Noname manuscript No.
(will be inserted by the editor)

Architectural Support for Task Scheduling

Hardware Scheduling for Dataflow on NUMA systems

Behram Khan · Daniel Goodman · Salman Khan · Will Tom · Polo Faraboschi · Mikel Luján · Ian Watson

Received: date / Accepted: date

Abstract To harness the compute resource of many-core system with tens to hundreds of cores, applications have to expose parallelism to the hardware. Researchers are aggressively looking for program execution models that make it easier to expose parallelism and use the available resources. One common approach is to decompose a program into parallel 'tasks' and allow an underlying system layer to schedule these tasks to different threads.

Software-only schedulers can implement various scheduling policies and algorithms that match the characteristics of different applications and programming models. Unfortunately with large-scale multi-core systems, software schedulers suffer significant overheads as they synchronize and communicate task information over deep cache hierarchies. In order to reduce these overheads, hardware-only schedulers like Carbon have been proposed, to enable task queuing and scheduling to be done in hardware.

This paper presents a hardware scheduling approach where the structure provided to programs by task-based programming models can be incorporated into the scheduler, making it aware of a task's data requirements. This prior knowledge of a task's data requirements allows for better task placement by

B. Khan
BT Research
E-mail: behram.khan@bt.com

S. Khan
Solarflare Communications
E-mail: salman.khan@gmail.com

D. Goodman · W. Tom · M. Luján · I. Watson
The University of Manchester
E-mail: Manchester@djgoodman.co.uk, tomsw@cs.man.ac.uk, mlujan@cs.man.ac.uk, watson@cs.man.ac.uk

P. Faraboschi
HP Labs
E-mail: paolo.faraboschi@hp.com

the scheduler which result in a reduction in overall cache misses and memory traffic, improving the program's performance and power utilization.

Simulations of this technique for a range of synthetic benchmarks and components of real applications have shown a reduction in the number of cache misses by up to 72% and 95% for the L1 and L2 caches respectively and up to 30% improvement in overall execution time against FIFO scheduling. This not only results in faster execution and in less data transfer with reductions of up to 50%, allowing for less load on the interconnect, but also results in lower power consumption.

Keywords Scheduling · Hardware Scheduling · Task based application · Dataflow

1 Introduction

Multi-core chips are now commonplace and provide applications with an opportunity to achieve much higher performance than the uniprocessor systems of the recent past. Furthermore, the number of cores on a chip is growing rapidly, thus increasing the performance potential. This trend has created a renewed interest in high-level dataflow and task-based parallel programming models such as OpenMP [1], Cilk [2], TBB [3], CUDA [4], OpenCL [5], StreamIt [6], OoOJava [7] and StarSs [8]. These models provide constructs to express parallelism and synchronization in a manageable way, and their runtimes take care of *resource management* and *scheduling*.

For efficient execution a scheduler should ensure that:

- The execution units are well utilized, performing load balancing if needed.
- The number of concurrently active tasks remains within reasonable limits so that the memory requirements are not unduly large.
- Only small scheduling overheads are imposed.
- Related tasks are placed on the same core and core cluster, if possible, in order to take advantage of data locality, through utilizing the cache and memory bank structure.

The ability of the scheduler to realize these properties is constrained by the information available from the programming model. In this paper we describe our hardware implementation and testing of scheduling approaches for dataflow programming models demonstrating how the structure present in such models can be combined with simple hardware scheduling to provide significant performance improvement. While this performance improvement is dependent on the use of appropriate programming models to construct the application, applications constructed with other programming models can still run on hardware supporting this scheduling, just without the performance benefit provided by the additional structure being exposed to the scheduler.

Schedulers can make decisions that improve data locality if they are aware of the data requirements of the tasks, by placing tasks on cores whose caches contain the required data. Currently in high performance applications running

on dedicated resources, this is generally addressed explicitly by the programmer. Leaving aside the extra complexity which makes code harder to write and more fragile, in the general case a program will not be executing on dedicated resources. When sharing resources with other programs the programmer is unable to gather the required information about the processing environment in order to make the correct decisions, leaving them dependent on the system scheduler.

In the general case information about which tasks use which data is absent as a consequence of the way in which conventional imperative programming models have been extended to include the ability to perform parallel execution. However, models like dataflow programming are designed for parallel execution and contain additional information about the access patterns of the computation. The main characteristics of the dataflow model are that the execution of an operation is constrained only by the availability of its input data. The computation is performed by side effect free tasks and the execution is triggered by the presence of data instead of the explicit flow of control. These constraints prevent both deadlocks and race conditions. Pure dataflow as described here introduces some limitations to the set of problems that can be solved. These limitations are addresses through work combining the pure elements with well managed areas of mutable state [9]. These modifications do not effect the work presented here and are outside of the scope of this paper.

In this paper we demonstrate how the structure provided to programs by the dataflow programming model can be incorporated into task schedulers, making them aware of a task's data requirements without any further help from the programmer. Our major contributions are:

- An examination of how the use of a dataflow programming model can, by allowing more intelligent scheduling techniques, improve system performance.

- We propose two novel scheduling policies, 'Token Scheduling' and 'Reference Scheduling' and demonstrate how these scheduling policies result in better resource utilization.

- We propose the design of a scalable hardware scheduler that has low hardware complexity and is relatively insensitive to the access latency of the hardware queues.

- A demonstration that the proposed architectural support has significant performance benefits (section 5) and the scheduling policies have much better resource utilization when compared with other scheduling policies. Our scheduling policies result in a reduction in cache misses by up to *72%* and *95%* for the L1 and L2 caches respectively and up to a *30%* improvement in overall execution times against FIFO.

The rest of the paper is organized as follows: Section 2 introduces the dataflow programming model explaining how it adds the required extra struc-

ture to programs which our schedulars can then take advantage of. Section 3 introduces our proposed scheduling policies. Section 4 describes our scalable hardware scheduler based on our scheduling policies. Section 5 presents our experimental methodology and evaluation results. Section 6 discusses related work, and Section 7 concludes the paper.

2 Dataflow

Dataflow programming is a model of computation that provides an effective means of constructing deterministic, race condition free parallel programs. It was first seriously examined in the 1970s and 1980s when it was thought that parallelism would soon become essential to achieve continuing increases in performance. This resulted in the construction of real dataflow machines [10, 11] which demonstrated the scalability of the approach, and a large amount of work on the implementation of programming languages suited to these systems [12]. Refinements to the dataflow model were also developed which provided data structure handling and support for lazy evaluation to enable the implementation of more advanced functional programming approaches [13–15]. However, chip development at the time was able to make parallel programming unnecessary in the general case. Now that this is no longer possible, dataflow programming is under a range of guises [16,7,17] once again a serious area of research.

In a dataflow model the program is split into independent deterministic pieces called dataflow threads or dataflow tasks, each of which is only dependent on a known set of inputs. Depending on the granularity of the program, dataflow tasks can vary in size from a single instruction to whole functions including calls to other functions. This allows arbitrarily large computational units. Only once the set of inputs has been passed to a dataflow task can it be scheduled for execution. To ensure that there are no race conditions introduced between the use and modification of task inputs, all data used as task inputs is immutable. When run, a task may generate outputs for other tasks and may also generate additional tasks. This flow of data between tasks means that the execution of the program can then be visualized as a directed acyclic graph where the nodes are the dataflow tasks and the vertices are the data dependencies between these. The dataflow model is asynchronous and self-scheduling since the execution of nodes is constrained only by the data dependencies. An example of this can be seen in Figure 1 and the corresponding code in Figure 2.

As dataflow tasks are only dependent on their input data and this data is immutable, the scheduling of tasks is only dependent on the flow of data between tasks, not on the explicit progression of a program counter. This adds to the program an explicit description of when data is passed and the guarantees that there are no additional data dependencies, so providing structure to the program that is absent in traditional procedural programs, and it is this structure that the work described in this paper builds on.

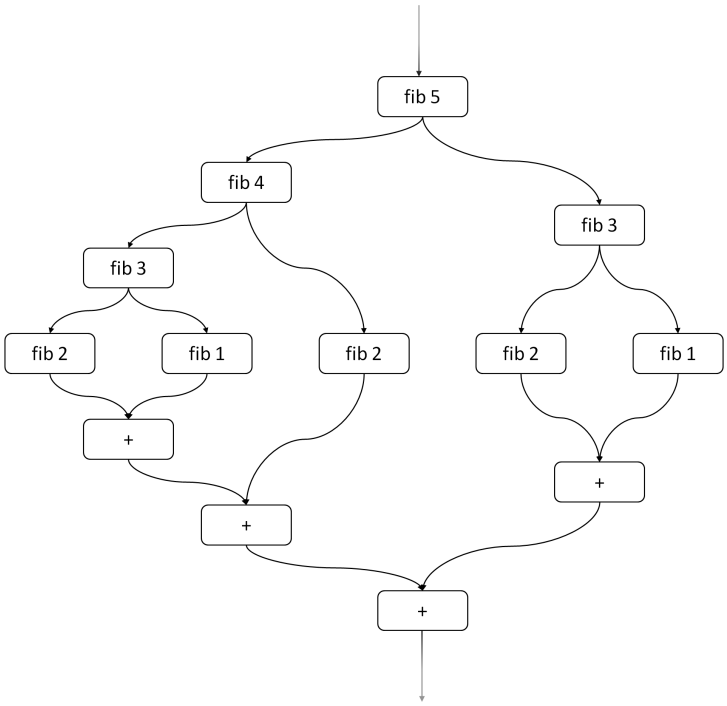


Fig. 1 An instance of a dataflow graph to calculate the 5th Fibonacci number.

```
void fib()
{
    int n = read(1); // receive n
    if (n < 2) {
        ......
    }
    else {
           f1 = schedule(&fib, 2);  // spawn fib1, waiting for 2 arguments
           f2 = schedule(&fib, 2);  // spawn fib2, waiting for 2 arguments
           f3 = schedule(&add, 3);  // spawn add, waiting for 3 arguments

           f1.arg1 = n-1;  // send fib1, n-1
           f1.arg2 = f3;   // send f1 the target task for its result

           f2.arg1 = n-2;  // send fib2, n-2
           f2.arg2 = f3;   //send f2 the target task for its resuult

           ......
           ......
    }
}
```

Fig. 2 A simplified part of a dataflow function for computing Fibonacci numbers. The function takes as an argument the value it is to compute, the Fibonacci number. It checks if this is the base case. If it is not, it constructs two new tasks, fib1 and fib2 which take `fib` as their function with the arguments `n-1` and `n-2` respectively.

