Research Dissertation

Andrew Wright [40006161]

May 5, 2013

Abstract

Here's an abstract

1 Introduction

This document will investigate how best to analyse the performance of task data flow and work-first based runtime systems for parallel programs taking into account data locality.

These runtime systems use a task data flow model to record and manage dependencies during parallel execution. This approach allows the programmer to concentrate on application development rather than the low level parallelisation techniques. This is useful because processing different tasks on different cores can be tricky, especially if there are implicit dependencies between tasks. They also provide an effective and flexible way for the scheduler to maximise the number of tasks that can be run at one time across the multiple processors. Using this method, we can rival highly tuned libraries with minimal tuning effort [13]. The other aspect of these schedulers is their provably good work first principle. Which will be discussed in further detail in section 1.1.2. The scheduler analysed is the first scheduler of this kind and is called 'Swan' [15].

To give a proficient introduction to the problem area I will discuss why it is useful to combine task dependencies with the work-first principle by looking at schedulers which focus on one method in particular and then present the unified solution which this research will analyse. It is important to understand some of the finer points of these run time systems as they will shape how the analysis tool is built. I will then discuss the existing methods for analysing parallel runtime systems focusing on methods which were considered in this research. The rest of the paper will go onto set out my approach and results.

1.1 The Problem Area - Parallel Scheduling

1.1.1 Task Based Parallel Programming

An implementation of task based parallel programming is *SMPSs* [13]. It attempts to provide a way for the programmer to avoid the arbitrary concerns of data dependencies and parallelisation as much as possible allowing them to focus on the problem at hand. To this effect, it is implemented as an extension of the C programming language with additional pragmas to identify functions as candidates to be run in parallel.

SMPSs uses the idea of a task dependency DAG (directed acyclic graph). Where each vertex is a task and the edges represent dependences between tasks. SMPSs uses what we call a non-strict DAG which means that one task can have a dependency on another which isn't on the same ancestral tree. We will discuss this concept further in Section 1.1.3.

SMPSs adopts a "work-stealing" approach whereby if a thread has no more tasks to execute, either on its own queue or the system queue, it will work on tasks from another thread's queue in FIFO order. Work-stealing is also implemented in Swan and Cilk as we will see.

A potential downside of SMPSs, compared to work-first schedulers (discussed later), is that it favours the critical path over work. Its master thread creates all dependent tasks for a function leaving the scheduler with an excess of idle tasks. There is a theory set out by the developers of Cilk [6] which proves that, based on an acceptable degree of parallel slack, it is actually more efficient to adopt a "work-first" approach. This means sacrificing critical path time i.e. more scheduling overheads, in order to reduce the amount of time it takes to complete a run on one processor. I will talk more about how this is achieved in the next section.

1.1.2 Work-First Scheduling

Cilk is where the idea of work-first scheduling originated and an implementation of it can be found in Cilk-5 [6]. Cilk is a parallel extension of C. The parallelism is introduced particularly through spawn and sync statements. Spawns allow a procedure to be run in parallel with its caller and sync requires the procedure to stall until the completion of all spawned procedures. The aspects of Cilk which we are particularly interested in, to provide a background on Swan, are the work-stealing and work-first properties of Cilk. Cilk uses a provably good work-stealing algorithm developed in Cilk-1 [2, 3, 10].

This "work-stealing" algorithm is implemented by giving each processor a ready deque (a double ended queue) data structure of call frames (a stack of related frames). Call frames can be inserted from the bottom and removed from either end. The processor uses the ready deque as a call stack, pushing and popping from the bottom. Processors which steal from this processor remove a call frame from the top. The reason that it steals from the top of the victim's (the processor from whom the call frame is being stolen) deque is to avoid contention and increase the probability of data locality.

The main development of Cilk-5 is a focus on the "work-first" approach to scheduling. We can characterise a Cilk computation by two quantities: its work, which is the total time needed to execute the computation serially, and its critical-path length, which is its execution time on an infinite number of processors. As its name suggests the "work-first" principle is:

> "Minimise the scheduling overhead borne by the work of a computation. Specifically, move overheads out of the work and onto the critical path."

In order for the work first principle to be effective we must assume that there is ample "parallel slackness". This means that the number of processors required to achieve full parallelism for the program is sufficiently greater than the number of available processors. For a mathematical definition of 'work', 'parallelism' and 'critical path' please refer to section 2.2. The reason that Cilk needs this slackness is to ensure that the extra expense of less efficient scheduling algorithm can be made up for by the efficiency of the work done per processor.

