Synchronization via Scheduling

Techniques For Efficiently Managing Shared State

Micah J Best
University of British Columbia
mjbest@cs.ubc.ca

Shane Mottishaw [*]
Craig Mustard Mark Roth
Alexandra Fedorova
Simon Fraser University
{smottish, cam14, mroth, fedorova}@cs.sfu.ca

Andrew Brownsword
Electronic Arts, Inc
brownsword@ea.com

Abstract

Shared state access conflicts are one of the greatest sources of error for fine grained parallelism in any domain. Notoriously hard to debug, these conflicts reduce reliability and increase development time. The standard task graph model dictates that tasks with potential 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 programmer often creates more dependencies than needed, which results in constrained graphs with large monolithic tasks and limited parallelism.

We propose a new technique, Synchronization via Scheduling (SvS), that uses the results of static and dynamic code analysis to manage potential shared state conflicts by exposing the data accesses of each task to the scheduler. We present an in-depth performance analysis of SvS via examples from video games, our target domain, and show that SvS performs well in comparison to software transactional memory (TM) and fine grained mutexes.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming; D.3.4 [*Processors*]: Compilers; D.3.4 [*Processors*]: Run-time environments

General Terms Design, Languages, Measurement, Performance, Reliability

Keywords parallel programming, shared state management, Synchronization via Scheduling, Dynamic Reachability Analysis

1. Introduction

Shared state access conflicts are the cause of majority of errors in parallel programming. Race conditions and corruption of shared data are common. These bugs can be notoriously difficult to track down as they often manifest rarely, depending on the state of not just one, but several threads of execution. Unfortunately most existing frameworks don't provide mechanisms to automatically protect shared state. Runtime systems such as the one driving Cilk [12], OpenMP [4] and the venerable pthreads are largely concerned with dispatching code for execution. Systems such as Intel's TBB [6] do provide a vast array of synchronization primitives, including a number of distinct types of mutexes, for the programmer to construct their own state protection. Unfortunately, they contain very little in the way of support for orchestrating these schemes. Software Transactional Memory(STM) [25] does address this problem and as STM is the closest in effect to our proposed technique, we will discuss it further in the context of our experiments in Section 3.

Very few domains have been as profoundly affected by the multicore revolution as video games. The complexity and high level of interaction of the systems which often rivals operating systems, a nearly inexhaustible demand for better performance, large programming teams and tight deadlines makes it an ideal testbed for parallel techniques. The volume of this software and its commercial appeal would make the problems worth solving even if they were unique to the domain, but any technique developed have application in many other domains. For these reasons we have chosen to let the needs of this domain drive our research. Our assumptions and choice of experiments reflect this.

Programmers faced with the difficulties in managing the data accesses of a large collection of tasks tend to leave many tasks monolithic, comprised sometimes of thousands of lines of codes. The embarrassingly parallel kernels, those without complex state interaction, will generally be dispatched in patterns similar to `parallel_for`. This has lead to the structure of the current generation of video games. Often the work of an entire subsystem or one its major components will be assigned to a single processing context, co-scheduled with other components that are guaranteed to be conflict free. Interspersed between the execution of these groups will be the execution of the embarrassingly parallel sections and a number of explicit synchronization points. This structure is common in the industry [1]. The lack of parallel width in many phases of execution leaves resources idle and this approach will not scale as the number of cores increase.

The parallel structure of the program is often represented in the standard task graph model where tasks that have not had a dependency declared between them can be scheduled concurrently. There are cases where ensuring ordering of tasks with dependencies is necessary for correctness. However, there remain many cases where there is not an explicit logical ordering between the tasks and a dependency is declared to prevent two tasks that touch the

[*] Partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC)

same data from running concurrently. This serializing of tasks is necessary even if the two tasks touch the same data rarely.

When a program is executed and tasks are serialized unnecessarily because of unneeded dependencies, parallelism is reduced and performance can suffer. This work proposes a mechanism to correct this deficiency. Our model only requires that the programmer state *explicit* inter-task dependencies, i.e., those that are required by the program logic. In cases where tasks that are not explicitly dependent on each other *may* touch the same data, our system automatically inserts a new *implicit* dependency between them, which will prevent these tasks from running concurrently and corrupting shared state. We use conventional static analysis to determine when implicit dependencies are necessary. During runtime, when we can determine more precisely what data the task will actually access, we detect and remove any overly constraining implicit dependencies – to accomplish this we propose new dynamic analysis techniques. Making the shared state access patterns of each applicable task available to the scheduler we are able to safely schedule tasks with potential conflicts. We call this technique *Synchronization via Scheduling*(SvS).

The general concept of SvS is not overly complicated. The chief difficulty lies in utilizing the information provided by the static analysis and refinement without adding much overhead. To achieve the highly desirable performance of 60 frame-per-second (FPS), a frame must be constructed in just over 16*ms*. For optimal parallelization many tasks will have runtimes in the tens or hundreds of microseconds. This strict time budget does not allow for much extra computation. One of the major contributions of this work is the description of algorithms to achieve this high speed organization and to demonstrate that they work in practical contexts.

SvS differs from optimistic techniques such as STM in that the work of evaluating the admissibility of a task is done prior to its execution. Though some of the mechanisms used to realize SvS are similar to those in some STM implementations this difference means that there are no expensive rollbacks and there is a much smaller requirement for extra bookkeeping during state access. Additionally, STM requires programmers to define atomic transactions while SvS is an automatic technique completely handled by the compiler and runtime components. We will show a comparison between SvS and STM in section 3.

The rest of the paper is organized as follows. In Section 2 we discuss the model, algorithms and implementation of SvS. Experimental results, drawn from several applications in our domain, are presented in Section 3. In section 4 we will discuss related work and in Section 5 we will conclude and give a short discussion of our future work.

2. SvS Model and Implementation

2.1 Motivation and Overview

Task graph models are a standard pattern for structuring parallelism in programs [20]. A prevailing problem with this model is the lack of automatic shared state management between tasks. Consider an example of skeletal character animation. Typically, multiple animations are applied to the bones of a single character to produce realistic looking motion [2]. For example, to produce a character that is walking and limping we may blend the "walking" and "limping" animations. The mathematical operations performed by these two routines are commutative, so there is no explicit ordering between them. However, it is unsafe to execute these animation routines in concurrent tasks, because they *may* touch the same bones of the same character. In this case there exists a special type of dependency between tasks, which we term *implicit*. An implicit dependency exists when there is no logical ordering between the tasks imposed by data or control dependencies, but the tasks may access the same shared state and so it is unsafe to run them concurrently.

Without automatic shared state management, programmers must manage shared state by inserting explicit dependencies where implicit dependencies exist. Protecting shared state via explicit dependencies has two problems. First, this is prone to programmer error, especially when considering dynamic, pointer-based memory accesses. Second, this unnecessarily constrains parallelism in cases where the tasks *may* incur conflicting accesses of the shared state, but do not *actually* perform them at runtime.

We address the issue of shared state management in task graph models by introducing a new technique called Synchronization via Scheduling (SvS). SvS provides *automatic* shared state management by combining static and dynamic analysis to determine if two tasks can potentially access shared state. The result of static analysis is a task graph with dependencies that guarantee the protection of shared state. Dynamic analysis then utilizes run-time information to potentially remove unnecessary dependencies between tasks, allowing for increased parallelism. In this way, SvS determines the set of *possible* memory accesses a task makes *before* it is executed and schedules tasks such that no two tasks concurrently access the same memory. In this section, we will provide the details behind the model and implementation of SvS. In section 2.2 we outline the framework for SvS, followed by a brief discussion of relevant background information in sections 2.3 and 2.4. Starting in section 2.5, we will go into detail on the model and implementation of SvS.

2.2 Framework

Figure 1 shows the four main components that comprise the SvS framework: an SvS compatible language, static analysis, dynamic analysis, and the task scheduler. An SvS compatible language allows a programmer to group blocks of code into units called tasks. Programmers can provide a logical ordering between tasks but do not need to manage shared state between them. Beyond providing a task-graph abstraction, an SvS compatible language must be type safe and disallow pointer arithmetic. We describe our prototypical implementation of an SvS compatible language in section 2.4.

At compile time, a program written in an SvS compatible language is passed through a static analysis phase that generates information pertaining to symbols (linguistic abstractions for memory accesses) and task dependencies. This information defines a static task graph which provides an initial scheduling of tasks that ensures the protection of shared state. Information from static analysis is stored for use during dynamic analysis.

