SYNCHRONIZATION VIA SCHEDULING: TECHNIQUES FOR EFFICIENTLY MANAGING CONCURRENT SHARED MEMORY ACCESSES

by

Shane Mottishaw
B.Sc., Simon Fraser University, 2009

a Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in the
School of Computing Science
Faculty of Applied Sciences

APPROVAL

Name:	Shane Mottishaw
Degree:	Master of Science
Title of Thesis:	Synchronization Via Scheduling: Techniques for Efficiently Managing Concurrent Shared Memory Accesses
Examining Committee:	Dr. Hao Zhang Chair
	Dr. Alexandra Fedorova, Professor, Computing Science Simon Fraser University Senior Supervisor
	Dr. Ghassan Hamarneh, Professor, Computing Science Simon Fraser University Supervisor
	Dr. Arrvindh Shriraman, Professor, Computing Science Simon Fraser University SFU Examiner
Date Approved:	July 29, 2011

Abstract

The recent shift to multi-core computing has meant more programmers are required to write parallel programs. This is a daunting task even for experienced programmers due to the complexity and challenges involved in parallel programming. Key among these issues is managing concurrent accesses to shared memory. Unfortunately, most existing approaches rely on synchronization primitives that leave the complexity of protecting shared memory to the programmer and those that have attempted to automatically protect shared memory have achieved limited success.

To address the issue of shared memory management while mitigating the limitations of existing approaches, we introduce a new technique called Synchronization via Scheduling (SvS). SvS provides efficient and automatic protection of shared memory by combining static and dynamic analysis to determine the set of possible memory accesses a block of code makes before it is executed and schedules these blocks such that no two blocks concurrently access the same memory.

Acknowledgments

I would like to thank my Senior Supervisor, Dr. Alexandra (Sasha) Fedorova, for her support, guidance and leadership. I experience her as someone who is extremely motivated, kind, and trusting, which are qualities I hold in high esteem and am inspired to build within myself. Her focus and results-orientated approach helped keep me on track in my research endeavours and has encouraged growth in not only myself, but all of her students. It is a privilege to have worked with and learned from someone as successful and experienced in her field as Sasha.

I would also like to acknowledge the help and support of my colleagues Micah Best, Craig Mustard, and Mark Roth. In addition to their collaboration efforts in the conceptualization and actualization of the work in this thesis, I have learned a great deal from their individual strengths and skills and am happy to have fostered a friendship with them.

Finally, I would like to thank my family and my fiancée for their support, interest, and encouragement in my academic pursuits – especially for the times when I would virtually disappear before a paper deadline and upon my re-emergence be greeted by their congratulations.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Locating and protecting concurrent accesses (i.e. reads and writes) to shared memory is a key challenge faced by any parallel programmer. Conflicting accesses occur when two processors request access to the same memory location concurrently where at least one access is a write. Such conflicting accesses are a prominent source of error causing race conditions and data corruption. These bugs are difficult to track down as they may only manifest under certain inter-leavings of multiple instruction streams. Unfortunately, most existing tools and languages lack support for protecting shared memory and those that do attempt automatic shared memory management suffer from limited parallelism or impractical overheads.

In lieu of providing mechanisms for automatic protection of shared memory, many existing parallel languages (e.g. Cilk [18, 21]) and runtime libraries (e.g. Intel Threading Building Blocks [34, 2]) focus on hiding the details of dispatching code for parallel execution by providing abstractions for expressing parallelism. In order to manage shared memory accesses in these systems, the programmer is required to use synchronization primitives provided by the system or another library (e.g. pthreads). These primitives not only rely on the programmer correctly identifying all *potential* accesses to shared memory (which is made difficult when memory is accessed through arbitrary pointer dereferencing), but also organizing the use of these primitives in a way that avoids deadlock and high communication overheads.

Traditionally, there have been two prominent approaches to automatic shared memory management: static analysis and speculative parallelism. Solutions in the static analysis category consist primarily of compiler techniques for automatic parallelization. These techniques use code analysis (such as data dependence analysis [9], array dependence analysis

[10], shape analysis [37, 30] and points-to analysis [8, 39, 11, 40]) to identify regions of code (often loop bodies) that can be safely executed in parallel. Due to the limited information available at compile time, these techniques often make conservative assumptions when two regions of code *may* access the same memory. In these cases, parallelizing compilers are forced to serialize these blocks of code, thus reducing parallelism. For example, if loop iterations write to an array and the location of each write is determined by input data at runtime, then the compiler may assume that each access could potentially write to the same location in the array and thus all loop iterations would be serialized, even though this may not be necessary. In speculative parallelism, regions of code are *optimistically* executed in parallel and if a violation is later detected (e.g. concurrent access to shared memory) then the offending operation is *squashed* and restarted. Software transactional memory (STM) is the canonical realization of speculative parallelism and has been heavily researched [38, 22, 23]. In STM, programmers annotate regions of code as being transactions. At run-time transactions are speculatively executed in parallel and the system records the reads and writes of transactions. After transactions have completed execution, their read/write sets are compared and if the read/write sets of two transactions overlap, then one of these transactions must abort, rollback its changes, and try again. The overhead costs of these transaction aborts remain a limiting factor that has prevented the widespread adoption of STM.

To address the issue of shared memory management while mitigating the limitations and inefficiencies of existing approaches, we introduce a new technique called Synchronization via Scheduling (SvS). SvS provides efficient and automatic protection of concurrent accesses to shared memory by combining static and dynamic analysis to determine the set of possible memory accesses a block of code makes *before* it is executed and schedules these blocks such that no two blocks concurrently access the same memory.

To achieve this, SvS structures programs using a standard task graph model. In task graph models, code is divided into discrete units of execution (tasks) and directed edges (dependencies) are placed between tasks in order to enforce a specific ordering or prevent two tasks from running concurrently. SvS uses static code analysis to determine whether two tasks *may* touch the same memory, and if they do, a dependency is placed between them. The resulting tasks and dependencies define a static schedule (task graph) which ensures a correct parallel execution of tasks. However, due to the limitations of static analysis, some dependencies may have been created between two tasks that don't actually access the same memory at run-time. To address this issue, we introduce a new dynamic analysis technique called *dynamic reachability analysis*. Dynamic reachability analysis monitors connectivity/reachability properties of memory objects (our online abstraction for regions of heap memory). Connectivity is represented as a graph where memory objects are nodes and references between memory objects are edges. Reachability refers to the set of memory objects that can possibly be reached from a memory object M by traversing heap references (i.e. edges). The results of dynamic reachability analysis can then be leveraged to determine more precise read/write sets for tasks that access memory objects. When a task is considered for scheduling, it compares its read/write set with its dependent children and, if there is no overlap, unnecessary dependencies are removed, potentially increasing parallelism. We call the process of removing dependencies *dynamic refinement*. Finally, in specific cases where dynamic refinement is not able to calculate read/write sets for tasks (e.g. when a task's accesses are input-dependent and the task is waiting for input), the checking of read/write sets is deferred to the scheduler. It is then the job of the scheduler to efficiently dispatch tasks such that no two tasks have overlapping read/write sets.

In summary, this work describes the following contributions:

- A new dynamic analysis technique called *dynamic reachability analysis*
- Scheduling algorithms and efficient implementations for dispatching tasks with non-overlapping read/write sets
- The SvS framework for automatic shared memory management and an efficient implementation that is demonstrated to work in practical applications

An incomplete implementation and shortened description of this work was presented in [15]; here we review the SvS framework in its entirety and augment its implementation. The remainder of this thesis is organized as follows. In chapter 2, we discuss the SvS framework, followed by a description of the algorithms and implementations for each component of this framework in chapter 3 to 6. In chapter 7 we provide experimental results for several benchmarks and applications. In chapter 8 we discuss related work and in chapter 9 we conclude and provide a discussion of future work.

Chapter 2

SvS Framework

2.1 Motivation and Overview

Many existing parallel runtime environments [2, 18, 28] employ task based parallel programming models. In these models, users express their programs as a collection of *tasks* (function-like, basic units of work) and the runtime environment orchestrates the creation, scheduling, and execution of tasks in order to express parallelism. As described in the previous section, a prevailing problem with most implementations of this model is the lack of automatic shared memory management between tasks. To further motivate this problem, consider the example of character animation. In order to produce a character that is transitioning from walking to running, we need to blend the "walking" and "running" animations. In order to blend two animations, each animation performs mathematical transformations to the "bones" of the character. While these transformations are commutative, it is unsafe for two animations to be applied to the same character concurrently, because two animations *may* apply a transformation to the same bone.

Without automatic shared memory management, programmers must protected accesses to bones using the synchronization primitives available to them in order to enforce mutually exclusive access to individual bones. A common method for achieving this in task based models, besides using traditional lock based primitives, is through task graphs [31]. In a task graph, *dependencies* are placed between two tasks A and B that may access the same state, or require a specific ordering. A dependency (A, B) means that A must complete before B can begin. However, protecting shared memory using dependencies has two problems. First, like other synchronization methods, dependencies are prone to programmer error, especially when dealing with pointer-based memory accesses. To address the issue of programmer error, static dependency analysis could be performed to automatically place dependencies between tasks. While this will guarantee the protection of shared state, it does not address the second problem: dependencies constrain parallelism in cases where tasks *may* access the same memory, but do not *actually* access the same memory at run-time.

We address the issue of protecting shared memory 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 memory. The goal of static analysis is to detect dependencies between tasks in order to generate a task graph that guarantees the protection of shared state. To address the limitations of static analysis, SvS performs dynamic analysis at run-time to more precisely determine the read/write sets of tasks and uses these read/write sets to potentially remove unnecessary dependencies and, if necessary, to enable the scheduler to dispatch tasks with non-overlapping memory accesses. In the next section, we describe how the SvS framework integrates the mechanisms involved during static analysis, dynamic analysis, and scheduling.

