Task Scheduling for Runtime-Assisted Parallelism

Aditya Venkataraman* adityav@cs.wisc.edu
Bhuvana Kakunoori* bhuvana@cs.wisc.edu
Kishore Kumar Jagadeesha* kishore@cs.wisc.edu
Tejaswi Agarwal* agarwalt@cs.wisc.edu
*University of Wisconsin - Madison

Abstract

Emerging task-based parallel programming models shield the programmer from the challenging task of managing parallelism by delegating the responsibility of identifying and scheduling independent tasks to a runtime system. The runtime system typically collects vast amounts of semantic information about the application to optimistically scout for independent work. However, compared to the techniques for identifying indepedent work, runtime systems adopt relatively simple scheduling disciplines for scheduling runnable tasks; the most common approach being work-first delegation with randomized work-stealing for load balancing.

In this work, we present evidence to show that such simple schedulers are incapable of handling non-uniform memory access effects that are now prevalent on a single chip and can greatly degrade the performance of applications with fine-grained tasks. We explore the design space for augmenting scheduling decisions with information collected by task-based runtimes. Our experiments reveal a performance improvement of 1.4× to 2.8× times the current randomized work-stealing approach. Lastly, we propose a sample scheduling discipline, 'WARR', that represents a viable design point for intelligent schedulers and compare its performance against the default randomized-work stealing approach. WARR scheduler provides a speedup improvement ranging from 0.97× to 2.19× over the maximum performance obtained from the default scheduling discipline. We also port a subset of our improvements to the Cilk scheduler.

1. Introduction

A common approach to parallel programming is to decompose a program into a set of tasks and distribute them among the available processing elements. Many task-based programming models have been proposed in the literature [14, 18, 7, 8] that shield the programmer from the challenging job of mapping, orchestrating and scheduling tasks. These responsibilities are instead delegated to a runtime system that provides simple APIs to the programmer for exposing parallel work. Runtime systems collect vast amounts of semantic information about the application such as task read/write data-sets, application data-structures, producer-consumer relationships etc. This information aids the runtime in efficiently calculating task dependences and identifying runnable tasks. Runnable tasks are then scheduled across the available processing elements through a scheduling policy. The policy aims to maximize the utilization and load-balancing across the system. Work-stealing based schedulers with work-first delegation are the most popular scheduling algorithms for dynamic task parallelism as evidenced by past work e.g., Cilk [8], Cilk++ [3], Intel Thread Building Blocks [19], Java's Fork-Join Framework [16], Microsoft Task Parallel Library [17] and StackThreads/MP [23]. A work-stealing (WS) scheduler employs a fixed number of threads called workers. Each worker has a local deque to store tasks. When a worker is idle and there is no local task in its deque, the worker will try to steal a task from the top of a randomly selected worker's deque [5, 6]. For a busy worker, tasks are pushed and popped from the bottom of its local deque and these operations are syncronization-free. Work-first delegation [9] implies that a newly spawned task will be immediately executed on the same worker while the rest of the program continuation can be stolen by a different worker.

Randomized work-stealing with work-first delegation has attractive theoretical properties for a particular class of workloads where steals are rare [4], but we note three major drawbacks with this approach. First, randomized stealing can work against the inherent locality among tasks [1]. Second, work-first delegation migrates the program continuation for each task-spawn, which can be very costly for flat spawn-trees. Work-lists of tasks with a single-level spawn-tree is emerging as a popular programming model for irregular applications such as graph-analytics [14]. Finally, new application domains in Recognition,

Mining and Synthesis are dominated by parallel sections with small task sizes [15]. Fine-grained tasks are also attractive for effective load-balancing. However, the adverse effects of an inferior scheduling discipline are exacerbated for fine-grained parallelism [13].

However, runtime-systems also present great opportunities for improving task-scheduling. Runtimes collect vast amount of information about task memory footprints which can be used to make better scheduling decisions. In this work, we explore some avenues for incorporating this information into the scheduling discpline and argue for a tighter integration of a runtime's functional components and its scheduling policy.

Contributions The main contributions of this paper are:

- *We motivate the idea that simple work-stealing based schedulers are ineffective for fine-grained applications and for systems exhibiting NUMA effects.*
- *Using an internally developed task-based runtime system, we explore the design space for improving scheduling decisions with information collected by the runtime for evaluation of task-dependences.*
- *We present a sample scheduling discipline, 'WARR', that represents a viable design point for intelligent schedulers and compare its performance against the default work-stealing scheduler*
- *We extend the work-stealing scheduler in Cilk.*

Paper Organization We first give an overview of our internal task-based runtime system (§2) and present results that motivated our study into task-scheduling policies (§3). Subsequently, we describe the design-space exploration for improving scheduling decisions (§4). We then present the WARR scheduler compare it against the default work-stealing scheduler. We also present our experience of incorporating some of our intelligence to the default Cilk scheduler and the challenges we faced (§6). We summarize the related works (§7)in this field and conclude (§8).