As previously mentioned Cilk enforces a fully strict DAG which means that direct dependencies cannot be created from one task to another which isn't on the same ancestor tree. This is unlike Swan where the task data flow structure allows dependencies between tasks on differing ancestor trees. See Figure.1 for a comparison. Because Cilk uses a strict DAG it means that it is best suited for applications that can adopt a 'divide and conquer' approach to problem solving and is most effective when there is regular parallelism throughout the program. It isn't possible to write some programs, which could benifit from parallel execution, in a this way for example pipelines.

1.1.3 A Unified Scheduler

The scheduler which this research is focused on is the Swan scheduler. Swan combines work-first scheduling (the approach which Cilk adopts) with dependency-aware scheduling (as seen in SMPSs).

Swan's language is similar to Cilk and the dependency aspect of the scheduler is realised at the object level. The concept of a versioned object is introduced here. These objects are passed to tasks as'indep', 'outdep' and 'inoutdep' arguments depending on their memory access type. These are defined if a given argument has a memory access mode of: input, output or input/output for a task. If a task has no dependency information associated with it, it is run as an unconditional spawn and is executed as it would be with Cilk.

The swan scheduler allows nesting of fork/join parallelism and task graph parallelism. As well as task graphs which are arbitrarily nested. Nested task graphs require care because if the same object is being referenced by both a parent and a child the versioned object meta-data must track the correct dependencies. Swan handles this by creating new counters in the child's object so that dependencies are tracked independently.

1.2 Scheduler Analysis

The aim of this research is to analyse parallel schedulers so I will now describe the most relevant work which has been done in this area. There are several analysing tools for parallel systems and the majority follow a similar process which debugs and analyses a post-mortem parallel program. This process is:

1. *Data collection :* Software is used to collect information about the execution of the parallel program (hardware can be used too).
2. *Data analysis :* Once the stream of events is collected, the analysis step can be started to order, filter, calculate some statistics or simulate a parallel machine.

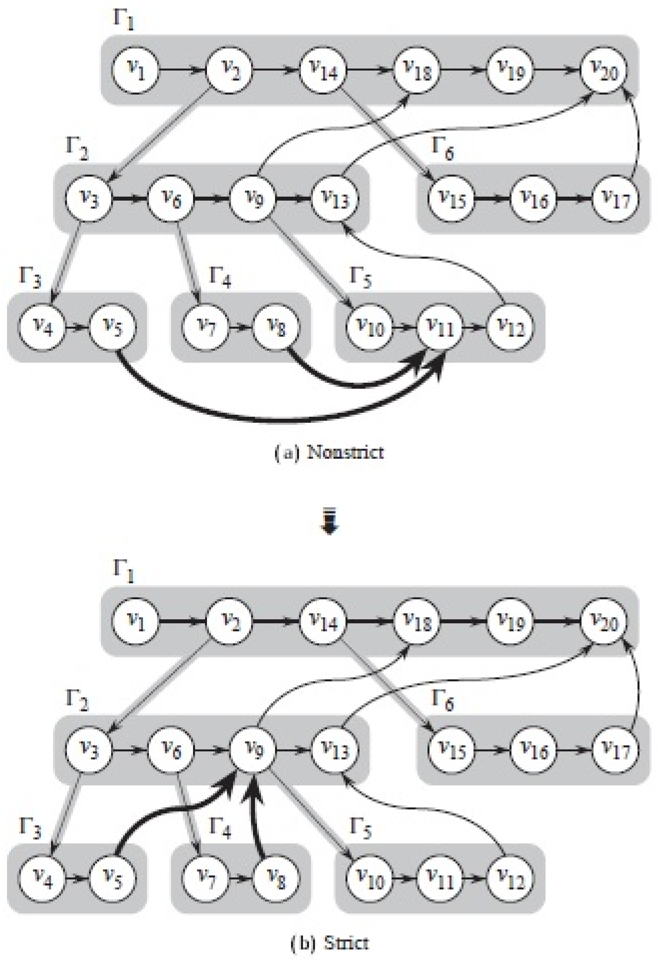

Figure 1: Comparison between strict and non-strict DAGs. Accredited to PhD Thesis by Robert D. Blumofe [2]

3. *Display :* Visualise the result provided by the analysis step.

