Synchronization Via Scheduling: Managing Shared State in Video Games

Micah J Best, Shane Mottishaw, Craig Mustard, Mark Roth, Alexandra Fedorova
Simon Fraser University, Canada
{mbest,smottish,cam14,mroth,fedorova}@cs.sfu.ca

Andrew Brownsword
Electronic Arts BlackBox, Vancouver, Canada
brownsword@ea.com

Abstract

Video games are a performance hungry application domain with a complexity that often rivals operating systems. These performance and complexity issues in combination with tight development times and large teams means that consistent, specialized and pervasive support for parallelism is of paramount importance. The Cascade project is focused on designing solutions to support this application domain. In this paper we describe how the Cascade runtime extends the industry standard job/task graph execution model with a new approach for managing shared state. Traditional task graph models dictate that tasks making conflicting accesses to shared state must be linked by a dependency, even if there is no explicit logical ordering on their execution. In cases where it is difficult to understand if such *implicit* dependencies exist, the programer would create more dependencies than needed, which results in constrained graphs with large monolithic tasks and limited parallelism.

By using the results of off-line code analysis and information exposed at runtime, the Cascade runtime automatically determines scenarios where implicit dependencies exist and schedules tasks to avoid data races. This technique is called Synchronization via Scheduling (SvS) and we present its two implementations. The first implementation uses Bloom filter based ‘signatures’ and the second relies on automatic data partitioning which has optimization potential independent of SvS. Our experiments show that SvS succeeds in achieving a high degree of parallelism and allows for finer grained tasks. However, we find that one consequence of sufficiently fine-grained tasks is that the time to dispatch them exceeds their execution time, even using a highly optimized scheduler/manager. Fine-grained tasks, however, are a necessary condition for sufficient parallelism and overall performance gains, so this finding motivates further inquiry into how tasks are managed.

1. Introduction

Video games have become a multi-billion dollar industry and consequently a great deal of effort is expended producing the software that drives them. Producing more and more compelling user experience implies a need to wring out the maximum performance possible. Video game engines, the core of the software, encompass many different computational styles and can rival operating systems in terms of complexity. The environment in which this software is produced further complicates development with tight deadlines and large development teams. This means that programmer productivity and software performance are both of key importance in the domain.

The switch from single to multi-core architectures and the new emphasis on parallelism has profoundly affected this domain. Parallel processing provides new opportunities for performance gains; however these gains are difficult to realize due to the complexity of the system. Currently, most parallelism is obtained by experts who build specific solutions to particular problems. The goal of our project, Cascade, is to take the techniques and processes that are used in industry and combine them in a cohesive and consistent platform and then to augment this model and exploit its particular attributes to realize significant parallelism and promote productivity.

In this paper we will illustrate the thrust of our project by discussing one concrete problem: Conventional task graph models are built on an assumption that there is no shared state among tasks unless the tasks are explicitly linked by a dependency. This puts the burden on the programmer to understand often complex sharing patterns and usually results in creation of large monolithic tasks, which prevent parallelism. To remove this limitation on sharing, we apply unconventional approaches to protect shared state between tasks and as a result enable finer grained tasks and ultimately increased parallelism.

1.1 Problem statement

A common approach to parallelization used by several large companies, is to represent the computation using the familiar *task graph* organization [2]. Tasks that depend on each other for control or data are executed sequentially, but tasks without dependencies can be executed in parallel. If any two tasks might access the same global state, the programmer must explicitly link these tasks via a dependency to avoid conflicts. In cases where the exact state accessed depends on input to the task, the dependency cannot be expressed at all in the task graph.

Given the difficulty in accounting for all *implicit* and *input-dependent* task dependencies programmers often tend to group tasks that might touch the same state into large monolithic tasks [2], artificially restricting available parallelism. Using locks is not considered a practical option given the complexity of the system and difficulty of lock-based programming. Adoption of software transactional memory (STM) is still in question given its high overhead and unpredictable performance [7]. As a result, game engines today are only *mildly* parallel. While this may be acceptable for today's systems that have only a handful of cores, in the future tasks will have to become a lot more fine-grained to leverage the hardware.