2. The PKLW System

In this section, we describe the internally-developed task-based runtime system - PKLW, at a high level to provide background for the rest of the paper. The PKLW system is a fork off Parakram, a concurrent, object-based shared memory model implemented on top of C++ [10]. The PKLW system was forked with the intention of serving as a suitable test-bed for exploring scheduling policies. It was enhanced with a flexible infrastructure for quickly implementing new policies and evaluating them through a detailed profiling framework.

```
1: while (work_list not empty) do
2:     ...
3:     <calculate task data_set>
4:     df_execute_lw (data_set, &function_df)
5: end while
6: df_seq()
7: ...
8: df_end()
```

Figure 1: Pseudocode for PKLW APIs

2.1. Client code

Programs written for PKLW are similar to conventional sequential C++ programs, but in addition to coding the algorithms, the user identifies: (i) potentially parallel functions, (ii) the objects shared between such functions, and (iii) their read and write sets, i.e. the data read & written by the function. The objects shared between such functions inherit from a token base class supplied by the runtime. The runtime associates tokens with dynamic instances of the class and uses these tokens to track dependences across tasks. The potentially parallel region is marked using a `df_execute_lw` interface provided by the library. The invocation should include a function pointer to the parallel function as well as the data-set that will be modified by that instance of the function. The invocation prompts the runtime to attempt execution of the function in parallel with the *continuation*, i.e. the remainder of the program. Figure 1 presents the pseudocode for an application using PKLW's APIs.

2.2. Runtime

The runtime houses active tasks within a reorder buffer (ROB) [24]. Each invocation of `df_execute_lw` is allotted an entry in the ROB. Hence, the order of tasks in the ROB is consistent with the sequential order of invocations within the program. Similar to ROBs in hardware CPU designs, the runtime allows out-of-order execution of tasks, but will retire them in program-order as defined by the ROB. This allows PKLW to achieve parallel execution with the added benefit of sequentially-consistent precise interruptibility. The runtime is built for scalability with minimum sharing of data

across cores and uses wait-free synchronization techniques where necessary.

Task Dependences Once an entry is alloted for a task in the ROB, the runtime will evaluate its read-/write sets. PKLW supports data-flow execution by constructing data dependence graphs (DAGs) for each task. Tasks whose data-sets do not clash with other tasks are deemed runnable. Tasks with data-dependences wait in the ROB till their parent tasks complete.

Scheduler PKLW's default scheduling policy is work-first delegation with randomized work-stealing for load-balancing. Each core has a standard deque to hold tasks assigned to it. Once a task is declared as runnable by the data-flow engine, it will be immediately executed on the same core. The program continuation will be pushed into the core's deque from where, it will most likely be stolen by another core and executed in parallel with the spawned task. Once a task completes, it will ungate the execution of any future tasks that are dependent on the current one. Also, if the completed task is at the head of the ROB, it will be retired from the runtime.

Profiling PKLW has an extensive profiling framework that is capable of capturing and logging the occurrence of events such as steals, spawns, task wait-times, task execution-times etc. This allows us to study the bottlenecks of scheduling policies.

3. Scheduling Overheads for Fine-grained Parallelism

In this section we will present the study that motivated our inquiry into task scheduling techniques for runtime-assisted parallelism. Parakram's underlying data-flow engine along with its default scheduling algorithm have exhibited impressive scaling for a range of workloads as published in previous work [10]. As discussed in literature [15, 21], fine-grained tasks are becoming important for task-based programming models due to a combination of architectural innovations, emerging applications in new domains and ease of load balancing for heterogeneous platforms with changing number of processing units. However, conventional wisdom suggests that task sizes in the order of ~1000 cycles cannot be profitably parallelized through a software runtime library as the mere overhead of executing runtime code for each task-spawn would overshadow the benefits of parallel execution.

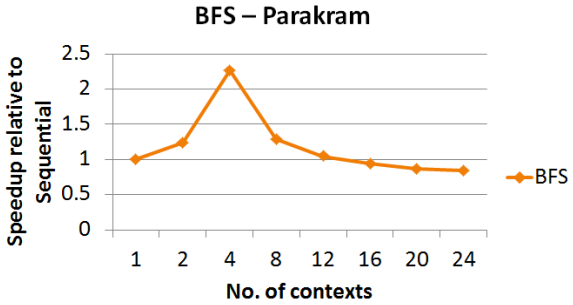

Figure 2: BFS: Initial speedup on PKLW

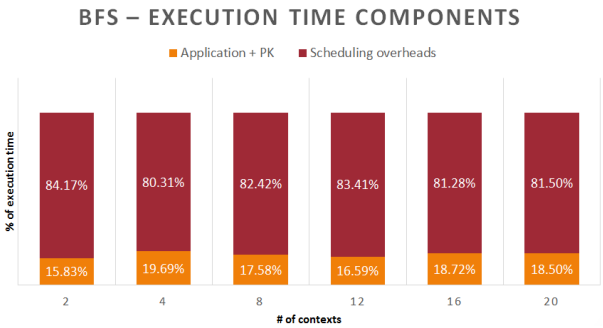

Figure 3: BFS: Execution time components on PKLW

Our initial results for Breadth First Search, a graph-analytics application with a task size of ~1500 cycles, were consistent with this expectation of modest speedups as shown in Figure 2. Similar trends were witnessed for other applications such as Single Source Shortest Path that exhibit fine-grained parallelism. Through profiling, we learnt that a significant portion of the execution time was spent on scheduling-related overheads such as migration of continuation, coherence-induced cache misses etc - time not accounted for by execution of any code. As shown in Figure 3, less than 20% of the BFS execution time is spent on executing either application or runtime code. BFS's parallelism is exposed to the runtime as a work-list of fine-grained tasks, an execution model that is popular for parallelizing such irregular applications [13, 18]. This suggests that scheduling overheads are particularly pernicious for irregular applications that expose fine-grained parallelism through work-lists of tasks.