I will look at each of the scheduler analysing tools in terms of these three areas.

1.2.1 General Parallel Program Analysis

There have been tools developed which can analyse the execution trace of a given parallel scheduler. Paraver is such a tool, it visualises parallel execution and displays it in a customisable way. Paraver looks at the actual execution time as a basis of its analysis. It uses various windows to display information to the user all of which is customisable so the user can view the collection of windows which most suits their purpose. These update as the user traces through the program, representing what is happening through various views. The ability to filter and to vizualise execution is useful however a lot of effort is required from the user in order to get a meaningful metric to help them improve their application. Paraver is also only interested in the display aspect of the analysis process. The data collection and exporting have to be applied to the program before Paraver is able to display it from a simulation file. This collection is very expensive and collecting a trace file can slow a runtime system down by an order of magnitude - leaving the user with an inaccurate measurement.

1.2.2 Work First Analysis

Cilkview is an analysing tool for Cilk which is useful to this research because Swan extends Cilk and benefits from much of the same analysis.

Cilkview first collects a program's parallelism information during a serial execution of the program code. It uses meta data in Cilk++ to identify the parallel control constructs in the executing application which allows it to construct a data dependency DAG. As this meta data is built into the Cilk++ binary it imposes no overhead on performance in a normal production environment. Instead of capturing exact timing measurements Cilkview uses instruction count as a surrogate of time. They feel that parallelism estimates accurate to within a binary order of magnitude suffice to diagnose problems of inadequate parallelism.

Cilkview focuses on three areas when analysing the results from *work, span and burdened span.* I will look at how each of these are calculated in turn and discuss why they are important.

Work is the total amount of time spent in all the strands. Let T_P be the fastest possible execution time of the application on P processors. Since the work corresponds to the execution time on 1 processor, we denote it by T_1. One reason that work is an important measure is that it provides a lower bound on P-processor execution time:

$$T_P \geq T_1/P$$

This is at least how much time it takes for P processors to execute the total amount of work done. Because theoretically P processors can do P instructions at one time. Using Cilk T_1/T_P is the most *speedup* that can be gained. Speedup is how much faster a program can run on a given environment this can be intuitively written as a program which is run on a machine with 5 processors can run 5 times as fast. This level of speedup is known as linear speed up.

The second area which Cilkview analyses is span. Span is also called the critical path and is the longest amount of time which is taken for a program to execute over any path of dependencies in the DAG. The critical path is the theoretical fastest time which a program could be run on an unlimited amount of processors. Due this it is denoted as T_∞ this also provides a bound on P processor execution time:

$$T_P \geq T_\infty$$

Cilkview uses the work and span to calculate the parallelism of a program T_1/T_∞. The parallelism is the average amount of work along each step of the critical path. From parallelism we can work out what the maximum number of processors are on which linear speed-up can be achieved for a particular program. Linear speed-up cannot be achieved on any number of processors greater than the parallelism. This information allows Cilkview to determine how scalable a program is.

The third aspect which Cilkview analyses is the burdened span. Although the execution time is reduced by tasks being run on various processors the overhead and book-keeping which is introduced to schedule these executions needs to be taken into account. This is what the burdened DAG does. In Cilkview a constant called the *span coefficient* δ is multiplied by the span to take into account the work stealing overhead which includes the book keeping and the cost of cache misses in order to migrate the stolen task's working set.

$$T_P \leq T_1/P + \delta T\infty$$

The burdened DAG places a burden on each continuation and return edge of the DAG. Cilkview computes the *burdened span* by finding the longest path in the burdened DAG.

If we let T_1 be the work of an application program, and let $\hat{T}_\infty$ be its burdened span. Then, a work-stealing scheduler running on P processors can execute the application in expected time:

$$T_P \leq T_1/P + 2\delta\hat{T}_\infty$$

Cilkview uses this burdened model to give a more accurate estimated lower bound on speedup. This is, a work-stealing scheduler running on P processors achieves speedup at least:

$$\frac{T_1}{T_P} \geq \frac{T_1}{T_1/P + 2\delta\hat{T}_\infty}$$

A detailed proof of this can be found in Cilkview's research paper [8].

Cilkview displays the following information to the user: amount of work done, the span, the burdened span, the parallelism, the burdened parallelism, number of spawns, number of syncs and the average maximal strand (which is how much work is down in between scheduler operations). Cilkview then gives estimated speedups for 2 - 32 processors. Cilkview also uses graphs to display this data in a meaningful way.

