrives as a result of a spawn. This keeps the overhead of the scheduler to a minimum.

3.1 Design Evolution

Our initial idea to study schedulers was to instrument a parallel program written in Cilk++ or OpenMP using Pin, and then use our scheduling algorithm as an intermediary between the trace provided by the instrumentation pintool and the Ruby memory module of GEMS — this design would have enabled us to take advantage of the Ruby's configurable cache coherence protocols and on-chip interconnections for chip multiprocessors (ref. Fig. 2). However, we found this approach to be too cumbersome and clumsy. Separating the GEMS memory module from the simulator and interfacing with it seemed to require significant amount of programming. After discussing this issue with Prof. Mowry, we decided on implementing our own **Thread Pool**. We now describe what our original design is, and why we decided to drop this plan. Furthermore, we had much difficulty trying to separate the GEMS simulator and interface directly to Ruby (the default method is to interface to Ruby through Simics). Therefore, we decided not to pursue in this direction, and subsequently came up with the cleaner and arguably much more realistic modeling of the scheduling problem.

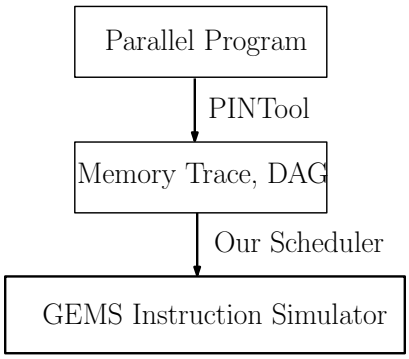

Figure 2: Original Design

4 Experiments

4.1 Setup

Because of problems with setting up the GEMS simulator, we decided to use a real machine to run most of our experiments. We used the `multi6` machine for most of our experiments. This machine has 4 dual core hyperthreaded processors (Xeon 7140m)- a total of 8 physical and 16 virtual processors. In most experiments, we chose not to use hyperthreading and use only 8 threads. Each core has it's own L2 (1MB) and L1 (16KB) caches. We only used the GEMS simulator for validating some of our programs using a bus interconnect under minimal cache latencies. Under such a setting, a parallel algorithm such as sorting written using our directives should display good speed ups as there are few artifices induced by memory access behaviour.

4.2 Work Load

Most Divide and conquer programs (such as sorting, matrix operations etc.) tend to have the following flavor: on an input n, divide the problem into certain parts, work on individual parts in parallel, and combine their result. For the algorithm to run well, the parallelisation is stopped after a certain stage, where the job is done sequentially. The amount of space requirement and work an algorithm does in this base case differs. While scans takes linear work and time ($w(n) = s(n) = s$), matrix multiplication does super linear work $w(n) = n^{3/2}$ and sort takes $w(n) = n \log n$ work. In algorithms involving super linear work, the same memory location is scanned several times. Our workloads model such behavior, they first cut down the input in to pieces that fit the lowest level of cache on which they operate sequentially. However, each workload does a different number of scans of the input they operate on. We evaluate the cases where the number of such scans is 1, $\log n$ and $n^{1/2}$. Further, at each fork, we split the jobs asymmetrically so as to induce job stealing. We note that our time taken by the scheduling thread is very low. When we modify above algorithms to fork threads and create new jobs and set the base cases to return with out doing any work, all programs complete in under 3 milliseconds.

Run times (average over 10 runs)			
Input size	$W(n)$	HR Scheduler	WS Scheduler
$512MB$	n	3.0($\pm$0.1)s	3.0($\pm$0.1)s
$8MB$	$n \log_2 n$	1.9($\pm$0.3)s	2.2($\pm$0.3)s
$8MB$	$n^{4/3}$	17($\pm$2)s	27($\pm$3)s

Table 1: Run times with different schedulers using 8 processors

4.3 Results

Table 4.3 shows the runtimes of work stealing scheduler as against the the scheduler we propose. In the first case of linear work, there is not much difference between the two. Both incur the cost of loading the memory one and little else. In the next cases, however, since each piece of the array is executed multiple times, our scheduler which locks jobs to caches instead of moving them does better. The work stealing has to incur the cost of loading the array into the L1 cache several times as jobs are migrated over L2 caches at times. Further, as the non-linearity of the work increases, the difference between the two schedulers increases.

