User-Assisted Storage Reuse Determination for Dynamic Task Graphs

Mehmet Can Kurt
The Ohio State University
kurt@cse.ohio-state.edu

Bin Ren
Pacific Northwest National Laboratory
bin.ren@pnnl.gov

Sriram Krishnamoorthy
Pacific Northwest National Laboratory
sriram@pnnl.gov

Gagan Agrawal
The Ohio State University
agrawal@cse.ohio-state.edu

Abstract

Models based on task graphs that operate on *single-assignment* data are attractive in several ways, but also require nuanced algorithms for scheduling and memory management for efficient execution. In this paper, we consider memory-efficient dynamic scheduling of task graphs, and present a novel approach for dynamically recycling the memory locations assigned to data items as they are produced by tasks.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: [Concurrent Programming — Parallel Programming]

General Terms Algorithms, Performance

Keywords Memory management, dynamic task graphs

1. Introduction

This paper focuses on memory-efficient execution of programs structured as task graphs and processed by a dynamic scheduler. To motivate the problem, consider a *data-flow* or a *single-assignment* task graph. The edges in such a task graph represent only the true dependences as ordering constraints. Each task in such a task graph produces a unique data item, which is not subsequently modified by other tasks. This representation maximally exposes the concurrency in the program and enables most effective utilization of available hardware parallelism. However, naively scheduling such task graphs can quickly overrun the memory available in a parallel system. Given that the definition-use relationships in a dynamically executing task graph are not known *a priori*, any given data item might have a potential future use, and thus cannot be garbage collected, until the task graph completes execution. Other approaches to optimize memory usage often require additional information about the number of uses or the last use of a given data item [2, 5]. Also, many approaches to addressing the memory issue require knowledge of the static task graph [1, 4, 6], a requirement that can limit the attractiveness of the task model itself.

PPoPP '16, March 12-16, 2016, Barcelona, Spain
Copyright © 2016 ACM 978-1-4503-4092-2/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2851141.2851180

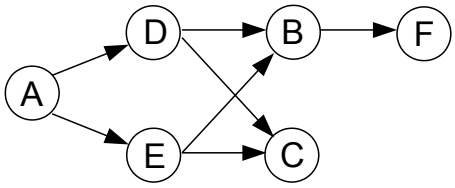

Figure 1: A task graph example, in which task B can safely recycle the store assigned to task A, but not the one assigned to task C.

We present a different approach for optimizing memory usage when scheduling such task graphs. Specifically, to improve the functionality and programmability of such programming models, we address the problem of memory minimization when the task graph is generated dynamically, and the information on last use (or the number of uses) of a memory block is not available. We present an approach to *recycling* the store across tasks in a task graph. We address a number of issues related to realizing a recycling based task graph execution. As a first step towards simplifying the programmer's task, we present an algorithm to verify that a given store recycling function is correct (e.g., the recycling does not cause any overwrites to live data or lead to a data race). We have developed mechanisms for such detection to be carried out with low overheads during program execution and enable recovery from incorrect overwrites within the same run. Thus, the programmer can execute an application with a potentially incorrect recycling function, and still obtain a correct execution without a large slowdown.

To further improve programmability, we present an approach towards automatically deriving a correct recycling function. Specifically, we consider compositions of incoming dependences as potentially valid recycling functions, determine which of them is correct, assess the reuse facilitated by correct functions, and choose the one that leads to the largest number of recycled stores. The correctness determination can be done by execution on a smaller problem size. While this does not ensure correctness of recycling function on a different input dataset (since dependencies can be input dependent), recall that we can further verify the correctness with low overheads during a production run.

2. Our Approach: Store Recycling Functions

A *store recycling function* maps each task T in a directed acyclic graph to another task such that T can reuse the memory segment containing the mapped task's output for its own output. For in-

stance, a specification $Recycle(B) = A$ states that the output of task B occupies the same memory region as the output of the task A. Here, we refer to task A as a *recycle predecessor* of task B and task B to be a *recycle successor* of task A. Several correctness and efficiency considerations arise in choosing recycling function(s). *A recycling operation dictated by a store recycling function is correct if and only if it complies with the causality relations among tasks, and hence does not lead to a data race*. As an example, consider the store recycling function that maps task B to task A in the task graph in Figure 1. A schedule that employs such a store recycling function should order the execution of A to precede the execution of B. A store recycling function is correct if and only if every valid schedule for the task graph satisfies this ordering constraint. In other words, any dependence induced by the store recycling function should be already implied by the edges in the task graph.

In addition, there are correctness requirements to ensure that we do not have an early overwriting of task outputs. Specifically, store recycling should not cause a task's output to be overwritten before all of its uses are complete. Figure 1 shows that task A has two immediate successors (D and E), which finish execution before B starts. Hence, B recycling A's output would not cause early overwriting of A's output. We formulate the above requirements and define the first constraint for a recycling operation to be correct as follows: **Constraint 1:** *If task B recycles task A's output, then A and all of A's immediate successors must causally precede B*. Note that this constraint can be ensured for a recycling operation by imposing additional dependences on the task graph. As an example, consider the concurrent tasks B and C in Figure 1. Although the recycling operation $Recycle(B) = C$ violates the first constraint, injecting an additional dependence edge from C to B is potentially possible. However, introduction of such additional dependences can lead to cycles and thus deadlock. They may incur performance penalties, because the scalability achievable by task graph schedulers is fundamentally limited by available parallelism, or alternatively, the computation's critical path length.