4. Design Space Exploration

In this section, we will present an overview of the design-space exploration for task-scheduling on runtime-assisted environments. This analysis is specific to PKLW runtime environment; in (§6) we extend the analysis to Cilk, a vastly different environment to PKLW. All experiments were performed on a 2-way hyperthreaded, 6-core, 2-socket, 24-context,

1.9GHz Intel Xeon E5-2420 (Sandy Bride; 32KB each L1I and D, 256KB private L2 caches per core; 15MB shared L3 cache) system. Programs were compiled with gcc 4.7.2, with the -O3 option on a Linux 2.6.32 kernel. In the rest of the paper, default scheduling refers to PKLW's default scheduling policy of work-first delegation with randomized work-stealing based load-balancing. Table 1 gives the applications that have been used to evaluate the scheduling disciplines.

The information that is, or can be made, available to PKLW falls under two broad categories: (i) Global information about the underlying system and application, and (ii) Local information about each spawned task. There are two sources of global information: (i) awareness of the system configuration through operating system and profiling and (ii) application's data-structures that can be made known to the runtime either directly through special APIs or indirectly through inheritance relationships. The sources of local information have been elaborated in a previous work [10]. Table 2 summarizes the information that is available to PKLW for a given application.

4.1. System-Awareness

Single chip NUMA effects are becoming increasingly prevalent for large-scale multicore systems [25]. To reduce execution time and energy consumption, data access locality ought to be exploited. Randomized work-stealing works against data locality as chunks of work can be stolen across the system with minimal or no control. Awareness of the underlying system architecture can be useful to identify a superior load-balancing technique. Towards this end, we introduced a '*System-Discover*' phase during PKLW's runtime startup. System-Discover has two stages: (i) from the operating system's sysconf APIs, PKLW will learn the number of cores, sockets in the system as well as the number of cores per socket and number of hyperthreads per core, and (ii) PKLW will spawn worker threads on each processing element and pass cache-line aligned messages across them. The measured round-trip latencies can be used to learn the relative distances between cores. Figure 4 gives the relative communication latencies for core 0. There are three distinct latency levels with increasing communication latency: (i) communication with neighboring hyperthread, (ii) communication with a core on the same socket (2× slower than hyperthread communication) and (iii) communication across sockets (6× slower than hyperthread communication). Based on this learnt layout, thread affinities are set and core distances are encoded into a compact bitmap.

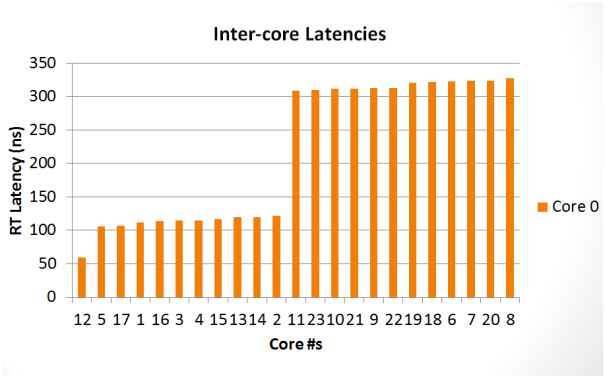

Figure 4: Inter-core communication latencies

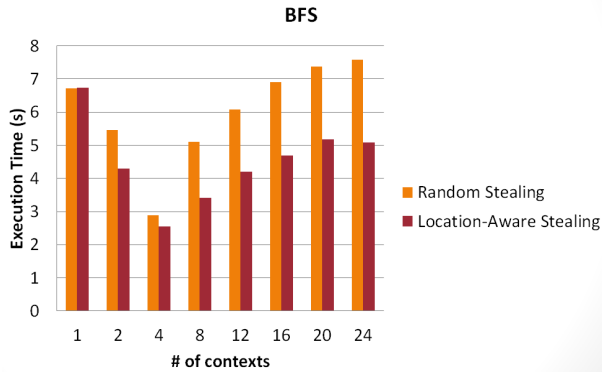

Figure 5: Locality-Aware Stealing in BFS

Locality-Aware Stealing Using the system layout we can implement a hierarchical, locality-aware stealing approach for dynamic load-balancing. When a core is out of work in its local deque, it will embark on heirarchical stealing where it will first snoop for work on its neighboring hyperthread. If there is work in the hyperthread's deque, the work can be stolen with least cost and minimal cache invalidations. Else, the core will snoop on cores that are part of the same socket. Eventually, the core will snoop on cores on a different socket. This hierarchical stealing was implemented using efficient bitmap operations to reduce time in critical path of stealing. Figure 5 shows the impact of locality-aware load-balancing on BFS speedups. Compared to randomized stealing, we witness a speedup improvement of upto 1.5×.

4.2. Migration of Continuation

The migration of the continuation as part of work-first delegation can be an expensive event involving the invalidation of several cache lines which can be particularly harmful for flat spawn-trees. To over-

Application	Remarks
Breadth First Search	Visit each node in a graph and compute its distance from the source node.
B-Tree	Construct a vast B-Tree with 1023 keys per node. Invoke a number of transactions that read a key or insert new keys into the B-Tree
Array-manip	Manipulate an array of objects in which each element accesses its neighbors
2D Convolution	Perform 2D convolution of a 4096x4096 image with a 5x5 Gaussian Filter
MM-fine/coarse/massive	Blocked matrix multiplication with small/medium/large block size
Cholesky Decomposition	Decomposition of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose

Table 1: Applications

Nature	Information	Source
Global	System Layout	Profiling, OS
	Application data-structures	Data-structure construction, special runtime APIs
Local	Task Sizes	Profiling
	Task Data Sets	Task spawn API
	Inter-task dependences	Data-flow DAGs