5 Surprises and Lessons

In order to implement our scheduler, we need to know which thread is assigned a processor above the OS level. Initially, we set about doing this by randomly assigning threads, and then finding out how far *apart* each pair of threads were by having these threads write to a common array. The farther the processors the thread pair was assigned to were separated, the greater their run time. However, we learnt that pthread library provides a function to specify which processor a thread should be preferentially allotted to. This would enable us avoid such diagnostics and assign threads to processors we want. While the pthread libraries largely obey such directives, we found that at times - even when the system has neglible interrupts from other active processes - threads tend to migrate between processes every few seconds. We need to further investigate the cause and solution of this problem.

6 Conclusions and Future Work

In this project, we have (i) studied different thread scheduling algorithms for chip multiprocessors; (ii) built a framework wherein plugging in a new scheduling algorithm and measuring the timing analysis/cache miss analysis, etc. is *extremely simple*. We have shown that work stealing scheduler may not be ideal for all workloads, and smarter schedulers are needed to make use of cache locality. Directions for future work include:

- The scheduler we proposed does not steal any jobs except from the caches above it, while work stealer steals from everywhere. A flexible scheduler that can be configured to steal only under certain conditions such as from a limited distance or only when idle for certain time might be better in certain cases. This needs to be studied.

- We could not analyze details of cache statistics because we used a real machine. It would be interesting to obtain these statistics on a simulator.
- We used simple workloads for our studies. It would be interesting to study real workloads by reprogramming them using our framework.

7 Acknowledgment

We borrowed some code for creating and handling threads from the Thread pool class implementation written by Dr. Ronald Kriemann obtained from
`http://www.hlnum.org/english/projects/tools/threadpool/index.html`.

References

[BCGRCK] Guy E. Blelloch, Rezaul A. Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, Provably Good Multicore Cache Performance for Divide-and-conquer Algorithms SODA '08: Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, 2008

[BG04] Guy Blelloch, Phillip Gibbons Effectively sharing a cache among threads Proceedings of the 16th ACM SPAA, 2004.

[BJKLRZ96] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, Yuli Zhou Cilk: An Efficient Multithreaded Runtime System Journal of Parallel and Distributed Computing, 55 69, August 25, 1996

[BL99] Robert D. Blumofe, Charles E. Leiserson Scheduling Multithreaded Computations by Work Stealing Journal of the ACM, 720 748, September, 1999.

[CGKLABFFHMW07] Shimin Chen, Phillip Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd Mowry and Chris Wilkerson Scheduling Threads for Constructive Cache Sharing on CMPs Proceedings of the 19th ACM SPAA, 2007.

[CPKCPRSPM05] Julia Chen, Juang Philo, Kevin Ko, Gilberto Contreras, David Penry, Ram Rangan, Adam Stoler, Li-Shiuan Peh, Margaret Martonosi. Hardware-modulated parallelism in chip multiprocessors. *SIGARCH Comput. Archit. News*, 2005.

[HFFA09] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, Anastasia Ailamaki/ Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches *Proceedings of the 36th ACM/IEEE Annual International Symposium on Computer Architecture*, 2009.

[LJWZ08] Wenlong Li, Aamer Jaleel, Tao Wang, and Yimin Zhang Thread Scheduling on Multiprocessor Systems. *Patent. Trop Pruner and Hu, PC*, 2008.

[N99] Girija J. Narlikar Space-Efficient Scheduling for Parallel, Multithreaded Computations Ph.D. thesis, Carnegie Mellon University, 1999.