NTNU
Norwegian University of
Science and Technology

Extending OMPT to Support Grain Graph Visualization

Peder Voldnes Langdal

Master of Science in Computer Science
Submission date: June 2017
Supervisor: Magnus Jahre, IDI
Co-supervisor: Ananya Muddukrishna, IDI

Norwegian University of Science and Technology
Department of Computer Science

Peder Voldnes Langdal

Extending OMPT to Support Grain Graph Visualization

Master thesis,
spring 2017

Department of Computer Science
Faculty of Information Technology and Electrical Engineering

Problem Description

The grain graph is a recent visualization method for OpenMP programs. To implement it, it is necessary to record a large number of properties per task and parallel for-loop chunk instance. Its authors used an in-house runtime system called MIR to do so.

The OpenMP Tools (OMPT) API is an upcoming extension to OpenMP for performance analysis. It has been shown that grain graphs can be generated from data obtained via OMPT, but that certain properties must be extracted with methods that are burdensome, that can incur significant performance overhead, and that could be more accurate if the interface supported obtaining these properties directly. Previous work indicated that these problems could be remedied by extending OMPT.

The master thesis will seek to investigate the ramifications of adding the proposed extensions to the OMPT specification. It should provide answers to the following two high-level questions, both motivated by the design objectives of OMPT: Does supporting the extensions require a notable increase in implementation complexity? Can the extensions be implemented without significant performance overhead?

To answer these questions, the student will look at how task creation and parallel for-loops are implemented in relevant OpenMP implementations such as GCC, Intel, and LLVM. Then, the student will implement the extensions in an open source OpenMP implementation with existing OMPT support, one example being the Intel OpenMP runtime. Next, performance overheads will be measured, possibly through a combination of micro-benchmarks and application benchmark suites.

Abstract

Because of physical constraints, performance gains of single-core processors has come to a halt. Computer architects have responded by adding multiple processor cores to their designs. However, for continued performance gains, multi-core designs require multithreaded applications. Manually managing individual threads becomes burdensome for large applications, and programmers therefore opt to use interfaces that abstract some of this complexity. OpenMP is one such interface. It is an industry-standard for parallel shared-memory programming.

There is currently an ongoing effort to add a profiling interface called the OpenMP Tools (OMPT) API to the upcoming OpenMP 5.0 specification. OMPT will allow creating portable, high-quality performance analysis tools for OpenMP programs.

Grain graphs is a recent visualization that simplifies OpenMP performance analysis. It has previously been found that the instrumentation callbacks of OMPT are almost sufficient to generate the data needed by grain graphs. However, OMPT does not describe events to measure the duration spent creating tasks, or tracing the execution of parallel for-loop chunks.

In this thesis, I propose extensions that provide the necessary descriptions, and evaluate the performance impact of these extensions in the LLVM/Clang toolchain. My evaluation shows that the overheads are low. Benchmarks from the EPCC OpenMP micro-benchmark suite provoke up to 3% increased overhead in the most important scenarios. Most HPC workloads from the BOTS and SPEC OMP2012 application suites don't see any change in execution time. While the proposed extensions are motivated by grain graphs, they can be used by other profiling methods as well.

Sammendrag

På grunn av fysiske begrensninger har ytelsesveksten til enkeltkjerneprosessorer stoppet opp. Datamaskinarkitekter løser dette problemet ved å inkludere flere prosesseringskjerner i sine design. Flerkjerneprosessor avhenger imidlertid av programmer som kan bruke mange tråder for å oppnå forbedret ytelse. I store programmer er det tungvindt å manuelt håndtere individuelle tråder selv. Many programmerere bruker derfor grensesnitt som abstraherer bort noe av denne kompleksiteten. OpenMP er et slikt grensesnitt. Det er en industri-standard for parallell programmering med delt minne.

For tiden pågår det et arbeid med å legge til et grensesnitt for ytelsesprofilering, ved navn OpenMP Tools (OMPT), til den kommende spesifikasjonen OpenMP 5.0. OMPT vil gjøre det mulig å lage plattformuavhengige, høy-kvalitets verktøyer for ytelsesanalyse av OpenMP-programmer.

Graingrafen er en nylig utviklet visualisering som forenkler ytelsesanalyse av OpenMP-programmer. Det har tidligere blitt vist at OMPT sine tilbakekall for instrumentering nesten tilfredsstiller de databehovene som Graingrafer har. Uheldigvis beskriver ikke programmeringsgrensesnittet hendelser for å måle tid brukt på å opprette OpenMP-oppgaver, ei heller går det an å spore utførelsen av parallelle for-løkke-deler.

I denne avhandlingen foreslår jeg utvidelser som tilfører de nødvendige beskrivelsene. Videre blir ytelsesvirkningen av utvidelsene evaluert i LLVM/Clang-verktøykjeden. Min evaluering viser at de påførte kostnadene er lave. Tester fra EPCC OpenMP mikrotestsamplingen viser opp til 3% økte ytelseskostnader i de viktigste scenarioene. De fleste høyytelsesprogrammer fra programpakkene BOTS og SPEC OMP2012 kjører like raskt med og uten utvidelsene. De foreslåtte utvidelsene er hovedsaklig motivert av Graingrafen, men kan imidlertid med fordel brukes av andre ytelsesprofileringsmetoder også.

Preface

This thesis is submitted to the Norwegian University of Science and Technology.

Acknowledgement

I would like to thank my main supervisor Ananya Muddukrishna for his excellent guidance and invaluable feedback throughout the thesis work. I would also like to thank my formal supervisor Magnus Jahre for helpful assistance and feedback along the way. Throughout this work, I have also been in contact with some keen individuals in the OMPT mailing list, particularly Joachim Protze (RWTH Aachen), Jonas Hahnfeld (RWTH Aachen), and Harald Servat (Intel). I would like to extend my gratitude to these individuals for helpful comments and suggestions regarding my proposed OMPT extensions. I would also like to thank Sergei Shudler (TU Darmstadt), who heard about my work and pointed out that it could be useful for data race detection. Lastly, I thank my family and friends for all their support and encouragement throughout my studies.

Peder Voldnes Langdal
Trondheim, June 11, 2017

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, I introduce background information on OpenMP and grain graphs to explain why I extended the OMPT interface. Then, the problem description is used to derive a set of requirements the thesis should fulfil. I then describe my contributions, before finally giving an outline of the thesis.

1.1 Background and Motivation

Because of physical constraints, performance gains of single-core processors has come to a halt. Modern computers combat this by increasing the number of processor cores. To take advantage of this, computer programs must be written to perform its computations in parallel. However, writing manual, low-level parallel code is cumbersome. It is often preferable to instead use an application programming interface (API) to simplify this process, one example being *OpenMP*. OpenMP is an industry-standard for parallel shared-memory programming [1]. It is an compiler-assisted API. By inserting OpenMP directives into otherwise serial code, compilers output parallel programs that are scheduled during execution by runtime systems.

Parallel programs are harder to understand than serial programs for a multitude of reasons. To create efficient and correct programs, one must reason about work partitioning and parallel access to shared resources [2]. Programmers often rely on tools that simplify this process.

The *grain graph* is a recent visualization method for OpenMP programs [3]. It displays a fork-join view of a program's tasks or for-loop chunks, collectively termed grains. Derived metrics such as parallel benefit can be visually encoded in the graph, guiding optimization decisions. The current reference prototype uses an in-house OpenMP runtime named MIR [4] to collect all necessary data during program execution.

I have earlier found that the upcoming *OpenMP Tools API* (OMPT) can provide almost all the data necessary to generate grain graphs [5]. However, a few key measurements are impossible to obtain through OMPT. Specifically, grain

graphs require measuring task creation time and tracing for-loop chunk events. It is desirable to collect the required data through OMPT because it is on course for inclusion into the OpenMP 5.0 specification. This means that one could generate grain graphs for all programs that use an up-to-date, standards-compliant implementation of OpenMP, instead of being bound to the MIR runtime.

In this thesis, I propose extensions to OMPT that provide the missing descriptions. I implement the extensions in a standard toolchain, and evaluate their overheads using the Schedbench and Taskbench micro-benchmarks from the EPCC OpenMP micro-benchmark suite [6, 7], and HPC workloads from the BOTS [8] and SPEC OMP2012 [9] application suites. My evaluation shows that the overheads are low. The Schedbench for-loop scheduling micro-benchmark reports up to 3% overhead in the two most important scenarios: when no tool is attached, and when a no-callback tool is attached. Likewise, Taskbench shows that task creation overheads stay below 3% in the same two scenarios. Applications from the BOTS and SPEC OMP2012 application suites generally don't see any change in execution time, but the worst performing application, Health, runs 2% slower. Although the extensions are motivated by grain graphs, the events they describe are general and can enable cost-effective, precise measurements in other profiling tools as well.

1.2 Project Goals

The problem description mandates the goal of the thesis. This goal is summarized below. Further dissection of the problem description yields a set of requirements for this thesis, which are also presented here:

Goal Extend OMPT to provide all data required by grain graphs.

Requirements

R1 Introduce the upcoming OMPT API, including its design objectives and architecture.

R2 Design the necessary extensions in a way that is consistent with the existing interface and its design objectives.

R3 Implement the proposed extensions and evaluate their impact on (1) implementation complexity and (2) program performance.

Based on the problem description, a set of optional tasks that could advance the thesis have been identified:

OR1 Perform a survey of the performance profiling field to assess if and how the extensions are useful to others.

OR2 Find an appropriate venue for proposing the extensions to the OpenMP Architecture Review Board, and solicit feedback.

1.3 Contributions

The work in this thesis carries the following contributions:

- I have showed that there is a very real demand for extended instrumentation in OMPT by surveying relevant profiling papers.
- I have explored the design space, and crystallized extensions that are consistent with OMPT design objectives and that allows generating grain graphs.
- A prototype OpenMP runtime system based on LLVM OpenMP has been developed and made open source. Along with this, I have modified LLVM Clang to support an extension that requires new code generation.
- The developed prototype is used to evaluate any added implementation complexity due to the extensions. Moreover, the prototype has been thoroughly benchmarked to quantify any overheads added by the extensions.
- My paper [10] about the extensions, written together with the supervisors, has been accepted for publication in the International Workshop on OpenMP (IWOMP), the premier venue for OpenMP research. This is a big step in securing that the extensions make it into OpenMP.

1.4 Outline

The outline of this thesis is as follows:

- Chapter 2 presents a wide range of background material required to understand OpenMP, performance analysis, the OMPT profiling API, and why the API is highly anticipated. This fulfils requirement **R1**.
- Chapter 3 presents the proposed extensions, the considerations behind them, and how they are implemented, fulfilling requirement **R2** and partially **R3**.
- Chapter 4 describes the experimental setup used to evaluate the overheads tied to the extensions, partially fulfilling requirement **R3**.
- Chapter 5 presents results from micro-benchmarks and HPC workloads used to evaluate the overheads, fulfilling requirement **R3**.
- Chapter 6 inspects similar interfaces and proposals, and further demonstrates how other profiling methods require descriptions provided by the proposed extensions. This fulfils requirement **OR1**.
- Chapter 7 concludes the thesis results, and gives an overview of future work.

Chapter 2

Background

The work in this thesis touches on a lot of interrelated topics. The extensions are designed such that they can be proposed for inclusion in the upcoming OpenMP 5.0 specification, and I therefore introduce OpenMP and the upcoming OpenMP Tools API (OMPT). To understand the motivations behind the extensions and OMPT itself, I briefly present the field of performance analysis and performance visualization. Finally, I discuss some popular OpenMP implementations.

2.1 OpenMP

Computer systems are becoming increasingly parallel. Writing efficient programs for modern multiprocessors with shared memory can be a difficult task. In addition to regular performance concerns such as algorithmic complexity, memory allocation, memory access pattern, and branching pattern, a multitude of other challenges are introduced when a program becomes parallel. Two of these are [2] partition work and efficient use of shared resources such as memory.

OpenMP is an industry-standard API designed to help overcoming these challenges [1]. By inserting directives into programs written in C, C++ or Fortran, otherwise serial code is transformed into a parallel program. The compiler reads the directives and generates appropriate function calls to an OpenMP runtime system. During program execution, the runtime system will manage worker threads, locks, and other resources on behalf of the program. It will also assist in distributing work among the workers. A simple example of how this can be done is shown in Listing 2.1

```
void array_sum(
    int *arr1,
    int *arr2,
    int *arr_out,
    int arr_length) {
  #pragma omp parallel for schedule(static, 10)
  for (int i = 0; i < arr_length, ++i) {
    arr_out[i] = arr1[i] + arr2[i];
  }
}
```

Listing 2.1: Statically scheduled parallel for-loop

A serial for-loop is turned parallel through the OpenMP `parallel for` directive. This directive first starts a parallel region, which is a region of the program that is executed by multiple worker threads. It then starts a worksharing for-loop, which divides the iteration space among available workers. The `schedule` clause is used to specify exactly how the loop should be scheduled. The static schedule ensures that subsets of the iteration space, commonly called *chunks*, are distributed among workers in a round-robin fashion. The second argument to the schedule clause specifies the chunk size, which is the number of iterations within individual chunks. Together with the `section` and `single` constructs, the `for` construct has been supported since the first release of OpenMP.

```
int fib(int n) {
  int a1, a2;
  if (n < 2)
    return n;
  #pragma omp task shared(a1) firstprivate(n)
  a1 = fib(n-1);
  #pragma omp task shared(a2) firstprivate(n)
  a2 = fib(n-2);
  #pragma omp taskwait
  return a1 + a2;
}
```

Listing 2.2: Naive task-based Fibonacci implementation

The third revision of the OpenMP specification introduced the task construct to support task-parallel programs. Tasks are independent units of work that can run on any worker thread. Unlike sections, tasks can be created dynamically based on conditions known only at run-time. Tasks can also create child tasks. Lastly, task synchronization differs from barriers, which are commonly used together with sections. Task synchronization pauses the execution of a task until its child tasks are complete. The worker that previously executed the parent task is then free to execute other tasks in the meantime . Barriers make threads wait until all threads have reached the barrier point. This illustrates how the tasking concept

simplifies the implementation of task-parallel programs, as threads and barriers are abstracted away. Shown in Listing 2.2 is a naive task-based implementation of a function that calculates the nth Fibonacci number. Each call to this function generates two child tasks and waits for them to complete before returning the result.

2.2 Performance Analysis Methods

To analyze the performance of a system, there are two main approaches. The first is to create a model of the system, and use the model to calculate the estimated performance. The second approach is to create the system, or a prototype of the system, and measure its performance. The latter approach is what is used in this thesis, but both are presented. I focus on methods relevant for analysis of parallel programs.

2.2.1 Performance Modelling

To design efficient parallel programs, it is important to understand the costs of one's design decisions. It is not feasible to implement all program permutations and measure their performance, so modelling can be used to prune the design space. Moreover, it can be used to evaluate scalability and identify bottlenecks of the chosen design, and indicate where optimization efforts are best spent after implementation. To give a brief overview of common performance modelling methods, I present Amdahl's law, the Roofline Model, and work-span analysis.

