Restructuring the OMPi Compiler for the Accelerator Era

MASTER THESIS

submitted to the designated

by the General Assembly of Special Composition
of the Department of Computer Science and Engineering
Examination Committee

by

Alexandros Papadogiannakis

in partial fulfillment of the Requirements for the degree of

MASTER OF SCIENCE
IN COMPUTER SCIENCE WITH EXPERTISE
IN COMPUTER SYSTEMS

February 2016

Dedication

To those who supported me, one way or another, throughout the years.

Acknowledgments

I would like thank Professor Vasslios V. Dimakopoulos, and Spyros Agathos for their help and time over the last five years.

I would also like to thank Christos Theodorakis, Thanasis Koufoulis, Nikos Papanikos and Andromachi Hatzieleftheriou for giving me something to look forward each week, and Tilemaxos Alpos and Tasos Moustakas for the time we spend together during my undergraduate years.

Last but not least I'd like to thank my family who has put up with me and supported me all these years.

Table of Contents

List of Figures

List of Programs

Glossary

abstract syntax trees (AST)

A tree representation of the abstract syntactic structure of source code written in a programming language.

application programming interface (API)

A set of routines, protocols, and tools for building software applications.In particular, it usually refers to the calls available by a software library to the application programmer.

barrier

A point in the execution of a program encountered by a team of threads, beyond which no thread in the team may execute until all threads in the team have reached the barrier and all explicit tasks generated by the team have executed to completion.

construct

An OpenMP executable directive and the associated statement, loop or structured block, if any, not including the code in any called routines. That is, in the lexical extent of an executable directive.

data environment

The variables associated with the execution of a given region.

declarative directive

An OpenMP directive that may only be placed in a declarative context. A declarative directive results in one or more declarations only; it is not associated with the immediate execution of any user code..

device

An implementation-defined logical execution engine.

COMMENT: A device may have one or more processors.

device data environment

A data environment defined by a `target data` or `target` construct.

directive

In C/C++, a line starting with *#pragma*, that specifies OpenMP program behavior.

EELib

execution entity libraries, libraries that implement the abstract execution entities of OMPi's runtime, providing the runtime with execution threads.

enclosing context

In C/C++, the innermost scope enclosing an OpenMP directive.

executable directive

An OpenMP directive that is not declarative. That is, it may be placed in an executable context.

host device

The device on which the OpenMP program begins execution.

implicit variable

or variable with implicitly determined data-sharing attributes is a variable referenced in a given construct, that does not have predetermined data-sharing attributes, and is not listed in a data-sharing attribute clause of the construct.

kernel

An executable created by a `target` construct that is offloaded onto a target device.

master thread

The thread that encounters a parallel construct, creates a team, generates a set of tasks, then executes one of those tasks with thread id 0.

nested construct

A construct (lexically) enclosed by another construct.

ORT

OMPi RunTime, the runtime system used by OMPi.

region

All code encountered during a specific instance of the execution of a given construct or of an OpenMP library routine. A region includes any code in called routines as well as any implicit code introduced by the OpenMP implementation. The generation of a task at the point where a task directive is encountered is a part of the region of the encountering thread, but the explicit task region associated with the task directive is not. The point where a target or teams directive is encountered is a part of the region of the encountering thread, but the region associated with the target or teams directive is not.

structured block

For C/C++, an executable statement, possibly compound, with a single entry at the top and a single exit at the bottom, or an OpenMP construct.

target device

A device onto which code and data may be offloaded from the host device.

threadprivate variable

A variable that is replicated, one instance per thread, by the OpenMP implementation. Its name then provides access to a different block of storage for each thread.

undeferred

A task for which execution is not deferred with respect to its generating task region. That is, its generating task region is suspended until execution of the undeferred task is completed.

Abstract

Alexandros Papadogiannakis.
M.Sc. degree in Computer Science, February 2016.
Department of Computer Science and Engineering, University of Ioannina, Greece.
Restructuring the OMPi Compiler for the Accelerator Era.
Supervisor: Vassilios V. Dimakopoulos.