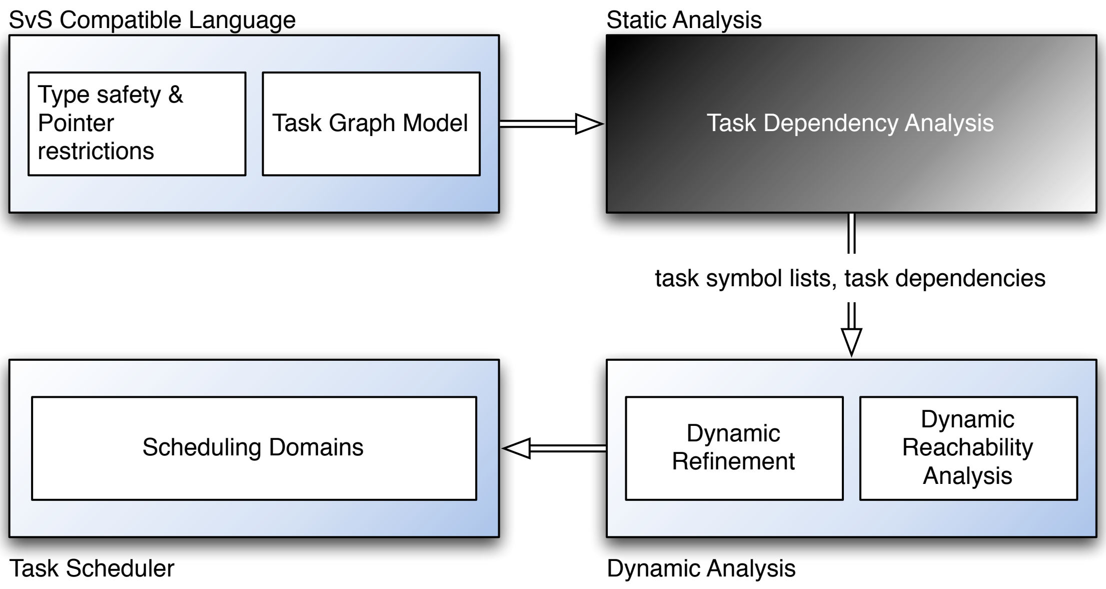


Figure 2.1: SvS Framework

2.2 Framework

Figure 2.1 shows the four main components of the SvS framework: an SvS compatible language, static analysis, dynamic analysis, and the task scheduler. An SvS compatible language enables a programmer to decompose a program into tasks: blocks of code representing units of work. Tasks in this language are assumed to be commutative, unless the programmer imposes an ordering constraint between tasks. The programmer does this by annotating a task B as having an *explicit dependency* with task A in that A must complete before B can begin. Beyond providing a task-graph abstraction, an SvS compatible language also provides type safety in that all accesses to memory (e.g. through pointers and variables) are accesses to SvS's abstraction for regions of memory called *memory objects*. This is analogous to how all object types in Java inherit from the Object class. Additionally, restrictions on pointer arithmetic is required such the result must always point to a memory object. Although not a strict requirement for SvS, our implementation of an SvS compatible language completely disallows pointer arithmetic in order to greatly simplify static code analysis without imposing any burden on the programmer. We describe our prototype SvS language in chapter 3.

A program written in an SvS compatible language is passed through a static code analysis phase that collects symbols (linguistic abstractions for memory access) that are accessed by tasks and creates dependencies between tasks that *may* access the same memory. Dependencies inserted during static analysis are called *implicit dependencies*. These dependencies, when combined with programmer specified explicit dependencies, define a static task graph which provides an initial schedule that ensures that no two tasks will concurrently access the same memory. The symbols collected by this phase are also used to generate task specific functions that are called during dynamic analysis and scheduling in order to obtain more precise read/write sets for tasks.

We purposely visualize the static analysis phase in figure 2.1 as a "black box" because SvS is indifferent to the techniques used to implement static analysis. Any implementation that extracts the list of symbols each task may access and produces dependencies that ensure the protection of shared memory are sufficient for use in the SvS framework. For example, a correct, but naive, static analysis implementation could simply place a dependency between all tasks, leaving all extraction of parallelism to be performed by dynamic analysis. Static analysis in SvS is formally defined as *task dependency analysis* in chapter 4. This chapter also describes how task dependency analysis can be implemented using existing techniques and provides details on our implementation of one of these techniques. The classic limiting factor of static analysis when applied to parallelization is that it is limited to compile time information. As a result, static analysis is sometimes forced to make conservative assumptions and place dependencies between tasks that are potentially unnecessary, thus restricting parallelism. This problem can be alleviated using dynamic analysis.

We introduce a new dynamic analysis technique called *dynamic reachability analysis*. Dynamic reachability analysis abstracts regions of heap memory as memory objects and during task execution it monitors changes to the connectivity properties of memory objects accessed by tasks. The result of dynamic reachability analysis is that a memory object M can be queried to return a set of all the memory objects it can potentially access (i.e. reach) through arbitrary dereferencing of M's "links" to other memory objects.

As tasks are considered for scheduling, the results of dynamic reachability analysis are used to generate and compare more precise read/write sets for tasks in order to potentially remove implicit dependencies that were created during static analysis, but are found to be unnecessary when information from dynamic reachability analysis becomes available at run-time. We call this process of removing implicit dependencies *dynamic refinement*.

Finally, the general role of the task scheduler is to ensure that concurrently executing tasks have non-overlapping read/write sets, in effect performing synchronization via scheduling. In most cases, this is achieved by simply executing tasks whose dependents have already completed. However, there are cases where the comparison of read/write sets that is normally performed during dynamic refinement must be deferred onto the task scheduler. In these cases the task scheduler is responsible for efficiently comparing the read/write sets of (potentially many) tasks in order to dispatch tasks with non-overlapping read/write sets. The method for orchestrating this process is called *scheduling domains*.

While the SvS framework is one of the contributions of this work, its implementation relies on both new and existing techniques. Chapters 3 to 6 will discuss how these techniques are integrated in order to implement the SvS framework.

Chapter 3

SvS Language

As described in chapter 2, a requirement of SvS is a language that provides a few restrictions to enable static and dynamic analysis and allows the programmer to write a program as a collection of tasks. In the following sections, we provide details on our implementation of an SvS compatible language and the underlying task graph model assumed by SvS.

3.1 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, a program is divided into units of work, called *tasks*. A *task graph* defines a static scheduling of these tasks, as dictated by directed edges called *dependencies*. Our task graph model has two types of dependencies: *logical* and *dataflow*. A logical dependency (A, B) means that A must complete before B can begin. When A has executed, we say that the dependency (A, B) has been satisfied. Logical dependencies are used to implement implicit and explicit dependencies in the SvS framework. In a dataflow dependency, the parent task (called a *producer*) sends data to the child task (called a *consumer*). In this case, the dependency is satisfied as soon as input is available, and the consumer task can run as soon as it receives a data item. 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 said to be *runnable* and we say that an *instance* of the task can be executed. A task instance is equivalent to calling the task's work kernel (the block of code encapsulated by the task) from one or more cores/threads. Tasks can be single instance or multi-instance. In our implementation, consumer tasks are multi-instance because an instance is executed for each data item received. We also describe consumer tasks as *dynamic* because its instances are dynamically "created" as data is received. Multi-instance consumer tasks allow for the expression of data-parallelism and the collection of all instances of a consumer task defines a single data parallel operation. All other tasks in our model are single instance tasks where a single call to the tasks work kernel is executed on a processor.

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 complete before executing the graph again. If we define a single execution of the task graph as an *iteration*, then iteration i must complete before $i+1$ can begin.

The features required to support this task graph model are implemented in our C++ parallel runtime environment, described in [12]. Additionally, the dependencies, tasks, and constructs described in this section satisfy the task graph requirements of an SvS compatible language, which we describe in the next section.

3.2 Cascade Data Management Language (CDML)

Programmers wanting to utilize the SvS framework write their programs in an SvS compatible language. Our prototype of an SvS compatible language is called the Cascade Data Management Language (CDML). Note that CDML itself is not a requirement for SvS. As described in chapter 2, a language is suitable for use in the SvS framework if it provides:

1. Support for defining task graphs
2. Minimal type safety in that all accesses to memory (e.g. pointers, variables) are accesses to memory objects (SvS abstraction for regions of memory)
3. Restrictions on pointer arithmetic in that any resulting pointer must still point to a memory object

We describe the concept of memory objects in greater detail in section 5.1. In order to greatly simplify static analysis without burdening the programmer, CDML disallows pointer arithmetic altogether. To differentiate pointers in CDML from traditional C/C++ pointers, we call pointers in CDML, *links*. Additionally, because our 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 in general. 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. More details on CDML and its future use and contributions can be found in [13].

As in any SvS compatible language, the user writes a program as a collection of tasks. The grammar for defining a task in CDML is shown in figure 3.1.

```
task := task_type task_name constraints? body
task_type := ‘itemizer’ | ‘transform’
constraints := (( send | receive | explicit ) ‘;’) +
sends := ‘sends’ ‘:’ (type output_var ‘=>’ task_name )
receives := ‘receives’ ‘:’ type input_var
explicit := ‘explicit’ ‘:’ task_name (‘,’ task_name)*
body := ‘{’ ( declarations | statements )* ‘}’
```

Figure 3.1: CDML task syntax

As shown in figure 3.1, there are two types of tasks in CDML: *transform* and *itemizer*. A transform is just a single instance task. An itemizer is a multi-instance consumer task where the body of the task is executed for each item received. In effect, an instance of an itemizer is defined by the execution of the task body with a received data item at run-time. Note that this means the data accesses made by an instance of an itemizer will depend on the data item received, and thus is considered to be *input dependent*. The importance of this distinction is described in section 5.2 and chapter 6. Additionally, because itemizers are data parallel consumer tasks, they are always the children of dataflow dependencies.

Tasks written in CDML are assumed to have no specific ordering by default. In order to enforce a specific ordering between tasks, a programmer must specify an explicit dependency using the *explicit constraint* syntax of a task. For a task T, all tasks referenced by `task_name` in the explicit constraint define an explicit dependency ($\texttt{task\_name}, T$) i.e. `task_name` must complete before T can be executed. At run-time, explicit constraints are never broken (i.e. they are not considered during dynamic refinement). In many cases, the programmer will not need to define an explicit ordering between tasks because the same outcome will be achieved regardless of the ordering. This is often the case in video games and scientific computing where many computations are commutative. A programmer can also define dataflow dependencies using the `sends` and `receives` constraints. For example, A task T with "`sends : int output => Consumer`" in its constraints means that T sends integers to the task `Consumer`. `Consumer` would then have the constraint "`receives : int input`" where `input` stores the received integer.

Because SvS provides automatic shared memory management between tasks, programmers do not have to protect accesses to shared memory i.e. programmers do not have to identify data dependencies or orchestrate synchronization or communication between tasks. Static and dynamic analysis determine the potential read/write sets of tasks and the run-time ensures that no two tasks with overlapping read/write sets are executed concurrently. In the cases where two tasks (or task instances) are discovered to have overlapping read-/write sets, an arbitrary ordering is chosen and the tasks are executed sequentially.

In order to compile CDML programs, we have implemented a translator in Java using the ANTLR Parser Generator [32]. Our translator converts CDML programs into C++ using classes defined by our run-time library. The translator also performs symbol collection and static analysis to generate implicit dependencies, which is described in the next chapter.