A store recycling function can potentially map two different tasks onto the same task. As an example, consider the recycling operation $Recycle(C) = A$, in addition to $Recycle(B) = A$. Existing dependences in the task graph indicate that A and A's immediate successors (again D and E) causally precede both B and C satisfying the first constraint. Although these recycling operations are individually correct, they cannot be both applied since B and C are concurrent and can lead to a data race on A's output. To prevent this, we define the second constraint as follows: **Constraint 2:** *Two tasks B and C can recycle the same task A's store only if B (or C) can recycle the store associated with C (or B).* As a correct recycling scenario exemplifying the second constraint, one can consider the recycling operations $Recycle(B) = A$ and $Recycle(F) = A$. Since F can potentially recycle B's output, these recycling operations can be both applied in the same run.

3. Overview of Techniques

The two constraints in Section 2 imposed on a store recycling function relate to causality relationships between tasks. A simple scheme can be to execute the task graph and retain predecessor and successor information for all tasks, and verify all recycling operations on termination. However, this could incur high overheads. Our idea is to perform *on-the-fly* checks, using *vector clocks* to concisely represent causality [3].

Our next goal is to be able to automatically derive recycling functions. Our approach here involves first generating a set of candidate recycling functions and then verifying their correctness. Our first observation is that the candidates that a given task can recycle safely should be a subset of the tasks's immediate and transitive predecessors. In general, such candidates for a task can be explored by traversing the task graph starting from the given task and exploring the task graph towards the source tasks (tasks with no incoming dependences) by using predecessor relations provided by the user. Our procedure works as follows. First, we require information on the dependence structure of the task graph from the user. Dependence structure of a task graph consists of the number of immediate predecessors for any task and a function to reach each of them. Next, the paths are enumerated to a bounded depth, referred to as *hops*, to limit the cost of verification.

Although we can detect incorrect recycling function using vector clocks, we remove vector clocks and related mechanisms during normal execution to minimize performance impact. Instead, the following two guarantees are maintained using other more efficient mechanisms. *A recycling function does not result in concurrent recycling.* and *A data item recycled before all of its uses are done is correctly computed again by re-execution of its producer task.*

4. Summary of Experimental Evaluation

To evaluate the effectiveness of automated recycling, we compared the performance of our scheme *auto-recycle* against hand-written code that does recycling. The auto-recycle versions for these experiments were generated by first exploring recycling functions automatically with a 2-hops traversal, and then employing the one which leads to the most number of simulated recycling operations. At 61 threads, our scheme leads to 4.6%, 1.8%, and 0.3% overheads over handwritten-recycle version, respectively for LCS, LU, and Cholesky. These small overheads demonstrate that recycling approach can be effectively automated. These overheads are because of synchronization required to detect concurrent and premature recycling in the automatic versions, and a more direct mapping and avoidance of indirect references in the handwritten versions.

We evaluate the costs to verifiy a set of candidate recycling functions. The overheads turn out to be minimal as long as we go with either 1 or 2 hops, For the 3-hops case, there is an observable increase in the execution time, especially in LU, where more than 70000 distinct recycling functions are checked for correctness. It is a high relative cost for execution with a small problem size. However, in absolute terms, the entire verification phase takes less than 1 second for these benchmarks. The generation of recycling functions is performed sequentially before verification begins, but the time ranges from a few microseconds to 0.035 seconds.

For each benchmark, we evaluate the impact of all recycling functions that are explored with 2 hops and determined to be incorrect. We start the production phase with one incorrect recycling function and switch to a correct function once an incorrect recycle operation is detected at runtime. We divide possible recovery overheads into 9 bins: $<$1%, 1-5%, 5-10%, ..., 50-60%. Results show that number of incorrect recycling functions that cause less than 1% overhead turns out to be 50%, 100%, and 84% of the total number of incorrect functions for LCS, LU, and Cholesky, respectively. Remaining portion of evaluated incorrect functions result in overheads between 1 and 5%.

References

[1] L. Gérard, A. Guatto, C. Pasteur, and M. Pouzet. A modular memory optimization for synchronous data-flow languages: application to arrays in a Lustre compiler. In *LCTES*, pages 51–60, 2012.

[2] M. C. Kurt, S. Krishnamoorthy, K. Agrawal, and G. Agrawal. Fault-tolerant dynamic task graph scheduling. In *SC*, pages 719–730, 2014.

[3] V. Raychev, M. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *OOPSLA*, pages 151–166, 2013.

[4] D. Sbirlea, Z. Budimlic, and V. Sarkar. Bounded memory scheduling of dynamic task graphs. In *PACT*, pages 343–356, 2014.

[5] D. Sbirlea, K. Knobe, and V. Sarkar. Folding of tagged single assignment values for memory-efficient parallelism. In *Euro-Par*, pages 601–613, 2012.

[6] P. Schnorf, M. Ganapathi, and J. L. Hennessy. Compile-time copy elimination. *Softw., Pract. Exper.*, 23(11):1175–1200, 1993.