2.1 Scheduling

Dataflow is an asynchronous and self-scheduling model where the execution of nodes is constrained only by data dependencies. From the code, Figure 2, we can see that each 'fib' function creates two new tasks 'fib1' and 'fib2'. The `schedule(&fib, 2)` creates a dataflow task which executes `fib` as its function once all the dependencies are computed. It also informs the dataflow scheduler that this task is waiting for only two pieces of input data to be ready for execution. Finally, `fib1`'s and `fib2`'s arguments are set to the value $n-1$ and $n-2$ respectively, and the task to pass their results to, making them ready for execution. Tasks can be passed as arguments to other tasks, so there is no requirement for all the input to come from the constructing task.

The sudo code used here is based on a C library and is used for familiarity, more advanced libraries such as DFScala [18] allow for cleaner more concise syntax.

Figure 3 shows how tasks are managed by a dataflow scheduler using example code of Figure 2. Note that this is just an abstract view of the scheduling of a task taking place, the discussion about the actual design and implementation of the scheduler is discussed in later sections. The process is split into the following three stages.

Task Creation The first step in the life of any task is its creation. Tasks are created by an existing tasks, in this case task `t1` running on core 1. Created tasks have a function that they are going to execute and a number of arguments that they require. Note they do not need to know where these arguments will come from. Newly created tasks are placed in the pending queue to await their arguments.

Receive Arguments and Calculate Affinity Over the course of a tasks lifetime it will receive arguments which are placed into its tuple. Once all of the arguments have been received the task can be moved into the ready queue to await execution. The scheduler maps a task to a core when it is moved to the ready queue. This can be seen in step 2. At any given time there will be a set of available cores and a set of available tasks that need to be mapped onto these cores. It is how best to construct this mapping given the available information that we explore here.

Schedule on a Core Once a task is in the ready queue it can be placed onto an available core when a core requests for a task (step 3), in this case core 2.

2.2 Using Dataflow Dependencies for Cache Aware Scheduling

In multi-core processors typically cores or small groups of cores will have private caches. If over the course of a program's execution, tasks provided to a thread pool are scheduled on different cores, but are working on the same

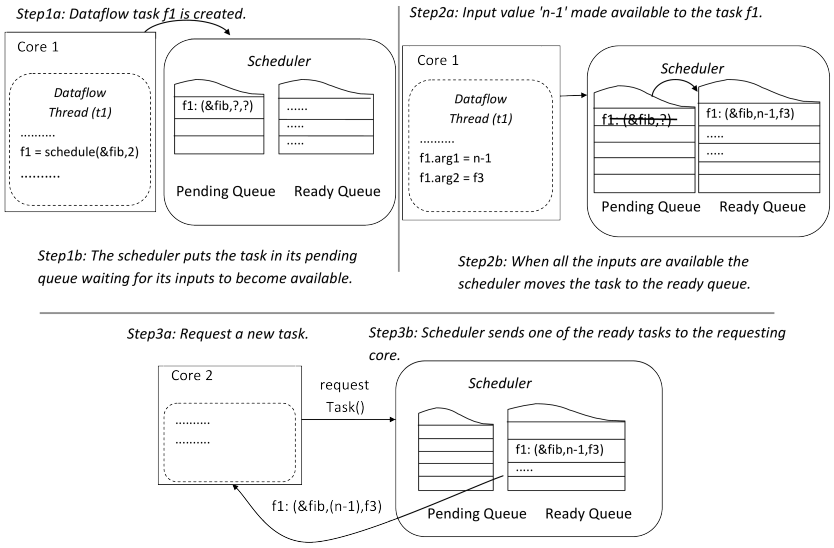


Fig. 3 Dataflow Scheduling (abstract view).

data, cache misses will occur resulting in delays to the programs execution and the use of system resources to transfer the required data to the computation. If both computations are scheduled to the same core this can be avoided, as demonstrated in Figure 4. Unfortunately in standard procedural programming models threads and tasks do not provide the scheduler with information describing the data they will access, so the scheduler is unable to take this information into account. In dataflow models, which data a task is going to use and where its data was generated and used recently is known in advance. This means that it is possible to group tasks that use the same data on the basis of these data dependencies. It is important to note that the use of this information is not an all or nothing situation, but is something that can be included into existing scheduling algorithms when it is available to improve their performance, falling back on traditional scheduling when the information is not available.

3 Novel Scheduling Policies

In this section, we describe and discuss two strategies to allow the task schedule to take advantage of the structure provided to programs by the dataflow programming model. By using this information the scheduler is aware of a task's data requirements and can make better decisions. The detailed design of suitable hardware to support these policies and the evaluation of their effectiveness is discussed in Sections 4 and 5.

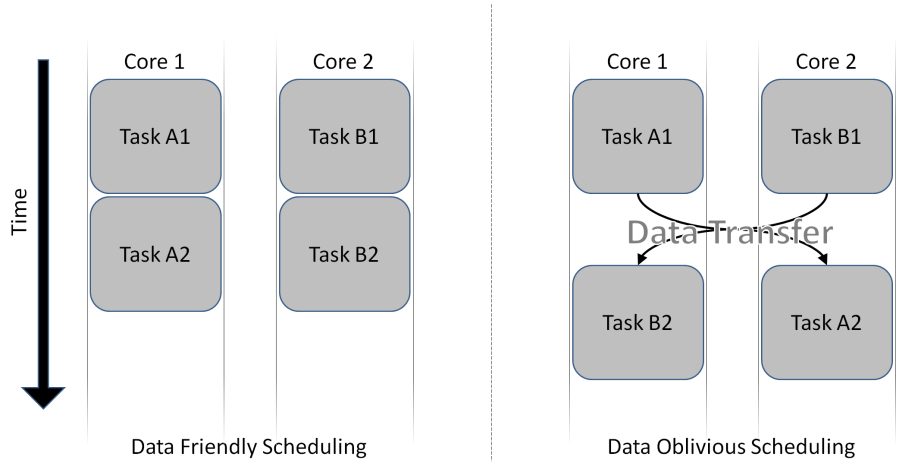


Fig. 4 An example of the time saving that can be achieved through data locality. On the left the time if data locality is observed, on the right the events if it is not and the resulting delay while data is transferred.

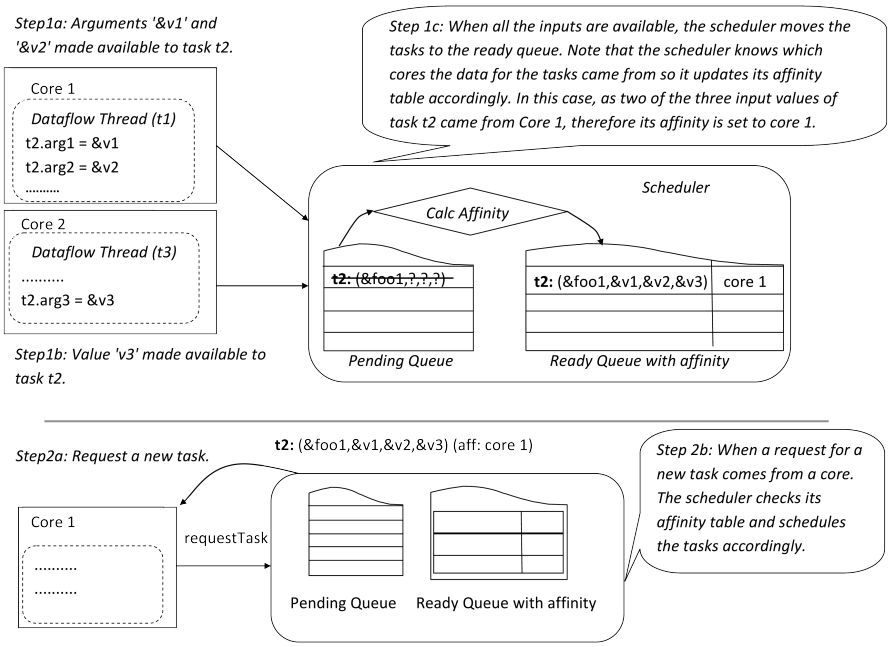


Fig. 5 Token Scheduling.