Chapter 4

Static Analysis

4.1 Task Dependency 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 [10], 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 chapter, 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

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

Figure 4.1: CDML symbol syntax

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 4.1 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 points-to analysis [8, 39, 11, 40], array dependence analysis [10], shape analysis [37, 30], and disjoint heap analysis [24].

Because it is not our goal to expand upon work that has already been done in static analysis, we chose to implement Berndl et al.'s flow- and context-insensitive points-to analysis algorithm [11]. Our implementation is based on the descriptions of this algorithm found in [6, 40]. An overview of this algorithm and its implementation in SvS is provided in the following sections.

$$\begin{aligned}
pts(v,h) &\ :-\ malloc(v,h) && (4.1)\\
pts(v_1,h) &\ :-\ assign(v_1,v_2), pts(v_2,h) && (4.2)\\
hpts(h_1,f,h_2) &\ :-\ store(v_1,f,v_2), pts(v_2,h_2), pts(v_1,h_1) && (4.3)\\
pts(v_1,h_1) &\ :-\ load(v_1,f,v_2), pts(v_2,h_2), hpts(h_2,f,h_1) && (4.4)
\end{aligned}$$

Figure 4.2: Points-to Inference Rules

4.2 Points-to Analysis

The essential problem of points-to analysis is: given a pointer variable v, find all memory locations h such that v *may* point to h. Points-to analysis can be context and flow sensitive or insensitive. Flow sensitive analysis takes into account the order of statements in a program in order to determine the locations h that v may point to at a specific program point. Context sensitive analysis considers the calling context when analyzing function calls. Context and flow sensitivity adds an immense amount of complexity that is far beyond the scope of this work, so we chose to implement Berndl et al.'s flow- and context-insensitive points-to analysis algorithm [11]. Furthermore, we currently do not implement inter-procedural analysis. For the purposes of our experiments, we were able to manually inline function calls as necessary to expose potential accesses to shared memory. The remaining function calls are assumed to be "pure" in that they do not access heap memory and thus can be safely ignored during task dependency analysis.

Berndl et al.'s points-to algorithm is compactly described by the inference rules in figure 4.2. The rules in figure 4.2 use the following relations and domains:

V is the domain of variables of the type "pointer to T" where T is a type. Note that variables are specified using a qualified name that encodes the variable name, the task it was declared in (if applicable), and the block it was declared in.

H is the domain of memory allocation sites. Allocation sites represent all heap objects that can be created by an object allocation statement.

F is the domain of field descriptors. Field descriptors reference a member of an object represented by a variable.

$malloc : V \times H$ is the relation representing object allocation statements of the form "h : v

= new Type" or "h: Type v" where $h \in H$ and $v \in V$. The first statement represents dynamically allocated objects and the second statement represents global variables. Therefore $malloc(v, h)$ means that there is an allocation site h that assigns a newly allocated object to v.

$assign : V \times V$ represents assignment statements. $assign(v_1, v_2)$ means that the statement "$v_1 = v_2$" exists in the program and that v_1 includes the points-to set of v_2.

$store : V \times F \times V$ represents field store statements. $store(v_1, f, v_2)$ means that the program contains the statement "$v_1.f = v_2$".

$load : V \times F \times V$ represents field load statements. $load(v_1, f, v_2)$ means that there is a statement "$v_1 = v_2.f$" in the program.

$pts(v, h) : V \times H$ is the relation that says a variable v_1 can point to heap objects from allocation site h.

$hpts(h_1, f, h_2) : H \times F \times H$ is the relation that the field f of a heap object from site h_1 can point to an object from site h_2.

Referring back to figure 4.2, rule 4.1 says that v can point to h if there is an allocation site h that involves v. Rule 4.2 performs a transitive closure of assignment statements. Rule 4.3 captures the effects of "$v_1.f = v_2$" statements saying if v_1 points to an object at site h_1 and v_2 points to an object at site h_2, then the field f of objects at h_1 points to objects at h_2. Finally, rule 4.4 captures the effects of "$v_1 = v_2.f$" statements saying if v_2 points to an object at h_2 and the field f of an object at h_2 points to an object at h_1, then v_1 points to an object at h_1.

Using the above relations and rules, points-to analysis is achieved as follows:

1. For each assignment statement in the program, create an appropriate *load*, *store*, *malloc*, or *assign* relation.

2. Apply the rules in figure 4.2 until the results converge (i.e. until no new relations are found)

The results of points-to analysis is a mapping of variables to sets of allocation sites that a variable can point to at some point in the program. The implementation of points-to analysis in SvS, and how it is used in task dependency analysis, is described in the next section.

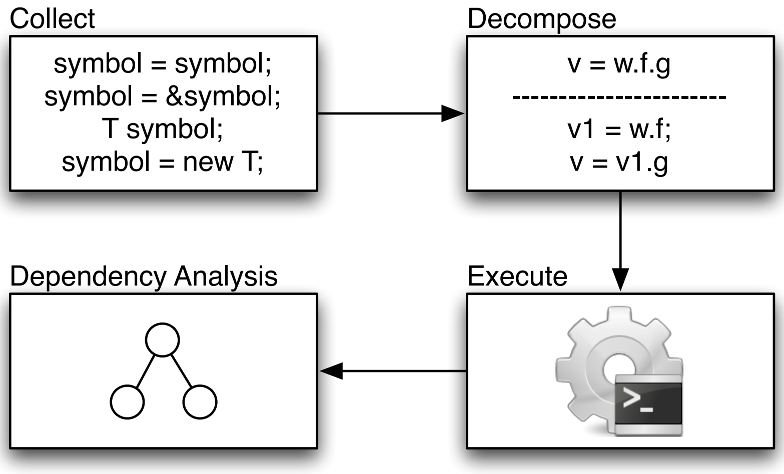


Figure 4.3: Stages of Points-to Analysis in SvS

4.3 Implementation of Points-to Analysis in SvS

Figure 4.3 shows the stages of points-to analysis that take place in our translator. The first stage is a symbol collection pass that collects all assignment and allocation statements in the program that involve links (CDML's pointer equivalent). It also collects symbols that are read as part of expressions, since these appear in statements other than link assignment. These assignments, allocations, and symbols cannot be immediately processed by our points-to analysis algorithm because it only accepts statements recognized by the *load*, *store*, *malloc*, and *assign* relations. Therefore the collected statements and symbols must first be decomposed into a series of *load*, *store*, *malloc*, or *assign* statements. For example, the statement "$v = w.f.g$" would be decomposed into "$v_1 = w.f$" and "$v = v_1.g$". Each time a new *load*, *store*, *malloc*, or *assign* statement is matched or created, the symbols/variables involved are added to a list belonging to the task in which the original statement was found. This list, called *pt_access_list*, is used for reference once points-to analysis results are calculated.

Note that the relations and rules defined in the previous section can be implemented directly in the Prolog language. Therefore, once all the *load*, *store*, *malloc*, or *assign* relations (now called *facts*) are created, the points-to inference rules are appended to create a Prolog program. Using the SWI-Prolog [41] JPL library interface, we are able to execute this program from within the translator and collect the results. The program is executed by querying for all solutions to $pts(V, H)$ and the results are stored in a hash map that associates a set of allocation sites for each variable.

In the last stage, we perform task dependency analysis. First, we iterate over the variables in the *pt_access_list* for each task that was generated during decomposition and use the results map to look up the points-to set for each variable. Each of these points-to sets are added to the cumulative points-to set for the task. Once the points-to sets have been calculated for each task, a pairwise comparison between tasks' points-to sets is performed and if the points-to sets of two tasks T_1 and T_2 overlap, then the translator generates the implicit dependency (T_1, T_2).

4.4 Summary

In our implementation of SvS, it is not our intention to match the state of the art or expand upon the work that has already been done in static analysis, but rather to address the unavoidable limitations associated with static analysis by developing and implementing feasible and beneficial dynamic analysis techniques. However, some form of static analysis satisfying the conditions outlined in section 4.1 is necessary for a complete implementation of the SvS Framework. Therefore we have opted to implement the rudimentary static analysis presented in this chapter for the purposes of completeness and proof of concept, not accuracy. Regardless of the sophistication of static analysis, an unavoidable limitation is the lack of information available at compile time. In the next chapter, we describe the dynamic analysis techniques that SvS employs to address the limitations of static analysis.

Chapter 5

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.

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*. We use *dynamic reachability analysis* (section 5.1) 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.

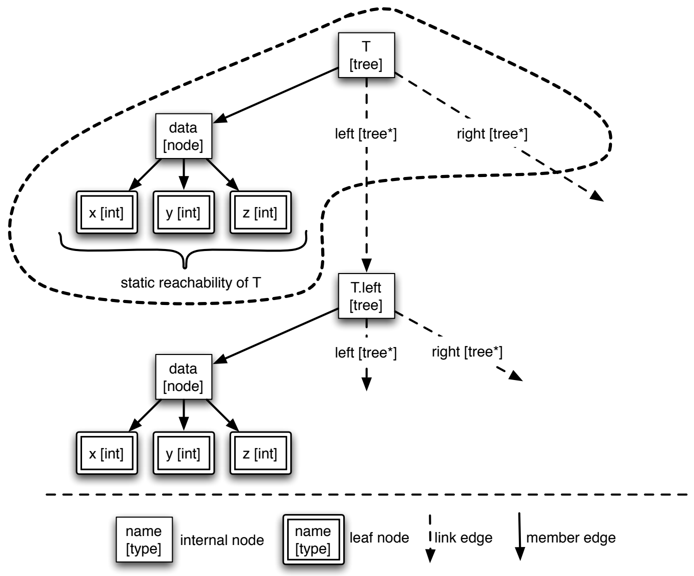
We will also introduce *signatures* (section 5.1.1), which are used to compactly represent read/write sets and efficiently determine which tasks have non-overlapping memory accesses.

After dynamic refinement is performed, it is the role of the scheduler to ensure that concurrently executing tasks have non-overlapping read/write sets. Note that if all of a task's dependencies are either satisfied or removed by dynamic refinement, then it is guaranteed to not conflict with any other task. In these cases, the task scheduler can immediately execute the task without any further checking. However, there are cases where the comparison of read/write sets that is normally performed during dynamic refinement must be deferred onto the task scheduler. In these cases the task scheduler is responsible for efficiently comparing the read/write sets of (potentially many) tasks/task instances in order to dispatch instances with non-overlapping read/write sets. Two new scheduling algorithms that accomplish this goal will be presented in chapter 6.