OpenMP is a widely used framework whose goal is to standardize and ease development of portable parallel applications. It consists of a collection of compiler directives and library routines. Version 4.0 of the specification adds device directives, among other things. These allow the programmer to offload a block of code onto available devices such as GPUs and co-processors.

In the context of this thesis, we will initially discuss about outlining, a method for transforming source code. Outlining is used to transform a block of code into a standalone function, and is used by many compilers for the transformation of OpenMP directives, including OMPi an open source C OpenMP compiler. Outlining in OMPi was part of the transformation of the `parallel` construct, so it was impossible to use it for implementing new constructs. With that in mind we redesigned the outlining procedure, making it independent and autonomous.

With the help of our new outlining procedure, we implemented the `task` construct, which is used for executing code on any device available on our system, along with the rest of the device directives which where introduced in version 4.0 of the OpenMP specification. These are `target data`, `declare target` and `target update`.

Finally we used the new infrastructure to support OpenMP on the Parallella board, which is a credit card sized computer, with a dual-core ARM CPU and a 16 or 64 cores Epiphany coprocessor. Epiphany is used as a targeted device with full OpenMP support.

Εκτεταμενη Περιληψη

Αλέξανδρος Παπαδογιαννάκης.
Μ.Δ.Ε. στην Πληροφορική, Φεβρουάριος 2016.
Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων.
Αναδιάρθρωση του Μεταφραστή OMPi για την Εποχή των Επιταχυντών.
Επιβλέπων: Βασίλειος Β. Δημακόπουλος.

Το OpenMP είναι ένα προγραμματιστικό πρότυπο που έχει στόχο τη διευκόλυνση ανάπτυξης παράλληλων εφαρμογών. Αποτελείται από ένα σύνολο από οδηγίες (directives) προς τον μεταφραστή και ρουτίνες βιβλιοθήκης. Μία από τις σημαντικές δυνατότητες που προστέθηκαν στην έκδοση 4.0 του πρότυπου είναι οι οδηγίες για συσκευές. Αυτές μας δίνουν την δυνατότητα να εκτελέσουμε μέρος της εφαρμογής μας σε διάφορες συσκευές που είναι διαθέσιμες στο σύστημά μας, πέραν του επεξεργαστή, όπως κάρτες γραφικών (GPUs) και συνεπεξεργαστές (coprocessors).

Στα πλαίσια αυτής της εργασίας, αρχικά θα μιλήσουμε για την μέθοδο outlining για τον μετασχηματισμό πηγαίου κώδικα γραμμένου στη γλώσσα C. Η μέθοδος outlining αποτελεί μια διαδικασία μετατροπής ενός τμήματος κώδικα σε ανεξάρτητη ρουτίνα και εφαρμόζεται σε πολλούς μεταφραστές για την υλοποίηση του πρότυπου OpenMP, συμπεριλαμβανομένου του OMPi, ενός μεταφραστή ανοιχτού κώδικα που αναπτύσσεται από την Ομάδα Παράλληλης Επεξεργασίας του Τμήματος Μηχανικών Η/Υ και Πληροφορικής. Η υλοποίηση της όμως στον OMPi, ήταν στενά συνδεδεμένη με τον μετασχηματισμό της οδηγίας `parallel`, πράγμα που καθιστούσε τη χρήση της για άλλες οδηγίες OpenMP ιδιαίτερα δύσκολη και μη μεταφέρσιμη. Για αυτό τον λόγο υλοποιήσαμε έναν νέο, γενικευμένο μηχανισμό outlining, έχοντας ως στόχο να είναι όσο το δυνατό ανεξάρτητος και αυτόνομος.

Χρησιμοποιώντας τη νέα υποδομή outlining, προχωρήσαμε στην υλοποίηση της οδηγίας `target`, η οποία δίνει εντολή να εκτελεστεί ένα κομμάτι κώδικα σε κάποια διαθέσιμη συσκευή, αλλά και συνολικότερα στην υποστήριξη του προτύπου OpenMP 4. Οι άλλες οδηγίες για συσκευές που παρέχει το πρότυπο και υλοποιήσαμε είναι οι `target data`, `declare target` και `target update`. Η οδηγία `target data`

χρησιμοποιείται για τη δέσμευση μνήμης, η οδηγία `declare target` για την δήλωση καθολικών μεταβλητών και ρουτινών και η `target update` για τη μεταφορά δεδομένων.