1.3 Other Metrics

There are other metrics which have been used to measure the performance of parallel applications, examples of these are: True Zeroing[11], Gprof[7] and Quartz NPT[1]. An overall study and comparison of each method has been carried out [9] and suggests that critical path analysis, as discussed in section 1.2.2, is most effective.

2 Investigation Method

As alluded to above the most useful metrics which currently exist to analyse parallel programs revolve around the program's span also know as it's *Critical Duration.*

2.1 Critical Duration Analysis

The critical duration is the shortest possible time taken for a given parallel program to execute. This is often represented as T_∞ as it is equivalent to the execution time of a program if it were run on an infinite number of processors. The critical duration combined with the amount of total work a program does allows us to calculate the degree of parallelism which, as seen in Section 1.2.2, will determine how scalable a program is. Scalability is a vital measurement in regards to parallel programming as the primary goal of parallel programming is to allow performance to increase as the number of processors available increase.

2.2 Measurement of Time

When considering how this research would measure time two options were considered. The first is instruction count which is the method that Cilkview [8] adopts. They argue that because the advantages of adding another processor would only be realised if there was a large difference between parallelism and actual speed-up there is no need for a precise measurement of time.

The other option was to attempt to take the actuate ‘wall clock’ time measurements. The benefits of this include the fact that time is easily relatable to, meaning a programmer can quickly evaluate the output of the program and understand it. It can also be easily validated by external counters unlike instruction count. It was because of these reasons that this research decided to adopt this definition of time.

Time was captured at a millisecond level to give an accurate result. This was chosen over cycles in order to increase portability and to ensure that changes brought about by CPU power saving options, such as CPU frequency being reduced, wouldn’t affect the measurements. The method ’gettimeofday()’ from the sys/time.h library was used over RDTSC [**?**] due to potential issues regarding CPU timers not being synchronised over multiple cores.

A negative of this method of timing is the fact that it is it will take into account the entire host environment rather than the individual process. So, if the user is running the program on a machine which is heavily loaded, meaning it has a lot of running processes, the time measurements could be slower than on the same machine when it is dedicated to the program being analysed. This could affect the level of parallelism to be expected.

2.3 Non-object Dependent Critical Duration Analysis

The critical path algorithm devised in this research can be broken into two parts. Firstly the work first aspect, whereby object dependencies are ignored and, secondly, the object aware aspect. We will now discuss the details of both these aspects in turn.

Task Begins Execution
The algorithm for obtaining the critical duration at run time without object dependencies can be seen in Figure 2. At first we calculate the parent's duration, this is done by subtracting the current time from the last set 'start time' of the parent. This will either be the time at which parent was first executed or the time it was last resumed from a returning task. We then add the duration to the accumulation of work done by the parent to get the parent's current work done total. The parent's critical path is then updated. All work that parent has done on the current strand, that it's duration, will get directly added into it's critical path. The spawned/called task is assigned the parent's current critical duration instead of set to 0 as with the work done. We will see the reason behind this when object dependencies are introduced in Section 2.4.2.

Task Finishes Execution
When a task is finished and returns to it's parent it's duration, work done and critical duration are calculated. The calculation of the critical duration takes into account the child's critical duration also. If a task has spawned children who have had a longer critical duration that the task its self then their critical duration should become it's critical duration. Once this has been determined the task's duration should be added to the critical duration because, like the parent tasks, the duration of a given task will always be a part of it's critical duration.

The parent's critical duration is then calculated. This is determined by first deciding if the task that was executed can possibly be executed in parallel, that is, if it was called using a 'spawn' method. If it can't then then the amount of work done will be added directly to its parent's critical duration.

If the task can be run in parallel then its critical duration will be compared to the current highest child's critical duration for that parent. The child's critical duration is necessary because when the parent reaches a sync or ends, it's critical duration will be determined by comparing it's current critical duration and the highest critical duration of each of its children, allowing analysis tool to correctly represent spawn and sync statements. This will give us the final critical duration of any task.

Finally, when a task is finished execution its parent's total work is calculated. As this value is accumulation of all work done in a given strand and its children, the work done is simply added to the parent's current value for work done.

Sync Statement Called
When a sync statement is called execution is stalled until all of the spawned tasks have returned. At this point, because we will record critical duration in serial execution, we know that all spawned tasks will have finished execution. We now want to re-evaluate the task's critical duration comparing it to the highest critical duration of it's children. The highest value will be the final critical duration of the task.