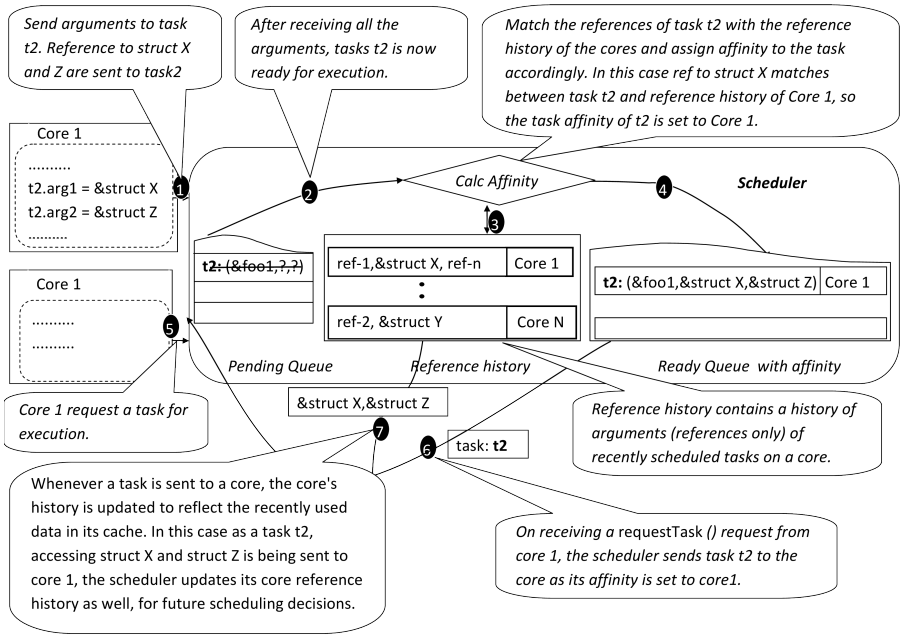


Fig. 6 Reference Scheduling.

3.1 Token Scheduling

This scheduling strategy relies on the assumption that if a reference to a data structure (Token) is passed from a task running on core x, the probability of that structure being cached by core x is high. So scheduling the receiving task on core x can result in better data locality and as a result better cache utilization. It is important to remember that a task can be sent tokens from many different tasks and not just the task that spawned it, and that it is not uncommon for a parent task to spawn tasks and then not pass any data to them.

To take advantage of this observation we change the following two steps in the scheduling process.

Receive Arguments and Calculate Affinity Record the received reference argument (Token) *AND* the core that the argument was received from. Once all of the arguments have been received the task can be moved into the ready queue to await execution. Note that the scheduler knows which cores the data for a task came from, so it updates the affinity information in the ready queue accordingly. The affinity of a task to a core is based on the number of references it receives from a core.

Schedule on a Core Once a task is placed in the ready queue it is placed on the core from which the highest proportion of references came from. Note this

may be a core from which no references came from, for example when the program first starts only one core will have passed any references to any tasks. In this scenario if any other core requests for a task, the scheduler randomly selects a task form the ready queue to be placed onto the requesting core. In the event of multiple tasks with the same percentage of references from the core, one can be selected at random from this set.

This is demonstrated in the example in Figure 5. In step 1a the task t1 running on core1 passes 2 arguments to task t2. In step 1b task t3 running on core2 sends a single argument to task t2 thus making t2 ready for execution. When core 1 requests a new task in step 2a, the scheduler looks at its list of ready tasks and then assign the task which has the highest affinity value for that processor as shown in step 2b.

3.2 Reference Scheduling

In reference scheduling, instead of the task recording where its inputs came from, the scheduler records which references each core has recently accessed and tasks are assigned to the cores by matching the set of references used by the task to each cores reference history. This variant is able to take advantage of the fact that the data may exist in multiple cores caches at the same time. To take advantage of this observation we change the following two steps in the scheduling process.

Receive Arguments and Calculate Affinity Record the received reference argument (Token) *AND* the core that the argument was received from. Once all of the arguments have been received the task can be moved into the ready queue to await execution. The scheduler maintains a reference history of each core and the affinity of a task to a core is calculated by matching the set of references used by the task to each cores reference history. The core with the reference history that provides the highest affinity with the task arguments is selected.

Schedule on a Core When a core sends a request for a task, the scheduler selects a task to be scheduled based on the affinity calculated in the previous step. Once the task is selected the scheduler updates the cores reference history with the tasks arguments. As the function pointer is also a reference this can also be included in the reference history allowing for improved caching of both code and data. Each cores reference history is small and bounded with a weighting placed on each entry to record its age. This reflects that older items are less likely to still remain in the cache. When new items are added existing items in the history are decayed until they are ultimately evicted on the assumption they are unlikely to still be in the cache.

An example of this can be seen in Figure 6 showing how the reference history is obtained and maintained by the scheduler using the task information present in the programming model. When task `t2` becomes ready, the scheduler

```
void func1()
{
    *struct_A = read(arg);  //receive an arg: Reference to structure 'A'.
    .....
    .....
    .....
    f1 = schedule(&func2, 1);  // spawn func2
    f1.arg = struct_A;         //pass the pointer to the next task

}
```

Fig. 7 A dataflow function which takes as an argument a reference to a structure and passes it onto the next function without 'using' it.

compares references passed to it in its argument list to the reference history of the cores. In this case the scheduler sees that the reference to data structure X is present in core 1's history, so it assigns the task `t2` to core 1. When the task `t2` is sent to core 1 for execution, the reference history of core 1 is updated so that it now contains the reference to structure Z as well.

Our reference scheduling model differentiates between 'used' references (references that are actually used by the dataflow task) and references that are not used and are just passed through to the next task in the dataflow graph. In Figure 7, reference to structure `A` is not considered when recording references of this task for scheduling. In reference based languages such as Java, and Scala [19] this status can be determined automatically at compile time with a high degree of accuracy. For pointer based languages such as C and C++ this would have to be inserted by the programmer.

All the affinities are only hints to the scheduler. The affinities do not bind the tasks to any particular core and in cases where a scheduler gets a request from a core for which it has no suitable tasks based on the affinities, the scheduler will reply with a task picked using conventional scheduling methods.

4 Architectural Support for Task Scheduling

In this section we describe the hardware design of our scheduler. A centralized queue is the simplest way to implement task queues in scheduling, and the block diagrams demonstrating our scheduling policies showed centralized queues for simplicity. In a centralized system, all the tasks are enqueued and dequeued from a single shared queue. While this is sometimes acceptable, a single queue can quickly become a bottleneck as the number of cores scale up. To address this bottleneck and allow better throughput and latency times, software and hardware schedulers often use distributed tasks queues with task stealing [20–24].

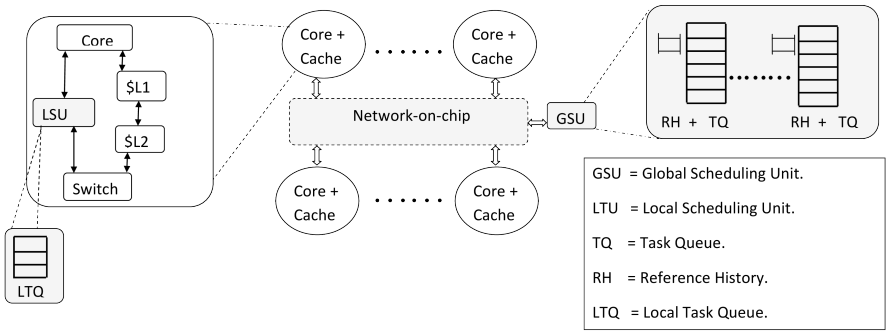


Fig. 8 Example of many-core system with our hardware support for task queues. The shaded portions are additional hardware for the scheduler.

4.1 Design

Our design provides for low overhead distributed task queues, and is tolerant to increasing on-die latency as the number of cores in the system scales. This is achieved by implementing the distributed task queues in the hardware. The tasks are stored in the hardware queues, scheduling is implemented in hardware, and we have hardware task prefetchers so that each hardware thread can start a new task as soon as it finishes its current one.

From the tasks queue hardware perspective, a task is simply a tuple. In this implementation, it is a tuple of 64-bit values; a function pointer and pointers to shared data passed to a task as arguments. Similarly the core reference history is also a tuple of 64-bit values. Reference histories could be implemented as bloom filters [25] to make the comparison between task reference and history references cheaper, thus making the hardware for comparison much simpler, quicker and more energy efficient. The hardware task queues have limited capacity, in order to support a virtually unbounded number of tasks for a given processor, and to support virtually unbounded number of processors, we can extend our model to move tasks out of the hardware tasks queues into the memory system using a mechanism like those used in Carbon [26].

Our design considers a Chip Multi-Processor (CMP) where the cores and last-level caches are connected by an on-die network. This design has two main components: a centralized global scheduling unit (GSU) and a per-core local scheduling unit (LSU). Figure 8 shows our design.

4.1.1 Global Scheduling Unit (GSU)

The Global scheduling unit holds enqueued tasks in a set of hardware queues with one queue per core in the system. This could be extended to implement a hardware queue per hardware context. To keep the hardware simple the hardware queues support insertion and deletion of tasks at either end of the queue but not in the middle. In Section 5 we use the simulator to evaluate

the cost of not allowing random access to the hardware queues in order to maintain simpler hardware. The global scheduling unit implements the task scheduling policies described in Section 3. Since the queues are physically located close to each other, the communication latency between the queues is minimized.