Τέλος, η νέα υποδομή χρησιμοποιήθηκε για την υποστήριξη του OpenMP στο σύστημα Parallella που είναι ένας ολοκληρωμένος υπολογιστής σε μέγεθος πιστωτικής κάρτας, με ένα διπύρηνο ARM επεξεργαστή και έναν Epiphany συνεπεξεργαστή με 16 ή 64 πυρήνες. Ο τελευταίος χρησιμοποιήθηκε ως συκευή, με πλήρη υποστήριξη OpenMP στον κώδικα που εκτελεί.

Chapter 1

Introduction

1.1 Background

Ever since the appearance of the first microprocessor, the traditional method of increasing computer performance was to decrease the size of transistors and logical gates. Unfortunately at some point, the performance gains provided by this approach started to diminish due to excessive energy leaks that resulted from silicon layers and devices coming closer and closer. By that time, not only did performance improvements of each new generation start to decrease, but also energy consumption started to skyrocket. This forced industry to turn to a new CPU model, and around 2005 multi-core processors were introduced in the personal computer market. Parallel systems started to become mainstream and are nowadays ubiquitous.

Up to that point, parallel systems were used only in data centers and research facilities for providing supercomputing power. The inclusion of multi-core processors in personal computers forced software developers to change their programming philosophy, if they wanted to take full advantage of the systems their code was executed on. The shift to parallel programming is not an easy task, since both industry and academia were focused exclusively on a standard serial way of programming [1].

To take full advantage of the available processors, there have been many proposals for new, inherently parallel programming languages (e.g. Cilk and Satin [2]). However, none of them has really proven popular and are mostly research prototypes.

Keeping the same base programming language seems to be the only viable solution, as far as productivity is concerned. In this spirit, OpenMP which is a framework that allows easy parallelization of C/C++ and Fortran programs, was realised in 1997. With OpenMP, running a loop in parallel is as simple as adding one code line before the loop they want to parallelize.

Technology continues to evolve, and multi-core processors have found their way into many electronic appliances, such as cellphones, tablets and smart-TVs. Just as our everyday electronics advanced, so did supercomputers. If we take a look at TOP500 list (a list of the fastest supercomputers), we will notice that supercomputers have started to shift from increasing the number of CPUs to adding an increasing amount of co-processors. For example each node of Tianhe-2, the worlds fastest supercomputer since 2013, with a peak speed of 33.86-petaflops, consists of two Intel Ivy Bridge Xeon processors and three Xeon Phi co-processor chips, and Titan, the second fastest supercomputer at the time of writing, at 17.59-petaflops, is pairing AMD Opteron CPUs with Nvidia Tesla GPUs.

Along with the technological improvements in hardware and system architecture, programming tools must also evolve. To that end, OpenMP v4.0 [3] was released in July 2013, which among other things introduced device constructs. These can be used to effortlessly execute code on accelerator devices such as co-processors, GPUs and DSPs among others. By using only a few extra lines of code, the programmer can execute parts of him program on the available devices, without the need to learn a new way of programming for each one of them.

1.2 Objectives

The objective of this work is to design and implement the necessary facilities to drive the OMPi compiler into the accelerator era. OMPi is an open source research compiler developed by Parallel Processing Group (PPG) of University of Ioannina. OMPi uses the outlining technique to transform the parallel construct. Because of how tightly connected the current outlining procedure was with the transformation code of the `parallel` construct, any attempt to use it elsewhere has resulted in duplicating the code, as with the case of `task` transformation.

The first major concern in this thesis was to design and implement a totally new, general, outlining infrastructure that could be applied to re-organize the transformation parts of the compiler in a straightforward way.