5.1 Dynamic Reachability Analysis

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*. Such a memory object can indirectly access all the memory that its members can access. So far, the definition of memory objects still implies static accesses. To allow for dynamic accesses, memory objects may also contain *links* and we say that the memory object that contains the link is the *parent*. 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. In this case, the link has no parent and is merely a reference or alias to the memory object it points to. 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 indirectly through members and links.

We formalize these definitions by representing the problem as a graph. Nodes in the graph represent memory objects. 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. In other words, it is just an alias to a node in the graph and therefore does not constitute an edge, nor does it affect the structure of



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

Figure 5.1: 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 and is called the *static reachability* of T. Related C++ definitions are provided on the bottom.

the graph. Since a link L can only change the memory object it points to, its parent never changes. 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*.

There are two important properties to note about graphs generated by the above formulation. First, leaf nodes will always be primitives, since they have an out-degree of zero and thus do not point to anything and do not have any members. Second, the only edges in the graph that can be modified are link edges and only by changing the child that a link points to. All other edges are static.

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. If we restrict the breadth first search to only follow member edges, then the resulting set of addresses is the *static reachability* of the node (i.e. a unique, static set of memory accesses). Figure 5.1 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 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.

5.1.1 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 [17] using a single hash function. Also, because signatures are constant in length and use hashing, there is the opportunity for false positives to 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 7.2.

5.1.2 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

The implementation of memory objects is analogous to the Object class in Java in that all types in an SvS compatible language are implemented as classes that inherit from a `MemoryObject` template class. The `MemoryObject` class stores the id of a memory object that is generated inside the class's constructor. Ideally, this id would be unique, but this is not a necessary condition for correctness and would require `MemoryObject`'s created on separate threads to synchronize. To avoid this, we use a fast, thread-safe random number generator to assign object ids. Recall that a newly created memory object will have a static reachability that denotes a unique and static set of memory accesses. Because these accesses are unique and static, they can correctly be represented by a single id: the id of the newly created (root) memory object. We will use this property as an optimization for calculating the reachability of a memory object.

The `MemoryObject` class contains a **getSignature** function that 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 recursively 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. However, 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. Recall that a link has a non-null parent only when it is a member of a memory object and that its parent never changes. Therefore a memory object M that contains a link as a member is responsible for setting the parent of the link to point to M and once the parent has been set, it is never changed. Parent pointers are set in the constructor of a class inheriting from the `MemoryObject` class and this code is generated by our translator since it knows the members of a declared or built-in type. A null parent represents the case where a link is just a reference or alias to the memory object it points to (i.e. it is not a member of a memory object so its parent does not get set) 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. Note that the only way to change the reachability of a memory object, and thus the reachability graph itself, is by changing 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.

The goal of master nodes is to logically group together nodes in the reachability graph to form super-nodes, thus reducing the effective size of the graph during updates. As such, 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 *neighbourhood* of M. A master M maintains a signature that accurately represents its reachability; this signature is shared by all nodes in the neighbourhood of M. M is also responsible for propagating changes in the reachability of nodes in its neighbourhood to all other masters that can reach M.

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

Algorithm 1: Link Assignment

```
Input: A link ($lhs$,$rhs$) where $lhs$ is the parent memory object of the link and $rhs$ is
       the new child that the link will point to as a result of the assignment
begin
    if inDomain($lhs$) then
        setMaster($lhs$)
    end
    if getDomainSize($lhs$) $<K$ or inDomain($rhs$) then
        incDomainSize($lhs$)
        if not isEmpty(owningMasters($rhs$)) then
            foreach $master \in$ owningMasters($lhs$) do
                push( notifyMasters( primaryMaster($rhs$)), $master$)
            end
        end
        if not isMaster($rhs$) then
            foreach $master \in$ owningMasters($lhs$) do
                push( owningMasters($rhs$), $master$)
            end
        end
    else
        setMaster($rhs$)
        foreach $master \in$ owningMasters($lhs$) do
            push( notifyMasters($rhs$), $master$)
        end
    end
    updateReachability( $lhs$, reachability($rhs$))
end
```

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 continuously 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 `owningMasters(`*node*`)` is the masters to whose neighbourhood *node* belongs. `notifyMasters(`*master*`)` is the list of masters that *master* must notify when its reachability changes, because these masters can reach the nodes in *master*'s reachability. Once the necessary updates to `owningMasters` and `notifyMasters` are complete, the reverse breadth first search to update scope is initiated by the call to `updateReachability`. Note that `updateReachability` also performs cycle detection in the reachability graph.

5.1.3 Summary

Because symbols are just references to memory and because all types inherit from the `MemoryObject` class, at runtime symbols are just memory objects or links to memory objects. To maintain the reachability (i.e. potential memory accesses) of a memory object, we perform dynamic reachability analysis every time a link assignment occurs. This means that dynamic reachability analysis takes place (potentially in parallel) during task execution. The next section discusses how the results of dynamic reachability analysis (signatures of master nodes representing reachability) are used to generate more precise read/write sets for tasks *before* tasks are executed in order to increase parallelism in SvS (e.g. by removing unnecessary implicit dependencies) while still maintaining correctness.

Algorithm 2: Refinement

```
Input: Task parent
Output:
1  begin
2      foreach child ∈ implicitDependents(parent) do
3          if not (isDynamic(parent)) and not (isDynamic(child)) then
4              psig = getSignature(parent)
5              csig = getSignature(child)
6              if not(conflict(parent,child)) then
7                  removeDependency(parent,child)
8              end
9          else
10             if doPropagate(parent) and (atomic_dec(parentsNotReady(child))
               == 0) then
11                 propegateSchedulingDomain(parent,child)
12                 removeDependency(parent,child)
13             end
14         end
15     end
16 end
```

5.2 Refinement

The purpose of dynamic refinement is to utilize the results of dynamic reachability analysis in order to generate more precise read/write sets and compare these sets between tasks in order to potentially remove unnecessary implicit dependencies that were created during static analysis. The algorithm for this process is shown in algorithm 2.

The refinement algorithm is executed when a task becomes runnable, as defined in section 3.1. As depicted in algorithm 2, a runnable task (*parent*) iterates through each *child* (i.e. dependent) for which an implicit dependency (*parent*, *child*) exists, and for each one, it checks to see if both *parent* and *child* are not dynamic. Recall from section 3.1 that a task is dynamic if it is the consumer task of a dataflow dependency. We describe the situation where one or both of *parent* and *child* are dynamic later in this section. Otherwise, if both tasks are not dynamic, then signatures representing their read/write sets are compared and if there is no overlap we can safely remove the dependency between *parent* and *child*. This may cause *child* to become runnable and be executed concurrently with *parent*, thus increasing parallelism.