Table 2: Information available to PKLW

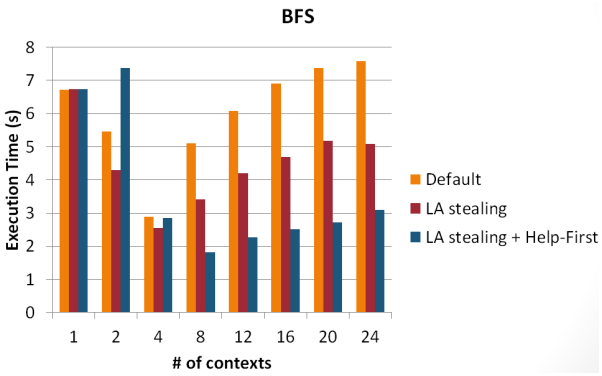

Figure 6: Help-First Delegation with Locality-Aware Stealing in BFS

come this drawback, we explored an alternate strategy called *help-first* delegation. In help-first delegation, the continuation continues to run on the same core while the spawned task is made available for stealing; the intuition being that the cost of migrating the task will be cheaper than the cost of migrating the continuation. Using help-first scheduling with locality-aware stealing, we notice a reduction in execution time of upto 2.8× for BFS relative to default scheduling, as shown in Figure 6. Our profiling revealed two drawbacks with this approach. First, all newly spawned tasks are pushed into the

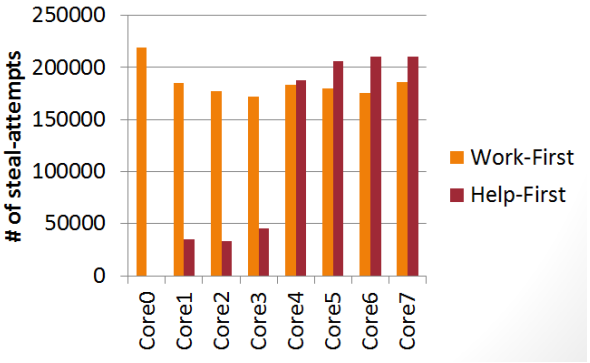

Figure 7: Work Distribution with Help-First Delegation in BFS

continuation's deque from where they are stolen by other cores, increasing the contention for that core's deque. Second, help-first delegation with locality-aware stealing leads to a skewed distribution of work across cores. Cores on the same socket as the continuation do more work as compared to cores on other sockets. The amount of work done by a core can be correlated with its number of attempted steals, as a busy core will have less reason to attempt stealing. As Figure 7 suggests, help-first scheduling causes cores on the same socket as the continuation to be 4× more loaded than other cores.

4.3. Work Assignment

To overcome the load imbalance and contention issues with help-first delegation, we attempted a work-assignment strategy. Given a runnable task, the continuation will decide to assign it to a particular core by pushing it to the relevant deque. The selection of the core is a policy question. In this work, we have evaluated two policies for work-assignment: round-robin and task data-set based assignment.

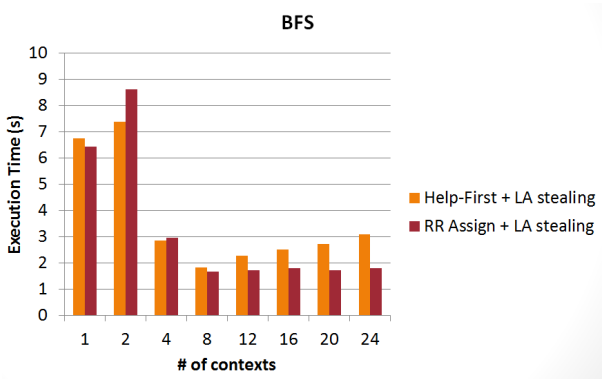

Figure 8: Round-Robin Work Assignment in BFS

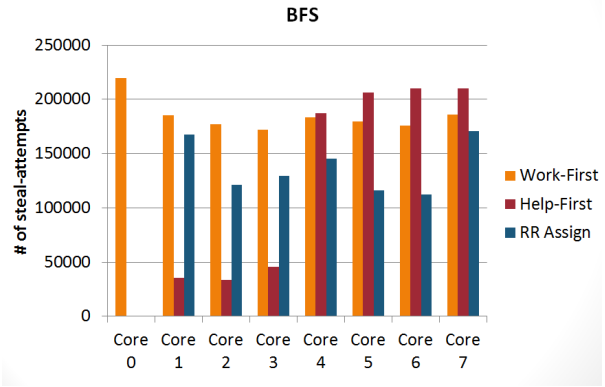

Figure 9: Work Distribution with Round-Robin Work Assignment in BFS

Round-robin assignment This policy is very simple as it assigns new tasks to cores in a round-robin manner. Despite its simplicity, round-robin assignment can perform quite well for typical applications on PKLW. Most applications structure the logical work-list of tasks as a queue of queues to avoid the usage of thread-safe containers, where the tasks in each queue are logically related to one other. A round-robin assignment will ensure that tasks in the same queue are assigned to the same core, exploiting the inherent logical relationships across these tasks. Cores that do not find any work assigned to them will steal work from other cores using the locality-aware stealing described before. As shown in Figure 8, round-robin assignment with locality-aware stealing provides upto 1.7× reduction in execution time compared to help-first delegation. Crucially, as seen in Figure 9, round-robin assignments achieves a more equitable distribution of work across worker cores as compared to help-first delegation.

Task data-set based assignment As mentioned in §2, PKLW knows each task's data-set at the time of task-spawn, which presents new opportunities for scheduling tasks. Tasks that operate on data located close-by in memory can be scheduled on the same or nearby processing units to maximize the data reuse. This requires the scheduler to maintain an active model of the contents of each cache on each core/-socket and map a new task's data-set to this model to determine its ideal destination. Initially we implemented a model similar to the one described in [20] in which we map each address in the task data-set to a cache model. However, this implementation greatly deteriorated the performance across all our applications. Core-assignment lies in the critical path of task-spawn; a heavyweight model for mapping task data can greatly reduce performance. Also, such cache models seek to exploit the temporal locality of cache contents across tasks which inherently offers diminishing returns over time.