2.2.1.1 Amdahl's Law

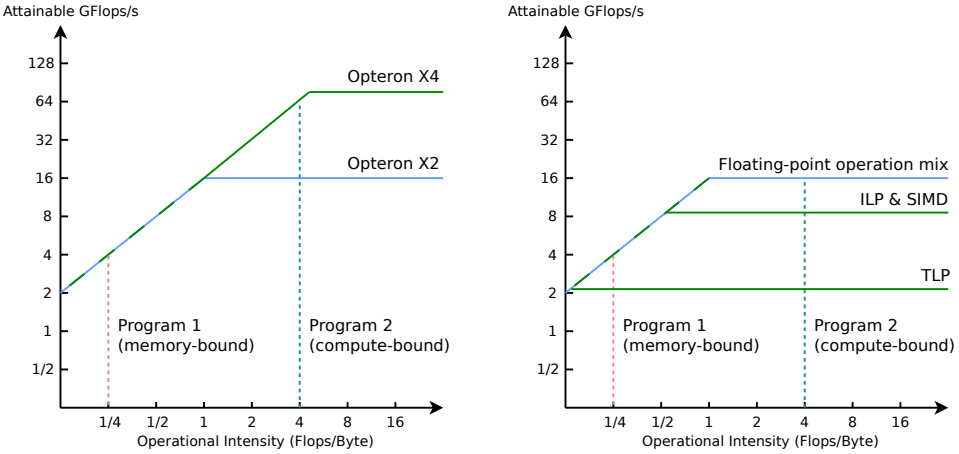
Amdahl's law [11] describes that the theoretical speedup of parallelizing an application is limited by the application's sequential component. More formally, the law can be formulated as:

$$S(s) = \frac{1}{(1-p) + \frac{p}{s}} \tag{2.1}$$

Here, S is the application's total speedup, s is the speedup in the part of the program that benefits from parallelization, and p is the proportion of execution time originally occupied by the parallel component of the program. It can be seen that as the parallel speedup s grows towards infinity, the speedup of the application S becomes:

$$\lim_{s \to \infty} S(s) = \frac{1}{1-p} \tag{2.2}$$

This implies that if 20% of a given program is sequential, the best-case speedup is 5. Said differently, the execution time can only be reduced by 80%. Amdahl's law is very simple. It was previously used to justify the belief that parallel computing



(a) The Roofline model for two AMD CPUs.

(b) The Roofline model for Opteron X2 with three performance ceilings.

Figure 2.1: The Roofline model. The figures are based on data from the original Roofline paper [13].

had limited utility except for a modest class of specialized applications [12], due to the infamous sequential component. However, since then it has been shown that most problems have parallel solutions and that the scalability of programs are often limited by other factors, such as communication costs. However, the law can still be useful to predict the speedup of incremental parallelization.

2.2.1.2 Roofline Model

The Roofline model [13] is a visual performance model designed to provide insights into factors affecting the performance of computer systems. Instead of prediction the exact performance of a program running on a given system, the model specifies upper bounds on program performance in floating-point operations per second (Flops/s). A basic premise in this model is that memory bandwidth is, and will continue to be, the constraining resource. The model defines *operational intensity* as number of operations per byte of (slow) DRAM traffic. Programs with low operational intensity are constrained by memory. Their attainable Flops/s is the product of the peak memory bandwidth (bytes per second) and operational intensity (Flops/byte). Programs with high operational intensity are instead constrained by the peak floating-point performance of the machine. Together, this gives the formula used to plot rooflines:

$$\text{Attainable Flops/s} = \min \begin{cases} \text{Peak Floating Point Performance} \\ \text{Peak Memory Bandwidth x Operational Intensity} \end{cases} \tag{2.3}$$

Figure 2.1a shows the basic Roofline model. The attainable Flops/s of two

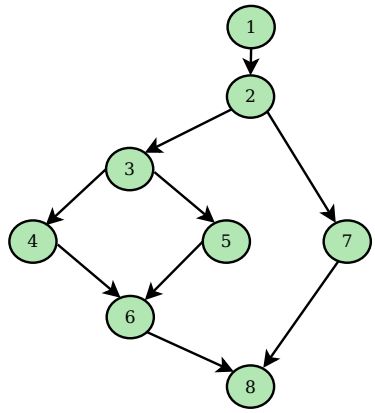


Figure 2.2: Parallel program in its DAG representation.

programs with different operational intensities is displayed. The rooflines of two different CPUs are shown, AMD Opteron X2 and AMD Opteron X4. An immediate insight is that Program 1 will not benefit from a processor with higher floating-point performance, since it is memory-bound. Program 2, on the other hand, benefits from higher floating-point performance due to its higher operational intensity. The basic Roofline model can be used to predict if your code is memory-bound or not. However, the model can be made more useful by further adding ceilings, which are upper performance bounds given which optimizations are performed. Figure 2.1b shows three optimizations that speed up programs running on the Opteron X2 [13].

The optimizations are ordered according to difficulty. The simplest optimization is using thread-level parallelism. The next optimization is to optimize for instruction-level parallelism, and to use SIMD instructions. To squeeze out maximum FLops/s, a significant fraction of instructions must be floating-point operations. Moreover, the proportion of additions and multiplication operations must be close to 50/50, since these can be done simultaneously with multiply-add instructions [13]. The ceilings allow quickly identifying which optimizations are likely to give the greatest performance gains. The weakness of the Roofline model is that finding the performance ceilings is not straightforward. Williams et al. suggest using optimization manuals and micro-benchmarks.

2.2.1.3 Work-Span Analysis

Parallel programs that make use of structured parallelism, such as tasks, can be modelled using the directed acyclic graph (DAG) model of multithreading [14]. In this model, every node represent an instruction, and the edges denote dependencies between instructions, as seen in Figure 2.2. An edge from node x to y means that x must complete before y can start. Figure 2.2 shows multiple *fork points*, where two nodes are dependent on one predecessor node, and *join points*, where

one node depends on two predecessor nodes. In the OpenMP tasking model, these occur when tasks are created, and when tasks synchronize, respectively. Using this model, parallelism can be defined precisely.

First, we define *work*. The work in a program is the total amount of time spent in all the instructions [15]. The work of the program in Figure 2.2 is 8, assuming that every instruction completes on 1 unit of time. We define the fastest possible execution time of a program running on P processors to be T_P. Since work corresponds to the execution time on a single processor, it is denoted T_1. Using these measures, the *work law* provides a power bound on P-processor execution time:

$$T_P \geq T_1/P \tag{2.4}$$

The work law presumes that processors cannot execute more than 1 instruction per unit of time, and P processors can therefore execute at most P instructions per unit of time. Next, we introduce the *span* measure. It is the longest path of dependencies in the DAG – in our example program it is found to be 6 by moving along one of the two leftmost paths. The span is equivalent to the theoretically fastest execution time when running on an infinite number of processors, and we thus denote it by T_∞. Span provides another bound on P-processor execution time, termed the *span law*:

$$T_P \geq T_\infty \tag{2.5}$$

We are now ready to define *parallelism*. It is the ratio of work to span, T_1/T_∞. It represents the average amount of work along each step of the critical path [15]. Parallelism bounds the speedup one can achieve by using more processors. Perfect linear speedup, i.e. a speedup of P when going from one to P processors, cannot be obtained for any number of processors greater than the parallelism. In our example program, the parallelism is only $8/6$, meaning that even on two processors we cannot achieve perfect linear speedup. More complex programs, such as one that does matrix multiplication of 1000×1000 matrices can have a parallelism in the millions. Thus, we see that work-span analysis is useful in quantifying the theoretical speedup of parallel algorithms.

2.2.2 Profiling and Tracing

Software profiling is the act of performing measurements on a piece of running software to capture one or more of its properties. These properties together form a *profile*. Tracing is similar to profiling, but also records the chronology of runtime events to form a *trace*. Traces therefore typically increase in size the longer a program runs. Both methods can be crucial in understanding the behaviour and performance of programs. Raw data for profiling and tracing can be collected in multiple ways with differing data granularity. The two main mechanisms used are instrumentation and statistical sampling, both of which I discuss below.

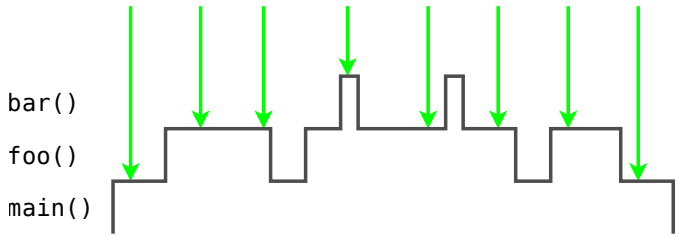


Figure 2.3: Periodic call stack sampling. At given times, the call stack is probed. Functions that take up a large portion of the total execution time are seen in many of the samples.

2.2.2.1 Instrumentation

Instrumentation of computer programs is a way to extract information from programs as they run. This is done by inserting extra instructions into the program at strategic points of interest. A common instrumentation use-case is instrumenting a specific function[1]. Depending on the informational needs, one can record its total call count or execution time, individual timestamps, individual execution times, and a whole host of other properties. However, excessively detailed instrumentation can lead to overheads that obscure or distort the bottlenecks in the original program [16]. Thus, instrumentation approaches must be frugal with respect to the degree of instrumentation, while also providing sufficient details to uncover problems.

Programs can be instrumented in multiple ways. Source code instrumentation inserts instrumentation code during program compilation. Binary instrumentation instead inserts instructions in the compiled program. This can be done dynamically at run-time [17] and requires no recompiling or relinking. Programs that make heavy use of a runtime system, such as OpenMP programs, can also be profiled or traced through an instrumented runtime system. It is then possible to create an uninstrumented and an instrumented version of the runtime system, using the latter only when needed. The drawback of this approach is that instrumentation is limited to runtime system code only.

2.2.2.2 Sampling

Sampling is the act of taking samples. In the context of performance analysis, sampling entails regularly inspecting the current state of the program. The sampling subroutine is customarily triggered by a system timer or a hardware performance counter exceeding a user-specified value [18]. The act of taking a sample consist of interrupting the execution on a particular thread and recording the location of the current instruction before resuming the thread. A common extension to this simple

[1] In computer science literature, the terms procedure, function, and subroutine, are all used to denote a callable sub-program within a larger program. I primarily, but not exclusively, refer to these as functions.

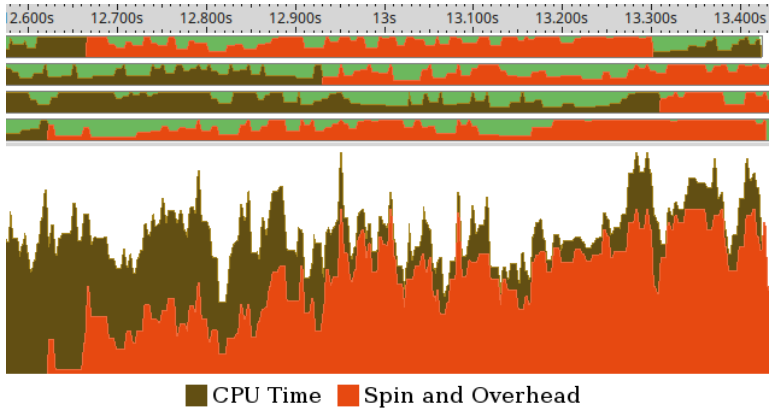


Figure 2.4: The thread timeline visualization of Intel VTune Amplifier.

function is to also probe the call stack, recording the return address of every frame. This exposes the chain of function calls that led to the current instruction. Figure 2.3 visualizes periodic call stack sampling. Instructions, and therefore also stack frames, can be mapped to their source code origin using debugging information inside the executable or in a separate file.

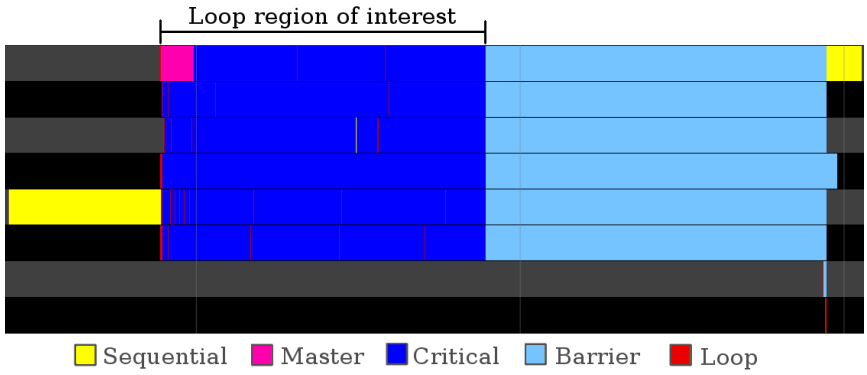
The overheads tied to sampling are low. Unlike with instrumentation, it is not necessary to insert extra function calls or variables into the program to be profiled. Except for when a thread is interrupted, the program executes its normal, non-instrumented flow of instructions. This also leads to lower influence on memory caches. Moreover, it is trivial to adjust the sampling overhead by adjusting the sampling frequency. However, sampling profilers are not exhaustive. Certain functions might evade the profiler because they are short, or more critically, because they are called with a period similar to the sampling period, but with an offset in time.

2.3 Visualization Methods

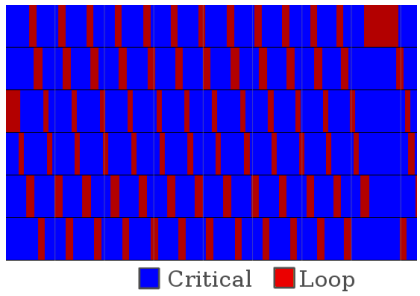
Data analysis commonly benefits from visualizations to quickly display patterns and relationships present in the data. In addition to general visualizations such as bar charts and scatter plots, performance analysis tools frequently employ one or more domain-specific visualizations. I present these below.

2.3.1 Timelines

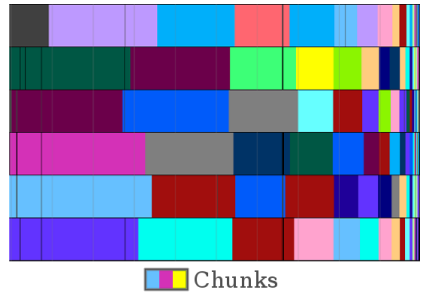
Timelines present the state of a given system over time. Many performance analysis tools implement a thread timeline visualization showing thread states of one or more threads chronologically. Some visualizations show discrete states such as *running* and *stalled*, while others show composite states, as seen in Figure 2.4.



(a) A thread state timeline of a small OpenMP program.



(b) A thread state timeline for a small section of the loop region displayed in Figure 2.5a.



(c) A for-loop chunk execution timeline, displaying the loop region of Figure 2.5a.

Figure 2.5: Three views from Aftermath. The x-axis shows time, while the y-axis shows different worker threads.

This timeline, from Intel VTune Amplifier, displays both thread state and CPU utilization per thread, and an aggregated timeline below. It can be used to identify problematic regions of program execution, for instance due to large amounts of overhead, or load imbalance between threads. However, the timeline gives no indication of what causes observed problems.

Performance tools such as Vampir [19], Aftermath [20] and hpc-traceviewer of HPCToolKit [21] include thread timelines that are OpenMP-aware. This means that they support a more fine-grained set of states that map to common OpenMP activities. Using Aftermath as an example, supported states include execution of a single/master construct, execution of a critical region, and waiting on a barrier. This makes it easier to pinpoint the cause of a problem. Aftermath also support separate timeline views for loop analysis. These allow displaying the time spent by each worker thread in completing a loop, and which iterations executed on each worker.

Figure 2.5a shows the execution of a simple OpenMP program with a parallel for-loop using Aftermath's thread state view. Zooming in on a small segment of the loop interval gives the timeline seen in Figure 2.5b, revealing an excessive amount of time spent inside OpenMP critical regions. This indicates that the algorithm should be refactored use less synchronization. Finally, Figure 2.5c displays the execution times of individual chunks. They can be seen to become shorter towards the end of the loop – a result of using OpenMP's guided for-loop schedule.