4.1.2 Local Scheduling Unit (LSU)

Centralized scheduling systems may not scale with the number of cores in the system. This issue is addressed with Local Scheduling Units. Each core in the system has a local scheduling unit that provides an interface between a core and the GSU. The LSU is used to hide the latency of dequeueing a task from the GSU by buffering a small number of ready tasks, this number would typically be a single task. To do this the LSU contains a task prefetcher and a small task prefetch buffer. Without the LTU, if a thread sends a task request to a GSU it will stall waiting for the response from the GSU, with the LTU the GSU only has to find a new task for each core in the time it takes a task to complete its execution, this should take many orders of magnitude longer than the task selection. Eventually there may be enough cores for this to become an issue which will require further parallelisation in the GSU or course grained tasks, but we do not believe this will occur in the near future within a single shared memory system.

When a task is requested by the core's hardware thread, the LSU returns a task to the core and sends a prefetch request for the next task to the GSU. The LSU buffer should be large enough to hide the latency of accessing the GSU. In our benchmarks we find that buffering a single tasks is sufficient to hide the GSU dequeue latency. Because the dataflow graph has provided in advance the information about which references a task will use, a buffer of 1 can be implemented without loss of precision.

4.2 Physical vs Virtual Addresses

All addresses currently stored by our implementation are virtual addresses. The likelihood of a collision on a specific address is small as this has to be the address that is passed to the task, not just an address reachable from the passed arguments. In the event that this does happen the code will still execute correctly, just with potentially less optimal scheduling, something we would like to explore as part of our further work. If it turns out that this does become an issue, it can be addresses either by flushing the reference history on a core every time it changes context as the likelihood of values surviving in the cache is small, or by converting to physical addresses and accepting the extra complexity.

Parameter	Configuation
#Processor	1-32
L1 ICache	Private, 64KB, 4-way, 2 cycles
L1 DCache	Private, 64KB, 4-way, 2 cycles
L2 Cache	Private, 2MB, 8-way, 20 cycles
Main Memory	500 cycles
Interconnection Network	2D-Mesh

Table 1 Architectural parameters used.

5 Evaluation

To conduct an evaluation we used the gem5 simulator [27]. gem5 includes a range of processor and memory models, with processor from purely functional to detailed out-of-order models and memory models from simple *classic* systems to the detailed Ruby modules. As our concern is the effect of scheduling on locality, precision in the micro architectural model was not necessary, so we model comparatively simple in-order X86 processors (the *SimpleTiming* model in gem5), accompanied by Ruby cache with a MOESI coherence protocol.

While we vary the number of processors simulated, all the experiments were carried out with 64KB private L1 data caches and a 2 MB unified private L2 caches. All caches were 4-way set associative. Table 1 summarizes the base system configuration.

To experiment with hardware scheduling, we add the hardware described in section 4 to the system. We applied a 20 cycle delay for an access (e.g., enqueue or dequeue) to the global task unit. This is in addition to the latency for the cores to message the GSU over the on-chip network. In Section 5.2.2 we evaluate if the sensitivity of the design and if this figure needs further consideration.

Within this simulator we implemented the five different scheduling policies: Random, FIFO, Source, Token and Reference Scheduling. These different strategies represent increasing levels of complexity for the scheduler. Our proposed policies of Token and Reference scheduling are already explained in detail in Section 3. Here we will briefly discuss the other scheduling policies with which we will be comparing our proposed schemes.

Random each processor is randomly assigned a task from a set of available tasks. This strategy is included to demonstrate that any improvements are not simply because we are introducing an element of randomness to the scheduling.

FIFO is our baseline and schedules tasks strictly in the order that they become ready.

Source Scheduling is a strategy that can take advantage of programs which are split into distinct parts. With this strategy, cores will preferentially run

tasks created by other tasks on that core. This approach is similar to one used by Carbon [26].

5.1 Benchmarks

To test the effects of our scheduling policies in the scheduler, we used a set of six benchmarks: Block matrix multiplication, iterative refinement for motion estimation, index searching, route planning, and two versions of kmeans. If we constrain these relatively small examples to the dedicated hardware available in HPC, it is in principle possible to manually achieve the same potential using conventional solutions, however this makes solutions that are fragile and may not exhibit performance portability. As programs increase in size and become more complex, or in more general scenarios where resources are not dedicated, but are shared with other programs, effective hard coding become untractable. The hard coding of strategies into the program also assumes that the programmer is able to correctly determine the appropriate strategy, real world problems are often too complex especially when such problems include input that is outside of the programmers control.

We will now discuss each of these benchmarks in more detail before discussing their performance for different memory scheduling techniques.

5.1.1 Block Matrix Multiplication

Block matrix multiplication of matrices A and B is a technique used to structure parallelism and reduce the size of the working set such that it fits within the processor's cache, improving overall performance. With this approach the resultant matrix C is split into blocks, each of which is calculated independently. Each independent calculation is computed from the matrix multiplication of two sub-matrices drawn from the rows of A and the columns of B. By correctly choosing the blocks size for moderate sized matrices it is possible to fit the whole of the two sub matrices in the cache, reducing accesses to main memory. Once a block is computed, if the next block to be computed is adjacent then only one of the sub-matrices needs to be replaced to continue, further reducing accesses to memory. To exercise the scheduler, each block in this example is computed by a separate task.

5.1.2 Iterative Refinement

With iterative refinement a transformation is repeatedly applied to a dataset until the data satisfies some convergence criteria. This is used for solving a range of numerical problems especially when there is no closed form solution. In this instance we examine the HornShunk [28] motion estimation algorithm, later extending this to a complete pipeline to calculate the transformation between 3D images. Examples of such images would be ultrasound scans of a beating heart.

In summary, the algorithm can be divided into the following tasks:

1. Calculate the derivatives of the image intensities with respect to spatial and temporal coordinates: I_x; I_y; I_z; I_t.
2. Calculate the cross-products of the derivatives.
3. Convolve each one of the components with a Gaussian filter.
4. Iteratively solve the system of linear equations described by the HornShunk algorithm to generate a motion field.
5. Apply the motion field to the moving image, resample and go back to step 1.

For this benchmark we test both the iterative solution computed at step 4 and the iterative solution computed by the entire pipeline. The structure of the graph is that the each task computes for part of the image, then the tasks pass information to an evaluator task that decides if another iteration is required, or if the solution has converged.

5.1.3 Sharded Index

This benchmark searches a brother-son index for the number of occurrences of a collection of words that appear in a corpus of books drawn from Project Gutenberg [29]. The index is sharded into independent pieces, with the shard that any given word belongs to being determined by a hash of the prefix of the word. Sharding structures in this way is a common way of distributing the memory foot print of a data structure across a cluster to improve performance, for example a web indexes where sharding allows the entire structure to be maintained in RAM. Here maintaining locality improves the portion of the data structure maintained in the level one and level two cache.

5.1.4 Routing

This benchmark uses Dijkstra's algorithm to determine the shortest path between a set of sources and a set of destinations. Each source and destination pair is passed to a thread as a pair of arguments and these are then scheduled for execution.

5.1.5 Kmeans and Kmeans-complex

K-means [30] is arguably the most commonly used clustering technique, clustering a set of points into a number of groups based on their spatial locality.

We experiment with two implementations of kmeans. In kmeans-a, the points and centers are stored in an array; In kmeans-l they are stored as linked-lists. The different implementations allow us to analyze the effect on data reuse, data locality and cache utilization of more complex structures relative to simple array accesses.

5.2 Results

When evaluating the scheduling policies, we observe improvements in data locality, measured through cache misses which can be seen in Figure 9. This shows the number of L2 misses as a percentage of those seen when using *FIFO* scheduling. We see that for all benchmarks our *Reference* and *Token* scheduling policies perform at least as well as the best alternate policy. In case of Routing benchmark, the *Reference* scheduling policy reduces the cache misses to 30% of the *FIFO* level on 32 cores. On the average the *Reference* scheduling policy reduces the L2 cache misses to about 50% of the *FIFO* level. The advantage of the *Reference* scheduling policy is that the scheduler knows exactly which data was most recently sent to which processor, which as expected results in the best data locality.

The *Source* policy which prioritises the processor where a given task was created shows almost identical performance to *FIFO*. This policy relies on the observation that a tasks children are more likely to share data requirements than a random task elsewhere in the system. Unfortunately this observation fails to work for a range of models including those that converge to a single task to perform some control logic before returning to work on the dataset. Examples of this model include MapReduce [16] and the ForkJoin [31] frameworks. *Random*, as expected, performs progressively worse as the number of processors increases. This is because the probability of finding a good schedule by chance decreases as the number of processors rises. The apparent improvements of *Random* for Matrix Multiply and Sharding reflects the failure of *FIFO* and *Source* to adapt well to these benchmarks as the core count increases.

In Figure 10, we see the reduction in L1 cache misses for four of the benchmarks. Iterative refinement and Routing show no significant effect on the L1 cache misses from changing scheduling policy. This is because the inputs for threads are too large to fit in the L1 cache. This emphasises that any scheduling policy will only obtain an advantage from locality if the data is partitioned such that it remains in the cache between threads. Scheduling cannot remove the need for appropriate partitioning of the problem.