Our goal is to rethink shared state management in task-graph systems. We propose a solution where explicit dependencies in the task graph are specified only if there is a logical or a control-flow dependency and any implicit data dependencies are managed automatically without involvement of the programmer. For example, consider two tasks in an animation system that each update the bones of a character skeleton to blend two different animations. One task blends walking and and the other blends limping and there are no *explicit* data or control dependencies between these two tasks. However, since they may update the same bones in the character skeleton, these tasks cannot run concurrently. Another example is an animation system that produces tuples of *(BoneID, Animation)* that are sent to a data-parallel task for the blending computation, increasing parallelism greatly. However, as before, two concurrent tasks cannot be processing the same *boneID*. In this case, it may not be possible to express these dependencies in an offline task graph at all due to the input data only available at runtime.

Our solution is a combination of offline and online analysis. The *implicit* dependency in the first example can be discovered offline and a dependency is automatically added to the task graph at compile time. The potential for conflicts in the second example is discovered offline and resolved at run-time when the input data is known. This solution allows a programmer to focus solely on logical dependencies between parts of the program, greatly increasing programmer productivity, and allowing greater expression of parallelism.

We implemented this solution in the framework of a new parallelization library, Cascade. In Section 2 we provide a brief overview of the system, and in Section 3 we focus on the main problem it addresses: managing shared state in task graph-based programs. We present experimental results in Section 4, discuss related work in Section 5 and conclude in Section 6.

2. Cascade

Our assessment of the current practices and unmet needs in the video game domain lead us to create an efficient parallel programming framework built to support the task graph organization of programs, dataflow between tasks and data-parallel tasks. C++ is the *de facto* language in the domain and so it was used to write the Cascade runtime. However, we discovered, as many others in the field have, that C++ was too permissive in many situations, allowing programmers to easily and unknowingly violate the constraints necessary for correct parallelization. Additionally, we wanted to add certain constructs for defining tasks that could not easily be expressed as classes or macros. To address this and simplify experimentation with our ideas we needed a way to quickly decompose programs into suitable tasks for the runtime scheduler. Cascade programs are written in a new language CDML (Cascade Data Management Language). The syntax is very similar to C++ with a small number of restrictions, such as forbidding pointers, and extra constructs necessary for the definition of tasks. A CDML program is translated into C++ using the Cascade runtime library before compilation. Creating a new language was not a primary goal of our research, but a side effect born of necessity, and so we do not detail it here.

3. Managing Shared State

Our system associates a set of *data constraints* with each task, which represent that task's read and write set. These constraints are derived statically through symbolic analysis and combined with programmer specified control flow to derive the task graph off-line. If static analysis determines that two tasks can touch the same data a dependency is inserted into the graph. However, static analysis can only determine the *potential* accesses of a task when *actual* state accesses are dependent on input data. When this potential set is greater than the actual set we refer to this as *overconstraint*. We use information exposed at runtime to make these constraints more precise through a process of *refinement*. Using this runtime analysis to drive scheduling decisions is referred to as *synchronization via scheduling* (SvS). We developed two implementations of SvS: *Signatures* and *Partitioning*. We now describe the static analysis performed, give a more detailed description of both techniques and discuss the limitations of SvS.

3.1 Static Analysis and Refinement

Use of single variables can be determined simply from textual analysis. If a task contains `x = y + 1;` both `x` and `y` will be added to the constraints as a write and a read respectively. However, consider the difficulty analyzing the expression `z[ i - 1 ] = y + 1` given that we do not know the value of `i` until runtime. Adding `z` to the write set results in overconstraint as the line only refers to a single element of `z`. Without online refinement this task would need to have exclusive access to `z` and so could not be co-scheduled with any other task that accessed `z`. There are additional examples of overconstraint, but in our exploration of the domain we have found the case of accessing indexible structures to be pervasive. As our work continues we will identify and address additional cases.

We can resolve overconstraints at runtime when variables (`i` in the example) are part of the input to the task and are unmodified before that access. A variable is considered part of the input to a task if either the task is a dataflow consumer and the variable is part of the received data or if it is part of global state and part of the task's read set. At runtime the values of these inputs are then used to determine the precise constraints of a task; this process is called *refinement*. Refinement results in a unique id for each state access and can be used to properly schedule tasks by avoiding concurrent execution of tasks with overlapping id sets. Significant overhead is avoided by only comparing id sets of tasks that have been determined at compile time to possibly conflict.