We purposely visualize the static analysis in figure 1 as a "black box" because SvS is indifferent to the implementation of static analysis. Static analysis techniques that produce a correct list of task dependencies and a list of all symbols that a task may access are suitable for use within the SvS framework. Static analysis in SvS is formally defined as *task dependency analysis* in section 2.5; this section also explains how task dependency analysis can be implemented using existing techniques. The classic limiting factor of static analysis when applied to parallelization is that it is limited to compile time information. As a result, it often is forced to create dependencies that are potentially unnecessary, thus restricting parallelism. This problem can be alleviated using dynamic analysis.

Throughout the execution of an SvS program, dynamic analysis maintains dynamic reachability information (i.e. potential memory accesses) for symbols accessed by tasks. As tasks are considered for scheduling, this information is used to generate and compare read/write sets of tasks in order to remove any implicit dependencies that were deemed necessary by the static analysis, but were found to be non-existent when dynamic reachability information became available at runtime. We call this process of removing unnecessary static dependencies *refinement*.

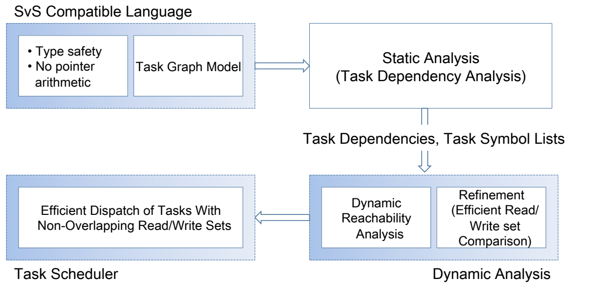

SvS Compatible Language
• Type safety
• No pointer arithmetic
Task Graph Model
Static Analysis (Task Dependency Analysis)
Task Dependencies, Task Symbol Lists
Efficient Dispatch of Tasks With Non-Overlapping Read/Write Sets
Dynamic Reachability Analysis
Refinement (Efficient Read/ Write set Comparison)
Task Scheduler
Dynamic Analysis

Figure 1. SvS Framework

Finally, the scheduler respects the set of refined dependencies by ensuring that all concurrently executing tasks have distinct read-/write sets (i.e. no implicit dependencies), in effect, performing synchronization via scheduling.

While the SvS framework is the new contribution of this work, its implementation relies on both new and existing techniques. In particular, our framework allows for the use of existing well-established static analysis techniques, but the algorithms used in dynamic analysis and in scheduling are new to this work.

2.3 Task Graph Model

In this section, we define the task-graph model that is assumed by the current implementation of SvS. In task-graph based execution, code is divided into discrete units called *tasks* and a *task graph* defines a static scheduling of these tasks through directed edges. If there is an edge (A, B) then the task A, the parent, must complete before B, the child, can be executed. These edges are referred to as *dependencies* and a dependency is *satisfied* when the parent completes execution. Our task-graph model also includes explicit dataflow where one task, a *producer*, 'sends' data to another task, a *consumer*. Two tasks involved in dataflow have a *dataflow dependency*. Currently, our task graph model does not allow the programmer to specify cyclic dependencies.

A task that has no parents or has all dependencies satisfied is considered runnable. Additionally, a consumer task must also have been sent data to be runnable. A task *instance* is a task running on a processor. When a task is runnable, we say that an *instance* of it can be scheduled. Task instances are generated in one of two ways. If the task is a consumer task, then a copy (i.e. instance) is executed for each data item received, and the collection of all instances define a data-parallel operation. We describe these instances as being part of a single data-parallel task. For all other tasks, an instance is generated "statically" at the beginning of the program.

Besides dependencies, there is also an implicit temporal ordering between executions of a task graph in that we execute all tasks in the graph and wait for them to finish before executing the graph again. More formally, if we define the execution of all tasks in a task-graph to be an *iteration*, then iteration i must complete before $i + 1$ begins.

The dependencies and constructs described in this section support the task-graph requirements of an SvS compatible language, which we describe in the next section.

2.4 CDML

To facilitate writing programs based on the task-graph model outlined in section 2.3 and enable the static analysis required for SvS, we have developed the Cascade Data Management Language (CDML). Because C++ is the standard language for game development, CDML is similar to C++ with a few added annotations and restrictions. Note that CDML is not a requirement for SvS; as described in section 2.2, essentially any language that includes the following features is suitable for SvS: (1) syntax for articulating the task graph model described in section 2.3, (2) type safety, (3) no pointer arithmetic.

Additionally, because the current specification for CDML does not yet support object-oriented programming, we assume no inheritance or polymorphism in our current implementation of SvS, but this is not a requirement. We plan to address inheritance and polymorphism in future work. Because we are not presenting CDML as a contribution of this work, the full syntax and features of CDML will not be discussed here.

The grammar for CDML tasks is shown in figure 2. There are two task types in our current language specification: *transform* and *itemizer*. A transform is just a (static) single instance task. An itemizer is used to implement data-parallel tasks, where multiple instances of the task's body are executed to process items received at run-time. An instance is created for each item received.

Our system assumes that there is no specific ordering between tasks, unless the programmer explicitly specifies a dependency. *Explicit* (i.e. "ordering") constraints are expressed in a tasks *constraints*. At run-time, explicit constraints specified by the programmer are never broken. In many cases, a programmer would not need to specify an explicit ordering between tasks, because the same outcome will be achieved regardless of the ordering. This is especially the case for video game engines and scientific computing applications where many computations are commutative. The programmer can also specify data-flow dependencies using the *send* constraint.

Programmers do not have to manage shared memory accesses between tasks. Static and dynamic analysis are used to automatically detect when two tasks can access the same memory. In the case where tasks may perform conflicting accesses to the same data, SvS chooses an arbitrary ordering for the tasks and runs them sequentially. Otherwise, tasks can be run concurrently.

We implemented a translator that converts CDML code into C++. The translator also performs the static analysis to detect implicit dependencies, as described in the next section.

```
task := task_type task_name ':' constraints? body
task_type := 'itemizer' | 'transform'
constraints := ( send | receive | explicit )+
body := '{' statements '}'
```

Figure 2. CDML task syntax

2.5 Static Analysis

We term the static analysis performed in SvS as *task dependency analysis*. The goal of task dependency analysis is to statically find implicit dependencies between tasks – that is, determine whether two tasks (or task-instances) can potentially access the same memory location. The collection of implicit and explicit dependencies define a task graph that ensures the protection of shared state. Because task dependency analysis is essentially a form of dependency analysis, we will present the definition of dependency analysis and derive from it a formal definition for task dependency analysis.

In traditional dependency analysis [9], the fundamental goal is to determine whether a statement T depends on a statement S. T depends on S if there exists an instance $S^{'}$ of S, an instance $T^{'}$ of T, and a memory location M such that:

1. Both $S^{'}$ and $T^{'}$ reference M, and at least one reference is a write
2. In the serial execution of the program, $S^{'}$ is executed before $T^{'}$
3. In the serial execution, M is not written between the time that $S^{'}$ finishes and the time $T^{'}$ starts

As mentioned in the previous section, the ordering between tasks in SvS, and thus their accesses, are assumed to be commutative unless the programmer enforces an ordering between tasks by inserting explicit dependencies. For the remaining pairs of tasks, we are not concerned with the order in which they execute. Because of this, conditions number 2 and 3 are not applicable to SvS. It follows from this that task dependency analysis is not concerned with whether the dependency is flow-dependent, anti-dependent, or output-dependent. Therefore, task dependence analysis in SvS can be restated as follows: A dependency exists between task T and a task S if there exists an instance $S^{'}$ of S, an instance $T^{'}$ of T, and a memory location M such that:

> Both $S^{'}$ and $T^{'}$ reference M, and at least one reference is a write. A task references M if there exists a statement X in the body of the task that references M.

In modern programming languages, a reference to a memory location M might be represented as a scalar variable, array, or pointer. In SvS, we refer to these abstractions for memory locations as *symbols*. The syntax for a symbol in CDML is provided in figure 3 and mirrors the syntax of C/C++ expressions for array, variable, and member access. Since symbols abstract references to memory, task dependency analysis becomes collecting the symbols in the body of a task and determining if a symbol x in task S can reference the same memory location M as symbol y in task T where at least one of the references is a write. The problem of determining if two symbols can reference the same memory has been thoroughly explored by research in static analysis including pointer analysis [8], array dependence analysis [21], shape analysis [19], and disjoint heap analysis [15].