2.3.2 Call Graphs

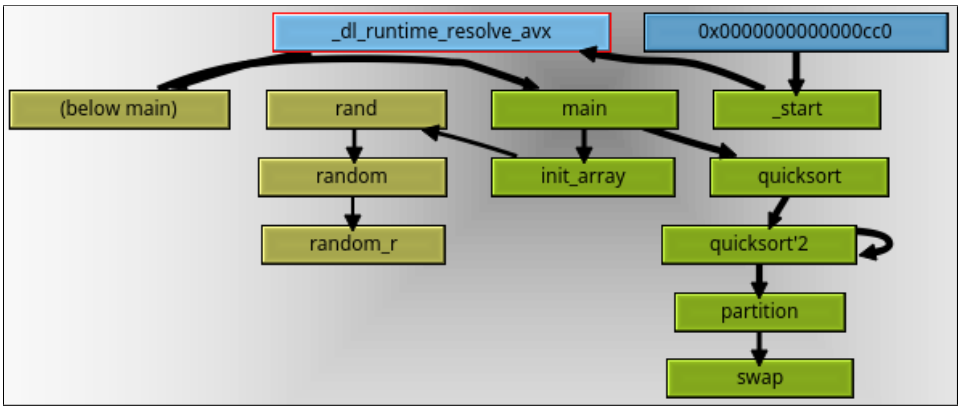
Function call graphs display caller-callee relationships between functions of a program. It is a directed graph, as seen in 2.6. An edge from a function `foo` to a function `bar` denotes that `foo` calls `bar`. Call graphs are versatile, and thus have numerous applications. When visualized, developers can use them to quickly understand the basic program structure. Other uses include tracking values used in function calls, identify functions that are never called, and quantify the execution time spent by each function. The latter use indicates how it can be used to guide program optimization. By focusing on functions that occupy a large proportion of the execution time, one can avoid optimizing functions that have little impact on the overall execution time.

For parallel programs, call graphs are typically generated per thread. When using a large number of processors, inspecting the call graphs of all threads can be a time-consuming task. A separate issue is that many runtime libraries, including popular OpenMP libraries, make a lot of calls to internal functions. This grows the call graph with many nodes that in many cases are not of interest, such as the two nodes named `random` and `random_r` in Figure 2.6.

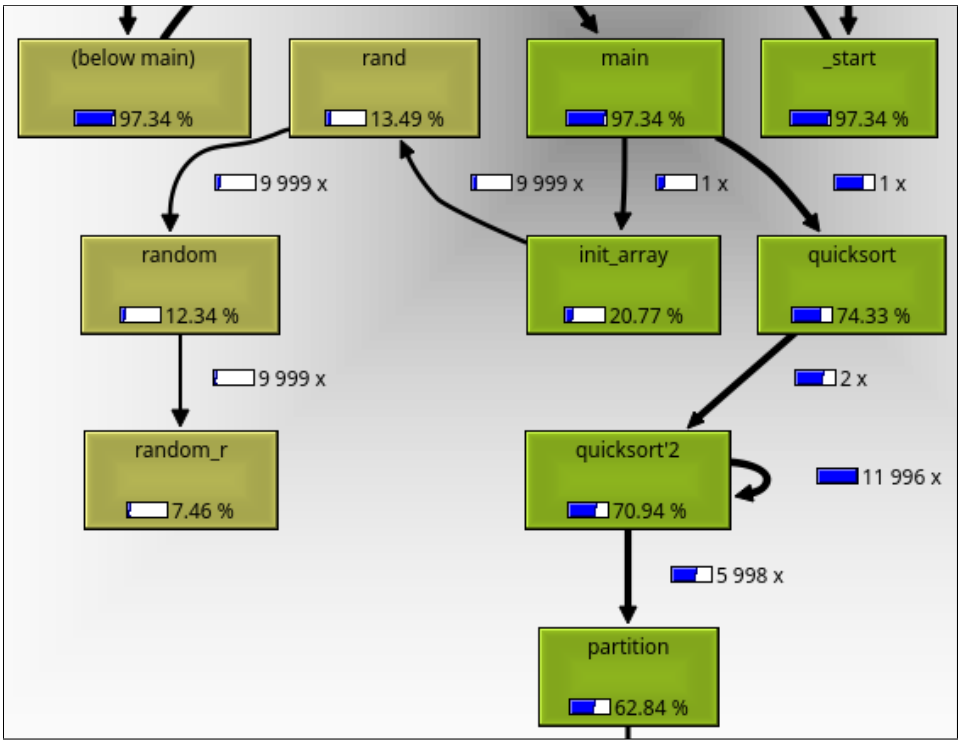
Figure 2.6 shows two call graph views from KCachegrind [22], a popular profile data visualization tool for GNU/Linux. It uses profiling data from Callgrind, an instrumentation tool from the Valgrind [17] framework. A number of different views are supported. Figure 2.6a shows a compact call graph, allowing a quick overview. Figure 2.6b shows a more detailed view – edges are decorated with call counts, and function nodes include the proportion of execution time spent inside the function. In both views, nodes that occupy less than 5% of the execution time are filtered out to remove noise. Clicking on a node gives access to more information, such as exact source code location.

2.3.3 Grain Graphs

The grain graph is a recent visualization for OpenMP programs [3]. The graph shows a fork-join view of *grains* – instances of tasks and for-loop chunks. Grains are placed according to the user-centric abstractions provided by OpenMP, giving a consistent graph regardless of OpenMP implementation differences. This view maps naturally to the typed structure of the program, making its interpretation intuitive. A set of performance metrics are calculated per-grain. Poor-performing grains are highlighted so that they are quickly discerned from other grains.

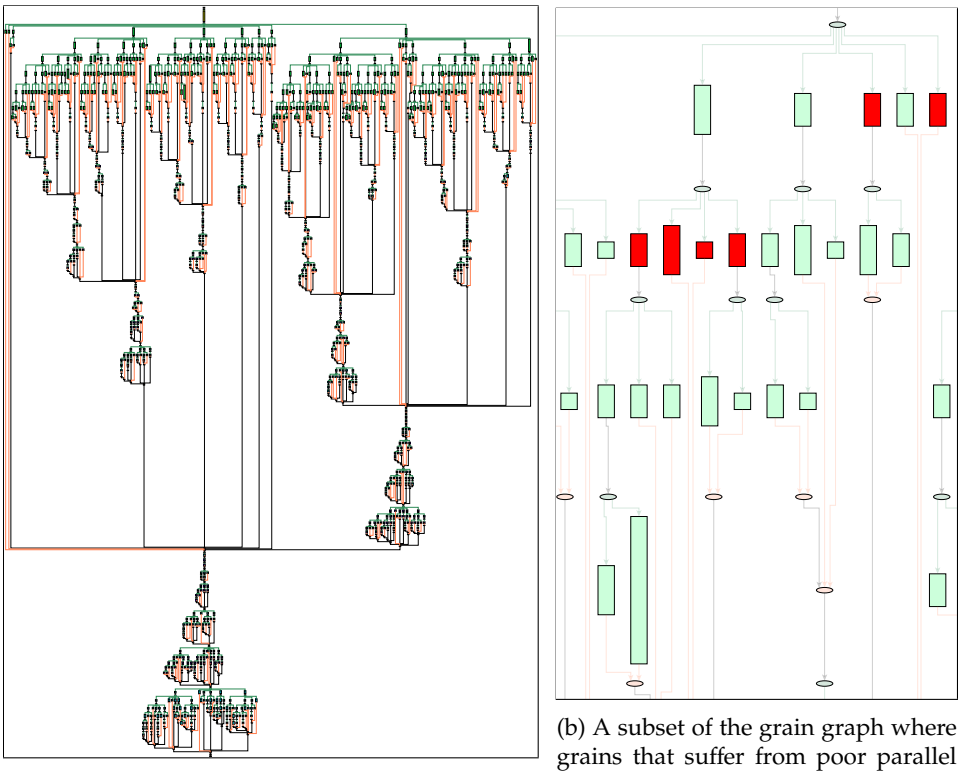


(a) A compact call graph overview.



(b) A cut-out of a more detailed call graph.

Figure 2.6: Two call graph views generated with KCacheGrind. The profiled program is a simple implementation of the quicksort algorithm. Green nodes are user functions, while other colours denote external functions.



(a) The full grain graph.

(b) A subset of the grain graph where grains that suffer from poor parallel benefit are highlighted.

Figure 2.7: The grain graph generated from a parallel merge sort implementation.

Figure 2.7 shows the grain graph for a parallel, task-based implementation of merge sort. Grains, fork points, and join points make up all nodes in the graph. Fork points occur when tasks spawn children tasks. Children tasks later synchronize with their parents, giving join points. Edges are directed and show various relationships between nodes, one example being a grain causing a fork point. The length of grain nodes encode their execution time. Clicking on a grain node reveals additional grain properties, one example being the source code location of the OpenMP construct that spawned it. The graph is acyclic, and therefore a directed acyclic graph (DAG).

To characterize the performance of grains, grain graphs rely on a set of derived performance metrics. One such metric is parallel benefit. A grain's parallel benefit is its execution time divided by the overheads of parallelization incurred by its parent. This overhead is equal to the sum of the grain's creation time and the average time spent by sibling tasks in synchronizing with the parent. Grains with low parallel benefit spend a proportionally large amount of time inside the runtime system, and should therefore be executed sequentially instead. Grains with poor parallel benefit are highlighted in Figure 2.7b. Other derived metrics include [3] load balance, work deviation, instantaneous parallelism, and scatter. When viewing the graph, the user can select one of these to pinpoint problematic grains according to the selected metric.

Grain graphs consist of three components. The first is a tracing component that collects a set of per-grain properties during program execution. Currently, this step is performed using measurements from the MIR runtime system [4], a task-based runtime system with OpenMP-support and powerful profiling capabilities. The second component performs post-processing and graph generation. This is implemented in a reference prototype [23]. The last component is a graphical user interface used to view and navigate the graph. The GraphML files generated by the reference prototype can be viewed by graph tools such as *yEd* or *Cytoscape*.

Data Collection Using OMPT

I have previously found that the OMPT API, discussed in Section 2.4, allows collecting almost all properties needed by grain graphs [5]. However, a few key elements are missing. First, to calculate a grain's parallel benefit, one must know how long it took to create it. At the time of writing, a relevant callback exists, named `ompt_callback_task_create`. This is a single event, meaning it is only called once during task creation. Moreover, its signature does not carry any timing information. It can therefore not be used to measure task creation duration.

Second, since grains can also be for-loop chunk instances, there must be a way to trace the execution of individual chunks. This includes obtaining the iteration range, execution time, and parallelization costs associated with each chunk instance. Currently, the API does not contain any chunk-related events. The event associated with parallel for-loops, `ompt_callback_work`, is shared among all OpenMP worksharing constructs, and is therefore too general to provide loop-specific information. I later propose extensions that allow OMPT to provide these properties. By obtaining all required event data from OMPT, one can portably

generate grain graphs for any program running on standards-compliant OpenMP runtime systems.

2.4 OpenMP Tools API

The OpenMP Tools API [24], called OMPT for short, is a recently proposed extension to the OpenMP standard. The API enables creating performance analysis tools for OpenMP programs. Multiple similar interfaces exist, but these are typically tied to a specific OpenMP runtime system. The prime motivation behind OMPT is to make it possible to create high-quality performance analysis tools that are portable. In the future, any standard-compliant OpenMP implementation will support OMPT.

2.4.1 Previous Influences

There have been multiple attempts to create a standard OpenMP profiling API. OMPT is based on two such efforts, namely the POMP API [25] and the Sun/Oracle Collector API [26].

The POMP API enables instrumentation of runtime functions as well as regions of user code through the use of source code instrumentation. Compiler directives can be used to control the degree of instrumentation, and compliment the existing directives used with OpenMP. Events are monitored through extra calls to the POMP monitoring system, inserted by the compiler.

The Sun/Oracle Collector API's primary focus is statistical sampling of call stacks. It also contains support for instrumentation, but this support is not as extensive as that of POMP. When sampling call stacks of advanced programs, the stacks will typically consist of user functions and runtime functions. Moreover, the call stacks of the master thread and slave threads will differ, since the master thread starts the program while slaves are spawned later. To simplify call stack analysis, the Sun/Oracle Collector API allows classifying stack frames as user functions. This allows tools to focus only on user or runtime regions if they so wish.

2.4.2 Design Objectives

OMPT is inspired by the two aforementioned APIs, and followingly supports both instrumentation and sampling. Its explicit design objectives [24] include that it should be possible to attribute costs to user regions and runtime regions. The API should be sufficient to construct instrumentation-based performance tools as well as low-overhead tools based on asynchronous sampling. Adding OMPT support to an OpenMP runtime system should not add significant overhead if no tool is using the interface. Lastly, the API should not place an excessive burden on runtime developers or tool implementors.

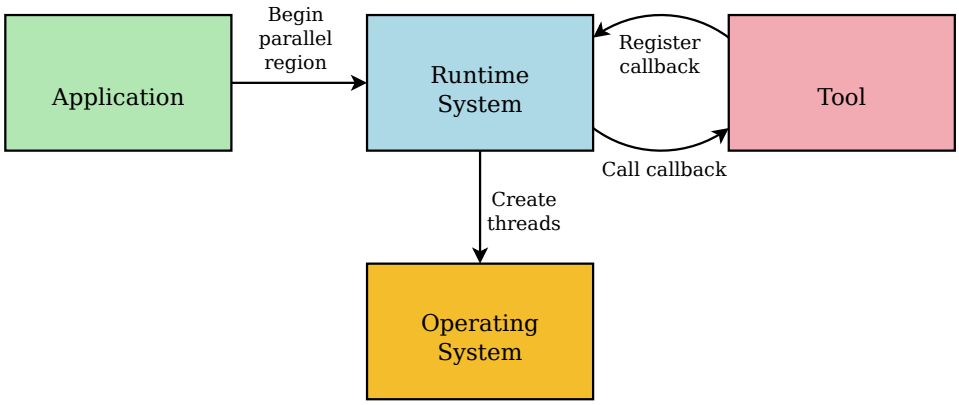


Figure 2.8: Common interactions between an application, an OpenMP runtime system, an instrumentation-based performance analysis tool, and the operating system.

2.4.3 Environment

Performance analysis tools based on OMPT, hereafter simply called tools, are composed of functions that reside in the address space of the program being profiled. During program initialization, the OpenMP runtime system will search for and call a function called `ompt_start_tool`, provided by the tool. This function returns pointers to the tool's initialization and finalization functions, to be called by the runtime before and after the program's main function.

The `ompt_start_tool` can be provided to an OpenMP implementation in three ways [27].

1. Statically link the function definition into the application.
2. Dynamically load a shared library containing the function definition into the address space of the application.
3. Specify the location of one or more tools using the environment variable `OMP_TOOL_LIBRARIES`. The first valid implementation from this list is loaded by the OpenMP runtime system.

Item 2 and 3 above shows that OpenMP programs do not have to be recompiled or relinked to work with tools. This is convenient, since one does not need separate production and profiling versions of the program. It also allows profiling programs for which one does not have the source code.

When the tool's initialization function is called, the tool obtains pointers to relevant OMPT functions. Common candidates are `ompt_get_thread_data`, `ompt_get_state` , and `ompt_set_callback`. The latter function can be used by event-tracing tools to register relevant callbacks. Tools that perform asynchronous sampling will also initialize resources for this purpose, typically a signal handler and a

signal-sending timer on POSIX systems. During program execution, the runtime will call any registered callbacks when their corresponding events occur. Figure 2.8 shows some common interactions between the program, the OpenMP runtime system, the tool, and the operating system. During startup, the tool registers callbacks. The program later enters an OpenMP parallel region, which causes the runtime system to create threads and call the `ompt_callback_parallel_begin` callback if registered for, before the execution of program code continues.