Instead of tracking individual addresses of the task data-set, we chose to track at page-granularity. Each data-set entry is mapped to its corresponding page in memory. These pages are mapped to a destination core according to a hash-table. The hash-table is structured such that nearby cores will receive close-by pages. This effectively mimics the partitioning of the application's data-structure across the available cores of the system. This implementation also failed to provide any benefits.

Eventually we discovered a *destructive interference* between the dataflow engine and our scheduler. Whenever a new task is spawned by the application, the dataflow engine will first determine its dependence with any currently executing task. This checking, which occurs on the continuation's core, causes the dataflow engine to *write* to each entry in the data-set. This defeats the purpose of partitioning the application's data structure across cores as the data will be repeatedly brought to the continuation's core. This is an implementation detail of PKLW's dataflow engine and may not be true for all runtime environments. To validate this expectation, we modified PKLW's dataflow engine and our applications to work with *dummy tokens* instead of actual data-objects of the application. Corresponding to each shared object in the application, we instantiated a dummy object in a different region of memory. PKLW's dataflow DAGs will be constructed using these dummy tokens, allowing us to partition the application's data-structures across cores. Since the dummy token to shared object mapping is statically fixed, this approach will not affect the functional correctness of the application. As can be seen from Figure 10, this approach outperforms the round-robin assignment by providing

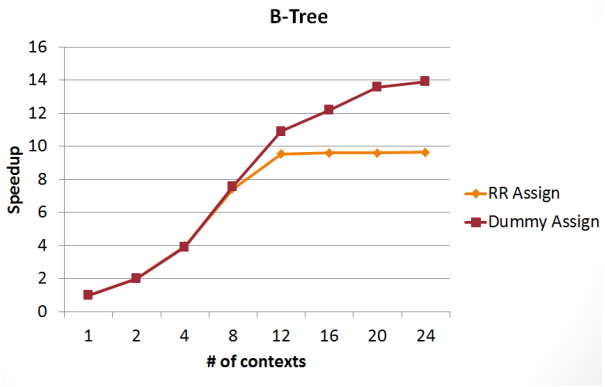

Figure 10: Page-based Assignment using dummy tokens in B-Tree

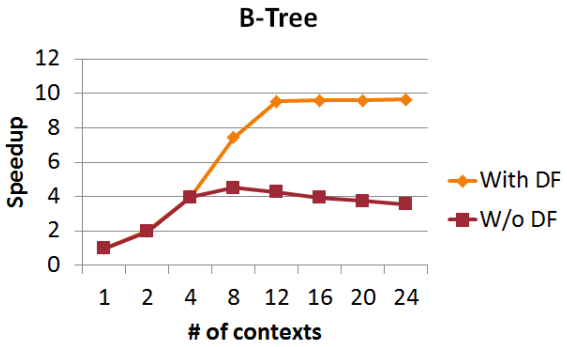

Figure 11: Data-flow scheduling in B-Tree

an additional speedup benefit of up to 1.4x. These promising results suggest that scheduling benefits can motivate new runtime designs that don't destructively interfere with scheduling optimizations.

4.4. Data-flow Dependences

A data-flow dependence between two tasks, A and B, implies that Task B reads/writes data that is written to by A. Until Task A completes, Task B is not runnable and hence will not be visible to the task scheduler. Once Task A completes, the data-flow engine will make Task B runnable and pass it to the scheduler which will then evaluate the best core assignment for Task B. In an ideal scenario, Task B will be assigned to the same core as Task A and pushed into the same deque. However, it is possible that there are several tasks ahead of Task B in the deque which will be executed ahead of Task B. While Task B waits in the deque, it could be stolen by a different core as well. A data-flow dependence offers the opportunity to exploit the temporal locality across tasks. The longer that Task B does not execute on the same core, the lower will be the chances of reusing Task A's data. To combat this latency, we modified PKLW's data-flow engine to expose the dependent task to the scheduler before it becomes runnable. The scheduler will annotate the task with the core assignment of its parent. This annotation allows the dependent task to completely bypass the scheduler's normal functioning. Once the parent task completes, it effectively hands over its deque-slot to the dependent task which will immediately execute on the same core - maximizing the benefits to be had from temporal locality. The advantage of data-flow scheduling depends on the number of data-flow dependencies in the application. As Figure 11 shows, data-flow scheduling is highly effective for B-Tree which has a lot of data-flow edges.

5. 'WARR' Scheduler

The previous section (§4) presented a generic exploration of the design space for intelligent task-scheduling. In this section we describe the 'WARR' scheduler, a sample scheduling discipline for PKLW and compare its performance against the default work-stealing scheduler. It implements the following optimizations that mutually complement one another: (i) system-awareness through profiling (§4.1) to learn inter-core distances, (ii) locality-aware stealing for load-balancing where idle cores will preferentially steal from neighboring cores, (iii) round-robin based work-assignment (§4.3) for runnable tasks and (iv) data-flow aware scheduling (§4.4) where data-dependent tasks completely bypass the round-robin assignment. WARR does not use the data-set based assignment as that would require extensive changes to the PKLW dataflow engine and applications.

As shown in Figure 12, WARR scheduling provides on aggregate a 36% benefit over the default randomized work-stealing approach. Figure 13 gives the speedup profile for each application. We note that WARR provides a greater benefit for fine-grained applications (maximum benefit up to 119.7%) such as MM-fine and BFS, without degrading the performance for coarse-grained applications (a maximum degradation of -2.2%) such as Cholesky Decomposition and MM-massive. WARR scheduling also provides greater scalability as speedups are witnessed till higher core counts than with default randomized work stealing.