Figure 11 shows the network traffic and network utilization for all the benchmarks. The network traffic consists of request controls, response data, write back data, write back control etc. As can be seen from the graphs, the reference scheduling policy generates the least amount of traffic which results in a reduction in network contention and also helps reduce power consumption. On the average, on 32 cores, the reference scheduling reduces the network traffic by about 50%. The reduction in network traffic has a direct and proportional impact on the dynamic power consumption of the network. This relation can be seen by looking at the power consumption graphs in Figure 12

The difference in cache misses leads to an improvement in execution time as well. Figure 13 shows the scaling in execution time for the scheduling policies evaluated. Speedups are shown relative to *FIFO* on the same number of processors. We can see that the changes in execution time closely track the changes in cache misses. In the case of the *Sharder Index* benchmark, we get a

(a) Matrix Multiply

(b) Iterative Refinement

(c) Sharded Index

(d) Routing

(e) kmeans-a

(f) kmeans-l

(g) Average

Fig. 9 Number of L2 misses, as a percentage of those seen with FIFO scheduling.

speedup of up to 30% and the average speedup for 32 processors is 14%. This shorter execution time combined with the lower network power usage results in an even better improvement in the overall energy consumed by the network.

The comparison of speedup graphs of *kmeans-a* and *kmeans-l* provides an insight into the effect of these strategies on different styles of applications.

(a) Matrix Multiply

(b) Iterative Refinement

(c) Sharded Index

(d) Routing

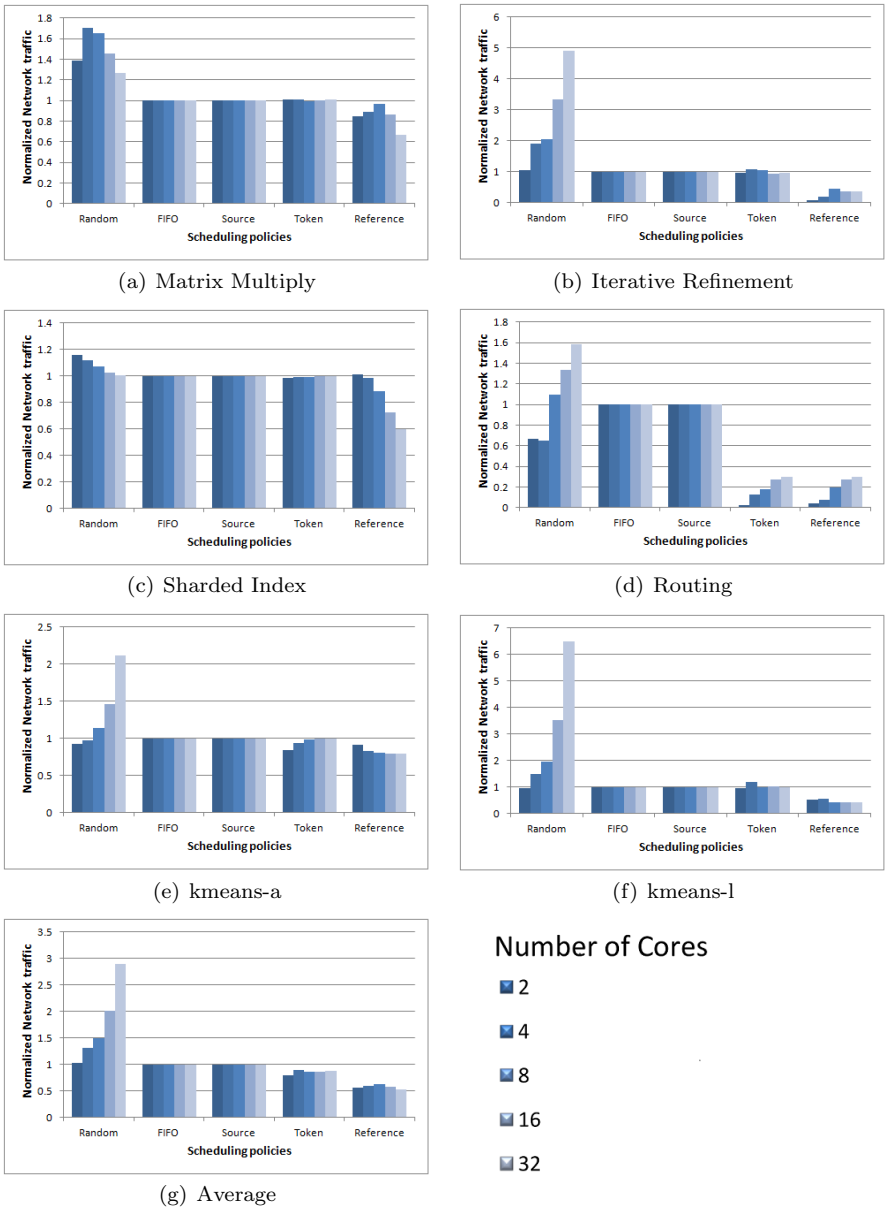
(e) kmeans-a

(f) kmeans-l

(g) Average

Fig. 10 Number of L1 misses, as a percentage of those seen with FIFO scheduling.

Even though both the benchmarks show a reduction in cache misses when using the reference scheduling policy, with kmeans-l, the reference scheduling policy results in significant performance improvement as compared to kmeans-a. The reason for this is the access patterns of the two benchmarks. As *kmeans-a* works on an array based structure, it has very consistent and predictable access



(a) Matrix Multiply

(b) Iterative Refinement

(c) Sharded Index

(d) Routing

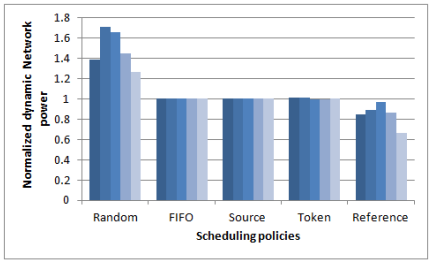
(e) kmeans-a

(f) kmeans-l

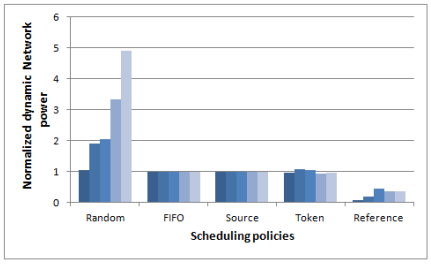
(g) Average

Fig. 11 Network Traffic, normalized against FIFO scheduling.

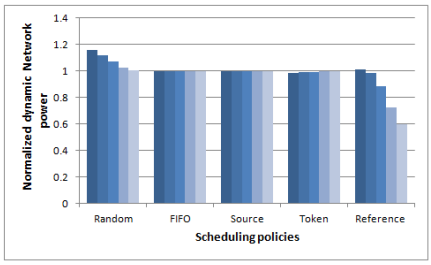
patterns which are very cache friendly, therefore the overall cache miss rate is very small and improving this miss rates does not have any significant impact on the performance of the benchmark. Matrix multiplication is also array based and demonstrates a similar pattern. However, kmeans-l uses a linked-list, and Sharded uses a brother-son tree making them more representative of general



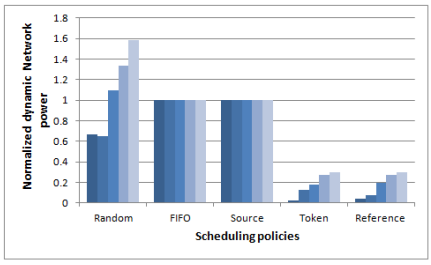
(a) Matrix Multiply



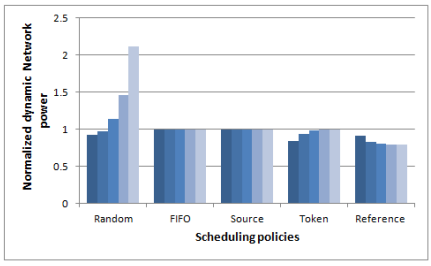
(b) Iterative Refinement



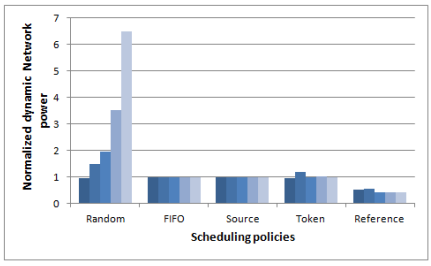
(c) Sharded Index



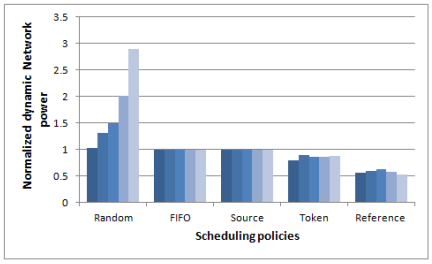
(d) Routing



(e) kmeans-a



(f) kmeans-l



(g) Average

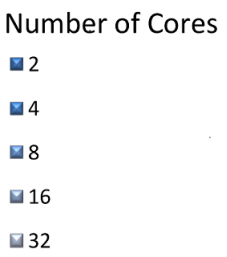
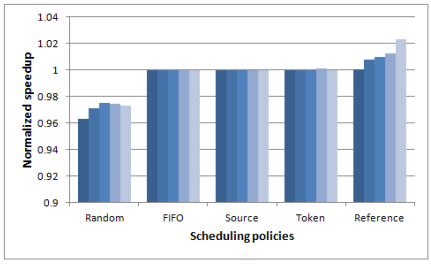
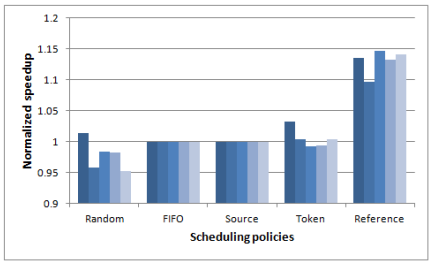


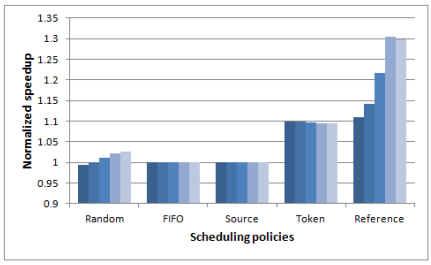
Fig. 12 Runtime dynamic power consumed by the network for each scheduling mechanism, normalized against FIFO scheduling.