2.3.1 Algorithm Design Choices

The Cilkview scalability analyser mentioned in Section 1.2.2 calculates critical duration in a similar way to the algorithm in Figure 2. The most notable difference between the algorithm described here and Cilkview's implementation is we ensure that the task's work done is self contained, reducing the amount of coupling between a child and it's parent. The Cilkview method resets the parent's work to 0 when each task is spawned. Therefore a given task will contain the total work done by the program up to that point. This also means that until a task returns the parent's value for the amount of work it has carried out up to that point will be incorrect. Our approach has avoided this because, for future development, we recognise it would be advantageous to ensure that the amount of coupling between child and parent is as low as possible. We will explore the reasons behind this later in the paper.

```
when each task is ready to execute:

   calculate parent's duration
   calculate parent's work done [p.wd + duration]
   calculate parent's critical path [p.cp + duration]

   assign task's critical path = parent's critical path

when a task is finished executing:

   calculate task's duration
   calculate task's work done [t.wd + duration]
   calculate task's critical duration:
         t.cd = MAX(t.child_critical_duration , t.cd)
         t.cd = t.cd + duration

   if the task hasn't been spawned:
        calculate parent's critical duration:
           p.cd = p.cd + duration
   else:
        calculate parent's child critical duration:
           p.ccd = MAX(p.ccd, t.critical_duration)

   increase parent's work done by task's work done

when a task is synced

   parent's critical duration is calculated [MAX(p.cd, p.
      child_critical_duration)]
```

Figure 2: Algorithm describing the critical duration as captured without object dependencies

2.3.2 Implementation

Data Capture and Storage
There were several metrics which needed to be recorded in regards to each task. Thankfully in swan there exists a metadata object relating to each task. This was extended in order to accept the information we needed with access functions introduced for each member. There were however, some elements which need to be accessed, which required a slight change of structure to way in which task metadata was constructed. One such example is linking the metadata regarding a task to the metadata of its parent. Before the development of this tool the only place to access a task's parent was through it's frame. We decided that it was logical to have a pointer stored to a task's parent in its metadata also, allowing us to access the parent from the object we had available. Some issues arose also from the availability of information regarding whether or not a task had been spawned. The only place that this was available originally was when invoking a stack frame. In order to get around this we were able to pass a flag into the initialiser for task metadata as this will never change.

Swan's Hooks
Within the scheduler there are several places which were useful in order to implement the designed algorithm.

Tasks Begin Execution
A function exists which is called when all arguments are issued to a task. When the task has no object dependencies then the argument issue function will be executed immediately before the task begins to execute this is because, when there are no arguments there is no chance that the task will have to wait on arguments completing. This is where the function called at the beginning of task execution is run from.

Tasks End Execution
A function exists which is called when a task releases it's arguments. At this point the task will completed its execution and therefore the system should be aware that the arguments it was using, in this case none, are available to be executed. This is therefore where the code to record a task finishing a its execution is called from.

Sync Statement Called
Because different functionality is used when a task syncs, that is the task's critical duration is calculated, it should be called from a different section of the scheduler. There is a procedure which handles calls to

the sync statement from the context of the parent's frame. We can then use the frame to access the metadata associated with the task - adjusting it accordingly.

2.4 Object Dependent Critical Duration Analysis

In order to ensure that critical duration analysis is effective when analysing a unified scheduler such as swan it is important that the algorithm, which was introduced in the previous section is modified to handle these objects.

2.4.1 Algorithm Modifications

The modification to the algorithm previously introduced is twofold as seen in Figure 3. The first addition is implemented before a task is executed. The algorithm will check if the task has any object dependants which are of the type indep or inoutdep. This means that the task has an read dependency on the objects. If the the task does have any object dependants of this type it will re-evaluate it's critical duration comparing its critical duration with the object's and assuming the highest. This is effectively saying that the task can't be run until the object is ready.

The second modification which is made to the algorithm is when at task has completed its execution. If the dependency type is either outdep or inoutdep that is the task will write to the object then the object will be effected when the task ends. If the task does have said dependency then the object's critical duration will be evaluated and assume the highest critical duration of its self or the task which is dependant on it. This can be phrased as the task object not being deemed 'ready' until the write tasks are finished writing.

A sync statement will not affect the critical duration of an object.