6. Enhancing the Cilk Scheduler

Cilk is a task-based multi-threaded language that provides programmer constructs for parallel control [8]. The programmer annotates user level code with parallel constructs which the compiler and the run-

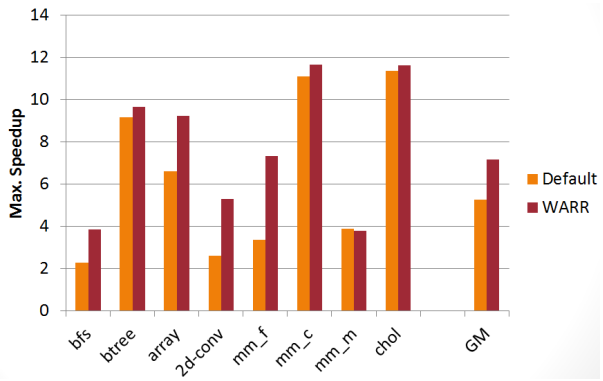

Figure 12: Maximum Speedups with WARR Scheduler

time processes for generating and scheduling parallel tasks. Cilk follows a work-first approach, thus maintaining the serial semantics of the program. Cilk scheduler uses a random work stealing for load balancing. On a task spawn, the runtime starts executing the new task and pushes the continuation stack frame on to its own work queue. When this continuation is stolen by other worker threads, a 'slower' copy of the continuation is created. The overhead of creating and restarting the slower clone is high, since the program state including local variables, program counter should be restored from the stack frame. Therefore, the runtime scheduler is optimized to reduce the number of steals and favors executing the faster clones predominantly.

6.1. Work Assignment

The design of Cilk mentioned in the previous section, makes it harder to implement work assignment for the spawned tasks without making drastic changes to its runtime and compiler. The runtime maintains the invariant that the newly spawned tasks are never stolen. When a task is spawned, Cilk does not create a separate activation frame for the thread, but instead relies on the C function call stack to execute the spawned task. A new stack frame is created for the continuation and is pushed to its worker queue. This whole process is hard-coded into its compiler and the runtime does not have access to the stack frame of the newly created task. There is a huge state information gap between the compiler annotations and the runtime, as a result of which, changes cannot be done in isolation without sharing data between them. Thus, the independent spawned tasks cannot be assigned or stolen in the current design of Cilk scheduler, thus leaving the scheduler to deal only with the slower continuation threads. Having understood these design limitations, we modified the scheduler to implement a variant of work assignment, which assigns the continuation instead of spawned tasks. Instead of relying on other worker threads to steal the continuation, the continuation is directly pushed to another core's work-queue, in a round-robin fashion. The idea is to reduce the contention on a single worker queue. For better load balancing, we allow other worker threads to continue to steal randomly after certain time. We evaluated this method for a set of benchmarks and the results are shown in Figure 14. We notice that the speedup gains are not significant and it in fact performs almost the same as the default work stealing approach. This behavior is expected since the overhead of continuation creation and migration is not removed.

6.2. Delayed work stealing

Cilk follows a greedy work stealing approach. The workers keep looking to steal tasks from other workers and the overhead of the continuation stealing itself is high. This overhead is tolerable if it can be hidden with the execution time of the individual spawned tasks. But for fine grained tasks, the overall execution time with these overheads can be worse than the sequential execution time. Delayed work stealing addresses this issue by delaying the work stealing on each individual worker thread. The worker thread sleeps for a certain amount of time before stealing the task from a neighboring work queue. This gives the victim thread a small window of priority over the tasks in its own deque and reduces the number of costly steals in the system. We evaluated this method for a set of fine grained applications and the results are shown in Figure 15. As seen from the figure, with the default scheduler, the speedup for fine grained tasks deteriorates as the number of cores increases. Delayed scheduling shows better performance and scalability for fine grained tasks.

7. Related Works

Randomized work stealing, introduced by Blumofe et al. [5], is based on a theoretical model for fully strict computations in which the time to run a computation on P processors is directly proportional to its work, which is the time to run in a single processor, and its span, which is the time to run the computation on infinite processors. It is shown that under the assumption of saturated parallelism, randomized work stealing is most communication-efficient. We believe that some of the assumptions of this model