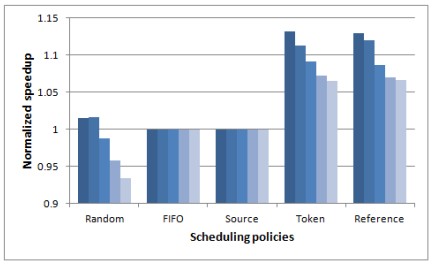
(a) Matrix Multiply



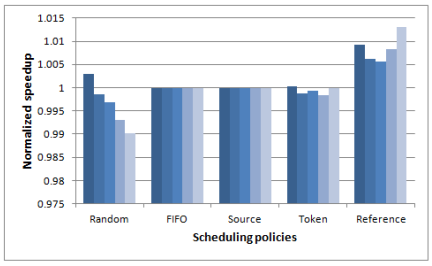
(b) Iterative Refinement



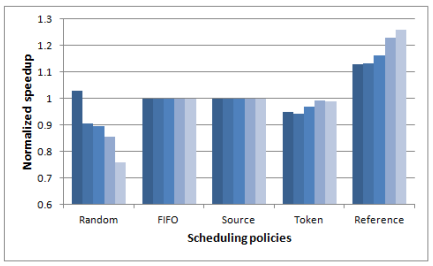
(c) Sharded Index



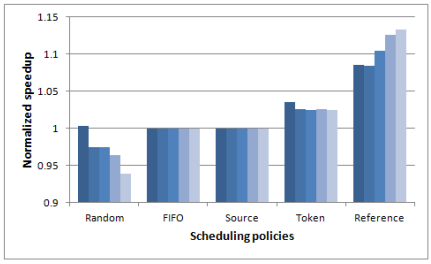
(d) Routing



(e) kmeans-a



(f) kmeans-l



(g) Average

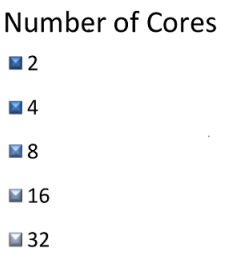


Fig. 13 Speedup, normalized against FIFO scheduling.

purpose applications. Figure 14 shows the comparison of L1 cache miss rates of kmeans-a and kmeans-l.

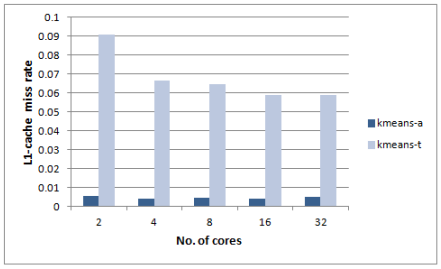


Fig. 14 L1 cache miss rate comparison using Reference scheduling, as the processor count varies from 2 to 32 cores.

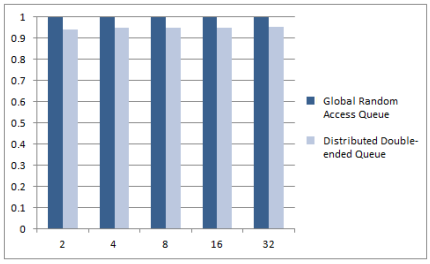
5.2.1 Comparison with random access global queue

In Section 4 we restricted the scheduler accesses to the top and bottom of each core's queue. We will now examine the performance effect of opting for this simpler design instead of allowing any ready task to be scheduled at any time. To do this we simulated a version of the design where a global queue contains all the ready tasks. When a request is sent to the scheduler, it looks at the entire queue to find the best match based on references. The task with highest affinity is then taken out of the queue and sent to the requesting core.

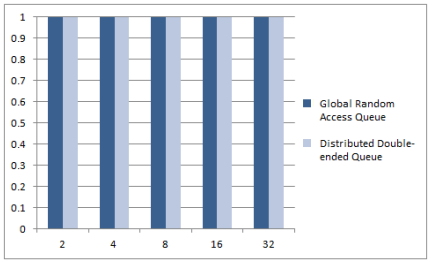
By allowing random access to the queue can result in better scheduling decisions as the task with the highest affinity is always selected by the scheduler producing better performance. In order to perform the comparison, the hardware scheduler in both the distributed and global queues takes the same amount of time to respond to a task request from a core. In reality inspecting the entire queue on every task request could make it take a much longer time to respond to and results in more complex hardware, but in this experiment we are examining if it would be worth it.

Figure 15 shows the comparison of our reference scheduling hardware model which uses distributed double-ended queues, with hardware scheduler which uses a random access global queue. From the results it is clear that the performance of our distributed queue model is very close to the global queue version. The performance of our model gets better with the increase in the number of cores. As there is a separate hardware queue for each core, with the increase in the number of cores the scheduler gets better opportunities to group similar tasks in the same queue and to get better data locality which results in better performance. On the average, for 2 cores, the distributed version performs at 96% of the global queue based scheduling which increases to 97% for a 32 core machine. This shows that additional queues improve performance, it also shows that mostly the performance gain is marginal.

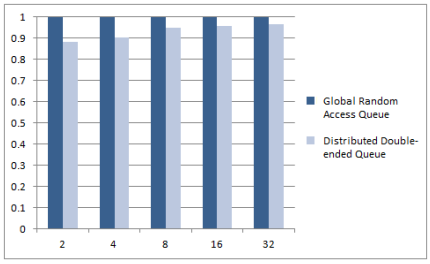
This is because for most applications it is either best to run the tasks they have just created, or the run tasks that where created at the same time as the current task. Collectively these represent points at each end of the queue. The notable exceptions are Matrix-Multiply and Kmeans-l which have a large number of tasks created and made ready at once. As future work we would



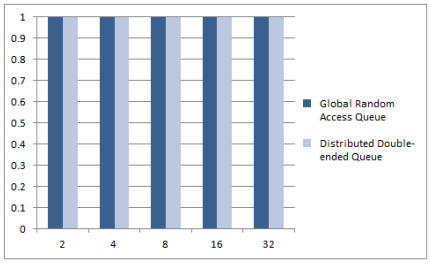
(a) Matrix Multiply



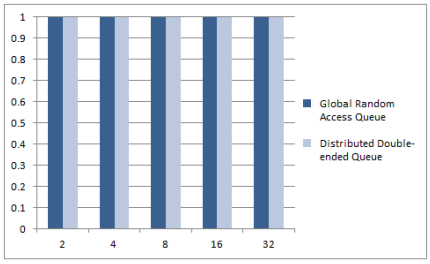
(b) Iterative Refinement



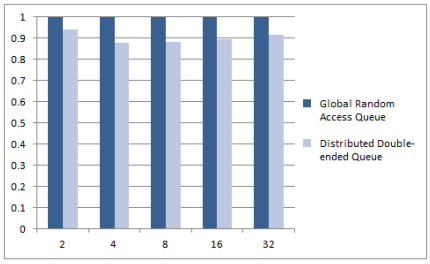
(c) Sharded Index



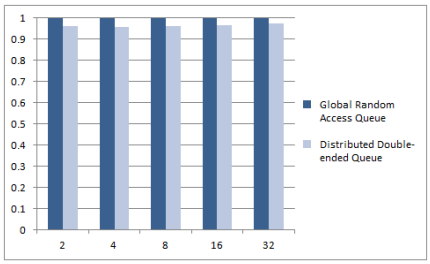
(d) Routing



(e) kmeans-a



(f) kmeans-l



(g) Average

Fig. 15 Speedup comparison of hardware scheduler with distributed double-ended queues vs. global random access queue. Speedup, normalized against hardware with a global random access queue, as the processor count varies from 2 to 32 cores.

like to investigate if this effect can be overcome by some form of sorting of the available tasks as this would allow relevant tasks to be grouped.

5.2.2 Sensitivity to the latency of the GSU

Another question with our design is the sensitivity of the design to the speed with which tasks can be dispatched. In our current design, the GSU takes 20 cycles to process each request. Like to other schedulers [26] our hardware scheduler uses LSUs to hide the latency of the GSU by buffering and prefetching tasks. In this subsection we measure the effectiveness of the latency tolerance mechanism.