Algorithm 3: 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
```

The signatures representing the read/write sets of *parent* and *child* are generated by calling `getSignature`, which is outlined in algorithm 3. Each task in SvS has a `getSignature` function that implements this algorithm; the code for `getSignature` is generated by the translator using the symbols collected during static analysis. Note that at run-time, these symbols become references to memory objects. Therefore, a composite signature for the task is created by iterating over each symbol and adding the signature representing the reachability of the underlying memory object to the signature representing the read/write set of the task. Note that we only iterate over symbols (memory objects) that are global in scope or are received links to memory objects, since they potentially reference shared memory. Symbols defined locally in a task are not visible outside the scope of the task and can be safely ignored because in order to reference shared memory, it must be the case that at some point a global or received symbol was aliased and any arbitrary link dereferencing/traversal will be encompassed by the reachability of that memory object. This follows from the points-to analysis rules in section 4.2 and the definition of dynamic reachability analysis.

Complications arise when one or both of *parent* and *child* are dynamic tasks. Because dynamic tasks are consumer tasks that receive data, their data accesses are dependent on the data they receive and therefore the result of `getSignature` is dependent on the received input. One way to resolve this issue is to make the conservative decision to leave the implicit dependency in place at the cost of potentially limiting parallelism. Instead, we attempt to optimistically remove the implicit dependency and defer read/write set comparison on to the scheduler. This is what is happening on lines 10 to 12 of algorithm 2 and the mechanism that allows us to optimistically remove dependencies is described in chapter 6. Note that optimistically removing a dependency is not the same as the optimistic execution that occurs in speculative parallelism. Optimistically removing a dependency during dynamic refinement just means that the dependency is removed without checking read/write sets because the checking of read/write sets will be performed by the scheduler instead. The scheduler still performs the checks to detect potential conflicts *before* tasks are executed, as opposed to performing checks during or after task execution as is the case with speculation.

There is another complication regarding dynamic tasks that is not addressed by dynamic refinement. Because the instances of dynamic tasks are generated dynamically, we do not have any dependency information for individual instances, only the collection of all instances when considered as a single, serial task. In order to avoid having to serialize all instances of a data-parallel consumer task, we again defer dependency checking on to the scheduler. It is then the scheduler's job to efficiently compare the read/write sets of (potentially many) task instances in order to dispatch task instances with non-overlapping memory accesses.

Chapter 6

Scheduling

The key role of the task scheduler is to ensure that any two concurrently executing task instances have non-overlapping read-write sets. In the absence of dynamic tasks, the job of the task scheduler is trivial: if all the dependencies of a task have been satisfied or removed by dynamic refinement, the task can be immediately executed. This is because any remaining implicit dependencies after dynamic refinement are sufficient for protecting shared state accesses. So if a task is runnable, all its dependencies have been satisfied and thus will not conflict with any other task.

However, as described in the previous chapter, dynamic tasks introduce more complexity. If, during dynamic refinement, an implicit dependency is optimistically removed, it is up to the scheduler to check to see whether a dependency in fact exists once input data becomes available. The scheduler is also responsible for efficiently comparing the read/write sets of (potentially many) task instances of data parallel consumer tasks in order to extract parallelism by dispatching task instances with non-overlapping read/write sets. The mechanism for coordinating this between instances of the same task or different tasks is called *scheduling domains*.

6.1 Scheduling Domains

A scheduling domain is comprised of three main components:

1. A set of runnable task instances to be scheduled

2. A signature representing instances that can be co-scheduled and/or are currently executing

3. A distributed algorithm that mediates the co-scheduling of instances and manages the shared signature and other state required by the algorithm

While there are many ways to implement the co-scheduling of task instances using scheduling domains, the basic process is as follows: as input becomes available, read/write sets are generated for task instances. Task instances then compare their read/write sets to the domain signature and if there is no overlap, they add their individual signatures to the domain signature and execute. If there is overlap, the execution of the instance is deferred until it is safe to run.

The process of assigning tasks to domains is described in lines 10 to 12 of algorithm 2. We call this method *strict propagation.* Strict propagation ensures the following set of properties:

1. A parent task shares a domain with its children (i.e. propagates its domain to a child)

2. A child that is assigned a domain does not propagate it to other tasks

3. If a child has multiple parents, it only shares a domain with one of them

4. Tasks are assigned a single domain

5. Multiple tasks may be assigned to the same domain

The first property just provides a source for creating and assigning domains, and the second property prevents domains from encompassing too large a region of the task graph. For example, without the second property, a domain created by a parent that is a root of the task graph may end up assigning all tasks in the graph to its domain. This is not optimal because there will be many task instances in the domain that will never access the same memory and therefore we incur unnecessary overhead by comparing the signatures of non-conflicting tasks against each other. The third and fourth properties reduce the complexity of the algorithm. Under a different scheme, we could allow tasks to be part of multiple domains. However, in order for an instance of a task to run, it would have to essentially "lock" each domain in order to prevent changes to the domains while the task instance compares its read/write set against each domain's signature, which would create a large critical section.

With relation to algorithm 2, the two conditions on line 10 ensure properties two and three. The call to `doPropagate` ensures property two by returning true if *parent* has not already been assigned an instance by one of its parents. The expression

$$\textbf{atomic\_dec}(\textbf{parentsNotReady}(\textit{child})) == 0$$

guarantees property three by checking to see if *parent* is the last of *child*'s parents to be made runnable. If these two conditions are true, then we (optimistically) remove the implicit dependency and assign the domain of *parent* to *child*.

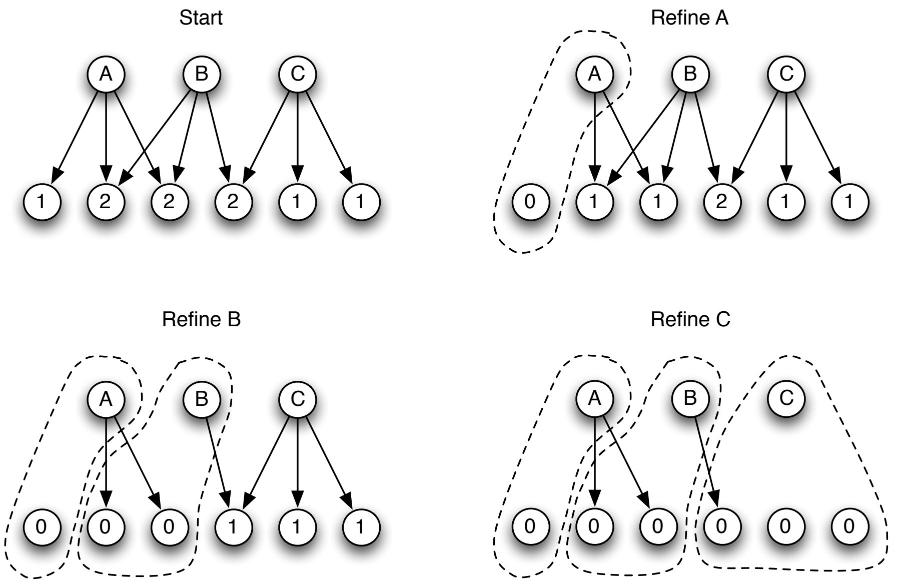


Figure 6.1: Example of SvS scheduling domain propagation. Domains are represented by the dashed boundaries.

Figure 6.1 demonstrates how strict propagation works. In the figure, tasks A, B and C are being considered for refinement. While this process happens in parallel, we will describe it in a serial fashion for ease of explanation. Assume that all tasks in the figure are dynamic. First, task A decrements the `parentsNotReady` counter of each of it's children. It succeeds in decrementing one of these counters to 0 and thus shares its domain (represented by the dashed region) with the child and leaves the dependencies with its other two children intact. B then does the same, removing dependencies and sharing its domain with two of its children while leaving one dependency unbroken. Finally, C succeeds in removing all of its dependencies with its children. Note that as dependencies are removed, some children become runnable and are able to execute concurrently in the same domain as their parent. While some dependencies remain, additional parallelism has (potentially) been uncovered. It is important to note that unless there is a remaining implicit dependency between tasks in different domains, two task instances from different domains will never conflict allowing instances in one domain to be scheduled independently of instances in other domains.

Once a task becomes runnable and is assigned a domain, it executes the domain's scheduling algorithm. We have developed two different domain scheduling algorithms, *progressive* and *staged*, which we present in the following sections.

Algorithm 4: Progressive Scheduling Domain Algorithm

```
Input: Task $T$, input queue $inputQueue$ and the SvS instance $I$ assigned to $T$
begin
    while needsThreads() do
        if not(isResetting($I$)) and pop($inputQueue$,$input$) then
            currentSig = getSignature($T$,$input$)
            if atomic_compare_and_update($I$,$currentSig$) then
                execute($T$,$input$)
            else
                push($inputQueue$,$input$)
            end
        end
        resetState($I$)
    end
end
```

6.1.1 Progressive

The progressive algorithm is outlined in algorithm 4. The primary object of this algorithm is to try and immediately execute task instances. The key mechanisms that facilitate immediate execution are encapsulated in calls to `atomic_compare_and_update`. This function atomically compares the signature of the current task instance to the domain signature representing all currently executing instances in the domain. If there is no conflict, the task's signature is added to the domain's signature and the function returns true. Otherwise, the domain signature is not updated and the function returns false. During each call to `atomic_compare_and_update`, some state related to the domain's signature is also atomically updated. If `atomic_compare_and_update` returns true, it is safe to execute an instance of the task. Note that once a task instance's signature has been added to the domain signature it cannot be removed. Therefore, after several tasks complete execution, the domain signature becomes stale and occasionally needs to be reset. When tasks call `resetState`, the state of the signature is checked and if it needs to be reset, a single task instance (and the thread it is running on) is elected to reset the domain signature and its state. While this is happening, other tasks are prevented from being executed by calling `isResetting`. Finally, the `needsThreads` function just signals to threads whether or not the domain can currently utilize more threads to execute the domain algorithm. If `needsThreads` returns false, threads will look for other domains to work on.

6.1.2 Staged

Algorithm 5 provides the staged method of executing task instances in a domain. A *stage* refers to a working set of task instances that can be safely executed in parallel where the creation and execution of one stage must complete before the next stage begins. When threads enter a stage, a single thread is elected to create the working set by comparing the signature of a task instance to the signature of the working set and if there is no conflict, the instance is added to the working set. If there is a conflict, the execution of the task instance is deferred to a later time. During this process, information related to the state of the signature is updated (as is done in the progressive algorithm) in order to determine when the working set/stage should be *released*. When it is determined a stage should be released, `doRelease` returns true and the stage is released for execution. Note that while the working set/stage is being constructed, no other task instances in the domain are being executed. Once released, threads concurrently execute task instances in the working set and all threads wait (by calling `stageWaitAndReset`) for all task instances in the working set to complete before moving onto the next stage. The key differentiation between the progressive and staged algorithms is that the staged algorithm alternates between phases of stage creation and execution rather than trying to immediately execute task instances. While the stage creation phase creates an, albeit small, bottleneck, when compared to the progressive algorithm the staged algorithm decreases contention over the *inputQueue*, requires less synchronization, and is better at balancing load across threads in situations where available parallelism is lacking. As a result, we have found the staged algorithm to generally be more efficient than the progressive algorithm.

Algorithm 5: Staged Scheduling Domain Algorithm