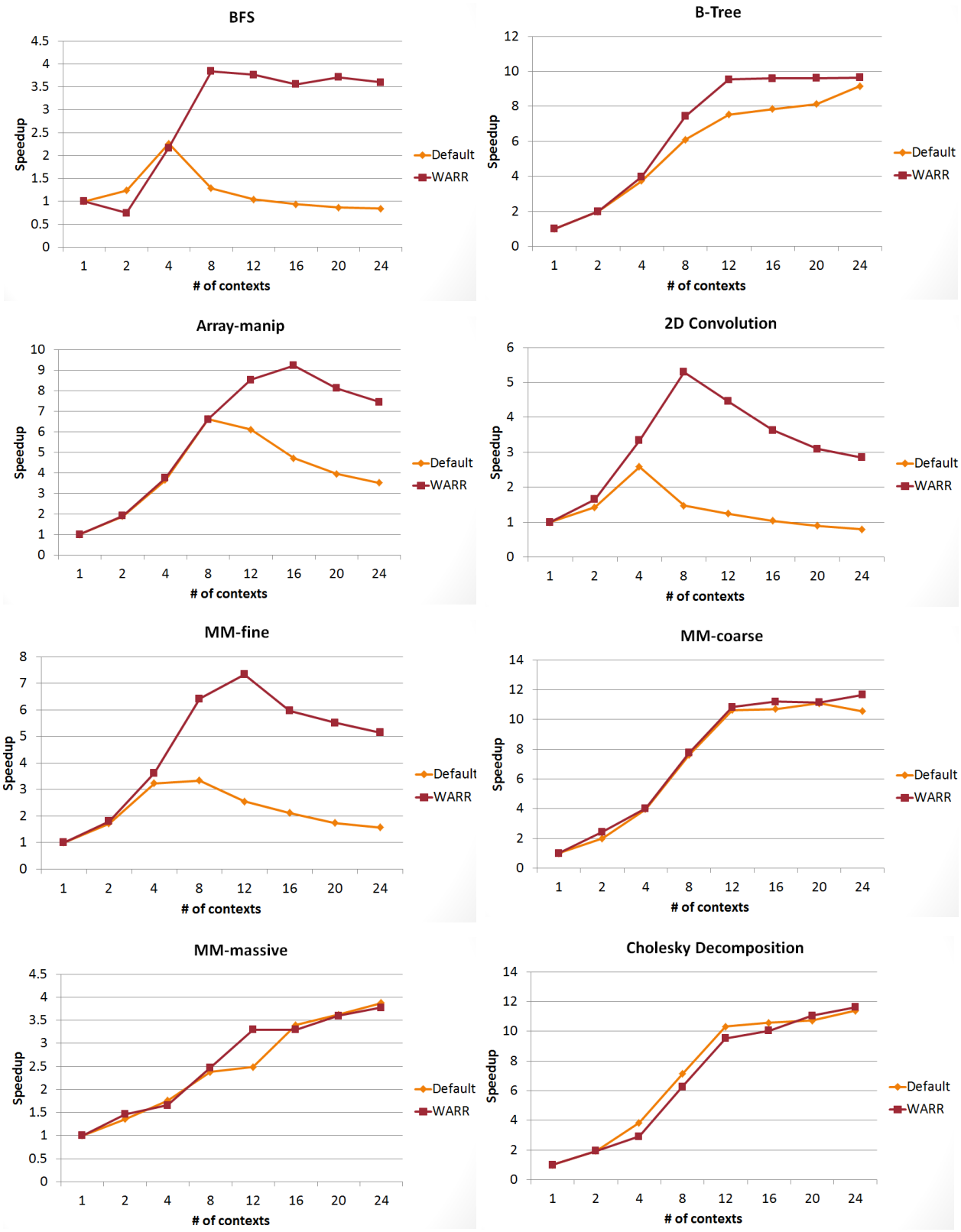

Figure 13: WARR Scheduling Speedup Profiles

are not applicable for fine-grained tasks. Keckler's [12] was one of the earliest to point out the importance of locality while scheduling tasks. He suggests a novel technique to assign processor affinity to each task. Guo et al. [9] propose a locality aware adaptive scheduling algorithm that addresses the problem of fixed task scheduling policy and the locality unawareness of randomized task stealing algorithms. For