Because it is not our goal to expand upon work that has already been done in static analysis, our current implementation is very conservative. As a result, our current approach performs the necessary symbol collection and produces a set of dependencies that generally result in a dependency being placed between each pair of tasks. In this case, dynamic analysis is exclusively responsible for uncovering parallelism. As it will be discussed in sections 2.6.4 and 2.7, this is achieved by collecting run-time information describing the potential memory accesses of the symbols extracted during static analysis in order to "recalculate" (i.e. refine) dependencies. We will show in section 3 that SvS is feasible even with dynamic analysis performing most of the work. However, we hypothesize that using more sophisticated static analysis could only reduce the amount of dynamic checks (i.e. dependency refinement), thus decreasing run-time overhead. Expanding the role of static analysis and implementing more sophisticated static analysis is a definite part of future work for SvS.

SvS is indifferent to what techniques are used to solve task dependency analysis as long as the output is a set of symbols for each task, and a set of dependencies between tasks that guarantees no two task instances can concurrently access the same memory location (which will be the case if the techniques correctly solve the task dependency analysis problem). Therefore it is not a goal of SvS to expand upon the work that has already been done in static analysis, but rather to address the deficiencies associated with static analysis.

2.6 Dynamic Analysis

Due to the limitations of compile time information, static analysis is often forced to make conservative assumptions. This may result in unnecessary dependencies, thus hindering parallelism. The goal of dynamic analysis is to potentially remove such dependencies at run-time. To achieve this, we use information available at run-time to generate more precise read/write sets for tasks. Then, as task instances are considered for scheduling, we efficiently compare their read/write sets to see if a dependency actually exists (a process we call refinement) and subsequently scheduling non-dependent tasks to execute concurrently.

```
symbol := identifier acessor*
identifier := [a-zA-Z_][a-ZA-Z0-9_]*
accessor :=
     '->'  identifier
  |  '.' identifier
  |  ('[' expression ']')+
```

Figure 3. CDML symbol syntax

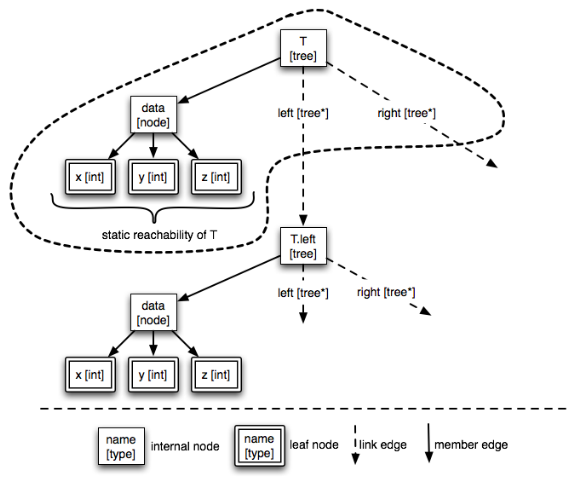
To calculate read/write sets, we monitor the connectivity and reachability properties of memory objects, our online abstraction for memory accesses, to determine the set of all addresses that can possibly be *reached* by a memory object. We call this set of accesses the *reachability* of a memory object and its connectivity properties are represented as a *reachability graph* (section 2.6.1). We use *dynamic reachability analysis* (sections 2.6.4 and 2.6.3) to maintain dynamic changes to reachability graphs as memory objects are created and linked together. Since symbols reference memory objects at run-time, this enables us to determine the reachability of symbols accessed by tasks and therefore more precise sets of potential reads and writes.

We will also introduce *signatures* (section 2.6.2), which are used to compactly represent read/write sets and efficiently determine which tasks have non-overlapping memory accesses. Tasks with non-overlapping read/write sets can then be scheduled concurrently. Two new scheduling algorithms that efficiently accomplish this goal will be presented in section 2.7.

2.6.1 Memory Objects, Links, Reachability and Reachability Graphs

We use the notion of a *memory object* to abstract memory accesses in SvS. In the simplest case, a memory object is a single primitive (e.g. `int`) and provides a direct access to memory. In general, memory objects may contain one or more primitives or other memory objects. Primitives and/or other memory objects that compose it are called its *members*. Memory objects may also contain *links*. A link 'points-to' a child memory object, which allows the parent memory object that contains the link to access all the memory addressable by the child memory object. The difference between members and links is that members are *static* – they cannot be removed from the object and their memory addresses within the enclosing object cannot be modified, whereas links are dynamic. The child that a link points to can be changed at any time, thus changing the set of memory addresses that a memory object can access. Links can also exist on their own, in that they do not need to be declared as a member of a memory object. Therefore, SvS tries to solve the problem of determining what memory objects a task can possibly access before the task runs, where memory objects can be accessed directly through members and indirectly through links.

We formalize these definitions by representing the problem as a graph. Memory objects represent nodes in the graph. A *member* edge is a directed edge defined as (X, Y) where memory object Y is a Member of X. A 'link' edge is a directed edge defined as (A, B) where A is a memory object that contains a link L that points to memory object B. We say that A is the parent of L and B is its child. If a link does not have a parent, it is essentially just an alias for the memory object it points to. Changing L to point to a different memory object C effectively removes the edge (A, B) and adds the edge (A, C). This graph represents the *dynamic reachability* of the memory object and is called the *reachability graph*.



```
struct node {          struct tree {
    int x;                 node data;
    int y;                 tree* left;
    int z;                 tree* right;
};                     };
```

Figure 4. The graph for a memory object representing a binary tree. The sub-graph inside of the dashed boundary represents the result of a breadth-first search that only follows member edges. The leaf nodes in the area are the containment of the memory object `T`. Related C++ definitions are provided on the bottom.

Given any node, (i.e. memory object), in the graph, the set of memory addresses that can be reached (i.e. accessed) by the node is called its *reachability* and is defined as the set of all leaf-nodes reachable by performing a breadth (or depth) first search starting at the given node. Because leaf nodes are primitives, they directly correspond to an addresses in memory, and thus define a set of memory addresses. Figure 4 provides an example of a graph that would be defined by a typical binary tree. The leaf-nodes inside of the dashed boundary represent the static reachability (unique, static set of memory accesses) of the root node of the tree.

By keeping track of the structure of the reachability graph for each memory object (a process we call *dynamic reachability analysis*), we are able to dynamically monitor reachability information providing significant insight into the potential memory accesses of tasks. This is particularly useful when dealing with dynamic data structures that allow for ambiguous accesses to memory. Implementation of reachability graphs and pertinent algorithms are described in the next sections.

2.6.2 Signatures: Representing Memory Accesses

Sets of memory accesses in SvS are represented as *signatures*: constant length bitstrings. When two signatures have the same bit set, it means they represent access to the same memory location (or memory object) and are said to overlap. To build a signature, id's (i.e. memory object id's) representing reads or writes are passed to a hash function to determine the bit to set in the signature. Signature overlap is checked using simple and efficient bit-wise operations.

Note that signatures are effectively Bloom filters [11] using a single hash function. Also, because signatures are constant in length and use hashing, there is the opportunity that false positives occur when comparing signatures. This does not affect correctness and its impact to performance can be greatly reduced by using large signature sizes with negligible impact to performance, which will be discussed in section 3.1.

2.6.3 Implementing Reachability Graphs and Dynamic Reachability Analysis

As discussed in the previous section, there are two main components to a reachability graph: memory objects and links. The goal of the implementation for these structures is to provide the meta-data and meta-functions necessary to efficiently maintain reachability graphs and extract the reachability of a memory object.

Memory Objects Memory objects are implemented as classes that inherit from a MemoryObject class template. The MemoryObject class stores the id of a memory object that is generated inside the class's constructor.

The **getSignature** function of the MemoryObject class returns a signature representing the reachability of a memory object. A straightforward way to implement this function is to simply perform a breadth first search by calling the **getSignature** function of each member, or the **getSignature** function of the memory object a link points to, and combine the returned signatures using a bitwise-or operation. For large reachability graphs, a breadth first search will be too expensive. Instead, we have implemented a more efficient method that utilizes the implementation of links as described in the next section.

Links Links are implemented in SvS as a smart pointer template class. Since a link is just an edge in the reachability graph, the smart pointer representing the link stores pointers to a parent and child memory object. The child pointer represents the memory object that a link "points-to" whereas the parent pointer denotes the memory object that the link is a member of. A null parent represents the case where a link is just a reference or alias to the memory object it points to and is not considered to be an edge in the reachability graph.

We now discuss how smart pointers are used to calculate and maintain the reachability of a memory object. The reachability of a memory object is changed when we change the child node of a link edge. This is equivalent to link assignment, and thus by overloading the assignment operator for the smart pointer class, we can detect a change in reachability and perform the necessary updates. The algorithm that performs these updates is called *dynamic reachability analysis*. First, consider the situation where each memory object stores a signature that accurately represents its reachability. Initially, the reachability of a memory object is just the signature created from its object id (i.e. the signature representing its static reachability). When a link L is changed to point to a memory object B, it means that the memory object $A = parent(L)$ can now access all memory objects reachable by B. It also means that all memory objects that can reach A can also reach memory objects reachable by B. Therefore, during link assignment, we could perform a reverse breadth first search starting at A, recursively updating the signature of each node to include the signature for the reachability of B. However, we want to reduce the cost of this breadth first search. To do this, we introduce the concept of *master* nodes.