```
Input: Task $T$, input queue $inputQueue$ and the SvS instance $I$ assigned to $T$
begin
    while needsThreads($I$) do
        if not (releaseStage($I$)) and becomeStageMaker($I$) then
            reset($I$)
            while not (doRelease($I$)) and pop($inputQueue,input$) do
                currentSig = getSignature($T,input$)
                if not conflict($currentSig$,signature($I$)) then
                    updateStateSuccess($I,currentSig$)
                    push( $workQueue, input$)
                else
                    updateStateConflict($I$)
                    push($inputQueue,input$)
                end
            end
            releaseStage($I$) = true
        end
        if releaseStage($I$) then
            while pop($workQueue, input$) do
                execute($T,input$)
            end
            stageWaitAndReset()
        end
    end
end
```

6.2 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 performs dynamic reachability analysis to generate more precise read/write sets for tasks and dynamic refinement uses these read/write sets to potentially remove unnecessary dependencies, thus enabling a greater degree of parallelism while still ensuring correctness. In the cases where SvS lacks static dependency information between dynamic task instances or dynamic refinement optimistically (i.e. without checking read/write sets) removes an implicit dependency, SvS defers the checking of read/write sets onto the task scheduler. The end result is that SvS automatically prevents conflicting accesses to shared memory by determining the potential read/write sets of tasks *before* tasks are executed, and schedules tasks such that no two concurrently executing tasks have overlapping read/write sets.

Chapter 7

Evaluation

In this chapter, we evaluate our implementation of the SvS framework on a series of micro-benchmarks, existing parallel benchmarks, and a large scale example from the video game domain. In section 7.1, we provide a discussion of the effects of static analysis on scheduling and define the focus of our evaluation. We begin our quantitative analysis in section 7.2, providing several experiments utilizing micro-benchmarks to demonstrate the primary parameters and costs associated with SvS. In section 7.3, we present two benchmarks from the PARSEC suite [16]: Fluidanimate and Canneal. PARSEC was chosen as a well known parallel benchmark suite that will help put the performance of SvS into perspective with existing parallel programming environments that require manual shared memory management. Finally, we have implemented a video game benchmark that focuses on entity management and artificial intelligence systems for managing and controlling the actions and movements of numerous agents. We chose to test SvS with a game benchmark because game engines have been profoundly affected by the shift to multi-core architectures and parallel programming. The complexity and tightly coupled interactions of numerous subsystems can rival operating systems, making the problem of protecting accesses to shared state extremely difficult [4], thus making it an attractive testbed for SvS. On top of this, they have high demands on performance and can greatly benefit from parallelization, if done efficiently. The complexity of these systems also means that converting a serial game engine into a parallel task-based model is almost insurmountable. Even in industry, it is instead often more preferable to re-implement a parallelized engine from the ground up. Our video game benchmark presented in section 7.4 represents our current efforts towards this goal.

For different experiments, we compare an SvS implementation, written in CDML, to implementations using Intel Threading Building Blocks (TBB) [2], Software Transactional Memory (STM) with the Dresden TM Compiler [20] and TinySTM++ library, and pthreads. In all SvS implementations, we use the staged scheduling domain algorithm as we have found it to perform better than progressive in most cases. 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.

7.1 Effects of Static Analysis on Scheduling

Conservative decisions during static analysis motivate the need for dynamic analysis, and are easily handled by dynamic refinement when considering static, single instance tasks. However, conservatism becomes problematic for scheduling when unnecessary implicit dependencies are placed between data parallel tasks. If the unnecessary implicit dependency is not optimistically removed, then the instances of one operation must complete before instances of the other begin, causing a much greater decrease in potential parallelism than there would be if two single instance tasks were being considered. Even if the dependency is removed, many task instances that do not conflict will share a scheduling domain, and thus we unnecessarily pay an extra cost (e.g. increased potential for false positives).

While the effects of static analysis on dynamic analysis and scheduling are an expected source of overhead in our system, our rudimentary static analysis, implemented primarily for completeness and proof of concept and not accuracy, has a greater propensity to exaggerate this overhead because it is far more conservative than more modern implementations. Our primary contribution, and thus the focus of our evaluation, is demonstrating the feasibility of using dynamic reachability analysis to determine a task instance's potential memory accesses *before* it runs and subsequently protecting concurrent access to shared memory by comparing read/write sets of many task instances and scheduling non-conflicting instances to run concurrently, as orchestrated by our scheduling domain algorithms.

For these reasons, we have chosen benchmarks that are composed primarily of pipelined or independent data parallel operations that require SvS to schedule instances within a data parallel operation. However, instances of different tasks in these tests do not access the same memory, and thus no implicit dependencies are required between tasks. In order to avoid being penalized by the simplistic implementation of our static analysis, we disable our points-to analysis in our experiments and only perform the symbol collection necessary for dynamic analysis. Therefore all protection of shared state is handled dynamically by scheduling domains and dynamic reachability analysis, which are the main contributions of this work.

7.2 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. False positives can lead to reduced parallelism, and there are several sources for false positives in SvS. First, having fixed sized signatures means that false positives occur as a result of signature hashing. Also, because internal nodes share the signature of a master node, there may be memory objects encoded in the signature that an internal node can not actually reach. Finally, our current implementation of dynamic reachability analysis conservatively provides results for the *full* reachability of a memory object, even though a task may only access a portion of this set of potential accesses.

The key parameters governing the costs of false positives and link assignment are signature size, master neighbourhood size, and reachability graph size and shape (i.e. connectivity). In the following sections, we break down our analysis into two categories: signatures and dynamic reachability analysis.

7.2.1 Signatures

As mentioned in section 5.1.1, 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 staging algorithm, this is the size of the *workQueue* each time it is released. 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.

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

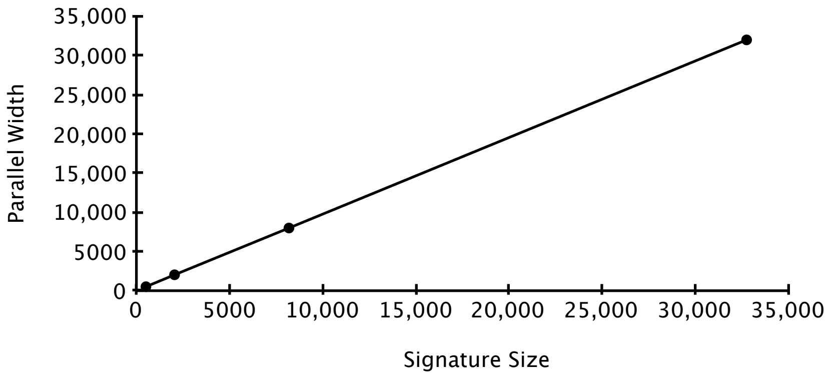


Figure 7.1: Parallel width for 128,000 task-instances under varying signature sizes

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 dynamic reachability analysis. Therefore, since all objects are unique, any detected conflicts are strictly due to false positives caused by signature size.

Figure 7.1 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). To measure parallel width, we just record the size of the *workQueue* each time a stage is released. 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 the staging algorithm is often successful in finding a task instance with a signature that does not conflict with the signature of the instances already added to the *workQueue*.

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. Finally, the bitwise operations used when comparing/calculating signatures are prime candidates for vectorization.

Because parallel width increases 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. In our experiments, we have found a signature size of

512 to be practical and sufficient.

7.2.2 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 neighbourhood sharing the same reachability signature: the signature of the master node representing that neighbourhood. Another source of false positives is conservativeness in dynamic reachability analysis. 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 neighbourhood: the larger the neighbourhood, 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 neighbourhoods. 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 neighbourhoods 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. We ran all three experiments using 8 threads and a signature size of 512.

Absolute Cost

For absolute cost, we measured the time it takes a consumer to perform a link assignment under varying neighbourhood sizes. The results are shown in figure 7.2, with the cost in microseconds on the y-axis and neighbourhood sizes (maximum number of internal nodes per neighbourhood) on the x-axis. Figure 7.2 demonstrates that as neighbourhood sizes increase, the cost decreases from about 7.2-4.6μs and 7.6-5.0μs for 20,000 and 40,000 objects respectively. There is also a slight overall increase (5.4%-7.6%) in cost going from 20,000 objects to 40,000 objects. Therefore, neighbourhood 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 20μs, 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 we are paying a cost during link assignment, SvS does not pay the cost of conflict resolution paid by other techniques such as STM. In sections 7.3 and 7.4 we demonstrate, with real applications, that the costs of SvS are outweighed by its benefits.

Parallel Width

Figure 7.3 demonstrates the change in parallel width (y-axis) as we increase neighbourhood sizes (x-axis). We measured the parallel width as described in section 7.2.1. Here we

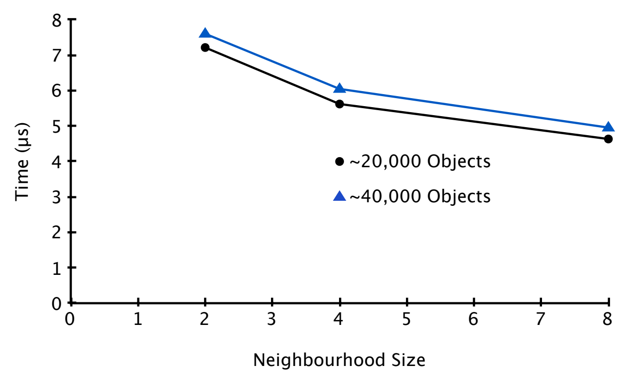


Figure 7.2: Cost of link assignment under varying neighbourhood and reachability graph sizes

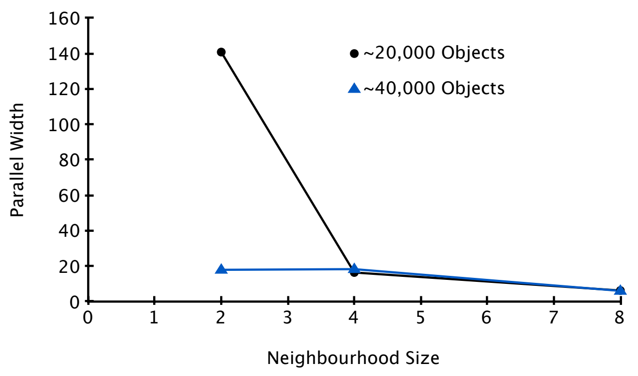


Figure 7.3: Parallel width under varying neighbourhood and reachability graph sizes

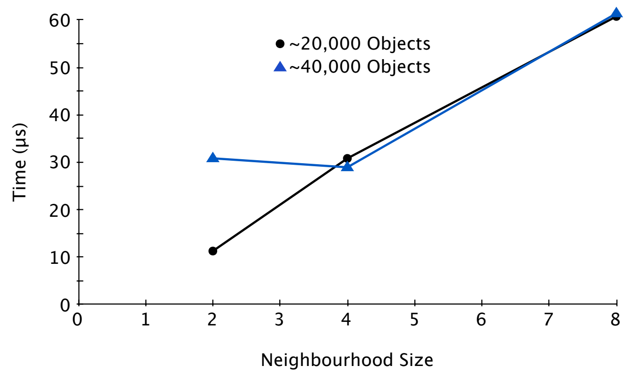


Figure 7.4: Average per task instance overhead (μs) of the consumer for varying neighbourhood and reachability graph sizes

see that parallel width is dramatically affected by the size of master neighbourhoods. As neighbourhood sizes 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 7.3. Note also that the size and shape of the reachability graph also have a dramatic effect on parallel width. For neighbourhood size 2, parallel width drops from 140 to 18 when going from 20,000 to 40,000 memory objects. This change is the result of how master nodes are placed in the reachability graph. When we construct a binary tree, it turns out that if the tree has an odd number of levels (as is the case for 20,000 objets) and a neighbourhood size of 2, all leafs of the tree are master nodes and thus have precise reachability. Because the signatures for task instances are built from the reachability of leafs, this means there will be fewer false positives when comparing the read/write sets of task instances. However, for an even number of levels, two leaves will share the master node of their parent (which is the parent itself). Because sibling leafs share a master node, they cannot be processed in parallel since they also share the same reachability signature. Additionally, leafs will have larger reachability (the reachability of their parents), further increasing false positives and affecting parallel width. How parallel width, along with link assignment, affects overall performance is described in the next section.

Overall Overhead

The goal of this section is to precisely define the actual cost incurred by using SvS in our system. If our system had zero overhead, we would be able to achieve the theoretical maximum parallel speed up, and thus our theoretical execution time of the consumer would be:

$$T_{parallel} = \frac{N * \delta}{C} \tag{7.1}$$

where N is the number of task instances to be scheduled, δ is the minimum task completion time and C is the number of available threads/cores. Note that $N * \delta$ just defines the minimum *serial* execution time of the consumer in a system with zero overhead. We can then calculate the *worst case* overhead of our system by measuring the execution time of the consumer in our micro-benchmark and calculating the difference between the actual parallel

Neigh. Size	Graph Size	Link Assign.	Scheduling	False Pos.'s	Total
2	20K	7.22	2.81	1.26	11.29
	40K	7.61	3.52	19.72	30.84
4	20K	5.62	3.19	22.01	30.82
	40K	6.05	3.46	19.44	28.95
8	20K	4.63	4.02	52.11	60.76
	40K	4.96	4.05	52.40	61.40

Table 7.1: Breakdown for the average per-task instance overhead (in μs) of SvS.

runtime to the theoretical runtime given by equation 7.1. However, using equation 7.1 as our baseline means that the calculated overhead will not only include the overhead of SvS, but also the unavoidable overhead of any task scheduling system. In order to isolate the overhead incurred by SvS, we use the following baseline: the execution time of the consumer when SvS is *disabled* in our system. When SvS is disabled, dynamic reachability analysis is not performed and the task scheduler does not compare the read/write sets of task instances in order to schedule them, it just schedules task instances to be executed immediately. We can then rearrange equation 7.1 as:

$$\delta = \frac{T_{parallel} * C}{N} \tag{7.2}$$

where $T_{parallel}$ is the execution time of the consumer when SvS is disabled. We can safely disable SvS in our micro-benchmark because no sharing actually occurs. Note that δ provides an *average* per task execution time which serves as more meaningful baseline as it includes the minimum execution time of a task instance *and* the average minimum overhead required to schedule a task instance in our system. We can then measure the execution time of the consumer with SvS enabled, and calculate the difference between δ(`SvSEnabled`) and δ(`SvSDisabled`) in order to provide a more precise measurement of the overhead of SvS. Figure 7.4 shows on the y-axis the resulting average per-task instance overhead of SvS – δ(`SvSEnabled`) – δ(`SvSDisabled`) – for different neighbourhood sizes (x-axis).

We see that for 20,000 objects, the per-task-instance overhead (μs) increases as neighbourhood sizes increase. This is because the decrease in parallel width dominates the decrease in link assignment cost as neighbourhood sizes increase. For 40,000 objects, there is virtually no decrease in parallel width between neighbourhood sizes 2 and 4 and therefore we see a slight decrease in overhead due to a decreased link assignment cost. After a neighbourhood size of 4, overhead begins to increase. Overall, we see overheads ranging from 11 to 61μs.

The per-task instance overhead of SvS in figure 7.4 includes the overhead of both link assignments and false positives (decreased parallel width) as well as the extra costs for scheduling in SvS (e.g. read-write set comparison to schedule tasks). In order to further break-down the overhead of SvS, we ran another experiment (which we label `ZeroSig`) where we changed the `getSignature` function for task instances to return an empty signature (all bits equal to 0). This means that SvS will not detect any conflicts (which are all false positives in our benchmark) and therefore will only include link assignment and SvS scheduling overhead. Therefore we can calculate the difference of the per-task overheads of `SvSEnabled` and `ZeroSig` in order to get the amount of overhead caused by false positives. Furthermore, because each task instance performs a single link assignment, we can just subtract the absolute link assignment cost from the overhead for `ZeroSig` in order to get the SvS scheduling overhead. The breakdown of overhead for false positives, link assignment and scheduling in SvS is provided in table 7.1.

7.2.3 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, communication overhead from synchronization, and 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 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.*

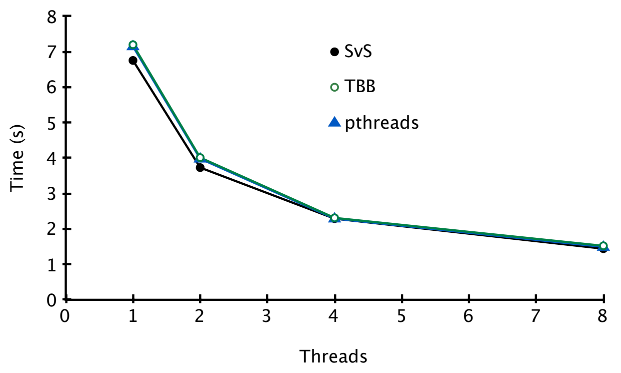


Figure 7.5: Runtime of Fluidanimate for varying number of threads

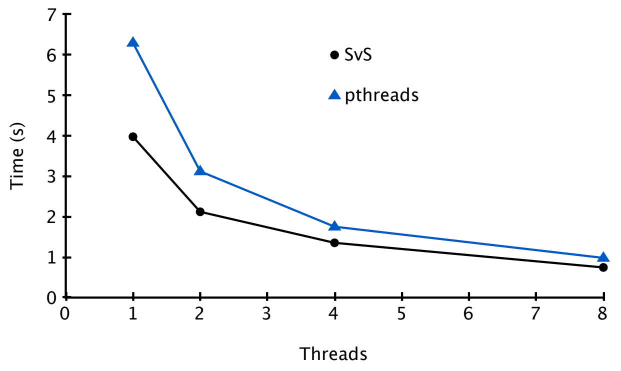


Figure 7.6: Runtime of Canneal for varying number of threads

7.3 PARSEC

PARSEC is a parallel benchmark suite designed to represent state-of-the-art parallel workloads [16]. While the majority of these benchmarks are embarrassingly parallel (i.e. no shared memory accesses) and thus do not need SvS, we converted Fluidanimate and Canneal which do have shared state conflicts and thus require synchronization. Both of these benchmarks operate on objects (often array elements) that do not have any links as members (i.e. objects with static reachability) and thus represent a real world case where dynamic reachability analysis is not actively updating reachability due to the absence of link assignments. For all PARSEC experiments, we used the *simlarge* dataset as input.

7.3.1 Fluidanimate

Fluidanimate performs fluid dynamics simulation by dividing a 3D space into partitions of cells containing particles where each partition is assigned a thread/core. When a cell is updated, the values of adjacent cells are are also updated. Internal cells of a partition (those not on a partition boarder) can be updated without synchronization, while boarder cells require locking. In the SvS implementation, writes to shared cells are never executed concurrently, thus automatically protecting boarder cells. This enabled us to increase the number of partitions and thus increase available parallelism. Figure 7.5 demonstrates the average runtime (in seconds, y-axis) for varying number of threads (x-axis) and compares SvS to the existing third party TBB and pthreads implementations of Fluidanimate. It is difficult to see in the figure, but the TBB and pthreads implementations have almost identical performance. SvS closely resembles the performance of both third party implementations and is 81 and 51 ms faster than TBB and pthreads respectively at 8 threads. Note that the TBB and pthreads implementations use fine-grained locking in order to protect shared state, while SvS does not require the programmer to perform any explicit synchronization. These results demonstrate that SvS is able to match (and even exceed) the performance of existing parallel programming techniques, with the added benefit that the programmer does not have to managed concurrent accesses to shared memory.

7.3.2 Canneal

The Canneal benchmark is a place-and-route simulation that uses simulated annealing to minimize routing costs on chips. The algorithm iteratively finds better solutions by choosing two random elements and swapping them if doing so leads to a more optimal solution. The provided pthread implementation performs swaps in parallel using *atomic pointers* which protect concurrent swaps using an atomic compare-and-swap (CAS) operation. In our SvS implementation, a data parallel consumer task receives integers used to look up pairs of elements in order to perform a swap. If two swaps share an element in common, then SvS prevents these swaps from happening concurrently. Figure 7.6 compares the execution time of the parallel section of Canneal (in seconds, y-axis) for varying number of threads (x-axis) between SvS and the existing pthreads implementation of Canneal. At 8 threads, SvS out performs the pthread implementation by 230 ms. The noticeable difference in performance is due to the pthreads version's heavy use of atomic CAS operations to perform swaps in parallel. SvS, however, avoids the communication and contention overhead of synchronization by only allowing the elements that can be swapped safely to be processed in parallel.

7.4 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 that is notoriously difficult to parallelize [5]. This is because AI involves arbitrary, complex logic, making it difficult to discover where, when and how data accesses occur. Also, the large number of interactions and movements involved in Entity Management means that several modifications may be made to a single object in one frame, where objects may contain arbitrary links and may be stored in several different dynamically linked data structures. These complicating factors, coupled with a potentially large amount of shared memory accesses, make this system a primary concern for parallelization.

We took the approach of Lupei et al [29] 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) that are governed by the following rules:

- Bombs explode reducing the health of citizens and techs within a set radius and if they are 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, ‘line-of-sight’ tests for occlusion are very expensive, thus making them prime candidates for parallelization. Before parallelization, we apply a common optimization employed in game engines: spatial partitioning. In QuakeSquad, the world is divided into a grid where each cell contains a linked list of the entities residing in that cell. As entities move from one cell to another, they are removed from the list of the previous cell and added to the list of the destination cell. Applying this optimization of spatial partitioning prevents each line-of-site test from having to consider every entity in the world.

However, even with this optimization, occlusion testing still dominates computation and so it is the focus of our parallelization efforts. These tests occur most frequently when citizens move from one cell to another and when bombs project fear onto unobstructed entities within their radius of effect. When parallelizing these tests, several concurrent shared memory accesses are exposed.

In the case of bombs projecting fear, we have an **UpdateBombs** producer task that for each bomb does a fast computation to collect all techs and citizens in the bomb’s radius of effect and stores these techs and citizens in separate linked lists. It then sends each of these lists (one for techs and one for citizens) to two separate data parallel consumer tasks. These consumer tasks then perform expensive line-of-site checks to see if the fear should be applied. Because two bombs may have overlapping radiuses, it is possible for two linked lists to contain the same tech or citizen. Therefore accesses to the entities in these linked lists must be protected. Because SvS uses dynamic reachability analysis to detect what memory objects are present in a list, task instances that are accessing linked lists with common entities will not be executed concurrently.

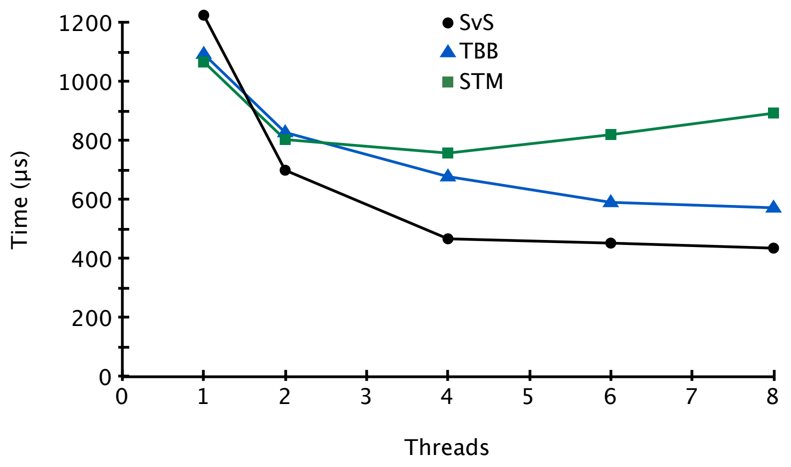


Figure 7.7: Comparison of frame computation times (μs) for SvS, TBB, and STM

Before a citizen moves, it performs line-of-site checks to avoid moving into an area with bombs. In order to perform these checks, it needs access to the adjacent cells surrounding the current location and the adjacent cells surrounding the destination cell. To parallelize this task, an **UpdateCitizens** producer task distributes citizens along with all the linked lists stored at each cell location involved in the computation. Because a citizen moving from one cell to another means it modifies the source and destination lists, it is possible for citizens to either modify the same linked lists or to modify a list that another citizen is reading. Again, SvS uses dynamic reachability analysis to produce a signature for each linked list, and the results are used to generate signatures for task instances.

We also implemented parallelized versions of QuakeSquad using TBB and STM. Figure 7.7 shows a comparison of the average frame computation times (in microseconds, y-axis) for varying numbers of threads (x-axis) between SvS, TBB, and STM. Using SvS, we were able to decrease average frame computation time from 1,226 to 436 μs. Due to the high cost of transaction aborts, STM only scales to 4 threads, after which performance begins to degrade. TBB performs similarly to SvS but has a frame calculation time that is approximately 137 μs slower than SvS at 8 threads due to latency and communication costs of fine grained locking.

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 and its comparison with other techniques demonstrate its ability to utilize 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.

Chapter 8

Related Work