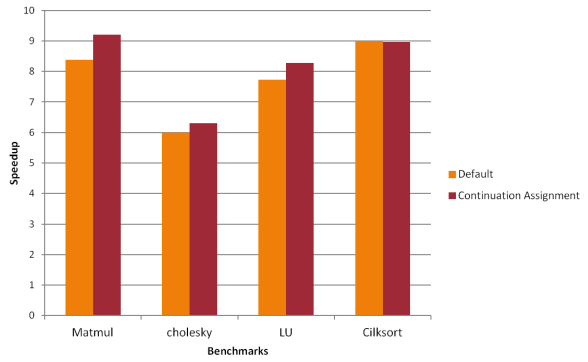

Figure 14: Speedup with Cilk Work Assignment

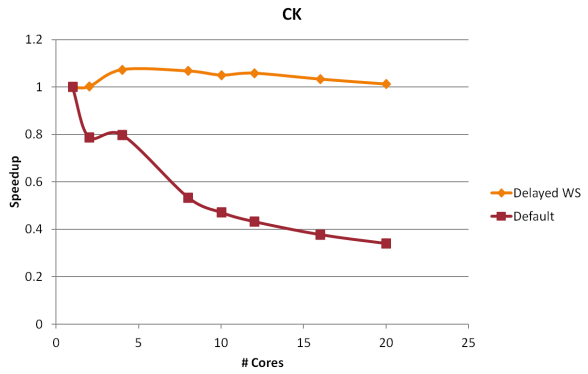

Figure 15: Speedup with delayed Work Stealing

the adaptive algorithm, the policy chooses between work-first and help-first algorithms dynamically at runtime, based on only the memory bounds of the application and does not work in coordination with the locality aware scheduling policy. Moreover, the locality aware algorithm itself depends on application and compiler hints and does not consider the hierarchical NUMA architecture of modern multicore machines. Philbin et al. [20] analyze the impact of cache locality on thread scheduling using a simple model to estimate the contents of the cache. This model and the task's data set are used to schedule tasks to processors. Mark s. Squillante and Edward D. Lazowska [22] present a novel way of modeling a cache for estimating a task's cache-reload miss ratio. In his thesis, Yoo [25] looks at scheduling policies employed by runtime systems. He contends that despite two decades of innovation, most software schedulers ignore cache hierarchy information while scheduling tasks. He develops a framework to analyze locality relationships between tasks such as producer-consumer relationships to schedule them on shared-memory many-core systems. Huseyin Gokseli Arslan [2] provides an adaptive control theory based scheduling framework which takes into account various metrics such as performance fairness and cache miss fairness. Though this seems to be a heavyweight approach, their mathematical approach provides a promising foundation for our work. Acar et al. [1] provide an upper and lower bound on the cache misses for work stealing algorithm. They propose a locality-guided work stealing algorithm, where a worker gives priority to the thread that has affinity to the processor. The affinities are set based on heuristics like initial placement. But, this policy is more suitable for iterative data parallel algorithms that can take advantage of the warm caches. It may not be applicable to irregular programs that have more random data sharing patterns. Task scheduling is a primary problem even for large cluster based systems. Matei Zahra et al [26] describe a delay scheduling technique for cluster scheduling . Since scheduling a task closer to the data is more efficient, the delay scheduling algorithm waits for a certain time until it is able to launch a task locally instead of migrating it off rack. To achieve fairness, it also migrates the task off rack if no node is able to schedule the task locally after a certain time. In Quincy [11], M.Isard et al introduce a framework for task scheduling on clusters by mapping tasks on nodes of a graph, and edge weights representing fairness and locality, thus generating a cost model which is then used to evaluate the best scheduling decision during runtime . The framework described here improves fairness, while still improving data locality. The mechanisms described above work well for large clusters when the average task size is high.

8. Conclusion

The industry is rapidly advancing towards large-scale chip-multiprocessors with tens to hundreds of cores. To efficiently utilize these cores, programs are using finer-grained tasks up to a granularity of ~1000 cycles. As task sizes get smaller, inefficient scheduling of these tasks to available cores leads to load imbalance and severe performance degradation due to NUMA effects. This paper has evaluated the drawbacks of the common randomized work-stealing schedulers for scheduling fine-grained tasks. With the increasing popularity of runtime systems in parallel programming frameworks, we proposed scheduling enhancements by utilizing the vast semantic information collected by runtime systems for identifying parallel work. We also proposed 'WARR', a viable scheduling discipline for fine-grained applications, without degrading performance for coarse-

grained applications. Finally, we also ported some of our ideas to the Cilk parallel programming library. In the future, this work can be extended to evaluate a tighter integration between the runtime library and the scheduler to further minimize the scheduling overheads. We believe this will lead to new research paths involving micro-architecture, operating systems and compilers.

9. Acknowledgements

We would like to thank Prof. Guri Sohi of the University of Wisconsin - Madison for providing access to the Parakram source code as well as multiple many-core systems for our experiments. We also acknowledge the contribution of Gagan Gupta to the design and evaluation of this work. We thank Vinay Gangadhar for providing us with the code for 2D convolution. Lastly, we thank our instructor, Prof. Michael Swift for his guidance and feedback throughout this project.

References

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*. ACM, 2000, pp. 1–12.

[2] H. G. Arslan, "Adaptive cache aware multiprocessor scheduling framework," Master's thesis, Queensland University of Technology, 2011.

[3] C. Arts, "Cilk++."

[4] R. D. Blumofe and C. E. Leiserson, "Space-efficient scheduling of multithreaded computations," *SIAM Journal on Computing*, vol. 27, no. 1, pp. 202–229, 1998.

[5] ——, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

[6] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2005, pp. 21–28.

[7] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[8] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *ACM Sigplan Notices*, vol. 33, no. 5. ACM, 1998, pp. 212–223.

[9] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12.

[10] G. Gupta and G. S. Sohi, "Dataflow execution of sequential imperative programs on multicore architectures," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011, pp. 59–70.

[11] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. ACM, 2009, pp. 261–276.

[12] S. W. Keckler, "The importance of locality in scheduling and load balancing for multiprocessors," 1994.

[13] M. Kulkarni, P. Carribault, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew, "Scheduling strategies for optimistic parallel execution of irregular programs," in *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*. ACM, 2008, pp. 217–228.

[14] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *ACM SIGPLAN Notices*, vol. 42, no. 6. ACM, 2007, pp. 211–222.

[15] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: architectural support for fine-grained parallelism on chip multiprocessors," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 162–173.

[16] D. Lea, "A java fork/join framework," in *Proceedings of the ACM 2000 conference on Java Grande*. ACM, 2000, pp. 36–43.

[17] D. Leijen, W. Schulte, and S. Burckhardt, "The design of a task parallel library," in *Acm Sigplan Notices*, vol. 44, no. 10. ACM, 2009, pp. 227–242.

[18] D. Nguyen, A. Lenharth, and K. Pingali, "Deterministic galois: on-demand, portable and parameterless," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. ACM, 2014, pp. 499–512.

[19] C. Pheatt, "Intel® threading building blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

[20] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li, "Thread scheduling for cache locality," in *ACM SIGOPS Operating Systems Review*, vol. 30, no. 5. ACM, 1996, pp. 60–71.

[21] D. Sanchez, R. M. Yoo, and C. Kozyrakis, "Flexible architectural support for fine-grain scheduling," in *ACM Sigplan Notices*, vol. 45, no. 3. ACM, 2010, pp. 311–322.

[22] M. Squillante and E. Lazowska, "Using processor-cache affinity information in shared-memory multiprocessor scheduling," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 4, no. 2, pp. 131–143, Feb 1993.

[23] K. Taura, K. Tabata, and A. Yonezawa, *Stackthreads/mp: integrating futures into calling standards*. ACM, 1999, vol. 34, no. 8.

[24] D. B. Witt and W. M. Johnson, "High performance superscalar microprocessor including a common reorder buffer and common register file for both integer and floating point operations," Jul. 22 1997, uS Patent 5,651,125.

[25] R. M. Yoo, C. J. Hughes, C. Kim, Y.-K. Chen, and C. 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*. ACM, 2013, pp. 315–325.

[26] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. ACM, 2010, pp. 265–278.