A master node M represents a bounded set of reachable nodes, i.e. a set of nodes X such that a path $M \rightsquigarrow X$ exists. We call the set of nodes X the *domain* of M. A master M maintains a signature that accurately represents its reachability; this signature is shared by all nodes in the domain of M. M is also responsible for propagating changes in the reachability of nodes in its domain to all other masters that can reach M.

All other nodes are called internal nodes. An internal node X can belong to multiple domains and keeps track of which masters (i.e. domains) it belongs to. X is responsible for notifying each of its masters when its reachability changes. We call the first domain an internal node is assigned to its *primary master*. An internal node belonging to multiple domains has multiple signatures that conservatively (but correctly) represent its reachability, so we arbitrarily choose the signature of the primary master to represent its reachability.

By introducing master and internal nodes, we essentially establish a tree of masters that is smaller than the original reachability graph and only maintain precise reachability information for masters. This decreases the cost of the reverse breadth first search required to monitor reachability. Under this implementation, the **getSignature** function of a memory object just returns the signature of its primary master. We are currently investigating more efficient methods for maintaining dynamic reachability information, but algorithm 1 provides our current implementation of dynamic reachability analysis, including how masters are created. In this algorithm, the list `Node.owningMasters` are the masters to whose domain `Node` belongs. `Master.notifyMasters` is the list of masters that the `Master` must notify on when its reachability changes, because these masters can reach the nodes in Master's reachability. Note that our algorithm also performs cycle detection in the reachability graph, but we omit the pseudo-code due to space limitations.

2.6.4 Implementation of Dynamic Refinement

The goal of dynamic refinement is to remove unnecessary implicit dependencies created by static analysis. To this end, the dynamic refinement process determines which memory objects the task may *actually* access by using the reachability of the referenced symbols and retrieving the corresponding signatures. The implementation of generating signatures for tasks during refinement is shown in algorithm 2.

Our algorithm is only concerned with the reachability of global symbols or received symbols (those that were received as an argument) (see line 4). Symbols that are local to the task are not of concern since they are invisible outside of task boundaries, unless they alias global or received symbols. Our implementation of static analysis keeps track of aliasing, so the potential shared accesses of local symbols would be represented as the reachability of the corresponding global or received objects.

Note that the signature generated by our algorithm is guaranteed to represent all possible memory accesses that a task will make during execution, even if the task performs link assignment, i.e. changes the reachability of a memory object. This is because we account for the reachability of *all* memory objects a task can access, and link assignment just changes the reachability of one memory object to include the reachability of another memory object, and thus does not change the cumulative reachability of all the memory objects in the task.

Each task in SvS has a **makeSignature** function that implements algorithm 2; the code for **makeSignature** is generated by the translator using symbols collected during static analysis. As link assignments occur during program execution, dynamic reachability analysis maintains signatures representing reachability (possible memory accesses) of memory objects. Because **makeSignature** builds a composite signature of symbols (which reference specific memory objects at run-time), the signature returned represents a description of memory objects that a task can access at that time. Therefore, the process of refining a dependency between two tasks is basically just calling **makeSignature** for each task and comparing the resulting signatures to see if a dependency in fact exists. By performing this check, we can dynamically re-calculate dependencies between tasks. This process is only performed for implicit dependencies. If an explicit dependency was specified by a programmer, then this dependency will not be removed.

Also note that false positives can occur during signature comparison. Besides the false positives caused by using signatures, there are two additional causes for false positives. The first is due to the conservative assumption that a task accesses the entire reachability of a memory object. The second is due to multiple memory objects sharing the signature of a master node, as described in section 2.6.3.

While refinement is conceptually a separate component in the SvS framework, its implementation is integrated with scheduling, as described in the next section.

Algorithm 1: Link Assignment

```
Input: A link $lhs$ with parent memory object $A$, and a link
       $rhs$ whose child is $B$
Output: New edge $(A, B)$
1  begin
2      if A.owningMasters = $\emptyset$ then
           /* A is its own master */
3          A.owningMasterList.add( A )
4          A.domainSize = 0
5      end
6      if (A.primaryMaster.domainSize < K) or
       not(B.owningMasters = $\emptyset$) then
7          if B.owningMasters = $\emptyset$ then
8              A.primaryMaster.domainSize++
9          end
10         foreach $a \in$ A.owningMasters do
11             foreach $b \in$ B.owningMasters do
12                 b.notifyMasters.add( a )
13             end
14             a.updateReachability( getSignature(B) )
15         end
16     else
17         B.owningMasters.add( B )
18         foreach $a \in$ A.owningMasters do
19             B.notifyMasters.add( a )
20         end
21     end
22 end
```

Algorithm 2: run-time calculation of a signature to represent all possible memory accesses that a task will make

```
Input: Task $T$
Output: Signature $S$
1  begin
2      Let $L_T$ = symbols($T$)
3      foreach $symbol \in L_T$ do
4          if symbol is not local then
5              $S$ += getSignature($symbol$)
6          end
7      end
8      return $S$
9  end
```

2.7 Scheduling Tasks

The key job of the task scheduler is to efficiently dispatch tasks with non-overlapping signatures to be executed concurrently. This effectively insures that each task's set of refined dependencies are respected.

Note that up to this point, we have discussed the process of refinement as comparing signatures between tasks in order to potentially remove dependencies. However, in cases where static analysis generates "many" dependencies (which is currently the case in our system), then performing pair-wise comparisons between tasks for each dependency as described in section 2.6.4 may not be very efficient. As mentioned in section 2.5, our current static analysis is extremely conservative and essentially ends up placing a dependency between each pair of tasks/task instances. In this case, refinement would be faced with approximately $\binom{T}{2}$ comparisons where T is the number of task *instances*, which can be large when dealing with data-parallel tasks. Therefore, instead of removing static dependencies, the scheduler essentially ignores this information and uses tasks' signatures (algorithm 2) to efficiently determine which task instances can be executed together concurrently. This is particularly useful for data-parallel tasks in general; data-parallel task instances are generated dynamically, making it difficult for static analysis to generate "meaningful" or "efficient" dependencies (e.g. if the operation is not obviously embarrassingly parallel, static analysis may just end up serializing all potential instances of the data-parallel task). In future work, we intend to incorporate dependency information in order to reduce the number of dynamic checks required to perform dynamic refinement and scheduling of tasks.

We designed and implemented two scheduling algorithms, which we present next. Due to space constraints we omit the pseudo-code for these algorithms and provide only textual descriptions.

2.7.1 Generations

The goal of generations scheduling is to create groups of task-instances such that no two instances have overlapping signatures; we call these groups generations. A single thread is elected to build generations. This thread tries to add a task to a generation in $delayList$, where each generation has a signature representing all tasks currently in the generation. A task can be added to a generation if the task's signature does not overlap with the generation's signature. If a task cannot be added to any of the generations in $delayList$, a generation from $delayList$ is released for processing and a new generation is added to $delayList$. The task is then added to this new generation and the signature of the new generation is initialized. Worker threads concurrently process tasks in a generation and ensure that tasks from different generations are never run concurrently by waiting for all threads to complete before advancing to the next generation in $scheduleList$.

2.7.2 Progressive

Progressive scheduling, attempts to execute tasks/task-instances as soon as possible, without violating dependencies. To do this, we maintain a signature, $workingSig$, that represents all tasks/task-instances currently running. For each task, a signature is created ($currentSig$) and we atomically compare $currentSig$ to $workingSig$, and if there is no conflict (i.e. no overlap between signatures), we atomically update $workingSig$ to include $currentSig$. If there is a conflict, $workingSig$ is not updated and the task is put back onto the queue for later execution. Otherwise, no dependencies exist between the current task and any tasks being executed, so the current task can be dispatched.

Because it is not possible to "subtract" from signatures when a task is completed, $workingSig$ will eventually become stale. This does not affect correctness, but it can affect performance in the form of false positives. To address this issue, we also keep track of the total number of signature updates and consecutive conflicts. If either of these values reach a threshold, we wait for all worker threads to finish executing tasks and then reset $workingSig$ and all flags.