The concept of SvS was first introduced by us in a workshop paper [14], which gave only a high level overview of the idea. A more developed model and implementation of SvS was later presented in [15]. Here, we expanded upon this implementation and provided a more complete and comprehensive description and implementation of SvS.

There are an ever increasing number of parallel environments and language/runtime combinations such as Chapel [19], Cilk [18], OpenMP [3], Gossamer [36] and Intel's Threading Building Blocks (TBB) [2] and Array Building Blocks (ArBB) [1]. However, none of these environments provide automatic mechanisms for protecting shared memory. Instead, they generally focus on providing better parallel abstractions to the programmer (e.g. tasks) that enable the runtime to efficiently break up and map blocks of code to processing cores, while leaving the programmer to manually manage shared memory accesses with the provided tools. Because many of these systems support task or annotation based parallelism, SvS or an SvS-like technique could be implemented in a number of these systems.

The Jade [35] language and Prometheus' Serialization Sets [7] both address shared memory 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 manually 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. SvS also allows for the expression of data parallel operations and automatically detects data dependencies between instances of a data parallel task. Prometheus' Serialization Sets work similarly to Jade, but they are applied to an object-oriented language and protect memory accesses within an object. Shared memory protection using SvS is more general.

While there is a large body of existing work on static dependency analysis and compiler assisted automatic parallelization, OoOJava [25] represents recent work in this field that similar to SvS attempts leverage simple programmer annotations along with static analysis to automatically protect against race conditions. While OoOJava focuses on powerful static analysis techniques, SvS compliments this work by addressing the limitations of static analysis and focuses on dynamic analysis techniques for automatically detecting data dependencies. OoOJava introduces a new static analysis technique, called disjoint reachability analysis [24] that abstracts collections of objects as heap region nodes and statically infers 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 reachability graphs and dynamic reachability analysis to provide an efficient way to maintain and extract complete reachability information.

Many techniques that dynamically address the automatic protection of shared memory are speculative in that they attempt to execute code without explicit synchronization, and "roll-back" or undo operations if a violation is detected (e.g. conflicting accesses to shared memory). Software Transactional Memory (STM) [38, 22, 23] is the most prominent of these techniques. Programmers using STM wrap code that requires protection in an atomic block (i.e. transaction) and the STM system tracks read/write accesses at runtime and automatically resolves any conflicting accesses by aborting and retrying transactions. The key difference between STM and SvS is that SvS determines whether or not two tasks will conflict *before* they are executed, whereas STM detects conflicts during and after execution. This means that STM is less conservative, potentially uncovering more parallelism, but may be subject to expensive rollbacks. Since rollback costs 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 due to the conservativeness of dynamic reachability analysis. This suggests an interesting direction for future work where we could combine SvS with STM, using STM in cases where conflicting accesses to shared memory are rare and SvS in cases where conflicts are more frequent.

Another prominent parallel programming model based on speculative parallelism is the Galois [27, 26, 33] framework. Galios differs from SvS in that it focusses on commutativity (as opposed to data dependency) analysis to automatically ensure a correct ordering of dependent tasks. Also, due to its speculative nature, it is subject to rollback overheads not present in SvS.

Chapter 9

Conclusion and Future Work

Synchronization via Scheduling (SvS) is a new framework for efficiently and automatically protecting concurrent access to shared memory in task graph models. SvS combines static and dynamic analysis in order to detect the potential memory accesses a task will make *before* it is executed, and schedules tasks such that concurrently executing tasks have non-overlapping read/write sets. SvS introduces a new dynamic analysis technique called dynamic reachability analysis that monitors connectivity properties of heap objects in order to determine more precise read/write sets for tasks. The SvS task scheduler also contributes new algorithms for efficiently comparing the read/write sets of (potentially many) task instances, thus enabling parallelism while avoiding conflicting accesses to shared memory. We also present our implementation of SvS and an evaluation of its overhead and performance in practical applications.

While this work demonstrates the feasibility of SvS, there are many opportunities for improvement and future research. One direction is to improve the performance of dynamic reachability analysis using more information from static analysis. For example, if static analysis determines that only a fixed “neighbourhood” of memory objects are accessed indirectly through memory object M, then reachability information needs only be maintained for this neighbourhood, thus reducing the cost of link assignment and false positives due to conservatism in our current implementation. Another source for optimization is better placement of master nodes. For example, instead of spatially determining master nodes, as is done in our current implementation, we could use semantic information to inform master node creation. For example, by using type information, if a list node of type `Node` assigns its `data` link member of type `Entity` to point to an `Entity` memory object, it may be beneficial to elect the `Entity` object to become a master because the parent of the link is a different type. Finally, we currently do not account for signatures in reachability graphs becoming stale. When a link assignment occurs, new reachability information is propagated, but due to the nature of signatures, the old reachability information cannot be "subtracted". One approach to fix this would be to periodically enter a phase that recalculates reachability information for memory objects in the system. This phase could be executed in parallel and infrequently, minimizing the impact on overhead.

We only presented two scheduling domain algorithms, but exploring other strategies for dispatching tasks with non-overlapping read/write sets is part of our ongoing research. Additionally, as the specification of the Cascade Data Management Language matures, we intend to conduct user studies to quantify its potential impact on the parallel programming community.

Bibliography

[1] Intel array building blocks. `http://software.intel.com/en-us/articles/intel-array-building-blocks/`.

[2] Intel threading building blocks. `http://threadingbuildingblocks.org/`.

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

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

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

[6] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Pearson Education, Inc., Boston, MA, USA, 2006.

[7] Matthew D. Allen, Srinath Sridharan, and Gurindar S. Sohi. Serialization sets: a dynamic dependence-based parallel execution model. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 85–96, New York, NY, USA, 2009. ACM.

[8] Lars Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, University of Copenhagen, Denmark, 1994.

[9] Utpal Banerjee. *Speedup of Ordinary Programs.* PhD thesis, University of Illinois at Urbana-Champaign, 1979. Department of Computer Science.

[10] Utpal K. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Norwell, MA, USA, 1988.

[11] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 103–114, New York, NY, USA, 2003. ACM.

[12] Micah Best, Alexandra Fedorova, Ryan Dickie, Andrea Tagliasacchi, Alex Couture-Beil, Craig Mustard, Shane Mottishaw, Aron Brown, Zhi Huang, Xiaoyuan Xu, Nasser

Ghazali, and Andrew Brownsword. Searching for concurrent design patterns in video games. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 912–923. Springer Berlin / Heidelberg, 2009.

[13] Micah J. Best, Shane Mottishaw, Craig Mustard, Mark Roth, Parsiad Azimzadeh, Alexandra Fedorova, and Andrew Brownsword. Schedule data, not code. In *Proceedings of the 3rd USENIX conference on Hot topics in parallelism*, HotPar'11, Berkeley, CA, USA, 2011. USENIX Association.

[14] Micah J Best, Shane Mottishaw, Craig Mustard, Mark Roth, Alexandra Fedorova, and Andrew Brownsword. Synchronization via scheduling: Managing shared state in video games. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, Berkeley, CA, USA, 2010. USENIX Association.

[15] Micah J. Best, Shane Mottishaw, Craig Mustard, Mark Roth, Alexandra Fedorova, and Andrew Brownsword. Synchronization via scheduling: techniques for efficiently managing shared state. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 640–652, New York, NY, USA, 2011. ACM.

[16] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[17] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[18] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30:207–216, August 1995.

[19] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21:291–312, August 2007.

[20] Pascal Felber, Torvald Riegel, Christof Fetzer, Martin Skraut, Ulrich Mller, and Heiko Sturzrehm. Transactifying applications using an open compiler framework. In *In Proc. of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, 2007.

[21] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[22] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, pages 388–402, New York, NY, USA, 2003. ACM.

[23] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

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

[25] James Christopher Jenista, Yong hun Eom, and Brian Charles Demsky. Ooojava: software out-of-order execution. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 57–68, New York, NY, USA, 2011. ACM.

[26] Milind Kulkarni, Patrick Carribault, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. Scheduling strategies for optimistic parallel execution of irregular programs. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 217–228, New York, NY, USA, 2008. ACM.

[27] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 211–222, New York, NY, USA, 2007. ACM.

[28] Daan Leijen and Judd Hall. Optimize managed code for multi-core machines. `http://msdn.microsoft.com/en-us/magazine/cc163340.aspx`.

[29] Daniel Lupei, Bogdan Simion, Don Pinto, Matthew Misler, Mihai Burcea, William Krick, and Cristiana Amza. Transactional memory support for scalable and transparent parallelization of multiplayer games. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 41–54, New York, NY, USA, 2010. ACM.

[30] Mark Marron, Mario Méndez-Lojo, Manuel Hermenegildo, Darko Stefanovic, and Deepak Kapur. Sharing analysis of arrays, collections, and recursive structures. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '08, pages 43–49, New York, NY, USA, 2008. ACM.

[31] Michael 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.

[32] Terence Parr and Kathleen Fisher. Ll(*): the foundation of the antlr parser generator. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 425–436, New York, NY, USA, 2011. ACM.

[33] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM.

[34] James Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[35] Martin C. Rinard and Monica S. Lam. The design, implementation, and evaluation of jade. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20:483–545, May 1998.

[36] Joseph A. Roback and Gregory R. Andrews. Gossamer: A lightweight programming framework for multicore machines. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, Berkeley, CA, USA, 2010. USENIX Association.

[37] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 16–31, New York, NY, USA, 1996. ACM.

[38] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[39] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[40] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM.

[41] Jan Wielemaker. An overview of the SWI-Prolog programming environment. In Fred Mesnard and Alexander Serebenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, december 2003. Katholieke Universiteit Leuven. CW 371.