3.2 Signatures

Signature-based SvS maintains constant length bitstrings, signatures, to represent the set of refinement ids for each possibly conflicting task. Each refinement id is passed to a hash function to determine the bit to set in the signature. Signature overlap is checked using simple bit-wise operations. Note that signatures are effectively Bloom filters [3] using a single hash function.

Cascade groups input data items into batches to ameliorate queueing overhead. A composite signature is calculated for each batch. Groups of these batches are created such that no two batches have overlapping signatures; we call these groups *generations*. Batches are added to a generation in order and when a new batch's signature would overlap with any other signature in the current generation this batch is used to start a new generation. State access conflicts are prevented because the tasks are never run concurrently with tasks in another generation.

3.3 Partitioning

Under partitioning, each core has a unique queue from which it executes tasks. The refined id of a task is used to determine what queue the task will be placed in. In this way, tasks with overlapping id sets are executed serially. Note that this technique applies only to tasks with a single id, whereas signatures applies to tasks with one or many ids. However, this process has the additional benefit of increasing data locality, and as with signatures, each core's queue uses batching to reduce overhead.

3.4 Current Limitations

We do not expect SvS to emerge as a general-purpose approach for managing shared state. Refinement is only possible with input-dependent state accesses when the input is left unchanged or only trivially modified by constants. In our experiments we have had success refactoring many algorithms to conform to these constraints. It remains to be seen if this refactoring is feasible in all circumstances without overly encumbering the programmer. When refinement fails tasks will be overconstrained and the system will execute many tasks sequentially. We do find that SvS is well suited for the particular application domain we are targeting. Our test implementation is centered around arrays since game engines often employ them for efficiency reasons. Expanding the process of refinement to include other types of data structures and determining the concrete limitations of SvS are part of our ongoing work.

4. Experimental Results

4.1 Methodology

To demonstrate SvS's viability we use the popular 3D Character Animation Library (Cal3D) [1] : a skeletal based animation library that operates by representing a character model as a hierarchical set of bones. Realistic motions are generated by mathematically blending multiple precomputed animations. Animations in Cal3D are a sequence of transformations to be applied to specific bones. If more than one animation is applied the results are 'blended' together and so a bone may be modified multiple times during the production of one pose. However, a bone is unique to a model and the processing of a model is independent from the processing of other models.

Our test application simulates an open world populated with several models. In order to establish a base-

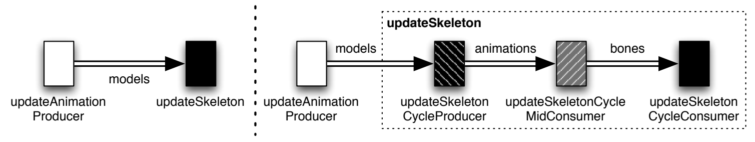

Figure 1. Task graph for animation pipeline: restricted (left) and SvS (right)