2.4.4 Tracing Functionality

The data needed by grain graphs must be obtained by tracing the creation and execution of OpenMP parallel regions, tasks, and parallel for-loops. I therefore briefly present OMPT's current tracing functionality. At present, the OMPT interface supports 28 callbacks. 12 of these are mandatory, meaning that an OpenMP implementation must implement all of them to be standards-compliant. The remaining 16 callbacks are optional. Below is a list of existing callbacks relevant to obtaining the data needed by grain graphs. All of them are mandatory, except for `ompt_callback_work`. This is the callback used to trace parallel for-loops.

- `ompt_callback_thread_begin`
- `ompt_callback_thread_begin`
- `ompt_callback_parallel_begin`
- `ompt_callback_parallel_end`
- `ompt_callback_task_create`
- `ompt_callback_task_schedule`
- `ompt_callback_implicit_task`
- `ompt_callback_work`

2.5 OpenMP Implementations

To extend OMPT, we need to first understand how support for the OpenMP specification is typically implemented. I therefore examine how the popular compiler toolchains GCC 7.1, LLVM/Clang 4.0, and ICC 17 incorporate OpenMP support. Some details have already been mentioned in Sections 2.1 and 2.4, but they are repeated here to help readability. The toolchains implement most OpenMP functionality similarly, and so I present their approaches collectively. I focus on the interaction between compilers and runtime systems and omit internal runtime system implementation details, such as what data structures are used, unless relevant to OMPT.

2.5.1 Overview

First, we look at how the toolchains implement OpenMP support. Both GCC, LLVM/Clang, and ICC, implement OpenMP through OpenMP-specific code generation in the compiler, in addition to a supporting OpenMP runtime system. In most cases, the compilers, upon encountering an OpenMP directive, generate code that call the supporting runtime system. This avoids adding undesirable complexity inside the compilers, since most of the work is done inside runtime functions.

Each compiler is essentially coupled with an OpenMP runtime system, and these runtime systems are developed and released separately from the compilers. GCC uses *libgomp*, Clang[2] uses the *LLVM OpenMP runtime library*, and ICC uses the *Intel OpenMP runtime*.

The LLVM OpenMP runtime library was donated by Intel to LLVM in 2014. This means that it is almost identical to the Intel OpenMP runtime, as improvements in one of these libraries make their way to the other. Moreover, this means that programs compiled with Clang can link with Intel's runtime, and vice versa. It also means that the primary difference between LLVM/Clang and ICC lies in the code generation. ICC's code generation is more involved, and uses a relatively greater proportion of the available runtime functions.

2.5.2 Parallel Regions and Tasks

Most OpenMP constructs are implemented similarly in all toolchains. Listing 2.3 shows a simple program with a parallel region. The corresponding instructions generated by GCC with optimization level *O3* is shown in Intel syntax in Listing 2.4.

The code inside the parallel region is outlined to its own function. The parallel region is started by calling a runtime system function called `GOMP_parallel`. The address of the outlined function is passed as the first argument. Inside the function call, libgomp starts a team of worker threads and calls the provided function pointer. This pushes a return address inside the runtime system to the stack. Inside the outlined region, the program jumps inside the `puts` function. When this function returns, the instruction pointer is restored to the return address inside the runtime system, which finalizes the parallel region. Note that the outlined function has been optimized to do a tail call, which is why it does not end with a `ret` instruction. Had there not been a function call at the end of the parallel region, the outlined function would end with `ret`.

This scheme is used in all three toolchains. Tasks are implemented similarly, and thus we omit showing more code. The runtime functions tied to tasks are however a lot more complex, since tasks can be seen as execution contexts that might execute immediately, might be pushed into a thread-local task queue, might need to synchronize with sibling tasks, and so on.

[2]For the remainder of this thesis, I refer to the combination of Clang and its supporting runtime library as LLVM/Clang, while using simply Clang to denote the compiler itself.

```
int main(int argc, char const *argv[]) {
  #pragma omp parallel
  {
    printf("foo\n");
  }
}
```

Listing 2.3: Program with a parallel region

```
.LC0:
  .string "foo"
main._omp_fn.0:
  mov     edi,
    OFFSET FLAT:.LC0
  jmp     puts
main:
  sub     rsp, 8
  xor     ecx, ecx
  xor     edx, edx
  xor     esi, esi
  mov     edi,
    OFFSET FLAT:main._omp_fn.0
  call    GOMP_parallel
  xor     eax, eax
  add     rsp, 8
  ret
```

Listing 2.4: GCC output

2.5.3 For-loops

Parallel for-loops using the `dynamic` or `guided` schedule types are dynamically scheduled. This means that chunks are assigned to worker threads at run-time. Such loops are implemented equivalently in GCC, LLVM/Clang and ICC. Listing 2.5 shows a simple for-loop, and Listing 2.6 shows how the code is transformed. In short, every worker keeps requesting chunks inside a while loop, until there are no more chunks left. The runtime system outputs the lower and upper iteration bounds of the next chunk by writing to stack variables pointed to by pointer arguments. The iterations within the iteration bounds are then executed in a for-loop.

```
#pragma omp for schedule(dynamic)
for (int idx = 0; idx < 100; ++idx) {
  foo();
}
```

Listing 2.5: A basic, dynamically scheduled for-loop

```
int lbound, ubound, idx;
// lb, ub, and iter. step used to find local bounds
bool continue = loop_dynamic_start(0, 100, 1, &lbound, &ubound);
while (continue) {
  for (idx = lbound; idx <= ubound; idx += 1) {
    foo();
  }
  continue = loop_dynamic_next(&lbound, &ubound);
}
loop_dynamic_end();
```

Listing 2.6: Approximation of code generated for dynamically scheduled for-loops

The OpenMP directives examined thus far are implemented almost identically in the three toolchains. However, they differ in their implementations of statically scheduled loops. This schedule distributes iterations in a block-cyclic manner. This means that chunks are assigned to worker threads in a round-robin fashion, and the workers can compute the iteration bounds of the next chunk by incrementing the previous bounds by a stride equal to the chunk size multiplied by the number of threads. OpenMP implementations therefore do not need to perform any per-chunk runtime calls like the `loop_dynamic_next` seen in Listing 2.6.

```
int lbound, ubound, stride, idx;
// lb, ub, iter. step and chunk size used to find stride and local bounds
loop_static_init(0, 100, 1, 10, &lower_bound, &upper_bound, &stride);
while (ubound = min(ubound, globalUbound), idx = lbound, idx < ubound) {
  for (idx = lbound; idx <= ubound; idx += 1) {
    foo();
  }
  lbound = lbound + stride; ubound = ubound + stride;
}
loop_static_fini();
```

Listing 2.7: Approximation of code generated for statically scheduled for-loops

GCC implements loops with this schedule entirely in the compiler, while LLVM/-Clang and ICC both call loop initialization and finalization functions. Listing 2.7 shows what the generated code would look like if the program in Listing 2.5 instead used the static schedule and a chunk size of 10. In LLVM/Clang and ICC, the initialization function calculates the initial lower and upper iteration bounds, and also the stride, which worker threads then use to execute all assigned chunks. In GCC, the call to `loop_static_init` is replaced with code that calculates these values directly.

2.5.4 OMPT

At the time of writing, there is no OMPT support in libgomp, the runtime system used by GCC. Furthermore, the way statically scheduled loops are handled in GCC will likely have to be changed to support the `ompt_callback_work` callback of OMPT. This callback signals the beginning and end of worksharing constructs such as parallel for-loops. Since the runtime system is responsible for invoking callbacks, I expect GCC to end up with code generation that is equivalent to that of Clang and ICC, shown in Listing 2.7.

On the other hand, both the LLVM and Intel OpenMP runtime systems support a substantial subset of the OMPT interface. However, the OMPT code inside these libraries target the API specified in the outdated OpenMP Technical Report 2 [28], which is now superseded by OpenMP Technical Report 4 [27]. Fortunately for us, the group behind OMPT have created a fork of the LLVM OpenMP runtime system, and added support for the most recent OMPT specification. The group is made up of developers that have also contributed to the current OMPT implementation of the LLVM OpenMP runtime. The improved OMPT support is planned for inclusion in LLVM OpenMP when the OpenMP 5.0 specification is finalized.

Chapter 3

Extending OMPT

This chapter first define a set of design criteria that our extensions must abide to make it into OMPT. Next, I present the design of the extensions, and the considerations that led to this design. I also describe a prototype implementation. Last, I discuss the impact that the extensions have on the complexity of OpenMP implementations.

3.1 Design Criteria

OpenMP is an industry-standard for parallel shared-memory programming. The API is widely used in high performance computing. Any potential additions to the API is scrutinized, as careless designs can result in significant overheads in the design's implementations. Within an OpenMP environment there are multiple different types of overhead. Not all overheads are of equal concern to the designers of OpenMP, for reasons now discussed. Knowing the types of overhead that can be tolerated, and which cannot, is essential when designing extensions to OMPT. These insights will result in a set of design criterions.

Traditionally, there has been no standard profiling API for OpenMP programs. This is about to change with the inclusion of OMPT into the OpenMP 5.0 specification. This is made possible because the design of OMPT focuses on low overhead when the API is not in use by any tools. One of the explicit design objectives of OMPT [24] is that incorporating support for the API in an OpenMP runtime system should add negligible overhead. This provides the first design criterion.

(1) The extensions should add negligible overhead when no tool is in use.

Criterion 1 means that it must be possible to implement extensions without significant impact on the normal, un-profiled execution of runtime system functions. Inside the implementation, this can be achieved by making OMPT-specific code conditional, that is to only execute it when a tool is attached. However, adding a large number of conditional branches can still impact performance. Furthermore,

OMPT code might require additional stack variables, which also has the potential of degrading performance. Extensions should therefore be relatively simple.

Another high-priority design objective states that OMPT should allow the construction of low-overhead performance tools based on asynchronous sampling. Such tools will cause OMPT to enter an *enabled* state, but might not register any instrumentation callbacks. This entails that to abide this design objective, our extensions must also be frugal when OMPT is enabled. This can be formulated as the second design criterion.

(2) The extensions should add negligible overhead when a tool is attached, but no callbacks are registered.

Criterion 1 and 2 have similar priority. In fact, in the introductory paper of OMPT [24], the design objective that led to the second criterion is listed higher, as objectives are listed in order of importance. However, since achieving criterion 2 without also achieving criterion 1 is an oxymoron, I list them in the opposite order. This concludes the set of design objectives that explicitly mention overheads.

The third design objective relates to tracing, which is the aspect of OMPT I seek to extend. It simply states that the API should define support for trace-based performance tools. From this, and from criterion 1 and 2, we can extract a third criterion.

(3) Given a choice, the extensions should defer overheads related to tracing to tools.

The OMPT design objectives assertively focuses on low overheads inside the runtime system, while at the same time paying little attention to the overheads associated with tracing. For instance, tracing typically involves recording the time or duration of an event. One could consider, then, to pass a duration parameter as part of an event callback. This could potentially reduce overall overhead, since the tool could make due with one callback instead of two. However, this would imply potentially costly calls to a timing function inside the runtime system. The OMPT way is instead to leave such tasks to tools.

3.2 Task Creation Duration

To calculate a grain's parallel benefit [3], one must know its creation overhead. For task grains, this corresponds to its task creation duration. The duration can be measured as wall-clock time or as the number of CPU cycles spent creating the task. An existing callback named `ompt_callback_task_create` can notify tools that task creation is taking place, but it does not allow measuring the duration of the event.

3.2.1 Why Estimates are Insufficient

During a previous project [5], and during a discussion in the OMPT mailing list, shown in Appendix A, it was suggested that one can estimate the task creation

duration instead of measuring it per task, and use the estimate when calculating parallel benefit. My experience shows that this is not a viable solution, which I will demonstrate here.

Creating a task instance typically involves pushing it into a thread-local or global task queue after allocating and initializing book-keeping data structures. The time spent performing these steps can vary greatly. During busy periods, multiple threads might attempt to push or steal tasks concurrently, resulting in queue contention. Dynamic memory allocation can lead to costly page faults. Furthermore, some implementations might resize the task queue when it becomes full, resulting in additional memory allocation and copying. In the case that the queue is not resized, the task must be executed immediately instead, which avoids overhead related to queue locking. To demonstrate this, I use the test program of Listing 3.1.

```
uint64_t avg_sum = 0;
#pragma omp parallel num_threads(num_threads)
{
  uint64_t before = get_timestamp_counter();
  for (int i = 0; i < num_tasks; ++i) {
    #pragma omp task
    {
      // Necessary to avoid GCC optimizing away the task
      __asm__ ("");
    }
  }
  uint64_t elapsed = get_timestamp_counter() - before;
  uint64_t thread_avg = elapsed / num_tasks;
  #pragma omp atomic
  avg_sum += thread_avg;
}
uint64_t avg_result = avg_sum / num_threads;
```

Listing 3.1: A simple code for finding the average task creation time.

The benchmark is intended to show that even the average task creation duration varies greatly depending on a few key parameters. I ran the test with different values for `num_threads` and `num_tasks`, and compiled the program with LLVM/-Clang and GCC. The test machine is described in Section 4.1. Results are shown in Table 3.1. As seen in Table 3.1a, creating a large number of tasks on a single thread incurs the least amoung of overhead. This is expected, as there is no need for thread synchronization, and because most tasks are executed immediately. Unsurprisingly, the highest overhead is seen with few tasks and many threads, where the average task creation duration is more than 16 times longer. A more surprising result is how different the task creation overhead of GCC is compared to LLVM/Clang. It starts very high but becomes the lowest when the task count is high.

The experiment shows how there can be no single satisfactory task creation du-

# threads	# tasks		
	10	10^3	10^6
1	450	250	100
10	1500	600	275
20	1650	625	285

(a) Average task creation duration with varying thread count, using Clang.

Toolchain	# tasks		
	10	10^3	10^6
Clang	1650	625	285
GCC	65000	4400	60

(b) Average task creation duration seen in LLVM/Clang and GCC with 20 threads.

Table 3.1: The average number of cycles spent creating a varying number of tasks.

ration estimate. Even if one attempted to quantify the approximate duration with parameters close to those used for a particular program of interest, the overhead can even change for different phases of the program, meaning that the derived parallel benefit metric would only be correct in some of these phases. I conclude that task creation duration must be measured per task.

3.2.2 Design

In order to extend OMPT with the capability to measure the duration of task creation, there are multiple candidate approaches. I have considered the following three:

1. Add an endpoint parameter to the `ompt_callback_task_create` callback, that is an enumeration with the two options `ompt_scope_begin` and `ompt_scope_end`. Some existing callbacks such as `ompt_callback_work` use this solution. The callback is invoked before and after task creation. This allows tools to measure the time between calls at the expense of changing the signature and semantics of an existing callback.

2. Introduce a new callback that denotes the beginning of task creation, and let the existing callback `ompt_callback_task_create` be called at the end of task creation. This approach is similar to the first but a new callback is introduced instead of changing an existing one. This has the advantage that the signature of the new callback can be tailored to only provide relevant parameters, as some of the parameters of `ompt_callback_task_create` are not needed until after task creation.

3. Measure the task creation duration inside the runtime, and report it to the tool as an extra parameter to `ompt_callback_task_create`. The advantage of this approach is that it avoids an additional callback invocation before the task creation event. However, tools might want to measure time differently. Some might need CPU cycles while others need microsecond precision. To complicate things further, the runtime system may decide to measure elapsed processor cycles using a hardware performance counter, thereby reducing

the amount of performance counters available to tools. Should multiple cycle counters exist, the tool would not necessarily know which counter is used by the runtime system.

Initially, the problems associated with the third approach led to the conclusion that the two other approaches were better candidates. The primary reason was that it violates the third criterion of Section 3.1. However, when designing the extensions tied to for-loops, discussed in Section 3.3, the same challenge of measuring really short events occured. Specifically, grain graphs also require the creation duration of for-loop chunks. To limit the overhead added to chunk scheduling, an approach similar to the third approach discussed above was reconsidered.