Figure 16 shows the comparison of execution times with a high latency GSU taking 300 cycles to serve a request and with GSU that takes 0 cycles. As can be seen from the graphs, increasing the latency of the GSU has nearly no effect on the execution times of the benchmarks. This result was expected, as the presence of the LSUs and the large size of individual task help in hiding the latency of the GSU. By increasing the buffer size of the LSUs, the hardware can hide the latency effect of the GSU even if the task size in the benchmarks is very small. These mechanisms can be used with random global queue scheduling mechanism in order to further improve the scheduler performance, as latencies of delaying the affinity calculation can be hidden by the LSU.

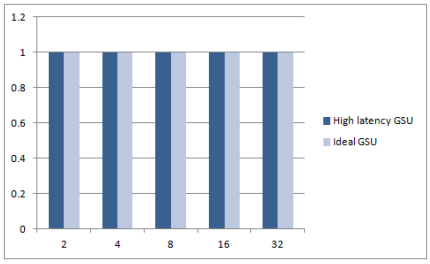
6 Related Work

We will now consider where else this strategy is already used and look at what existing schedulers do to maximize cache reuse.

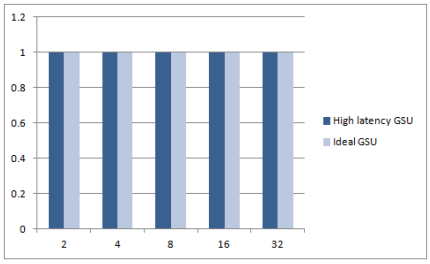
6.1 Cluster Computing

Although dataflow codes are not yet common enough for hardware schedulers to have been constructed for commodity multi-core processors, scheduling computations that map onto dataflow for locality of data can be found in many cluster computing environments. Probably the most well-known of these and the most important as it is also being used as a model for programming multi-core devices is MapReduce [16]. MapReduce is a functional programming model developed by Google to make it easier to construct parallel codes. It has been widely accepted both for large scale computing projects and multi-core machines.

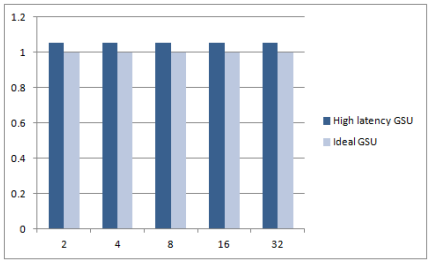
In their paper describing this work [16] Google make reference to the effect that scheduling for data locality has on network traffic and congestion. By scheduling tasks to execute on the same machine as their Google File System (GFS) block resides they are able to avoid inter node transfers and so reduce delays to the specific task, but also network contention. As not all data will be local, so some data movements must occur, and reducing delays in the network



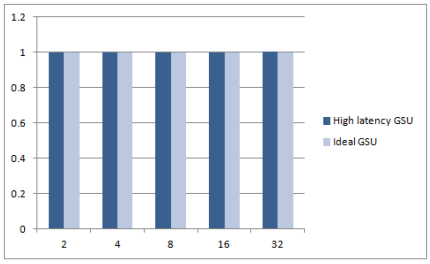
(a) Matrix Multiply



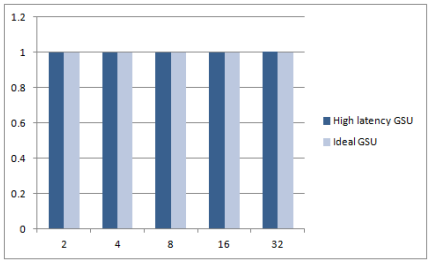
(b) Iterative Refinement



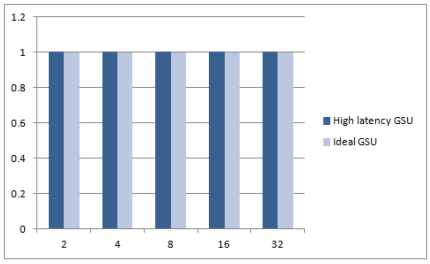
(c) Sharded Index



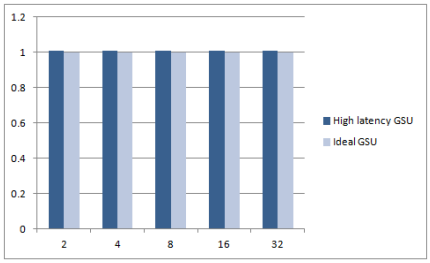
(d) Routing



(e) kmeans-a



(f) kmeans-l



(g) Average

Fig. 16 Execution time comparison of high latency (300 cycles) GSU with ideal (0 cycle latency) GSU. Execution times are normalized against ideal GSU. The processor count varies from 2 to 32 cores.

reduces delay to the program as a whole. It is clear from our results that the technique applied to GFS when executing dataflow programs on a cluster can also be effectively applied to scheduling for cache locality on multi-core processors.

6.2 Schedulers

Schedulers for more traditional parallel frameworks are software based. There exists a large body of work on dynamic task scheduling in software [32, 2, 33, 3, 34, 1, 35]. Software-only schedulers maintain queues in software, and threads communicate and exchange work implicitly through shared memory. There have been several optimizations that have been proposed to avoid the use of locks in most local enqueue/dequeue operations [2] or to use non-blocking stealing protocols [36, 23]. However, to get effective speedup the tasks constructed within these frameworks have to be sufficiently large such that the overhead of the scheduler does not dominate the execution time. For tasks of this size it was considered fair to ignore reuse of the cache as the time to replace every element in the cache is an insignificant part of the overall execution time.

To allow multi-core processors to be used for more general problems it is necessary to be able to efficiently construct and use smaller tasks. This has lead to the design of hardware schedulers such as Carbon [26] which have sufficiently low latencies to allow small tasks to be used efficiently. Carbon uses a centralized *global task unit (GTU)*, which contains one hardware FIFO queue per hardware thread. Applications then use special instructions to enqueue and dequeue task descriptors directly to and from registers. Task descriptors have a fixed size of 4 64-bit words. A small *local task unit (LTU)* per core is used as a task buffer to hide enqueue and dequeue latencies from GTU. At this scale, which is more suited to general purpose use of multi-core processors, improvements in cache locality were observed, however the frameworks lacked any way to observe the dependencies of the tasks. Instead the benefits occur as a by product of the separate queues that are constructed for each core, as a tasks' children are more likely to share data requirements than a random task located elsewhere in the system. Unfortunately this observation fails to work for a range of models including those that converge to a single task to perform some control logic before returning to work on the dataset. Examples of this model include MapReduce [16] and the ForkJoin [31] frameworks. This can be seen with our experiments with the source scheduling scheme.

Sanchez et al [40] present a combined hardware-software approach for scheduling that proposes asynchronous direct messages (ADM). This provides direct exchange of asynchronous messages between threads in the CMP without going through the memory hierarchy. This work focuses on programs that use fine-grain parallelism, with tasks as small as a thousand cycles, and on on minimizing the custom hardware structures by introducing simple general primitives that have multiple uses rather than fixed-function hardware like

those in Carbon. The evaluation is done by simulating cache-coherent, tiled CMPs with a packet switched interconnect. Although the work claims to have reduced the amount of hardware required in comparison with Carbon, it still requires custom hardware by adding ADM messaging unit per core and using an extra virtual network [42] in the packet-switched interconnect to route message packets, which requires extra buffering in the routers.

Yoo et al [43] provide a quantitative analysis of exploiting task locality for unstructured parallelism. The work involves developing a locality analysis framework and an offline scheduler that takes workload profile information as input and generates schedules that are optimized for the target cache hierarchy. The effectiveness of the scheduler is evaluated on three specific many-core cache hierarchies that they claim represent distinct and very different points in the many-core design space. Through a graph-based locality analysis framework and a generic, recursive scheduling scheme, the paper demonstrate that significant potential exists for locality-aware scheduling. The simulation results of three distinct 32-core systems show significant performance improvement over randomized schedule and the baseline schedule, Parallel Depth First (PDF) scheduling [41]. By increasing the hit rates in the caches closer to the cores, a locality aware schedule also reduces the average energy consumption in the memory hierarchy beyond the L1 caches relative to the random and baseline schedule. The work also highlight the importance of locality-aware stealing when the tasks are scheduled in a locality aware fashion, and demonstrate that a recursive stealing scheme can effectively exploit significant locality while load balancing.

Chen et al [39] evaluates the impact of thread scheduling algorithms on on-chip cache sharing for multithreaded programs. Many multithreaded programs provide opportunities for *constructive* cache sharing, where concurrently scheduled threads share a largely overlapping working set, as opposed to *destructive* competition for cache spaces among threads. It compares the performance of two schedulers: Parallel Depth First (PDF) [37,38], scheduler designed for constructive cache sharing, and Work Stealing (WS). The paper claims to be the first in demonstrating the effectiveness of PDF on real benchmarks, providing a direct comparison between PDF and WS. This study demonstrates that the PDF scheduler, which was designed to encourage cooperative threads to constructively share cache, either matches or outperforms the WS scheduler on a CMP machines for all the fine-grained parallel programs studied. The paper also shows that the task granularity plays a key role in CMP cache performance.

7 Conclusion

To take advantage of increasing number of cores in a CMP, applications must expose their thread-level parallelism to the hardware. One common approach to doing this is to decompose a program into parallel tasks and allow an underlying software layer to schedule these tasks to different threads.