2.4.2 Object Dependencies

As you will notice in Figure 4, where numbered circles represent tasks and round cornered rectangles dependent objects, there are 3 types of dependencies which are handled by the critical duration analysis. These are indep, outdep and inoutdep. We will now discuss how each is handled by swan and how that effects the analysis. In order to

```
when each task is ready to execute:

    if task has in/inout dependee objects:
        set task's critcal path = max critical path of dependee
            objects

when a task is finished executing:

    if task has has out/inout object dependees:
        set object's critical duration [MAX(obj.cd, task.cd)]
```

Figure 3: Algorithm describing the critical duration as captured with object dependencies

review these we will study Figure 4 which is a visual representation of the dependencies in test program dependants.cc which can be found in Appendix A.

- *Out Dependencies*: Outdeps specify relationships between tasks and objects in the case that a task will write to that object. An example of this can be seen in Figure 4 where task $T0$ has an out dependency to the object. As we discussed in the previous section an outdep will not affect a task's critical duration. We can see the reason for this by studying Figure 4. Consider for instance the true statement that $T2$ could be executed before or even in parallel with $T1$. Initially we may be inclined to disagree with this statement as $T2$'s out dependency relationship with the object could mean that, in the event of $T1$ reading from the object after $T2$ has written to it, $T1$ would read incorrect data. The reason we can be confident that this won't happen is that, in such a scenario, the scheduler will rename the objects, allowing parallelism.

  As shown in the algorithm the out dependency will affect the object's critical duration as any task with an in dependency to the same object, called after the out dependent task, will have to wait for the object to be ready. This can be illustrated in Figure 4 by studying the interactions between task $T0$ and task $T1$. $T1$ has an out dependency on the object which $T0$ is writing to. This means that, in order to ensure correctness, $T1$ cannot execute until $T0$ has completed. To reflect this in the analysis tool we ensure that the object's critical duration is at least as

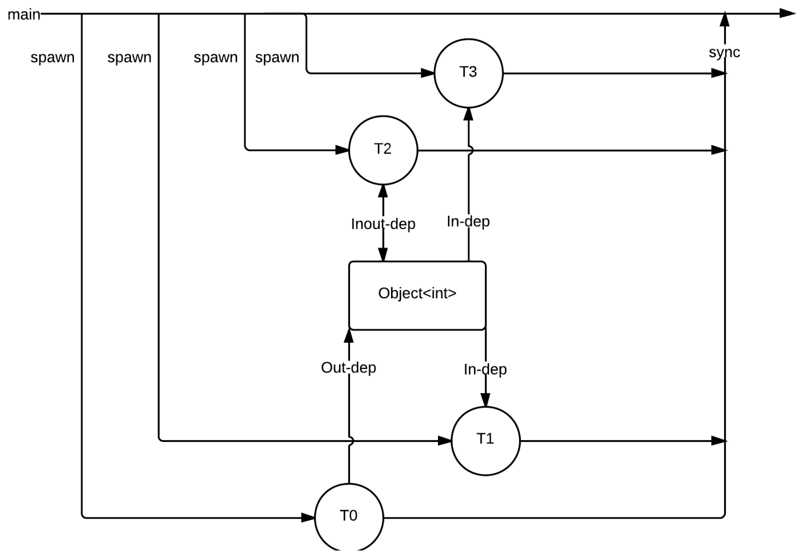

Figure 4: An illustration of object dependencies found in a task graph

long as $T0$'s thus ensuring that $T1$'s critical duration is at least as long as $T0$'s.

- *In Dependencies*: An Indep relationship is one in-which the task depends on the output of an object. This can be seen in the case of task $T3$. If we weren't taking object dependencies into account we could say that, because $T3$ is spawned from the main method, its critical duration would be equal to the length of the main method up to that point. However this is not the case because of the in-dependency which means that if $T3$ is to be correct it must have a critical path at least as long as the object i.e. the object's writer must have finished writing to it before the object can be read. In this case, because $T2$ is the task responsible for writing to the object before it can be read by $T3$, the critical duration of $T2$, the object and $T3$ will all be equal.
- *In-Out Dependencies*: An in-out dependency encompasses both types of dependencies described above. It will affect both the task's critical duration and the object's critical duration. This is because its in-dependency requires access to the most up to date value of the object and its out-dependency forces the critical duration of the object to be at least as long as this task's critical duration.