The time agreement disadvantage was solved by allowing tools to register a function in tool-space that returns the current time. Tools therefore control how time is measured, and the overhead depends on the timing subroutine chosen by the tool. This function is called by the runtime system before and after task creation. The difference between the two time values is passed as an extra parameter to `ompt_callback_task_create`. This leads to the function signatures seen in Listing 3.2.

```
// Signature of tool-provided time function
typedef double (*ompt_tool_time_t) (void);
// The proposed new signature of ompt_callback_task_create
typedef void (*ompt_callback_task_create_t) (
  ompt_data_t *parent_task_data,
  const ompt_frame_t *parent_frame,
  ompt_data_t *new_task_data,
  ompt_task_type_t type,
  int has_dependences,
  double event_duration,           // New duration parameter
  const void *codeptr_ra
);
```

Listing 3.2: The proposed task creation callback and tool-time function.

The new `event_duration` parameter is a double-precision floating point number. This allows high precision time values and is consistent with the existing `omp_get_wtime` / `omp_get_wtick` functions. If the tool does not register an `ompt_tool_time` function, the event duration is reported as 0. An alternative solution is to let the runtime system fall back to a low-precision timer consistent with `omp_get_wtime`. Experiments indicated that this solution adds significant overhead. In requiring that tools register their own time function to obtain the task creation duration, the design also allows them to opt-out of timing overhead by not registering one. We also specify that event durations calculated using `ompt_tool_time` function can be be reported as 0 if the compiler or runtime system decides that the duration is lower than the overhead to call the tool-time function twice. This is a way to reduce overhead for very short events. As such, a return value of 0 is used to signal either that no tool-time function is registered, or that the event is too short.

3.3 Extended For-loop Events

The grains of grain graphs are a generalization of task and chunk instances. When visualizing a parallel for-loop in a grain graph, each chunk is shown as a separate node. It is therefore necessary to obtain certain chunk properties such as execution time, iteration range, and creation overhead. This is currently impossible. The only callback directly related to parallel for-loops is `ompt_callback_work`. Because this callback is shared among all worksharing constructs, it does not provide any information specific to loops. This led to the realization that OMPT needs at least two new callbacks to allow collecting the required data about for-loop execution – a new loop callback and a chunk callback.

3.3.1 Design

The grain graphs method requires tracing for-loops and chunks. In this section, I propose extensions to this end. First, the loop callback is discussed, whose signature is shown in Listing 3.3.

```
// The proposed ompt_callback_loop signature
typedef void (*ompt_callback_loop_t) (
  omp_sched_t loop_sched,          // Actual schedule type used
  ompt_scope_endpoint_t endpoint, // Begin or end?
  ompt_data_t *parallel_data,      // The parallel region
  ompt_data_t *task_data,          // The implicit task of the worker
  int is_iter_signed,              // Signed loop iteration variable?
  int64_t step,                    // Loop increment
  const void *codeptr_ra           // Runtime call return address
);
```

Listing 3.3: The new loop callback signature.

Readers familiar with OMPT will notice that the loop callback is quite similar to the existing `ompt_callback_work`. The parameters `endpoint`, `parallel_data`, `task_data`, and `codeptr_ra` are reused in the new callback, and have unchanged semantics. The new `loop_sched` parameter carries the scheduling type used by the runtime system. This cannot always be deduced from source code, as programmers can specify the **`auto`** or **`runtime`** schedules to leave the decision up to the compiler and/or runtime system. To correctly deduce the number of iterations performed in the loop, the `step` parameter is required, together with the chunk callback discussed later. Last, the `is_iter_signed` is necessary so that tools can cast the iteration bounds reported by `ompt_callback_chunk` to the correct type. Since OpenMP allows both signed and unsigned 64-bit iteration variables, this cannot be avoided without having separate callbacks for these two cases.

Next up is the chunk callback. This callback allows tools to map chunks to worker threads, and to obtain the chunk iteration range. Furthermore, it allows recording the time spent performing the necessary per-chunk book-keeping, also referred to as the chunk creation duration. I reuse the tool-time function proposed

in Section 3.2.2. A tool-supplied timing function is called by the runtime before and after the chunk book-keeping procedures, and the difference between the obtained time instants is passed back as a parameter. This leads to the chunk callback seen in Listing 3.4.

```
// The proposed ompt_callback_chunk signature
typedef void (*ompt_callback_chunk_t) (
  ompt_data_t *task_data, // The implicit task of the worker
  int64_t lower,          // Lower bound of chunk
  int64_t upper,          // Upper bound of chunk
  double create_duration, // Interval found from tool-supplied instants
  int is_last_chunk       // Is it the last chunk?
);
```

Listing 3.4: The proposed chunk callback.

The `task_data` parameter allows tools to map chunks to implicit tasks. Every worker thread will execute their own implicit task, so this is equivalent to mapping chunks to workers. The `lower` and `upper` parameters unsurprisingly provide the chunk's iteration range bounds. The time spent on chunk bookkeeping is supplied in `create_duration`. Finally, the `is_last_chunk` parameter is used to inform tools that the last chunk is starting execution. This means that its execution will be completed when `ompt_callback_loop` is called.
These callbacks require modest changes to existing OMPT implementations. The proposed `ompt_callback_loop` simply replaces the existing `ompt_callback_work` in code that handles parallel for-loops. Calls to `ompt_callback_chunk` can be inserted in runtime system functions responsible for assignment of chunks to worker threads that execute dynamically scheduled for-loops.

3.3.2 Timing the Bookkeeping of Block-Cyclic Chunks

Statically scheduled loops make use of block-cyclic chunks, and OpenMP implementations therefore do not need to perform any per-chunk runtime calls. In this case, compilers should additionally generate calls to `ompt_callback_chunk`, preferably through a call to the runtime system. However, calling the runtime system for every chunk is expensive for small chunk sizes. Additionally, this overhead is not necessary unless a tool has actually expressed interest in chunk data by registering the chunk callback. This overhead can be avoided by checking if the callback is registered, and only conditionally calling the runtime system.

The OMPT interface differentiates between mandatory and optional instrumentation callbacks. Should a variant of the proposed extensions make it into OMPT, they would likely be optional. When registering for optional callbacks, the implementation will return its level of support for the callback. A return value of `ompt_set_sometimes` indicates that the callback will be invoked for an implementation-defined subset of associated event occurrences. This allows OpenMP implementors to gradually improve their support for more demanding changes,

such as the code generation change required to support the chunk callback for statically scheduled loops.

As an example, generating code equivalent to the pseudocode of Listing 3.5 allows tracing individual chunks and their iteration ranges, but does not allow measuring the per-chunk bookkeeping time, since there is only a single call to the runtime system before every chunk. As mentioned in Section 3.2.2, the event duration should be reported as 0 in this case. To capture the full bookkeeping duration of a chunk, additional runtime calls are needed to capture the operations that result from the while loop and the assignments of the iteration bounds, `LB` and `UB`.

```
bool callbackPerChunk = __omp_runtime_should_callback_per_chunk();
while (UB = min(UB, GlobalUB), idx = LB, idx < UB) {
  if (callbackPerChunk) {
    __omp_runtime_for_static_chunk(...)
  }
  for (idx = LB; idx <= UB; ++idx) {
    BODY;
  }
  LB = LB + stride; UB = UB + stride;
}
```

Listing 3.5: Basic code generation for partial chunk callback support.

3.4 Implementing the Extensions

In this section, I describe how the proposed extensions were implemented. The task creation extension presented in Section 3.2.2, as well as the for-loop extensions discussed in Section 3.3.1, were implemented in a version of the LLVM OpenMP runtime system. The official version of this runtime system only supports an outdated OMPT specification. However, the group behind OMPT has improved the LLVM OpenMP runtime system with support for the recent OpenMP Technical Report 4 [27]. I use this version as the basis for my extensions. The code is available for review [29, 30].

The LLVM OpenMP runtime system has a clean and simple design, although its documentation could be improved. I had to inspect a large amount of source code both in the runtime system and in Clang to understand how they fit together. I also had to compile and run test programs, and examine assembly code generated by Clang to verify some of my assumptions. After gaining the required understanding, I started modifying the runtime system with my extensions.

3.4.1 Task Creation Duration

Most tasking-related functions exist inside the file `kmp_tasking.cpp`. Particularly of interest are the two external API functions `__kmpc_omp_task_alloc` and

__kmpc_omp_task. Whenever a task construct is encountered in source code, the compiler generates two runtime library calls. The first is to __kmpc_omp_task_alloc. As the name suggests, this function allocates the data structures for a task and returns a pointer to the allocated root structure. This call is then typically followed by a call to __kmpc_omp_task[1]. This function attempts to push the task into a thread-local task queue. Should the queue be full, the task is executed immediately.

Because these two functions are always called sequentially, one can start measuring the task creation duration at the beginning of __kmpc_omp_task_alloc, and stop measuring at the end of __kmpc_omp_task. To give an impression of what the runtime system code looks like, a trimmed example is shown. The code responsible for the call to ompt_callback_task_create inside the modified runtime system is shown in Listing 3.6.

```
#if OMPT_SUPPORT
  if (ompt_enabled) {
    ...
    if (ompt_callbacks.ompt_callback(ompt_callback_task_create)) {
      double create_duration = 0;
      if (ompt_callbacks.ompt_callback(ext_tool_time)) {
        const double start =
          thread->th.ompt_thread_info.last_tool_time;
        create_duration =
          ompt_callbacks.ompt_callback(ext_tool_time)() - start;
      }
      // The actual callback
      ompt_callbacks.ompt_callback(ompt_callback_task_create)(
        ...,
        create_duration,
        ...);
    }
  }
#endif
```

Listing 3.6: Implementation of the extended task creation callback.

As seen, the code is only compiled if a macro named OMPT_SUPPORT defined, as is customary for OMPT code in LLVM OpenMP. At runtime, the ompt_enabled variable indicates if a tool has been registered, and most OMPT code checks this variable to avoid doing unnecessary work when there is no tool. I do the same, and also check if the ompt_callback_task_create callback has been registered, and later if the tool has supplied a time function. If all conditions are met, the recorded start instant is fetched, and subtracted from the instant received from a call to the tool-supplied time function. If the tool has not registered a time function, a creation duration of 0 is used instead. The ompt_callback_task_create callback is then called. Equivalent code is inserted in other task scheduling functions as

[1]Depending on the OpenMP clauses that decorate the task, other function calls might be needed instead. For instance, an if clause will result in a conditional call to __kmpc_omp_task_begin_if0.

well, as `__kmpc_omp_task` is not the only function where task creation ends. Because creating tasks in LLVM OpenMP requires more than 100 cycles, as shown in Section 3.2.1, we never return 0 to signal that the event is too short.

3.4.2 Extended For-loop Events

In this section, the implementation of extended for-loop tracing events is discussed. Since statically and dynamically scheduled loops are implemented differently, I describe them separately.

3.4.2.1 Static Schedule

In LLVM OpenMP, statically scheduled loops result in two runtime calls on every worker thread. The code resides in `kmp_sched.cpp`. Before loop execution, one out of four functions is called depending on the type of the loop iteration variable. The function names start with `__kmpc_for_static_init_`, and end with `4`, `4u`, `8`, or `8u`. The suffixes denote the size in bytes and signedness of the iteration variable. These functions are simply entry points that call instances of an internal C++ template function where the actual loop initialization takes place. Other loop-related entry points follow the same scheme. The template function receives pointers to upper and lower chunk iteration bounds, as well as the chunk stride. The function calculates these values and writes them to the locations pointed to by the pointers. When the call returns, these values are used by worker threads to compute their chunks, as described in Section 3.3.2. I remove the existing code that calls `ompt_callback_work` and replace it with a call to `ompt_callback_loop`.

The chunk callback relies on modified code generation. The code generation scheme previously shown in Listing 3.5 is implemented in Clang. This means that the implementation only has partial support for the chunk callback with statically scheduled loops. This is due to time limitations and the complexity of correctly implementing full support in both the code generation and the runtime system. However, even this partial support is useful when evaluating the overhead tied to the extensions, as it allows us to measure if adding a branch and a runtime system call inside the while-loop significantly impacts chunk scheduling overhead. The chunk callback requires four library entry points, for the same reason that the loop initialization procedure does. Similarly, these entry points call instances of an internal C++ template function. This function simply contains a call to `ompt_callback_chunk` with the appropriate parameters. The `create_duration` argument is passed as 0 to indicate to tools that the duration is too short.

When a worker thread has completed all of its designated chunks, the library function `__kmpc_for_static_fini` is called. This function is used to call the `ompt_callback_loop` callback with the endpoint parameter set to `ompt_scope_end`, to signal that the loop has completed execution.

3.4.2.2 Dynamic and Guided Schedule

The dynamic or guided schedules differ from statically scheduled loops in that chunks are assigned dynamically at run-time. Both schedules are implemented using the same functions inside `kmp_dispatch.cpp`. The start of a for-loop with either schedule results in a call to a function named `__kmp_dispatch_init`. A function parameter is used to indicate if the schedule should be dynamic or guided. The function performs basic initialization, and is also used to implement the first call to `ompt_callback_loop`.

After this, the program will repeatedly request one chunk at a time by calling a runtime function named `__kmp_dispatch_next`. This makes implementing the proposed chunk callback straightforward. At the very beginning of this function, the runtime system notes the current time using a tool-supplied time function. The function then finds an unassigned chunk, and writes the new iteration bounds to pointers provided by the program. Just before the function completes, another call to the tool-time function enables calculating the bookkeeping duration, which is followed by a call to `ompt_callback_chunk`. When the loop is complete, the end-call to `ompt_callback_loop` is performed.

3.4.3 Implementation Complexity

After implementing the extensions, we are able to evaluate their impact on the complexity of both the runtime system and compile. The LLVM OpenMP runtime system is a large piece of software, compromised of roughly 79,000 lines of code spread over 80 source files[2]. It supports advanced features such as a work-stealing task scheduler. To do this, it makes use of advanced data structures such as thread-local task queues from which tasks can be stolen by other threads in a thread-safe manner. This is complex code. In contrast, implementing the extensions was fairly uncomplicated.

Most of the difficulty experienced while implementing the extensions came from learning the interactions between the compiler and the runtime system, and from understanding the LLVM OpenMP code base. Once this was done, most changes were fairly trivial. For instance, calls to the proposed callback `ompt_callback_loop` were placed where calls to `ompt_callback_work` had been previously. Most of the arguments to the new callback were already available, and the code needed to get the remaining arguments was straightforward.

The extension to the `ompt_callback_task_create` callback was equally simple to implement. First, the `event_duration` parameter was added to the callback signature. Since task creation is a thread-local event, I could then leverage the fact that every thread is assigned an OMPT bookkeeping struct named `ompt_thread_info_t`, in which I added a single field of type `double` to hold tool-supplied time instants. I then called the `ompt_tool_time` function when a task was about to be created, and wrote the returned instant to the newly added time field. Subsequently, this value

[2]Found by counting newlines inside files that reside in the *src* directory with the file extensions `.h`, `.c`, or `.cpp`.

was read after task creation to calculate the time difference, as shown previously in Listing 3.6.

The `ompt_callback_chunk` callback, being a completely new callback, was slightly harder to implement. For dynamically scheduled loops, the arguments were already largely available inside the per-chunk function call. However, for statically scheduled chunks, no such runtime function existed. Implementing the required runtime entry points was fairly straightforward, as it was only a matter of figuring out the minimal set of parameters needed to implement the chunk callback. The biggest challenge was to implement the required code generation changes, discussed in Section 3.3.2. This is not unexpected, since Clang is a much larger and more complex code base compared to LLVM OpenMP. However, thanks to the clean design of Clang, even this was not very troublesome. I expect that adding full support for the `event_duration` of statically scheduled chunks will increase the complexity slightly, but it will still be a relatively minor increase compared to the considerable complexity present in other parts of the compiler and runtime system.