The above infrastructure paved the way for our second objective, that of supporting attached devices (GPUs, coprocessors, accelerators, etc). In particular, we implement the new `target` construct, which also needs outlining during its transformation, and is used to execute code on arbitrary accelerator devices. Along with the `target` constructs, we also implement other device construct. The `target data` is used to create data environments on the targeted devices, or in other words to allocate variables on the device's memory. The `target update` construct is used to move data to and from the device's memory. Finally the `declare target` construct can be used to create global variables in the devices, and make functions available.

We will also describe the additions made to the OMPi compiler in order to accommodate the new constructs, and study the extensions to the existing runtime and the addition of modules. Modules are new libraries, one for each type of device we want to offload code to, and are in charge with communications between the host program that runs on the cpu and the kernels, executables that run on the devices.

Finally we present a module which allows full exploitation of the popular Parallella [4] board. In particular, we present the first implementation of OpenMP for this credit-card-sized 18-core system that treats a dual-core ARM processor as the host and a 16-core Epiphany co-processor as an accelerator device, allowing full OpenMP code on both sides.

1.3 Thesis Organization

A brief description of the following chapters:

- Chapter 2 introduces OpenMP. First we present the most common directives available. Then we focus on the `parallel` and `task` constructs, and the clauses available for each of them.

- In Chapter 3 we discuss about OMPi, an OpenMP compiler and runtime system for the C language. We walk through the compilation process before inspecting the compiler and introducing ORT, its runtime system.

- Chapter 4 explains the concept of *outlining*. Although outlining is not a new idea, and was already present in OMPi, we design and implementa new outlining infrastructure which is quite general in its applicability. Base on this, we give new transformations of `parallel` and `task` constructs.

- In Chapter 5 we give a brief overview of the additions in version 4.0 of OpenMP, and we explore the new device directives and the capabilities they offer.

- In Chapter 6 we discuss how the device directives were implemented in the context of the OMPi compiler. The outlining infrastructure from Chapter 4 is used here to transform the `target` construct.
- In Chapter 7 we talk about the additions to OMPi's runtime system that were necessary in order to support the device directives. We also demonstrate the available modules, libraries that are used to target each different device type. We place particular emphasis on the Parallella board, a miniature 18-core system, for which we present the first available OpenMP implementation.
- Finally Chapter 8 represents a summary of what was discussed during the thesis, and examines future directions of work and research possibilities.

Chapter 2

OpenMP

OpenMP [5] is an application programming interface (API) whose purpose is to simplify the parallelization of serial programs. It provides a model for parallel programming that is portable across shared memory architectures from different vendors. In contrast to other APIs such as POSIX threads, OpenMP is a higher lever API which allows the programmer to parallelize a serial program in a simple, controlled and incremental way.

OpenMP consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. It provides directives for expressing parallelism, worksharing and synchronization. The directives are added to an existing serial program written in C/C++ or Fortran in such a way that they can be safely discarded by the compilers that do not support OpenMP (thus leaving the original program unchanged). As a consequence, OpenMP extends rather than changes the base language (C/C++ or Fortran). A “hello world” can be seen in Program 2.1.

In the following sections we present an overview of the core functionality of OpenMP up to Version 3.1 of the standard, and then focus on `parallel` and task constructs, and their clauses. The implementation of these two constructs in OMPi (Chapter 3) uses *outlining*, a technique whose redesign is discussed in Chapter 4. The new directives introduced in the recent Version 4.0 of OpenMP, and in particular the device model, are presented separately in Chapter 5.

```c
#include <stdio.h>
#include <omp.h>

int main(void) {
    #pragma omp parallel num_threads(4)
    printf("Hello World from thread %d.\n", omp_get_thread_num());
}
```

(a) Code

```
Hello World from thread 2.
Hello World from thread 1.
Hello World from thread 3.
Hello World from thread 0.
```

(b) Output

Program 2.1: OpenMP “Hello World”

2.1 Overview of OpenMP Directives

The OpenMP directives consist of three parts. Their format in C is as follows:

```c
#pragma omp <directive-name> [clause[[,] clause]...]
```

All the OpenMP directives begin with *#pragma omp* followed by the name of the directive. A number of zero or more clauses may be specified, seperated by commas or spaces. These control various aspects of the directive, such as variable scopes. Most of the directives apply to the succeeding statement, which must be a structured block.