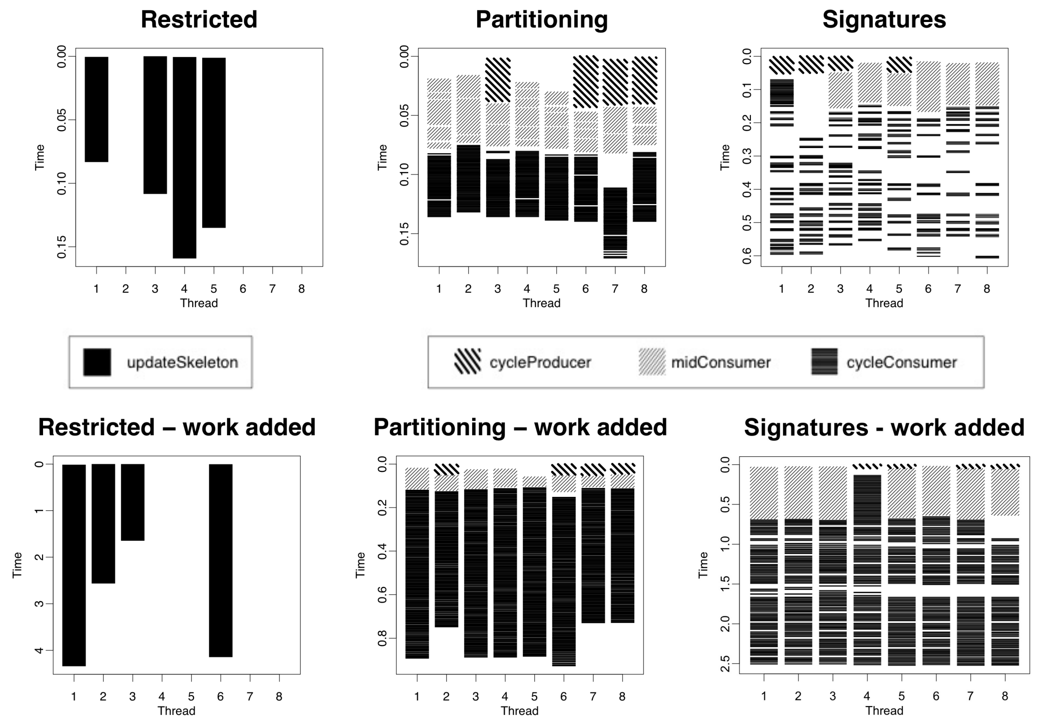

Figure 2. Activity graphs with synthetic work

line for comparison, we first implemented a version of our test application with only explicit dependencies providing state protection. As a result, this *restricted* version is limited to processing models in parallel. Figure 1 (left) shows the task graph of this version, where the task **updateAnimationProducer** sends model references to a data-parallel consumer, (**updateSkeleton**), which performs all computations. This version's parallelism is limited by the number of models.

SvS allowed us to safely increase the granularity of tasks from the *restricted* version. Potential state conflicts are managed and so we can process bones in parallel. The task graph for this version is shown in figure 1 (right). Like before, **updateAnimationProducer** distributes models to a data parallel consumer, **updateSkeletonCycleProducer**, which distributes the updated animations to the **updateSkeletonCycleMidConsumer** which then iterates over the animation and sends the associated bone ids to the **updateSkeletonCycleConsumer**. Finally, **updateSkeletonCycleConsumer** updates the bones stored in an array indexed by the bone ids.

4.2 Results

Figure 2 illustrates the assignment of tasks (shaded boxes) to threads (x-axis) over time (y-axis, in milliseconds) in what we call an activity graph. Rectangles with the same shading represent a data-parallel task. Note that in the Cascade runtime threads are bound to cores. The white gaps between rectangles represents the time spent in the Cascade runtime waiting for new tasks to execute, or dequeueing a data item to process.

Our experiments were run on a machine with 2 Intel Xeon E5405 chips with 4 cores each. Two cores share a 6 megabyte L2 cache for a total of 12 megabyte per chip. For all our tests, we used 4 models and 8 animations running with 8 threads. These parameters represent the case where there are bountiful resources and most reflects probable future conditions. The results for a typical frame are presented in the upper 3 graphs of figure 2. The upper leftmost shows a frame of the *restricted* version of our test application. We can see that parallelism is limited to only 4 tasks, due to our chosen parameters. The average execution time was 0.145ms (with a standard deviation of 0.008). The upper middle and rightmost activity graph shows an execution frame using SvS with partition and signature based scheduling. We can see that with SvS we are able to achieve much more parallelism in the system. Note that the execution times exceeds that of the *restricted* case with average runtimes of 0.210ms and 0.673ms and standard deviations of 0.029 and 0.093.

The increased execution times for SvS are a result of overhead in the Cascade runtime. Notice that the fine-grained tasks execute on the order of a few microseconds, but have significant relative overhead. This is not a failure to optimize the Cascade library, but reflects the unavoidable fact that scheduling a task requires a minimum amount of computation. SvS has allowed us to achieve more parallelism than can actually be used. These results underscore the necessity to rethink how parallel computations are organized on this fine grained level.