As a result of implementing the extensions, `kmp_tasking.cpp` grows from 3266 lines to 3315 lines, and its function count grows from 64 to 65. `kmp_sched.cpp` grows from 995 lines to 1050, while the number of functions increase from 15 to 21. The new functions consist of one template function used to call the chunk callback, four library entry point functions that simply call the aforementioned template function, and a library function used to determine if statically scheduled loops should perform the per-chunk runtime call. Last, `kmp_dispatch.cpp` grows from 2781 lines to 2817, and its function count grows from 65 to 67. To conclude, the extensions can be implemented without new data structures, without thread synchronization, and without too much work. The impact on OpenMP implementation complexity is low.

Chapter 4

Experimental Setup

The proposed extensions have been extensively benchmarked to guide the design and evaluate the end result. I here present the experimental setup used for these benchmarks.

4.1 Computer System

The test machine has two Intel Xeon E5-2630 2.2Ghz 10-core processors, giving 20 CPU cores in total. Each processor has a shared 25MB L3 cache. Each core has private 32KB L1 instruction and data caches, and a 256KB L2 cache. The system has 64GB RAM and runs CentOS Linux with kernel version 3.10. Intel's simultaneous multithreading implementation, Hyper-Threading, can give a speed-up or slowdown depending on the workload. For my tests, it is disabled. OpenMP threads are pinned to physical cores to limit variability between test executions.

4.2 Benchmarks

To evaluate the overheads tied to the extensions, a wide range of benchmarks are utilized. Micro-benchmarks are used to quantify any overheads added by our extensions. A wide range of HPC applications are run to show whether added overheads significantly impact the execution times of actual applications. In this section I describe these benchmarks.

4.2.1 EPCC OpenMP Micro-benchmark Suite

The EPCC OpenMP micro-benchmark suite [6, 7] contains multiple microbenchmarks, each intended to capture specific overheads in the implementation of OpenMP constructs. I use the most recent version of the suite, version 3.1. The proposed extensions require changes to code related to task creation and for-loop scheduling. I therefore run the two benchmark programs related to these areas,

Taskbench, and *Schedbench*, both described below. Both programs rely on a set of parameters: *Outer repetitions* specifies how many times to repeat the test within a benchmark execution, *test time* specifies the target duration for each test, and *delay time* specifies the busy-wait duration inside loop iterations and tasks.

Taskbench

Taskbench measures overheads related to tasking. It contains a total of 10 different tests. These tests capture overheads related to task creation, nested task creation, conditional task creation, task synchronization, and barriers inside tasks. The three most relevant tests are used to evaluate the extensions. I describe them below. For each test, the execution time of a serial reference routine is subtracted from the execution time of the equivalent OpenMP routine. The amount of work assigned to OpenMP test versions is equal to the work performed by the reference versions multiplied by the number of threads. It follows that the expected execution time for the sequential and OpenMP routines are the same. The overhead is defined as the difference in execution times. The tests used are:

MasterTasks All tasks are generated by the master thread.

ParallelTasks Tasks are generated on every worker thread.

NestedTasks Tasks are generated on every worker thread, and these tasks further generate more tasks.

The parameters used for Taskbench are: Outer repetitions is increased from 20 to 50 to reduce the variance in the overheads reported by the program. For the same reason, the test time is increased from 1000 µs to 30 000 µs. The default delay time is used, 0.1 µs. There is some variability between different executions of Taskbench. The author speculate [6] that this is due to synchronisation variables being placed at different physical memory locations from run to run. Another reason could be background jobs that interfere by using CPU and memory resources. I minimize accidental measurement bias by reporting the median of 20 different runs.

Schedbench

Schedbench quantifies overheads tied to for-loop scheduling. The three scheduling types of OpenMP; `static`, `dynamic`, and `guided`; are tested with different chunk sizes. Each scheduling type is tested with every chunk size that is a power of two, up to 4096. In other words, the sequence of tested chunk sizes is equal to $\{1, 2, 4, 8, \ldots, 4096\}$. As with Taskbench, OpenMP routines are compared with a sequential reference version. The overhead is the difference in execution times.

Schedbench is parameterized as follows: As with Taskbench, I use 50 outer repetitions, a test time of 30 000 µs, and a delay time of 0.1 µs. In this benchmark, each worker thread executes a set number of loop iterations. The number of iterations per thread is increased from the default of 128 to 4096. This means that the benchmark spends proportionally more time scheduling chunks, which

Application	Domain	Computation structure	Nested tasks	Number of tasks (medium inputs)
Alignment	Dynamic programming	Iterative	No	4950
FFT	Spectral method	At leafs	Yes	≈ 10 million
Fibonacci	Integer	At each node	Yes	≈ 40 billion
Floorplan	Optimization	At each node	Yes	≈ 67 million
Health	Simulation	At each node	Yes	≈ 17 million
NQueens	Search	At each node	Yes	≈ 377 million
Sort	Integer sorting	At leafs	Yes	≈ 2 million
SparseLU	Sparse linear algebra	Iterative	No	39480
Strassen	Dense linear algebra	At each node	Yes	≈ 1 million

Table 4.1: Summary of applications from BOTS [8].

is what is most affected by my changes. Schedbench is run 20 times, and median results are reported.

4.2.2 Barcelona OpenMP Tasks Suite

The Barcelona OpenMP Tasks Suite [8] (BOTS) is a collection of applications that makes use of the task construct of OpenMP. Programs from BOTS are commonly used by researchers that do work related to tasks [31, 32, 33], and are therefore fairly well understood [34]. All programs are written in the C programming language. Key properties of the programs that make up BOTS are shown in Table 4.1. Where available, large inputs are used, and medium inputs otherwise. I report the median results of 20 runs.

4.2.3 SPEC OMP2012 Benchmark Suite

The SPEC OMP2012 benchmark suite [9] contains a set of applications from different science domains. The applications employ a variety of OpenMP directives and are written to stress different parts of HPC systems, including the processor core and various memory hierarchies. The suite includes programs written in Fortran, C, and C++. My extensions rely on modified code generation in the compiler. As mentioned in Section 3.4.2, these modifications are implemented in Clang. Therefore, the Fortran programs of SPEC OMP2012 are not used. Table 4.2 contains an overview of the programs used. The programs use reference inputs, except for 376.kdtree. The original grain graphs paper [3] reported a bug in this program that SPEC has since acknowledged. Using reference inputs with a fixed version of this program results in excessive execution times. Increasing the cutoff from 2 to

Application	Domain	LOC	OMP directives
352.nab	Molecular modeling	11,485	`parallel`, `for`, `barrier`, `flush`
358.botsalgn	Sequence alignment	1,277	`parallel`, `for`, `task`
359.botsspar	LU factorization	209	`parallel`, `single`, `task`, `taskwait`
367.imagick	Image processing	96,810	`parallel`, `for`, `single`, `master`, `section(s)`, `critical`
372.smithwa	Optimal pattern matching	2,561	`parallel`, `barrier`, `flush`
376.kdtree	Sorting and searching	287	`parallel`, `for`, `task`, `taskwait`, `single`, `atomic`

Table 4.2: Overview of applications written in C or C++ from SPEC OMP2012 [9].

8 fixes this. The programs are run 12 times, and the median execution times are reported.

352.nab makes use of nested parallelism. Non-deterministic scheduling in the LLVM OpenMP runtime system causes the application's execution time to fall into one of three intervals which are quite far apart. For this program, nested parallelism is disabled by setting the environment variables `OMP_NESTED` and `OMP_DYNAMIC` to false. This eliminates the non-deterministic execution time binning.

4.3 OpenMP Runtime Systems

LLVM, the LLVM OpenMP runtime system, and Clang are developed separately, but they follow the same release cycle. At the time of writing, the most recent stable release for all three projects is version 4.0. This version of LLVM OpenMP supports an outdated OMPT specification. I will refer to this runtime as *TR2*, as it supports a subset of the OpenMP Technical Report 2 [28]. The group behind OMPT has augmented the TR2 runtime system with support for the more recent OpenMP Technical Report 4 [27], and it is therefore referred to as *TR4* in this thesis. The extensions are implemented by modifying the TR4 runtime system. I refer to this runtime system as *TR4E*. Every benchmark is run with the TR2, TR4, and TR4E runtime systems.

4.4 Compilers

Benchmark programs, OpenMP tools, and different versions of the LLVM OpenMP runtime are generally compiled using LLVM Clang version 4.0 with $-O3$ optimization. Because the chunk scheduling extension relies on slightly different code generation, a modified version of Clang is used when compiling benchmark programs that use the TR4E runtime. Specifically, the code generated when the OpenMP

Runtime system	**for** construct	**task** construct
TR2	ompt_event_loop_begin ompt_event_loop_end	ompt_event_task_begin
TR4	ompt_callback_work	ompt_callback_task_create
TR4E	ompt_callback_loop ompt_callback_chunk	ompt_callback_task_create

Table 4.3: Callbacks registered by test tools are runtime system specific.

for directive is encountered has been changed to match the scheme presented in Section 3.3.2.

4.5 Tools and Callbacks

All benchmarks are run with and without tools attached. When a tool is attached, two tool variants are used: a *no-callback tool* that registered no callbacks, and a *with-callbacks tool* that registered relevant callbacks that are empty. Callbacks registered by the tools are shown in Table 4.3. These differ across runtime systems because they support different versions of OMPT. In the TR2 runtime system, there is no direct equivalent to the ompt_callback_task_create callback of TR4. ompt_event_task_begin is used instead, as it is the closest match. Both callbacks are called once before a task starts execution.

Chapter 5

Results and Discussion

In this chapter I present benchmark results and discuss their meaning. The overhead of supporting the extensions is examined by comparing the results for the TR4 and TR4E runtime systems. The TR2 runtime is included in the results to also show the overheads in the current official LLVM OpenMP runtime system. To avoid verbosity, I refer to callbacks that describe parallel for-loop and chunk events as loop and chunk callbacks respectively. Benchmarks are run on 20 threads, one thread per physical core. Threads with even and odd thread numbers run on different CPUs. The results discussed here are also available [35] in a tabular format for review.

5.1 Schedbench

Schedbench measures for-loop scheduling overhead. First, we look at overheads incurred with no tool attached, and with a no-callback tool attached, since these are the two most important scenarios. These scenarios matter the most due to the design objectives of OMPT, discussed in Section 3.1. We start by looking at statically scheduled loops. The measurements are shown in Figures 5.1a and 5.1b. It can be seen that the TR2 runtime schedules the loop with less overhead. This is a result of only incorporating support for an outdated OMPT specification with less features, and is also seen for other schedule types. Since the extensions are implemented in the TR4 runtime system, I focus on comparing TR4 and TR4E.

Scheduling overhead with no tool attached is shown in Figure 5.1a. TR4E performance is virtually identical to TR4 for small chunk sizes, and becomes 1% faster for chunk sizes of eight or higher. This is due to the proposed loop callback being slightly simpler to implement than its predecessor, and therefore slightly faster. Figure 5.1b shows the same benchmark with a no-callback tool attached. The trend of up to 1% lower overhead is also observed here. This means that when not using instrumentation callbacks, the for-loop extensions do not add significant overhead to statically scheduled loops.

We now focus on for-loops that use the dynamic schedule, shown in Figures

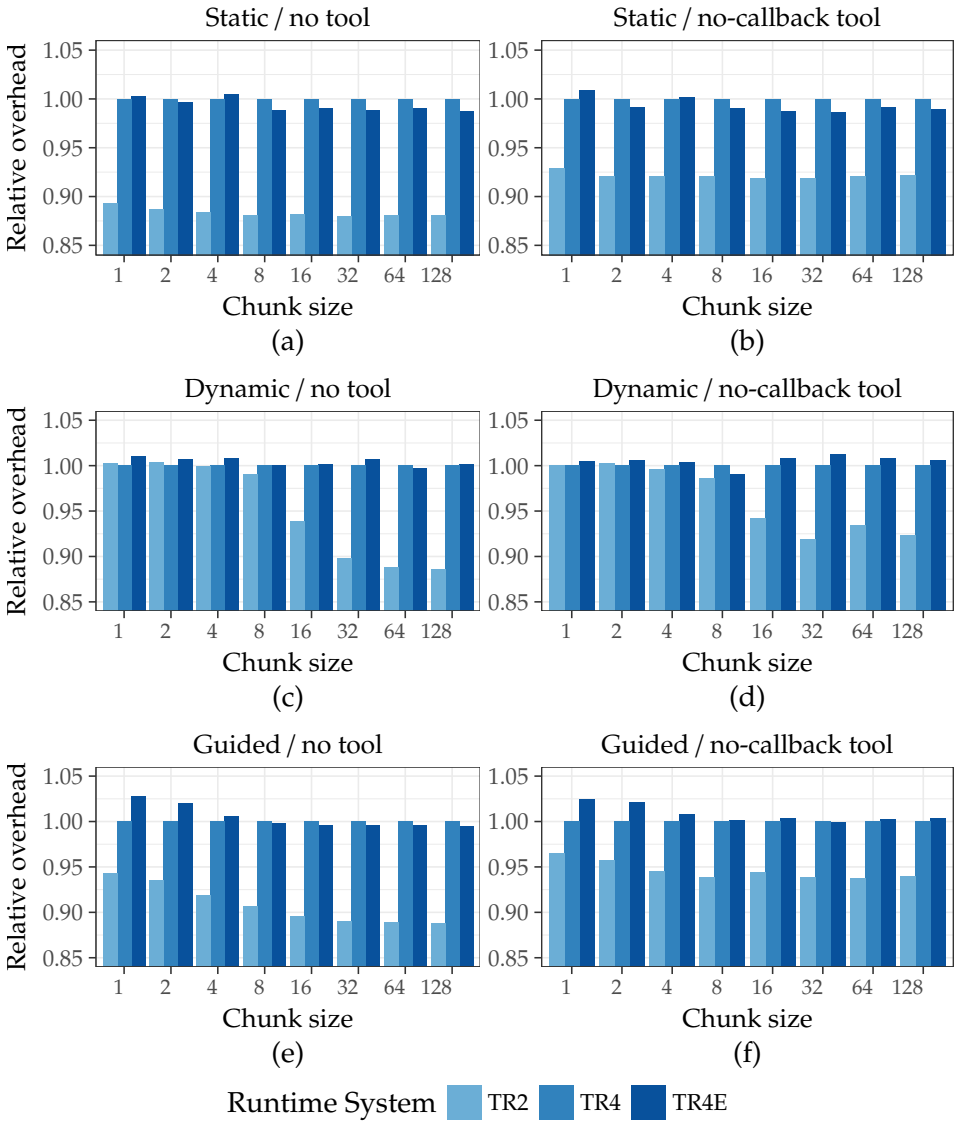


Figure 5.1: Relative overhead of loop scheduling measured with Schedbench in two different scenarios: no tool attached and a no-callback tool attached. TR4 is the baseline.

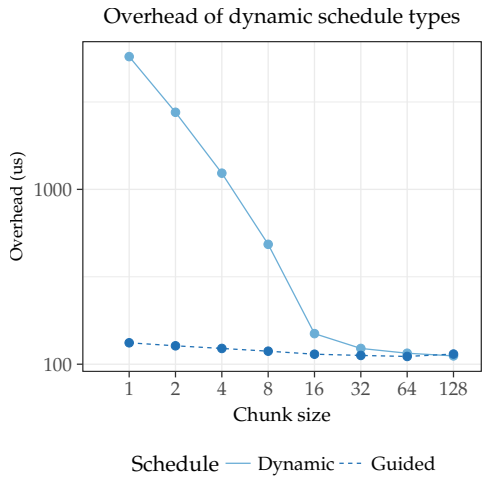


Figure 5.2: The `dynamic` schedule type incurs excessive overhead compared to `guided` for small chunk sizes.