It is now apparent that it is essential for a task not only to be aware of its own critical duration but also the critical duration of the entire program. If this were not the case, it would not be possible for the analysis tool to correctly set the object's critical duration.

2.4.3 Swan's Hooks

Similarly to the functions which handled the non-object dependent portion of the critical duration analysis, the object dependent functions are also executed when arguments are issued and released. In order to handle object dependency swan uses functors, a C++ construct which can be thought of as "executable objects" that we run on each argument in a parameter pack. For each parameter a template override will be matched in the functor executing the corresponding functionality. This is useful because, in the initialisation of each functor, the generic task-begin or task-end code, that is the code which is necessary for the critical duration analysis without objects, can be run. The object specialisations can then override any necessary values such as the critical duration. This is useful because it allows the program to have a single task beginning function and a single task ending function regardless of whether the task has object dependencies or not.

References

[1] T.E. Anderson and E.D. Lazowska. *Quartz: A tool for tuning parallel program performance*, volume 18. ACM, 1990.

[2] R.D. Blumofe. *Executing multithreaded programs efficiently.* PhD thesis, Massachusetts Institute of Technology, 1995.

[3] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. *Cilk: An efficient multithreaded runtime system*, volume 30. ACM, 1995.

[4] C. Ding and T. Chilimbi. A composable model for analyzing locality of multi-threaded programs. Technical report, Technical Report MSR-TR-2009-107, Microsoft Research, 2009.

[5] M. Frigo, P. Halpern, C.E. Leiserson, and S. Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Proceedings of the*

twenty-first annual symposium on Parallelism in algorithms and architectures, pages 79–90. ACM, 2009.

[6] M. Frigo, C.E. Leiserson, and K.H. Randall. The implementation of the cilk-5 multithreaded language. *ACM Sigplan Notices*, 33(5):212–223, 1998.

[7] S.L. Graham, P.B. Kessler, and M.K. Mckusick. Gprof: A call graph execution profiler. *ACM Sigplan Notices*, 17(6):120–126, 1982.

[8] Y. He, C.E. Leiserson, and W.M. Leiserson. The cilkview scalability analyzer. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 145–156. ACM, 2010.

[9] J.K. Hollingsworth and B.P. Miller. Parallel program performance metrics: a comprison and validation. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 4–13. IEEE Computer Society Press, 1992.

[10] C.F. Joerg. *The cilk system for parallel multithreaded computing.* PhD thesis, Citeseer, 1996.

[11] B.P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S.S. Lim, and T. Torzewski. Ips-2: The second generation of a parallel program measurement system. *Parallel and Distributed Systems, IEEE Transactions on*, 1(2):206–217, 1990.

[12] Stephen L. Olivier, Bronis R. de Supinski, Martin Schultz, and Jan F. Prins. Characterizing and mitigating work time inflation in task parallel programs. 2012.

[13] J. Pérez, R. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. pages 142–151, 2008.

[14] Andreas Sandberg, David Black-Schaffer, and Erik Hagersten. Efficient techniques for predicting cache sharing and throughput. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 305–314, New York, NY, USA, 2012. ACM.

[15] H. Vandierendonck, G. Tzenakis, and D.S. Nikolopoulos. A unified scheduler for recursive and task dataflow parallelism. pages 1 –11, oct. 2011.

A dependants.cc

```
#include "wf_interface.h"

using obj::object_t;
using obj::indep;
using obj::outdep;
using obj::inoutdep;

void task0( outdep<int> out ) {
    printf( "Task 0 start %p\n", out.get_version() );
    sleep(2); // secs
    out = 3;
    printf( "Task 0 done %p\n", out.get_version() );
}

void task1( indep<int> in ) {
    printf( "Task 1 start %p\n", in.get_version() );
    sleep(1);
    printf( "Task 1 done %p\n", in.get_version() );
}

void task2( inoutdep<int> inout ) {
    inout++;
    printf( "Task 2 increments to %d %p\n", (int)inout,
        inout.get_version() );
}

void task3( indep<int> in ) {
    printf( "Task 3 reads %d from %p\n", (int)in, in.
        get_version() );
}

int my_main( int argc, char * argv[] ) {
    object_t<int> v;
    spawn( task0, (outdep<int>)v );
    spawn( task1, (indep<int>)v );
    spawn( task2, (inoutdep<int>)v );
    spawn( task3, (indep<int>)v );

    ssync();

    return 0;
}
```