Other researchers have also pointed out the importance of supporting fine-grained tasks and proposed hardware based solutions [9, 11]. Without hardware support, efficiently executing fine grained tasks is only possible by considering the effects of scheduling decisions on the whole system. This makes previously minor performance factors such as cache contention very important. Having a large number of tasks to possibly execute means that the implications of executing a given task must be taken into account. In certain cases it would be advantageous for threads to do other work or even remain idle so as improve the performance of the system. These kind of low-level optimizations have so far been achieved by tuning applications for specific platforms. Our goal with Cascade is to generalize these techniques and apply them automatically. We have shown how to use information unique to a task graph to protect shared state and we plan to use this information to make better scheduling decisions.

To confirm our assertions about the nature of the overhead we added a small amount of synthetic work to tasks in our experiments. This changed the scale of the experiment without affecting the granularity. The lower three graphs in figure 2 show the same test with this additional work. The average execution times from left to right are 4.15ms, 1.21ms and 2.52ms with standard deviations of 0.209, 0.143 and 0.202. This helps confirm that if sufficient scheduling techniques are developed then techniques like SvS can be used to realize real-world performance and productivity gains.

5. Related Work

A variety of new programming environments, languages and paradigms have emerged in response to challenges of parallel computing. Due to space constraints, we focus on two, Jade [10] and Prometheus [5], that most closely relate to our key focus. Jade proposes a set of parallel extensions to C. A programmer denotes blocks of code as tasks and specifies their data constraints. Jade does not explicitly support a task-graph model and tasks are spawned dynamically. Although Jade also schedules tasks based on their constraints there are fundamental differences. In Jade the constraints are specified by the programmer whereas in our system they are derived automatically, thus freeing the programmer from the need to concentrate on implicit and hard to spot data dependencies. In Jade, memory references are checked dynamically during execution for compliance with constraints. In Cascade, no such per-reference overhead is incurred. A task's constraints is based entirely on the information available *before* the task runs. Prometheus' Serialization Sets work similarly to Jade, but they are applied to an object-oriented language and protect from races within an object. Shared state protection in Cascade is more general.

Software transactional memory (STM) may play a role in Cascade's future. The cases of overconstraint in section 3.4 are cases where STM may be of use. If there is a small probability of a conflict tasks could be executed as transactions since costly aborts would be rare. STM was applied in the domain of game engines before [6] [7], but not considered as a fallback from another synchronization method.

Traditional static techniques for extracting parallelism rely on loop and subscript analysis[4, 8], but are only valid when there are no dependencies between iterations. Therefore these techniques cannot handle dataflow with shared state.

6. Conclusion and Future Work

In this paper we proposed a new way of managing shared state in task-graph based programming models. We distinguished between explicit and implicit dependencies and proposed a new approach to manage implicit dependencies automatically, without programmer involvement and without relying on traditional methods for managing shared state, such as locks. While our approach, SvS, has limitations as a general approach, it fits nicely for the game engine domain. Future work includes further exploration of the applicability and limitations of SvS and enhancing Cascade with other optimizations such as data locality, GPU integration, prefetching and other avenues that our explorations uncover.

References

[1] Cal3d character animation library `http://home.gna.org/cal3d/`.

[2] Andersson. Parallel futures of a game engine `http://repi.blogspot.com/2009/11/parallel-futures-of-game-engine.html`.

[3] Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.

[4] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. *SIGPLAN Not.*, 21(7):162–175, 1986.

[5] Allen et al. Serialization sets: a dynamic dependence-based parallel execution model. In *PPoPP '09*, pages 85–96, 2009.

[6] Baldassin et al. Lightweight software transactions for games. In *HotPar '09*, 2009.

[7] Zyulkyarov et al. Atomic quake: using transactional memory in an interactive multiplayer game server. In *PPoPP '09*, pages 25–34, 2009.

[8] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. Interprocedural parallelization analysis in suif. *ACM Trans. Program. Lang. Syst.*, 27(4):662–731, 2005.

[9] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 162–173, 2007.

[10] Martin. Rinard and Lam. The design, implementation, and evaluation of jade. *ACM Trans. Program. Lang. Syst.*, 20(3):483–545, 1998.

[11] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. Flexible architectural support for fine-grain scheduling. In *ASPLOS'10: Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.