5.1c and 5.1d. Here, a slight increase in overhead can be observed, up to 1.5%. This is not too surprising, since code has been inserted at the start end end of the function that assigns chunks. Figures 5.1e and 5.1f shows the overheads incurred for the guided schedule. For this schedule, TR4E has up to 3% overhead for chunk sizes below 4. Since the chunk assignment function used for guided chunks is the same as the one used for dynamic chunks, one might expect to see the same in the dynamic schedule charts. However, small dynamic chunks suffer from excessive synchronization overhead because worker threads are constantly requesting new chunks, and this overhead dominates any overhead added by the extensions. This information is not readily apparent in the bar charts since they show overhead relative to TR4. Figure 5.2 shows the absolute overhead of these two scheduling types on the TR4 runtime system with no tool attached, which explains why the added overhead is more visible for guided chunks.

Finally, we look at the overhead incurred when registering the loop and chunk callbacks. Results are shown in Figure 5.3. The dynamic and guided schedule types incur up to 2.7% added overhead. This is expected, since registering for the chunk callback causes per-chunk callbacks. Another unsurprising result is that tracing statically scheduled chunks adds significant overhead. This happens because the overhead tied to block-cyclic chunks are normally very low, roughly 10 cycles. Registering for the chunk callback causes per-chunk runtime system calls that further calls the tool callback. The end result is an increase in scheduling overheads up to 46% for chunk sizes below 16. For chunk sizes above 16, TR4 and TR4E have overheads that are within 1%.

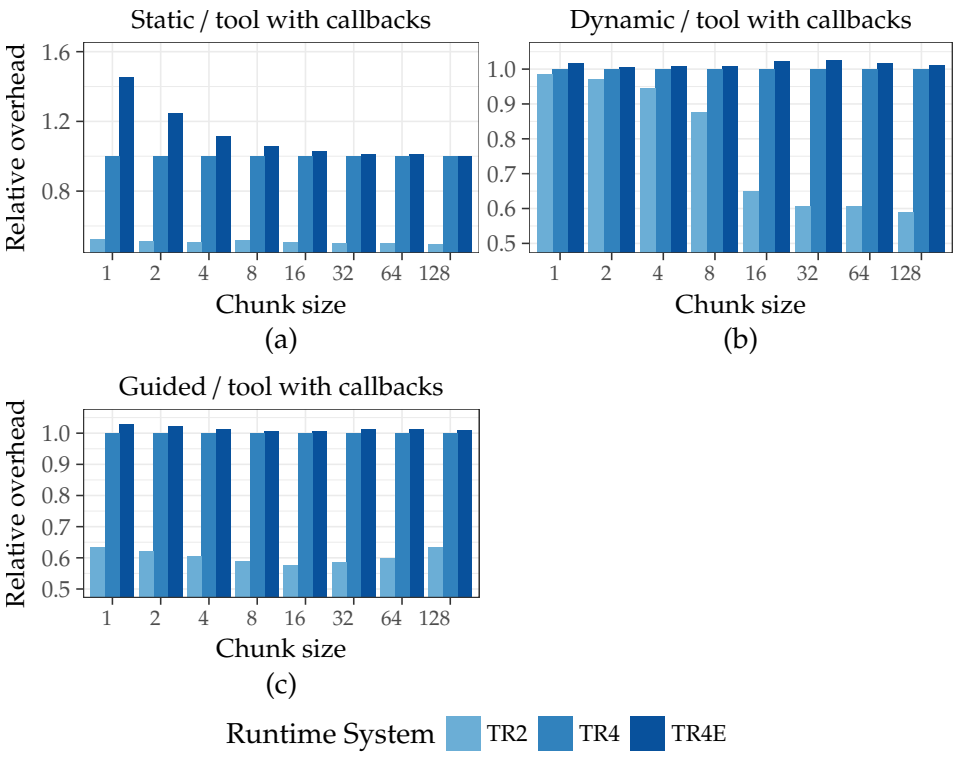


Figure 5.3: Relative overhead of loop scheduling measured with Schedbench when a tool registers callbacks. TR4 is the baseline.

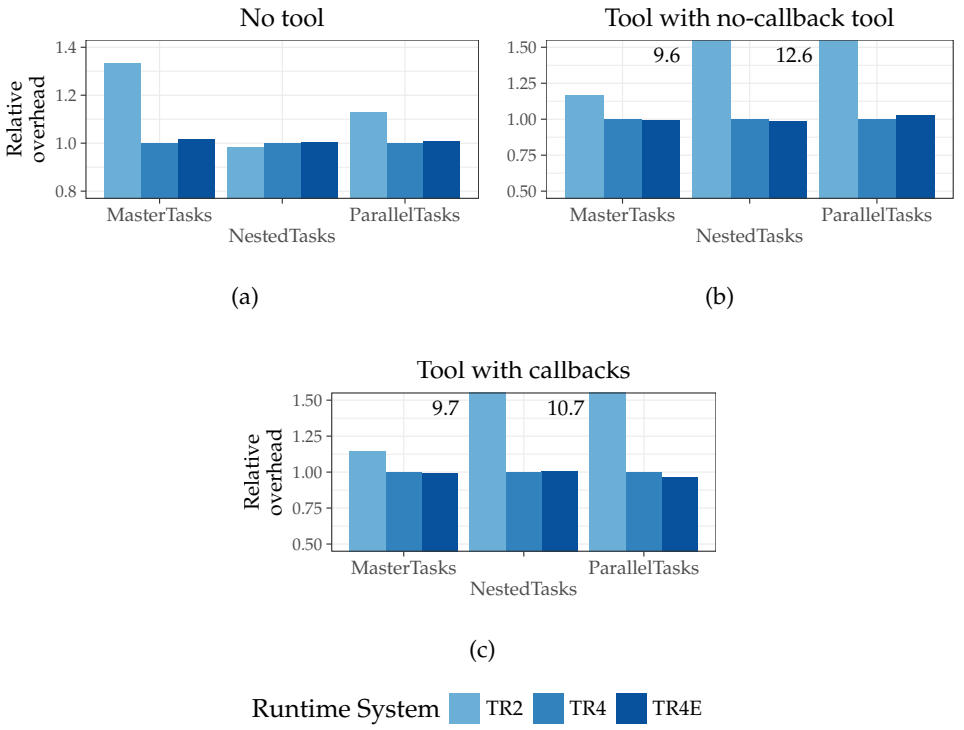


Figure 5.4: Relative overheads of the proposed task creation extension measured with Taskbench. TR4 is the baseline.

5.2 Taskbench

Taskbench captures overheads tied to various aspects of tasking. I present the results of tests within Taskbench that focus on task creation, since this is what the proposed task creation extension is concerned with. Figure 5.4a displays relative task creation overheads with no tool attached. The biggest difference between TR4 and TR4E is seen for the MasterTasks test, where TR4E adds 1.7% overhead. Besides this case, the overheads are within 0.6%. Figure 5.4b shows the results when attaching a no-callback tool. The ParallelTasks test suffers from 2.7% higher overheads on TR4E, while other tests are within 0.5%. Last, attaching a tool that registers for the task creation callback results in no observable performance difference between TR4 and TR4E, as seen in Figure 5.4c. In fact, the ParallelTasks test reports slightly lower overhead on TR4E. This is due to an incidental memory access optimization in the TR4E implementation.

TR2 shows poor task creation performance in general, but in the two cases where a tool is attached, the two benchmarks ParallelTasks and NestedTasks perform catastrophically compared to the other runtimes. This is because the older OMPT specification mandates that tasks are assigned unique integer IDs by the

runtime system. In the TR2 runtime, this is implemented with a atomic fetch-and-add operation. When creating short tasks on multiple threads, this can become problematic. The culprit is verified to be the atomic fetch-and-add operation by temporarily assigning an ID of 0 to every task instance. This results in task creation overheads in TR2 that resemble those of the other runtime systems. The MasterTasks benchmark is naturally not affected, as tasks are created on a single thread.

5.3 Application Suites

Applications from the BOTS and SPEC OMP2012 application suites are used to figure out if our extensions increase the execution times of HPC workloads. Results are shown in Figure 5.5. Starting with the no-tool scenario, shown in Figure 5.5a, we see that the results for 376.kdtree are the only ones that really stand out. TR2 lets the program finish in 88% of the time needed by TR4. This is somewhat surprising, considering that it is a task-based program, and considering that the Taskbench results did not show a clear advantage to TR2. The exact reason for the performance regression in TR4 is not known, and should be investigated later. For the same program, TR4E shows 4% less overhead, due to incidental optimization opportunities used by the modified Clang 4 compiler. TR4E generally performs on par with TR2 and TR4. Its worst result is for Health, which runs 1% slower.

Next we look at the scenario where a no-callback tool is used. The results are shown in Figure 5.5b. 352.nab runs significantly faster on TR2 than on the other runtime systems when a tool is attached. The exact reason is not known. TR4 and TR4E shows identical overhead. Last, we look at the scenario where a tool registers relevant callbacks. The results are presented in Figure 5.5c. Running on all runtime systems, the programs mostly spend the same amount of time executing. However, Health runs 2% slower on TR4E compared to TR4.

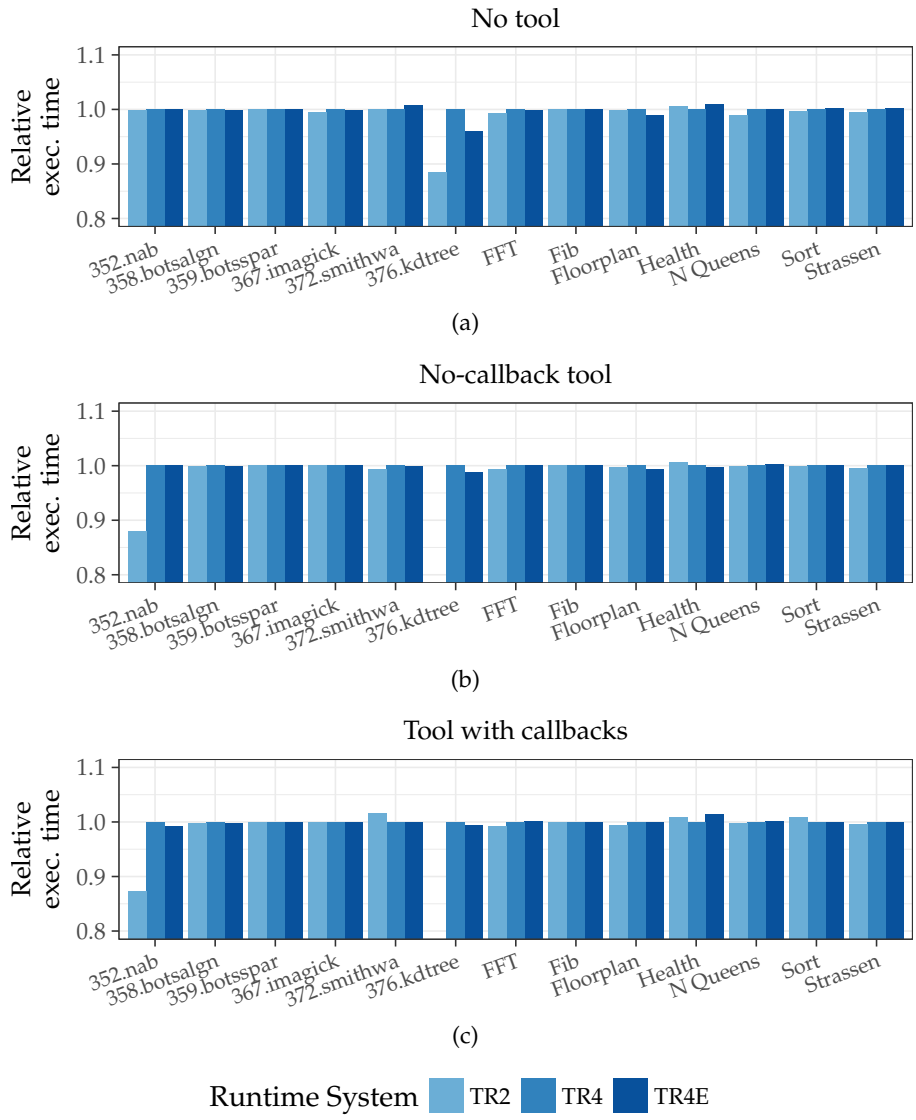


Figure 5.5: Relative execution times of applications from SPEC OMP2012 and BOTS. TR4 is the baseline. Programs with alpha-numeric names are from SPEC OMP2012.

Chapter 6

Related Work

To allow generating grain graphs from data collected through OMPT, I have proposed extending the API with new chunk and loop events, along with a way to measure the duration of task creation and chunk creation. In this chapter, I first discuss similar capabilities in other profiling APIs. Second, I argue that the extensions, while motivated by grain graphs, are useful to others as well.

6.1 Similar Interfaces and Proposals

The POMP API [25] is an earlier attempt to define a standard performance monitor interface for OpenMP. The authors propose a design where both OpenMP programs and OpenMP API functions are instrumented to report about execution events. The proposal includes an extensive set of suggested events. Since the paper precedes the OpenMP 3.0 specification, where the tasking concept was introduced, there is no mention of task-related events. However, POMP includes a list of interesting loop events: `Loop_enter`, `Loop_exit`, `Loop_chunk_begin`, `Loop_chunk_end`, `Loop_iter_begin`, `Loop_iter_end`, and `Loop_iter_event`. As shown, this proposal includes chunk-level events, of which OMPT has none. Furthermore, it even includes events for loop iterations. The last event listed, `Loop_iter_event`, is included by the authors to avoid high overhead for wrapping short events with a matching pair of API calls.

Around the time that the development of OMPT was announced [24], there were multiple groups suggesting additions to the interface. Qawasmeh et al. proposed [36] new extensions for task profiling. The authors propose a set of task-related events, including an event that indicates that a parent task has started creating a child task. The authors also propose two events that denote that task creation has ended. The first event additionally indicates that the created task will start execution immediately, while the second indicates that it will start execution later. The `ompt_callback_task_create` callback of OMPT is similar to the latter two events in that it is called after task creation, but OMPT has no events corresponding to the start of task creation.

6.2 Other Use-Cases

I here show how other researchers and tools demonstrate the utility of the proposed extensions.

6.2.1 Profiling Task Creation Behaviour

In addition to proposing an extension that allows measuring time spent on task creation, Qawasmeh et al. also show an application for this information [31]. Based on the premise that there is no single optimal task scheduling algorithm for task-based programs, the authors want to predict the optimal algorithm on a per-program basis. A trained artificial neural network (ANN) predicts the optimal algorithm based on 14 inputs, all of which are program properties obtained through profiling. Two of these properties are the time spent creating tasks on all threads and the number of cache misses during task creation on all threads. The proposed extensions allow the former, but not the latter. To allow counting cache misses, one would instead need a design where the tool is called before and after task creation.

6.2.2 Tracing and Visualizing For-loop Execution

Aftermath is a trace-based tool for performance analysis, briefly discussed in Section 2.3.1. Drebes et al. [20] extend Aftermath with support for analysis of OpenMP programs. Program execution is traced using an instrumented version of the LLVM OpenMP runtime system, similar to the approach I have used. As discussed in Section 3.3.1, there are no runtime API calls made for block-cyclic chunks. The authors do not solve this problem, and therefore cannot trace the execution of individual chunks in statically scheduled loops – a dominant parallelization pattern. Excluding 367.imagick, 91/105 parallel for-loops in SPEC OMP2012 are statically scheduled.