2.8 Ensuring Correctness

It is important to underscore that SvS always generates a correct parallelization of the code written in CDML. The first step of this is the static task dependency analysis of the code, which builds a task graph that may contain unnecessary dependencies, but guarantees that shared memory accesses are protected. At run-time, SvS will dynamically recalculate dependencies and schedule tasks to prevent conflicting memory accesses between task-instances at run-time, enabling a greater degree of parallelism while still ensuring correctness.

3. Evaluation

Video games are a collection of tightly integrated systems (rendering, gameplay, physics, simulation, AI, animation, audio, user input, networking, GUI, etc) [1] that operate in concert on a rapid and repetitive timeline. Given the significant amount of code involved in a full game engine, studying one in its entirety is a difficult proposition. The complexity of commercial game engines and their attendant tool chains and development environments means that even building the project can be a daunting task. Thus converting an engine from the traditional sequential model into a modern task-based model is almost insurmountable and is not often attempted, not even in industry where it is usually preferable to instead re-implement from the ground up. So it necessary to isolate a particular facet or subset of features in order to study the effects of a particular technique. Therefore, we evaluate SvS using a collection of existing benchmarks and real applications.

We present two game based experiments. Firstly, Cal3D [2] is a third-party open-source skeletal animation engine used in several video games. Chosen because it has a relatively compact and clean code base, Cal3d represents typical computations performed in modern game engines. Secondly, QuakeSquad, our own video game benchmark, focuses on spatial partitioning, entity management, AI and managing numerous agents.

While not strictly game related, we also present three benchmarks from the PARSEC suite [10]: Canneal, Fluidanimate and Blackscholes. We chose these because PARSEC is a well known and respected benchmark suite and will help put our results in context.

To provide an evaluation of the primary parameters and costs associated with SvS, we developed micro-benchmarks and several experiments which are presented in the next section.

All SvS tests were written in CDML and executed using the Generational scheduling algorithm which our initial testing shows performs slightly better than Progressive. Further optimization of these algorithms is future work. For the sake of comparison we also parallelized Cal3D and QuakeSquad using Intel TBB 3.0 [6] and software transactional memory (STM) using the Dresden TM Compiler [14] and TinySTM++ library. In each case we found that the encounter time locking (ETL) algorithm performed the best for STM. The PARSEC benchmarks we used were available already parallelized with pthreads and in some cases TBB. Our experiments were run on a machine with two Intel Xeon E5405 chips with four cores each. Each two cores share a 6MB L2 cache for a total of 12MB per chip.

3.1 SvS Overhead

In this section, we provide an evaluation of the primary parameters and costs associated with SvS. SvS has two main run-time costs: false positives and the absolute cost of performing dynamic reachability analysis during link assignment. The key parameters governing these associated costs are signature and master domain sizes. In the following sections, we break down our analysis into two categories: signatures and dynamic reachability analysis.

3.1.1 Signatures

As mentioned in section 2.6.2, false positives can occur during signature comparison, potentially limiting parallelism. We define parallel width to be the number of tasks that are able to execute concurrently at a given time. In the the simplest case where a task accesses a single memory object, using signatures limits the theoretical maximum parallel width to the size (in bits) of the signature.

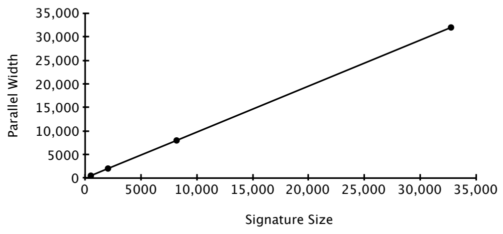


Figure 5. Parallel width for 128,000 task-instances under varying signature sizes

To demonstrate how signature size affects parallel width, we have designed an experiment consisting of a single producer task that sends unique memory objects to a data-parallel consumer task. The consumer task simply writes to a field of a received object. Note that the objects sent by the producer are single memory objects with no links as members (i.e. its reachability is static). Therefore, when an object is queried for its reachability, it just returns a signature representing its static reachability: a signature with a single bit set by hashing the id of the memory object. This means that there will be no false positives due to master nodes or conservative assumptions during refinement. Therefore, since all objects are unique, any detected conflicts are strictly due to false positives caused by signature size.

Figure 5 provides the average parallel width (y-axis) for varying signature sizes (x-axis) when the producer sends 128,000 objects. (This number was chosen to reflect the number of particles involved in modern fluid dynamics simulations). Note that because we use the generations algorithm, the parallel width at any given time is the size of the currently executing generation. Therefore to measure parallel width, we just record the sizes of each generation. The average parallel width was calculated over 100 executions of the producer and consumer.

Note that for all signature sizes, we (approximately) achieve the theoretical maximum parallel width and therefore the graph shows a linear increase in parallel width as signature size increases. This demonstrates that when conflicts occur, the generations algorithm is often successful in finding a generation in $delayList$ with a signature that does not conflict with the current object.

It is also important to note that the computational cost of increasing signature size is negligible. We have experimentally determined the cost of setting a bit to be about 10 cycles, and the cost of checking overlap on a 64-bit machine to be about $\frac{n}{64} * 10$ cycles, where n is the number of bits in the signature. This cost is further minimized by the fact that some signature operations can happen concurrently with executing tasks. Finally, the bitwise operations used when comparing/calculating signatures are prime candidates for vectorization.

Because parallel width increases linearly with signature size and the computational cost of increasing signatures is small, the overall cost of using signatures does not have a significant impact on the performance of SvS.

3.1.2 Dynamic Reachability Analysis

Dynamic reachability analysis has two primary costs associated with it that contribute to the overhead of SvS. The first cost is the absolute cost of performing dynamic reachability analysis, i.e. performing a link assignment. The second cost is false positives that occur due to memory objects in a domain sharing the same reachability signature: the signature of the master node representing that domain. Any false positives will in turn affect parallel width.

In general, absolute cost and parallel width are affected by the size (number of memory objects and links) and *shape* (i.e. layout/connectivity) of reachability graphs. In the case of absolute cost, larger reachability graphs potentially (although not necessarily) lead to more expensive reverse breadth first searches during link assignment. Also, since memory objects share the signature of a master and the reachability of a master is greater than the reachability of its successors, the larger the graph, the larger the potential for false positives due to sharing master-node signatures. The effective size of reachability graphs is regulated by the size of a master node's domain: the larger the domain, the fewer the master nodes in a reachability graph.

The following experiments demonstrate how absolute cost and parallel width are affected by the size of a reachability graph and the size of master domains. Because dynamic reachability analysis is also affected by the shape of reachability graphs, it is important to give consideration to the data-structures that we used for these experiments. The micro-benchmark that we implemented builds a binary space partitioning (BSP) tree of depth d. BSP trees are commonly used data-structures in computer graphics algorithms and are generated by continuously bisecting a space and creating nodes to represent each resulting bisection. It is also common for the leafs of a BSP tree to store pointers to all the objects (e.g. game entities or polygons) that are located in the space represented by each leaf. Therefore each leaf also contains a linked list of objects (in our case game entities). If the spaces represented by leafs are small enough, each leaf will likely point to one or zero objects. The entities pointed to by leafs are also stored in a global linked list and each entity contains a list of "items".

To simulate the assignment of entities to partitions represented by the leafs of a BSP tree, the producer sends out $(leaf, entity)$ pairs and the consumer performs the associated link assignment, along with synthetic work. The $(leaf, entity)$ pairs sent by the producer ensure that each entity is assigned to a unique leaf. In this case, no synchronization is actually required to protect the assignment of the entity to the leaf.

Using this micro-benchmark, we perform three experiments, which respectively demonstrate how absolute cost, parallel width, and overall overhead varies as the number of memory objects, and the size of domains change. In all experiments, we demonstrate results for approximately 20,000 ($d = 10$, $entities = 1000$) and 40,000 ($d = 11$, $entities = 2000$) total memory objects.

Absolute Cost For absolute cost, we measured the time it takes a consumer to perform a link assignment under varying domain sizes. The results are shown in figure 6, with the cost in microseconds on the y-axis and domain sizes (maximum number of objects per domain) on the x-axis. Figure 6 demonstrates that as domain sizes increase, the cost decreases from about 7.8-4.7μs and 8.5-4.9μs for 20,000 and 40,000 objects respectively. There is also a slight overall increase (4%-7%) in cost going from 20,000 objects to 40,000 objects. Therefore, domain size appears to have a more significant affect on cost than the size of reachability graphs.

Note that it is important to put the absolute cost of dynamic reachability analysis into perspective. For example, acquiring a mutex lock (that does not actually protect any code) can take anywhere from a hundred cycles to as much as 20 microseconds, depending on the level of contention. The cost of dynamic reachability analysis (and SvS in general) is not affected by the amount of contention/sharing in an application. Also, although way are paying a cost during link assignment, SvS does not pay the cost of conflict resolution paid by other techniques such as TM. In the following sections we demonstrate, with real applications, that the benefits produced by SvS are outweighed by its costs.