In this paper we have described how task based programming models can have their runtime extended to improve the locality of the data used by taking advantage of the explicit tracking of the movement of data within a computation. We propose two scheduling techniques, 'token scheduling' and 'reference scheduling' that make use of the information provided by the dataflow programming model. We also propose a scalable hardware scheduler design that has low hardware complexity and demonstrate that it is relatively insensitive to the access latency of the hardware queues.

Simulations of this technique for a range of synthetic benchmarks and components of real applications have shown that our scheduling policies can reduce the number of cache misses by up to 72% and 95% for the L1 and L2 caches respectively and up to 30% improvement in overall execution time against FIFO. This not only results in faster execution, and in less data transfer, allowing for less load on the interconnect, but also results in lower power consumption.

7.1 Future Work

There are currently five areas we would like to continue the exploration of this work into:

Larger Applications The size of the applications that we have run has been restricted by the run time of the simulator. While benchmarks such as Iterative Refinement form the kernel of real applications we would like to expand the runs out to larger applications including all the supporting code.

Multiple Applications We would like to perform experiments exploring what measures are required when executing multiple applications. Specifically how to balance fairness with through put, as an application what has a history of using many cores will likely maintain the use of those cores due to greater affinity. While this is beneficial for through put, it is less good for the responsiveness of applications.

Alternative Queuing Mechanisms As demonstrated in Section 5.2.1 while most of the time a double ended queue per processor is sufficient, when spawning a large number of simultaneously ready tasks there is potential benefit to having a single random access queue. However, the complexity, power consumption, scalability and responsiveness of such a solution are likely to render such an option infeasible. As such we would like to explore different heuristics and queue structures to further improve the available performance. Such solutions could include some form of sorting of tasks based on their arguments at the point that they become ready.

Evaluation on many-core design space One of the areas we are interested in is to evaluate the effectiveness of our scheduling policies on various many-core cache hierarchies representing distinct and different points in many-core design space. These experiments will be similar to the work done by Yoo et al [43].

Better Scheduling Algorithms We are already working on improving our current scheduling techniques and looking into ways of getting information regarding structure and size of the data from a program into the scheduling decision making process. For example, the scheduler not only looks into the number of references but also size of the data that is being referred to.

Acknowledgements The authors would like to thank the European Community's Seventh Framework Programme (FP7/2007-2013) for funding this work under grant agreement no 249013 (TERAFLUX-project). Dr. Luján is supported by a Royal Society University Research Fellowship.

References

1. OpenMP. OpenMP application program interface version 3.0 (2008)
2. M. Frigo, C.E. Leiserson, K.H. Randall., *The implementation of the cilk-5 multithreaded language.* (1998)
3. Intel., *Intel Thread Building Blocks, http://www.threadingbuildingblocks.org* (2006)
4. Park, W.J. Dally., *Buffer-space efficient and deadlock-free scheduling of stream applications on multi-core architectures.* (2010)
5. K. Opencl, A. Munshi. The opencl specification version: 1.0 document revision: 48 (2009)
6. W. Thies, M. Karczmarek, S. Amarasinghe., *StreamIt: A language for streaming applications.* (2001)
7. J. Jenista, Y. hun Eom, B. Demsky, SIGPLAN Notices **46**(8), 57 (2011)
8. J.M.Perez, R.M.Badia, J. Labarta., *A dependency-aware task-based programming environment for multi-core architectures.* (2008)
9. *The TERAFLUX project, http://www.teraflux.org* (2010)
10. J.R. Gurd, C.C. Kirkham, I. Watson, Commun. ACM **28**, 34 (1985). DOI http://doi.acm.org/10.1145/2465.2468. URL http://doi.acm.org/10.1145/2465.2468
11. G.M. Papadopoulos, D.E. Culler, in *Proceedings of the 17th annual international symposium on Computer Architecture* (ACM, New York, NY, USA, 1990), ISCA '90, pp. 82–91. DOI http://doi.acm.org/10.1145/325164.325117. URL http://doi.acm.org/10.1145/325164.325117
12. D. Cann, Commun. ACM **35**, 81 (1992). DOI http://doi.acm.org/10.1145/135226.135231. URL http://doi.acm.org/10.1145/135226.135231
13. I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg, J. Sargeant, in *ISCA* (1988), pp. 124–130
14. J. Darlington, M. Reeve, in *Proceedings of the 1981 conference on Functional programming languages and computer architecture* (ACM, New York, NY, USA, 1981), FPCA '81, pp. 65–76. DOI http://doi.acm.org/10.1145/800223.806764. URL http://doi.acm.org/10.1145/800223.806764
15. S.L. Peyton Jones, C. Clack, J. Salkild, M. Hardie, in *Proc. of a conference on Functional programming languages and computer architecture* (Springer-Verlag, London, UK, 1987), pp. 98–112. URL http://portal.acm.org/citation.cfm?id=36583.36590
16. J. Dean, S. Ghemawat, Commun. ACM **51**, 107 (2008). DOI http://doi.acm.org/10.1145/1327452.1327492. URL http://doi.acm.org/10.1145/1327452.1327492
17. D. Peng, F. Dabek, in *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (USENIX Association, Berkeley, CA, USA, 2010), OSDI'10, pp. 1–15. URL http://dl.acm.org/citation.cfm?id=1924943.1924961
18. D. Goodman, S. Khan, C. Seaton, Y. Guskov, B. Khan, M. Lujan, I. Watson, in *Proceedings of the proceedings of Data-Flow Execution Models for Extreme Scale Computing* (IEEE, 2012)
19. M. Odersky, L. Spoon, B. Venners, *Programming in Scala: [a comprehensive step-by-step guide]*, 1st edn. (Artima Incorporation, USA, 2008)

20. M.T. Vandevoorde, E.S. Roberts, *Workcrews: an abstraction for controlling parallelism.*, vol. 17 (1988)
21. E. Mohr, D.A. Kranz, B. Venners, R.H.H. Jr, *Lazy task creation: a technique for increasing the granularity of parallel programs* (1990)
22. D. Hendler, N. Shavit, *Non-blocking steal-half work queues* (2002)
23. D. Chase, Y. Lev, *Dynamic circular work-stealing deque* (2005)
24. U.A. Acar, G.E. Blelloch, R.D. Blumofe, *The data locality of work stealing* (2000)
25. B.H. Bloom., *Space/time trade-offs in hash coding with allowable errors.*, vol. 13 (1970)
26. S. Kumar, C. Hughes, A. Nguyen, ACM SIGARCH Computer Architecture News **35**(2), 162 (2007)
27. N. Binkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, et al., ACM SIGARCH Computer Architecture News **39**(2), 1 (2011)
28. B. Horn, B. Schunck, Artificial intelligence **17**(1), 185 (1981)
29. Project Gutenberg. http://www.gutenberg.org/ (1971)
30. J. Bezdek., *Pattern Recognition with Fuzzy Objective Function Algorithms.* (1981)
31. D. Lea, in *Proceedings of the ACM 2000 conference on Java Grande* (ACM, 2000), pp. 36–43
32. R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, Y. Zhou., *Cilk: an efficient multithreaded runtime system.* (1995)
33. R.H.H. Jr., *Implementation of multilisp: Lisp on a multiprocessor.* (1984)
34. Y.K. Kwok, I. Ahmad., *Static scheduling algorithms for allocating directed task graphs to multiprocessors*, vol. 31 (1999)
35. E. Su, X. Tian, M. Girkar, G. Haab, S. Shah, P. Petersen., *Compiler support of the workqueuing execution model for Intel SMP architectures.* (2002)
36. N.S. Arora, R.D. Blumofe, C.G. Plaxton., *Thread scheduling for multiprogrammed multiprocessors.* (1998)
37. Guy E. Blelloch and Phillip B. Gibbons. *Effectively sharing a cache among threads.* In Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures (SPAA '04). ACM, New York, NY, USA, 235-244. DOI=10.1145/1007912.1007948 http://doi.acm.org/10.1145/1007912.1007948
38. Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. *Provably efficient scheduling for languages with fine-grained parallelism.* J. ACM 46, 2 (March 1999), 281-321. DOI=10.1145/301970.301974 http://doi.acm.org/10.1145/301970.301974
39. Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. *Scheduling threads for constructive cache sharing on CMPs.* In Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures (SPAA '07). ACM, New York, NY, USA, 105-115. DOI=10.1145/1248377.1248396 http://doi.acm.org/10.1145/1248377.1248396
40. Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. *Flexible architectural support for fine-grain scheduling.* In Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems (ASPLOS XV '10). ACM, New York, NY, USA, 311-322. DOI=10.1145/1736020.1736055 http://doi.acm.org/10.1145/1736020.1736055
41. Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. *Scheduling threads for constructive cache sharing on CMPs.* In Proc. of the 19th SPAA, pages 105115, 2007.
42. W. Dally and B. Towles. *Principles and Practices of Interconnection Networks. Morgan Kaufmann Publishers Inc.* 2003.
43. Richard M. Yoo, Christopher J. Hughes, Changkyu Kim, Yen-Kuang Chen, and Christos Kozyrakis. *Locality-aware task management for unstructured parallelism: a quantitative limit study.* In Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures (SPAA '13). ACM, New York, NY, USA, 315-325. DOI=10.1145/2486159.2486175 http://doi.acm.org/10.1145/2486159.2486175