For statically scheduled loops, they instead record the time elapsed executing all chunks per worker thread, and calculate the chunk iteration ranges that make up this time interval. In a previous iteration of the extensions, I used the same approach for statically scheduled loops. However, because the exact mapping of chunks to worker threads is left implementation-defined, passing enough parameters to portably map iteration ranges to threads inflates the loop callback signature and leaves the actual calculation to tools. It might be possible to improve upon this approach, but since the generation of grain graphs requires per-grain execution time, I opt to trace block-cyclic chunks individually.

Using the traces generated by their instrumented runtime system, Aftermath is able to diagnose load imbalance problems. While the for-loop tracing implementation used by Drebes et al. is more limited than what the proposed extensions would allow, the authors clearly demonstrate the utility of exposing such information to performance analysis tools.

6.2.3 Data-Race Detection for For-loop Chunks

Recently, a new class of dynamic race detection tools that leverage structured parallelism, such as tasks, have emerged [37, 38]. Raman et al. [37] leverage structured parallelism to overcome limitations of existing dynamic race detectors. Yoga et al. [38] build on this work to support programs with locks. A key idea for both race detectors is to use structured parallelism constructs, such as tasks, to determine whether conflicting memory accesses can execute in parallel. They rely on constructing a tree structure called Dynamic Program Structure Tree (DPST), conceived by Raman et al. [37]. The tree captures the dynamic parent-child relationship between tasks. It also captures task synchronization, and can therefore be used to see which tasks may run in parallel. This information is used together with instrumentation of locks [38] and shared memory operations to reveal apparent data races. For task-based OpenMP programs, the DPST can be built using task-level events already present in OMPT.

The proposed loop and chunk callbacks allow building a DPST for programs with parallel for-loops as well, since chunks can be treated like tasks. A DPST contains three types of nodes. *Step nodes* denote regions of normal program execution. *Async nodes* shows task creation, and *finish* nodes show task synchronization points. For statically scheduled loops, the chunks that execute on a given worker thread cannot run in parallel, and can be seen as a set of sequentially executed tasks. These will together form a step node, whose parent will be an async node, who in turn has a parent finish node. This presumes that the thread count does not change between program executions. For dynamically scheduled loops, every chunk may run in parallel, and must therefore have their own step node. Thus it is possible to detect conflicting memory accesses across chunks. We see that because chunks can be treated like tasks, existing methods for task-parallel programs can be used on parallel for-loops as well.

6.2.4 Portable For-loop Metrics

Intel's VTune Amplifier [39] is a commerical, sampling-based performance profiler. It recently gained support for debugging and profiling OpenMP applications. VTune Amplifier is able to determine loop schedule types, chunk sizes, and time spent on loop scheduling. These features rely on source code inspection and call stack sampling. Provided that profiled programs use either the Intel or GCC OpenMP runtime systems, the profiler can map time spent inside observed stack frames to different types of OpenMP overhead. The proposed extensions to OMPT enable tools to portably compute similar metrics without the need for source code inspection or mapping library functions to OpenMP activities.

Chapter 7

Conclusion and Future Work

In this thesis, the goal was to extend OMPT to provide all data required by grain graphs. I have presented extensions to OMPT that add a time duration parameter to the task creation callback, improve information provided by the loop callback, and introduce a new callback to describe chunk events. Together, these extensions allow constructing grain graphs portably from any OMPT-compliant runtime system. The overheads incurred by the extensions are low. As measured with the Schedbench micro-benchmark, for-loop scheduling suffers up to 3% increased overhead, except when using the chunk callback for statically scheduled loops with chunk sizes below 16. Since this causes a costly per-chunk runtime system call, the overhead is up to 46% higher compared to TR4, or compared to TR4E when only the loop callback is registered. The Taskbench micro-benchmark reports that TR4E adds up to 2.7% overhead to task creation. HPC workloads from BOTS and SPEC OMP2012 generally show negligible slowdown when running on TR4E, but Health runs up to 2% slower.

The extensions adhere to OMPT design objectives. They are implemented in a consistent and maintainable manner in a standard toolchain, namely LLVM/Clang. Both the runtime system [29] and the modified Clang compiler [30] are publically available. The impact on source code complexity is found to be low. While the extensions are motivated by grain graphs, the events described by the extensions can be useful to other profiling methods as well. I find examples of other researches that trace chunk execution and measure task creation time. By including such extensions in the OMPT specification, researchers would not have to create custom profiling implementations of OpenMP runtime systems to obtain this data.

Moving forward, this work will be followed up by a presentation at the International Workshop on OpenMP (IWOMP) in New York in September, since my supervisors and I submitted a paper [10] on this topic. This is a good opportunity to reach out to members of the OpenMP Architecture Review Board and discuss the steps required to add the proposed extensions to the OMPT interface. However, even at this stage, a few likely next steps are clear.

Due to time and resource constraints, the proposed extensions were implemented in one runtime system, and tested on one computer architecture. OpenMP

programs are executed on a variety of runtimes and architectures. Therefore, it would be preferable to conduct more performance evaluations on other configurations. This would provide a more comprehensive understanding of the impact that the extensions might have on source code complexity and performance. Once the right people are convinced to include the extensions into OMPT, there would likely also have to be a discussion on the exact design that will make it into the specification. This should include feedback from OMPT users, since these are the individuals that truly know what events and callback parameters are needed inside tools.

Bibliography

[1] L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Computational Science and Engineering*, vol. 5, pp. 46–55, Jan. 1998.

[2] P. E. McKenney, "Is Parallel Programming Hard, And, If So, What Can You Do About It?," *Linux Technology Center, IBM Beaverton*, 2011.

[3] A. Muddukrishna, P. A. Jonsson, A. Podobas, and M. Brorsson, "Grain Graphs: OpenMP Performance Analysis Made Easy," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, (New York, NY, USA), pp. 28:1–28:13, ACM, 2016.

[4] A. Muddukrishna, P. A. Jonsson, and P. Langdal, *anamud/mir-dev: MIR v1.0.0*. Mar. 2017. DOI: 10.5281/zenodo.439351.

[5] P. V. Langdal, "Generating Grain Graphs Using the OpenMP Tools API," 2017. Available at: https://brage.bibsys.no/xmlui/handle/11250/2434632. Master project report.

[6] J. M. Bull, "Measuring Synchronisation and Scheduling Overheads in OpenMP," in *Proceedings of First European Workshop on OpenMP*, vol. 8, p. 49, 1999.

[7] J. M. Bull, F. Reid, and N. McDonnell, "A microbenchmark suite for openmp tasks," in *International Workshop on OpenMP*, pp. 271–274, Springer, 2012.

[8] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP," in *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, (Washington, DC, USA), pp. 124–131, IEEE Computer Society, 2009.

[9] M. S. Müller, J. Baron, W. C. Brantley, H. Feng, D. Hackenberg, R. Henschel, G. Jost, D. Molka, C. Parrott, J. Robichaux, P. Shelepugin, M. van Waveren, B. Whitney, and K. Kumaran, "SPEC OMP2012 - an Application Benchmark Suite for Parallel Systems Using OpenMP," in *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, IWOMP'12, (Berlin, Heidelberg), pp. 223–236, Springer-Verlag, 2012.

[10] P. V. Langdal, M. Jahre, and A. Muddukrishna, "Extending OMPT to support Grain Graphs," in *To appear in: International Workshop on OpenMP*, Springer, 2017.

[11] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 483–485, ACM, 1967.

[12] I. Foster, *Designing and Building Parallel Programs*, vol. 191. Addison Wesley Publishing Company Reading, 1995.

[13] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[14] R. D. Blumofe and C. E. Leiserson, "Space-efficient Scheduling of Multithreaded Computations," *SIAM Journal on Computing*, vol. 27, no. 1, pp. 202–229, 1998.

[15] C. E. Leiserson, "The Cilk++ Concurrency Platform," in *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE*, pp. 522–527, IEEE, 2009.

[16] J. K. Hollingsworth, B. P. Miller, and J. Cargille, "Dynamic Program Instrumentation for Scalable Performance Tools," in *Proceedings of IEEE Scalable High Performance Computing Conference*, pp. 841–850, May 1994.

[17] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, (New York, NY, USA), pp. 89–100, ACM, 2007.

[18] N. Froyd, J. Mellor-Crummey, and R. Fowler, "Low-overhead Call Path Profiling of Unmodified, Optimized Code," in *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, (New York, NY, USA), pp. 81–90, ACM, 2005.

[19] H. Brunst and B. Mohr, "Performance Analysis of Large-Scale OpenMP and Hybrid MPI/OpenMP Applications with Vampir NG," in *OpenMP Shared Memory Parallel Programming*, pp. 5–14, Springer, Berlin, Heidelberg, 2008. DOI: 10.1007/978-3-540-68555-5_1.

[20] A. Drebes, J.-B. Bréjon, A. Pop, K. Heydemann, and A. Cohen, "Language-Centric Performance Analysis of OpenMP Programs with Aftermath," in *OpenMP: Memory, Devices, and Tasks*, pp. 237–250, Springer, Oct. 2016.

[21] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCTOOLKIT: tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, pp. 685–701, Apr. 2010.

[22] J. Weidendorfer, "Sequential Performance Analysis with Callgrind and KCachegrind," in *Tools for High Performance Computing*, pp. 93–113, Springer, 2008.

[23] A. Muddukrishna and P. Langdal, "anamud/grain-graphs: Grain graphs v1.0.0," Mar. 2017. DOI: 10.5281/zenodo.439355.

[24] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis," in *OpenMP in the Era of Low Power Devices and Accelerators*, pp. 171–185, Springer, Berlin, Heidelberg, Sept. 2013. DOI: 10.1007/978-3-642-40698-0_13.

[25] B. Mohr, A. D. Malony, H.-C. Hoppe, F. Schlimbach, G. Haab, J. Hoeflinger, and S. Shah, "A performance monitoring interface for OpenMP," in *Proceedings of the Fourth Workshop on OpenMP (EWOMP 2002)*, pp. 1001–1025, 2002.

[26] M. Itzkowitz, O. Mazurov, N. Copty, Y. Lin, and Y. Lin, "An OpenMP runtime API for profiling," *OpenMP ARB as an official ARB White Paper available online at http://www. compunity. org/futures/omp-api. html*, vol. 314, pp. 181–190, 2007.

[27] OpenMP Language Working Group, "OpenMP Technical Report 4: Version 5.0 Preview 1," Nov. 2016. Available at: http://www.openmp.org/wp-content/uploads/openmp-tr4.pdf.

[28] OpenMP Tools Working Group, "OpenMP Technical Report 2 on the ompt interface," Mar. 2014. Available at: http://www.openmp.org/wp-content/uploads/ompt-tr2.pdf.

[29] P. V. Langdal, *LLVM OpenMP TR4E Alpha Release*. May 2017. DOI: 10.5281/zenodo.570288.

[30] P. V. Langdal, *LLVM Clang 4.0 With Code-generation for TR4E*. June 2017. DOI: 10.5281/zenodo.802855.

[31] A. Qawasmeh, A. M. Malik, and B. M. Chapman, "Adaptive OpenMP Task Scheduling Using Runtime APIs and Machine Learning," in *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pp. 889–895, Dec. 2015.

[32] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "SLAW: A Scalable Locality-aware Adaptive Work-stealing Scheduler," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12, IEEE, 2010.

[33] S.-J. Min, C. Iancu, and K. Yelick, "Hierarchical Work Stealing on Manycore Clusters," in *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS11)*, 2011.

[34] C. B. Zilles, "Benchmark Health Considered Harmful," *SIGARCH Comput. Archit. News*, vol. 29, pp. 4–5, June 2001.

[35] P. V. Langdal, "Extending OMPT to support Grain Graphs - Dataset," June 2017. DOI: 10.6084/m9.figshare.5086837.v1.

[36] A. Qawasmeh, A. Malik, B. Chapman, K. Huck, and A. Malony, "Open Source Task Profiling by Extending the OpenMP Runtime API," in *OpenMP in the Era of Low Power Devices and Accelerators*, pp. 186–199, Springer, Berlin, Heidelberg, Sept. 2013.

[37] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, "Scalable and Precise Dynamic Datarace Detection for Structured Parallelism," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, (New York, NY, USA), pp. 531–542, ACM, 2012.

[38] A. Yoga, S. Nagarakatte, and A. Gupta, "Parallel Data Race Detection for Task Parallel Programs with Locks," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, (New York, NY, USA), pp. 833–845, ACM, 2016.

[39] Intel, "Intel VTune Amplifier Webpage," May 2017. Available at: https://software.intel.com/en-us/intel-vtune-amplifier-xe.

Appendix A

OMPT Mailing List Correspondence

During the fall of 2016, I worked on a preporatory project where I examined OMPT to determine what event descriptions were needed to construct grain graphs. During this time, I also initiated contact with the OMPT mailing list, as seen in the email below. The contents of my original email can be pieced together from the quoted sections. Towards the end, Mellor-Crummey suggests benchmarking task creation and synchronization costs and using the results as constants in calculations needed by grain graphs.

```
Subject: Re: [Omp-tools] A few questions regarding OMPT
Date: 2016-11-09 22:55 CET

John Mellor-Crummey          Professor
Dept of Computer Science     Rice University
email: xxx@yyy.zzz           phone: 111-222-3333

On Nov 9, 2016, at 3:27 PM, Peder Voldnes Langdal <zzz@yyy.xxx> wrote:

> Hi,
>
> I am currently writing a paper that looks at whether or not OMPT can be
> used to implement a recent visualization method called grain graphs[1].
> It is a task-based profiling method, and it relies on collecting many
> performance measurements per task instance. Up until this point I have
> been experimenting with the LLVM OMP(T) implementation when uncertain,
> but as it only implements features from TR2, not everything can be
> answered. As such, I have some questions:

> 1) In an effort to measure overhead associated with a task instance,
> the in-house runtime used to produce grain graphs collects the number
> of cycles used to create a task and resume execution. This is used to
```

```
> calculate a metric called parallel benefit, which is a task's execution
> time divided by the overheads incurred for the parent (task creation,
> task synchronization). Since there is no "task create end" callback in
> OMPT, this seems impossible to do. Is this correct? It could also be
> achieved if there was some other callback that was always called after
> "task create", but I'm assuming there is not.

There is a task_schedule callback when the task gets scheduled. Creation
and scheduling are separate.

> 2) How locked is the OMPT Specification at this point, and are you still
> taking suggestions?

We have contributed a version of OMPT to the OpenMP specification TR,
which (if approved by the OpenMP ARB) will be a draft release for OpenMP
5.0. It is locked for the moment.

> 3) If I, or someone else, wanted to suggest adding callbacks or other
> changes, what would be the appropriate course of action? Note that I'm
> not necessarily talking about the case mentioned in 1), but rather in
> general.

At present, we should wait a week and see what they approve. If you want
to propose a new feature, we could discuss this on a future OMP tools
telecon. Right now we are taking a break for a few weeks after some
furious work getting OMPT into the draft standard.

> Thank you for your time!

Would it work to simply measure task creation and synchronization costs in
a benchmark and then use that constant in producing your grain graphs?
Don't you just need to know how long it takes to create and finish a task?
Perhaps that could be measured with a loop that creates and synchronizes
with empty tasks.

<< Email content ends here. The line below is the only part of the
   original email not quoted by Mellor-Crummey >>

[1] A. Muddukrishna et al. Grain graphs: OpenMP performance analysis made
easy. URL: http://dl.acm.org/citation.cfm?id=2851156
```