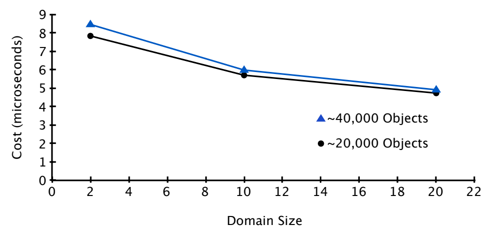


Figure 6. Cost of link assignment under varying domain and reachability graph sizes

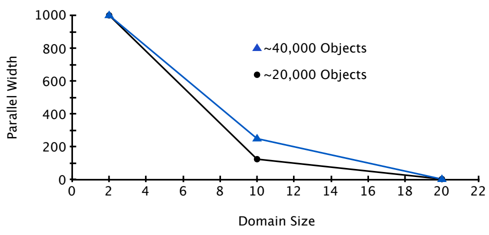


Figure 7. Parallel width under varying domain and reachability graph sizes

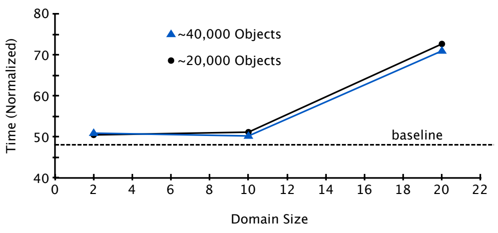


Figure 8. Overall run-time overhead (normalized) of the consumer for varying domain and reachability graph sizes

Parallel Width Figure 7 demonstrates the change in parallel width (y-axis) as we increase domain sizes (x-axis). We used a signature size of 8192 and measured the parallel width as described in section 3.1.1. Here we see that parallel width is dramatically affected by the size of master domains. As master domains increase, more memory objects share the same signature and master nodes decrease. As the number of master nodes decrease, their respective reachability increases, thus increasing the chances of conflict between the reachability of master nodes. This accounts for the dramatic decreases in parallel width demonstrated by both curves in figure 7. As in the previous section, the size of the reachability graph does not appear to have a significant affect on parallel width.

Overall Overhead For this experiment, we again used a signature size of 8192. Also, because no sharing actually occurs, we can compare the run-times of our system with SvS enabled and disabled in order to get a worst case scenario overhead for SvS. This overhead not only includes the cost of false positives and link assignment, but also any costs associated with refinement and scheduling, thus providing an overall worst-case cost of dynamic analysis performed at run-time.

Figure 8 demonstrates the overall run-time (y-axis, normalized to the number of $(leaf, entity)$ pairs sent by the producer) for different domain sizes (x-axis). We see that there is essentially no change between domain sizes 2 and 10, since the decrease in absolute cost is negated by the decrease in parallelism when domain

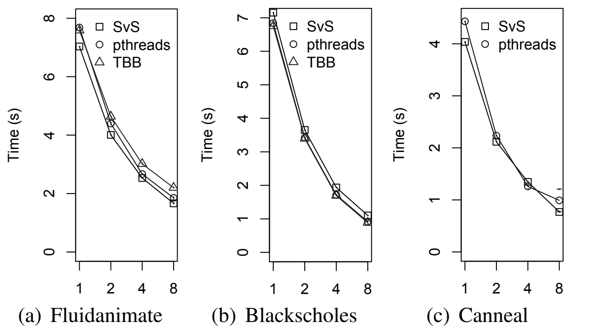


Figure 9. Performance and scalability of the PARSEC benchmarks.

sizes increase. After a domain size of 10, we see a sharp increase due to the sharp decrease in parallelism that figure 7 demonstrated. The dotted-line labeled "baseline" is the run-time of the benchmark when SvS is disabled. This means that the overall overhead of SvS, for a domain size of 2, is about 5% for 20,000 objects and about 6% for 40,000 objects.

3.1.3 Discussion

One crucial characteristic of SvS is that its overhead is not dependent on the amount of sharing in the system. Rather, it depends on a few internal parameters and, more predominantly, the size and shape of data-structures and their resulting reachability graphs. This is fundamentally different from existing techniques where performance decreases as the amount of sharing increases (e.g. contention over shared locks, cost of transaction aborts). This is not the case for SvS. In fact, since SvS knows the memory accesses of tasks before they execute, in can mitigate sharing conflicts by grouping together non-conflicting tasks. This is what generations accomplishes by using look-ahead. This is an important distinction between SvS and existing techniques. We demonstrate in the next sections that this distinction leads to SvS being able to perform as well as, or better than several existing techniques, with the added benefit that it performs shared state protection *automatically*.

3.2 PARSEC

PARSEC is a parallel benchmark suite designed to represent state-of-the-art parallel workloads [10]. While the majority of these benchmarks do not need SvS, we converted Fluidanimate and Canneal which do have shared state conflicts. We also converted Blackscholes, which has no conflicts, in order to show the performance of SvS even when not required.

Blackscholes is a benchmark from the financial domain which calculates prices for stock options. Option prices can be calculated independently from one another and the results are stored in an array. SvS is used to protect the array from having the same array slot written to simultaneously.

Fluidanimate, as the name implies, performs simulation of fluid. The existing parallel implementation divides the 3D space of fluid cells into partitions. During an update, a cell only needs to modify the values of adjacent cells. Therefore the internal nodes of a partition can be processed without any synchronization. Cells on a common border, ghost cells, require locking before being modified. In the SvS implementation, ghost cells are protected from race conditions automatically. For the SvS version we used the number of partitions that is much larger than the number of cores and SvS is then able to process partitions that do not share ghost cells in parallel.

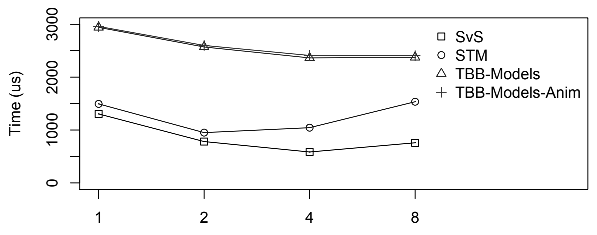


Figure 10. Scalability comparison between SvS, TBB and STM implementations of character animation.

The Canneal benchmark is a place-and-route simulation that uses simulated annealing to minimize the routing cost on the chip. The algorithm iteratively finds a better routing by picking two elements at random and then swapping them if this is determined to be beneficial. The third-party TBB implementation for this benchmark was not available. The pthread implementation uses a construct called an *atomic pointer* in order to swap two elements, relying on compare-and-swap (CAS) operations to ensure atomicity. The implementation purposefully allows for data races to occur [10]. However, the algorithm is designed to recover from those race conditions. We replace the use of atomic pointers with SvS in order to provide a safe way of swapping the elements in parallel. To accomplish this SvS applied to pairs of elements is used to automatically determine which swaps can safely execute.

We report runtimes calculated over an average of five runs using the *simlarge* dataset. Times reported for Canneal and Blackscholes are from the parallel section of the code. Fluidanimate has parallel sections throughout and so the total execution time is reported. Standard error was negligible in all cases except the Canneal pthread version with eight threads where it was 22%, which we believe to be caused by unpredictable latency of CAS's.

In Figure 9 we show the performance with different number of threads for SvS and the third-party pthreads and TBB implementations. In the pthreads and TBB implementation fine-grained mutexes are used to provide synchronization unless otherwise noted earlier. The results demonstrate that SvS is able to match performance of the pthreads[1] and TBB even though it does not require the programmer to explicitly protect access to shared state. This suggests that SvS may accomplish similar performance as other models with less programming effort although user studies would be needed to confirm this statement.

3.3 Cal3D

The Cal3D library implements a typical character animation algorithm. Shown in Figure 11, the algorithm iterates through all character models, blending several animations on each. Animations are blended by iterating through a model's bones and modifying a bone's position and rotation according to the current state of the animation. Animations typically modify some bones of a model, but not all of them. For example, an animation of a running motion updates the positions and rotations of bones of the legs and the arms, but not the chest bones. A waving animation updates the bones of one arm.

In order to correctly parallelize animation, two animations must not update the same bone concurrently. Different models do not share bones, so the iterations of the first loop in Figure 11 can run in parallel. However, because different animations may touch the same bone, the second loop cannot be parallelized without protecting against concurrent accesses. Thus, there are four ways to parallelize character animation: restrict parallelism to models, or process models and animations in parallel and protect accesses to bones with locks, transactional memory or SvS.

[1] SvS performs *better* than pthreads in Canneal, because the pthreads implementation of Canneal is limited by the heavy use of CAS's. SvS avoids this bottleneck by only allowing the elements that can be swapped safely without synchronization to be processed in parallel.

```
foreach(model in modelList){
  foreach(animation in model.animList){
    animation->calculateBonePositions(timeDelta);
    foreach(bone in animation.bones){
      skeleton.bones[bone.ID].blend(bone)
    }
  }
}
```

Figure 11. The character animation algorithm using a conventional loop notation.

Figure 10 compares performance and scalability of four parallel implementations of the main animation loop in Cal3D. To drive the loop we use the Cally animation example included in the Cal3D distribution, and we use 4 models and 8 animations per model. These numbers reveal several interesting facts.

TBB-Models processes models in parallel, and since we are processing 4 models, reaches maximum performance at 4 threads. TBB-ModelsAnims parallelizes processing models and animations, protecting accesses to bones with locks. Despite extra parallelism, TBB-ModelsAnims performs similarly to TBB-Models due to high lock contention over shared bones.

For the STM version, we used our parallel runtime system to create a transactional task for each model, animation and bone combination. We present the best performing STM algorithm. Since each transaction is a guaranteed write, it seems that STM performance suffers due to a high conflict rate between transactions writing to the same bone.

To use SvS, we created tasks as in STM, but each task is scheduled using SvS. SvS achieves better performance than other implementations due to the greater parallelism uncovered. Despite many potential conflicts, demonstrated in TBB and STM, the large number of runnable items available to be scheduled allows SvS to achieve good parallelism and performance.

Performance for SvS and STM stops improving when we have more than four threads. The reason has nothing to do with the method of synchronization, but is rooted in the very fine-granular nature of tasks in this example. Each task only takes 3500 cycles to complete, and the overhead of work-stealing dominates the computation. We found that if we implement a semi-static version of the scheduling algorithm that places tasks into thread-local queues and restricts work-stealing we are able to achieve scaling beyond four threads and improve performance in the eight-threaded case by more than a factor of three (these results are not shown).

This suggests an important direction for future research: investigation of semi-static scheduling techniques (in contrast to traditional work-stealing) in order to accomplish good performance for systems with very fine-granular parallelism, or automatically determining the right task granularity in order to minimize the overhead of handling fine-grained tasks. Although there are hardware proposals aiming to reduce the overhead of task scheduling [24], we believe that maximum efficiency can be obtained when software is also structured to avoid the overhead.

3.4 QuakeSquad

Artificial Intelligence (AI), determining the actions of game entities, and Entity Management, managing the movements and interactions of game objects, together make up one common game subsystem and are notoriously difficult to parallelize [5]. This difficulty has two main sources. First, AI logic tends to be arbitrary and complex being defined by game programmers to fit the circumstances instead of fitting some mathematical formalism. Secondly, the large number of interactions involved in Entity Management mean that several modifications may be made to a single object in one frame. These interactions can often set off chains of interactions that cluster in unforeseen ways. These two complicating factors, and the large amount of shared state that can potentially affected, make this system a primary concern for parallelization.

We took the approach of Lupei et al [18] with their SynQuake benchmark and created an application, QuakeSquad, that captures the essential computational patterns and data structures of video games while remaining simple enough for meaningful testing.

QuakeSquad, consists of a two dimensional world with four types of entities: bombs, walls, citizens and techs. These are governed by a few simple rules:

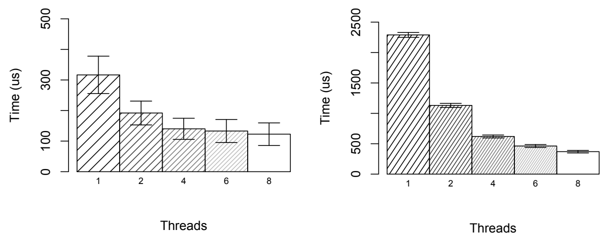
- bombs explode reducing the health of citizens and techs within a set radius and not obstructed by a wall.
- bombs 'project' fear onto citizens and technicians who are within a set distance and in the line of sight of the bomb.
- fearful citizens will move away from the closest source of fear while a calm citizen will move randomly.
- calm citizens will not move into an area where it would be subject to fear.
- techs will move toward the closet source of fear and if the tech touches a bomb it is disarmed.

With a large number of entities in the system, the 'line-of-sight' tests for occlusion are by far the most expensive. Without further optimization each test would have to consider every entity in the world. To reduce these tests, the world is divided into a grid where each cell is associated with an unordered list of every entity in that area. When an entity moves from one cell to another it will remove itself from its current list and add itself to the new one. The cell size is set such that when making a line-of-sight calculation only the current cell and adjacent cells need be considered. This division of entities, analogous to the spatial partitioning structures used in 3D environments, reduces the number of tests by at least two orders of magnitude. However, even with this optimization, occlusion testing still dominates the computation and so would benefit most from parallelization. These tests occur most frequently when citizens move and when bombs project fear onto citizens and technicians. The same tests also occur when a bomb explodes, but bombs explode infrequently and so we focus on parallelizaton of citizen movement and fear projection.

When these aspects are transformed into data parallel operations where entities are concurrently updated, potential shared state conflicts are exposed. During bomb updates it is common for bomb radii to overlap and they may modify the same entity simultaneously. During citizen movement a large number of citizens cross grid boundaries and thus expose the associated entity lists to potential concurrent modifications. In both cases SvS can be used to ensure that no race conditions occur, which we detail below. We perform each test with 35 bombs, 100 citizens, 40 techs and 120 walls. 2048 bit signatures are used for SvS.

3.4.1 Sending lists (Updating Bombs)

First, we focus on updating the bomb entities and determining which human entities are affected. The runtime of this operation is completely dominated by line-of-sight checks. First a task determines the entities in each bomb's radius and builds temporary linked lists of potential techs and citizens to scare. These lists of potential candidates are then sent to a data-parallel consumer which perform line-of-sight testing. If an entity is not occluded, the entity become fearful. The reachability of the underlying list is queried to return a signature representing its contents, which is passed to the scheduler. Figure 12(a) shows the execution times averaged over 100 frames. With one thread, the processing takes 1944 μs. At 8 threads the execution time drops to 675 μs. The standard deviation in all cases is bellow 100 μs



(a) Bomb Update (b) Citizen Movement

Figure 12. Scalability of the two QuakeSquad phases.

3.4.2 Modifying lists (Updating Citizens)

When a citizen moves, it will avoid moving into areas with bombs. This update is also dominated by line of sight checks. A producer task determines the potential new location for each citizen and then sends this to a data-parallel consumer which will perform a line of sight check and move the entity if no bomb is visible. This case is complicated by the grid of entity lists. When an entity moves from one grid to another, a reference is removed from one list and added to another. If two entities move into the same grid simultaneously the linked lists will be subject to concurrent modification. Additionally, the line of sight checks require reading the eight adjacent cells around current location cell and the eight cells adjacent to the prospective destination. Errors will occur if the structure of one of these lists is modified while it is being read. We use SvS to prevent these potential state access conflicts. Again, the reachability of the list queried to produce a signature for the scheduler. Figure 12(b) shows the execution times of this phase averaged over 100 frames. Executions times go from 4617 μs with one thread to 866 μs with eight. The standard deviation in all cases is bellow 90 μs.

3.4.3 Putting it together

The previous discussion has shown that both of these major tasks scale well in isolation. We now consider results for entire frames of QuakeSquad, which combines modifying lists and sending lists. For comparison we created a version using TBB with mutexes and another using STM. The results, averaged over 100 frames, are shown in figure 13. Scaling from one to eight threads in the SvS version reduces the frame execution time from 7633 μs to 1442 μs. Shared state accesses conflict approximately 10% of the time on average, meaning SvS detects and manages a conflict 1 in every 10 accesses. While the TBB version performs similarly to the SvS version, the STM version fails to benefit from extra threads. A closer examination showed that roll-backs were causing the data-parallel instances to lengthen and increase total runtime.

QuakeSquad is a comprehensive example representing a previously difficult to parallelize subsystem of modern game engines. The performance and scalability achieved by SvS in the results demonstrate its ability to utilize reachability graphs and dynamic reachability analysis to efficiently determine the reads/writes of tasks that access linked data structures and subsequently concurrently schedule tasks with non-overlapping read/write sets.

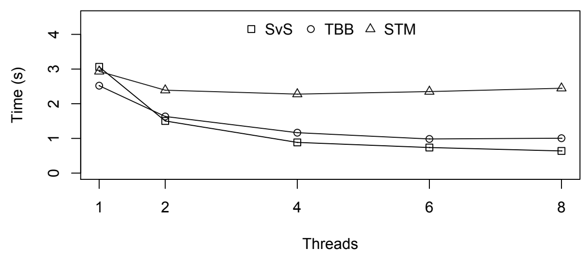


Figure 13. Scalability comparison between SvS, TBB and STM versions of QuakeSquad.

4. Related Work

SvS was previously introduced by us in a short workshop paper, which gave only a high-level overview of the idea, but the system was not fully specified or implemented at that time. This paper contains the first, self-contained, presentation of the model and implementation of SvS as well as detailed specification of algorithms, and evaluation with multiple benchmarks.

There is a great need in the video game industry for domain appropriate parallelization techniques. Developers for major games studios such as EA [1], Epic [5] and Valve [26] have expressed the need for comprehensive and efficient parallelism and have cited shared state management as a major roadblock.

There are an ever increasing number of parallel environments and language/runtime combinations such as Chapel [13], Cilk [12], OpenMP [4], Gossamer [23] and Intel's TBB [6] and Ct\RapidMind [3]. However, they don't provide automatic mechanisms for shared state protection, generally focusing instead on providing tools for the programmer to manually manage state. SvS or an SvS-like technique could be implemented in a number of these systems.

The Jade [22] language and the Prometheus [7] package both address shared state protection. Jade proposes a set of parallel extensions to C where a programmer denotes blocks of code as tasks and specifies their data constraints. Although Jade also schedules tasks based on their constraints there are fundamental differences. Jade is based around task-parallelism and constraints must be specified by the programmer whereas in SvS they are derived automatically, thus freeing the programmer from the need to concentrate on implicit and hard to spot data dependencies. A task's scheduling 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 using SvS is more general.

While there is a large body of existing work on static dependency analysis, OoOJava [16] represents recent work in this field that similar to SvS attempts to combine static and dynamic analysis. OoOJava abstracts collections of objects as heap region nodes and uses disjoint reachability analysis [15] to statically infer connectivity between objects. The result is a set of reachability states that are used to determine if two objects x and y are disjoint i.e. cannot reference the same heap node. If it is determined that they might reach the same heap node, in very specific cases they are able to check at run-time if $x = y$ in order to test for disjointness. Otherwise, they are forced to conservatively assume a dependency between x and y since they do not have full reachability information at compile time. SvS addresses this issue by introducing the concepts of reachability and reachability graphs and using dynamic reachability analysis to provide an efficient way to maintain and extract complete reachability information.

Many techniques that address shared state are optimistic in that they attempt to do computations without explicit synchronization and 'roll-back' or undo conflicting operations. Software transactional memory (STM) is the most prominent of these techniques and provides database-like transactional atomicity. A programmer wraps code that requires protection in an atomic block and the STM system automatically handles conflicts. The key difference between TM and SvS is that SvS determines whether or not two tasks might conflict before they are executed, whereas TM detects conflicts during execution. This means that TM is less conservative but may be subject to expensive rollbacks. Since rollback cost are high, STM performs well when most transactions are able to complete successfully. So STM may be advantageous to SvS in cases where actual conflicts between tasks are extremely rare, but SvS would serialize them to avoid potential races. This suggests an interesting opportunity for *combining* STM and SvS: using STM when actual conflicts are rare and using SvS when the conflicts are frequent. Such adaptive use of synchronization primitives may enable to exploit the best of both models and is an interesting direction for future work.

There has also been some work in the STM community on deliberately co-scheduling transactions that appear (based on static or dynamic information) to be unlikely to conflict with one another [27]. However, this work uses the history of previous conflicts, and perform co-scheduling for performance. SvS performs scheduling for correctness. SvS also relies on static and dynamic analysis to determine the potential memory accesses of a task before it executes, rather than conflict history recorded after the fact.

Galois [17] is an optimistic framework, focusing on data-parallelism, that falls outside of STM. Galois focuses on the parallelization of 'irregular applications', those with interdependent loop iterations. This framework differs from SvS in that it focusses on commutativity analysis as opposed to dependency analysis and is optimistic and must contend with overhead created by roll-backs.

5. Conclusion and Future Work

We presented SvS – a new framework for automatic protection of shared state in task graph models. We demonstrated that SvS performs comparably to other synchronization techniques, without requiring the programmer to explicitly manage shared state.

While this work demonstrates the feasibility of SvS, there are many opportunities for further research. First of all, there is an opportunity to incorporate more powerful static analysis, including disjoint reachability analysis [15]. In doing so, we may be able to extract greater parallelism statically, which will reduce the complexity and overhead of the algorithms that we use at runtime. Second, as we discovered in the character animation example, there are opportunities for investigating better software techniques for handling fine-grained tasks, including scheduling and determining the optimal task size. Finally, we are interested in expanding the scope of SvS codebase and performing user studies in order to fully understand the impact of SvS on programmer productivity.

References

[1] Parallel futures of a game engine `http://publications.dice.se/attachments/Sthlm10_ParallelFutures_Final.ppt`.

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

[3] Intel ct `http://software.intel.com/en-us/data-parallel/`.

[4] The openmp specification for parallel programming `http://www.openmp.org`.

[5] The quest for more processing power:part two: Multi-core and multi-threaded gaming `http://www.anandtech.com/show/1645/3`,.

[6] Thread building blocks `www.threadingbuildingblocks.org`.

[7] M. D. Allen et al. Serialization sets: a dynamic dependence-based parallel execution model. In *PPoPP '09*, pages 85–96. ACM, 2009. ISBN 978-1-60558-397-6. doi: http://doi.acm.org/10.1145/1504176.1504190.

[8] L. O. Andersen. Program analysis and specialization for the c programming language. Technical report, 1994.

[9] U. K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988. ISBN 0898382890.

[10] C. Bienia et al. The parsec benchmark suite: characterization and architectural implications. PACT '08, 2008.

[11] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/362686.362692.

[12] R. D. Blumofe et al. Cilk: An efficient multithreaded runtime system. In *J. of Parallel and Dist. Comp.*, pages 207–216, 1995. doi: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.3175.

[13] B. L. Chamberlain et al. Parallel programmability and the chapel language. *Intl. J. of High Perf. Computing Applications*, 21:291–312, 2007.

[14] P. Felber et al. Transactifying applications using an open compiler framework. In *TRANSACT*, August 2007.

[15] J. Jenista et al. Disjointness analysis for java-like languages. *Technical Report UCI- ISR-09-1*, 2009.

[16] J. C. Jenista et al. Ooojava: An out-of-order approach to parallel programming. In *HotPar '10*, 2010.

[17] M. Kulkarni et al. Scheduling strategies for optimistic parallel execution of irregular programs. In *SPAA '08*, pages 217–228. ACM, 2008. ISBN 978-1-59593-973-9. doi: http://doi.acm.org/10.1145/1378533.1378575.

[18] D. Lupei et al. Transactional memory support for scalable and transparent parallelization of multiplayer games. EuroSys '10, 2010.

[19] M. Marron et al. Sharing analysis of arrays, collections, and recursive structures. In *PASTE '08*, pages 43–49. ACM, 2008. ISBN 978-1-60558-382-2. doi: http://doi.acm.org/10.1145/1512475.1512485.

[20] M. D. McCool. Structured parallel programming with deterministic patterns. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1863086.1863091`.

[21] W. Pugh et al. Constraint-based array dependence analysis. *ACM Trans. Program. Lang. Syst.*, pages 635–678, 1998. ISSN 0164-0925.

[22] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of jade. *ACM Trans. Program. Lang. Syst.*, 20(3):483–545, 1998. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/291889.291893.

[23] Roback et al. Gossamer: A lightweight programming framework for multicore machines. In *HotPar '10*, 2010. URL `http://www.usenix.org/event/hotpar10/tech/full_papers/Roback.pdf`.

[24] D. Sanchez et al. Flexible architectural support for fine-grain scheduling. *SIGARCH Comput. Archit. News*, 38(1):311–322, 2010. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/1735970.1736055.

[25] N. Shavit et al. Software transactional memory. In *PODC '95*, pages 204–213. ACM, 1995. ISBN 0-89791-710-3. doi: http://doi.acm.org/10.1145/224964.224987.

[26] Valve. Dragged kicking and screaming: Source multicore. In *Game Developers Conference*, 2007. URL `http://www.valvesoftware.com/publications/2007/GDC2007_SourceMulticore.pdf`.

[27] R. M. Yoo and H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 169–178, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-973-9. doi: http://doi.acm.org/10.1145/1378533.1378564. URL `http://doi.acm.org/10.1145/1378533.1378564`.