In the following sections, we will present the most important programming constructs (directives, clauses, runtime routines etc) that OpenMP offers. However, we will concentrate mainly on the *parallel* and *task* constructs which constitute the two core parallelization constructs of OpenMP and are of direct interest in this thesis. Their implementation is based on a complex procedure called *outlining* which is described in Chapter 4.

2.1.1 Constructs for Generating Threads and Tasks

In OpenMP v3.1 [5] there are two constructs that generate threads and tasks. These are:

- *#pragma omp parallel:* The fundamental construct of OpenMP. When a thread encounters a `parallel` construct, a team of threads is created to execute the parallel region. The thread that encountered the `parallel` construct becomes the *master* thread of the new team, with a thread number of zero for the duration of the new parallel region. More information on `parallel` construct can be found in Section 2.3.

- *`#pragma omp task:`* The `task` construct generates an explicit task from the code for the associated structured block, whose execution may be deferred until later. More information on `parallel` construct can be found in Section 2.4.

2.1.2 Worksharing Constructs

A worksharing construct distributes the execution of the associated region among the members of the team that encounters it.

- *`#pragma omp for:`* The `for` construct specifies that the iterations of one or more associated loops will be executed in parallel. The iterations are distributed across threads that already exist in the team executing the parallel region to which the loop region binds.

- *`#pragma omp sections:`* The `sections` construct is a construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team. Each structured block is executed once by one of the threads in the team.

- *`#pragma omp single:`* The `single` construct specifies that the associated structured block is executed by only one of the threads in the team (not necessarily the master thread).

2.1.3 Other Constructs

- *`#pragma omp master:`* The `master` construct specifies a structured block that is executed by the master thread of the team.

- *`#pragma omp critical:`* The `critical` construct restricts execution of the associated structured block to a single thread at a time, i.e. it defines the following region as a critical region; it implies mutual exclusion.

- *`#pragma omp barrier:`* The `barrier` construct specifies an explicit barrier at the point where the construct appears.

- *`#pragma omp taskwait:`* The `taskwait` construct specifies a wait on the completion of child tasks of the current task.

- *`#pragma omp atomic:`* The `atomic` construct ensures that a specific storage location is accessed atomically, rather than exposing it to the possibility of multiple, simultaneous reading and writing threads that may result in indeterminate values.

- *`#pragma omp ordered`:* The `ordered` construct specifies a structured block in a loop region that will be executed in the order of the loop iterations. This serializes and orders the code within an ordered region while allowing code outside the region to run in parallel.

2.2 Runtime Library Routines

In addition to compiler directives, OpenMP offers a number of routines that allows the programmer to control dynamically the parallelization, and to obtain information about the program and its execution environment. They are divided into three categories:

- ***execution environment routines:*** These affect and monitor threads, processors, and the parallel environment. They include, but are not limited to, functions that query and change the number of threads (`omp_set/get_num_threads()`), change the scheduling loop policy (`omp_set_schedule()`), find out the calling thread's id (`omp_get_thread_num()`), and tell us if the call to the current executing routine is enclosed by an active parallel region (`omp_in_parallel()`).

- ***lock routines:*** General-purpose lock routines that can be used for synchronization. There are two types of locks supported, simple locks and nestable locks.

- ***timing routines:*** They provide a portable wall clock timer.

2.3 Parallel Construct

When a thread encounters a `parallel` construct, a team of threads is created to execute the parallel region. The thread that encountered the `parallel` construct becomes the master thread of the new team, with a thread id of zero for the duration of the new parallel region. All threads in the new team, including the master thread, execute the region. Once the team is created, the number of threads in the team remains constant for the duration of that parallel region.

Within a parallel region, thread numbers uniquely identify each thread. Thread numbers are consecutive whole numbers ranging from zero for the master thread up to one less than the number of threads in the team. A thread may obtain its own thread number by a call to the `omp_get_thread_num()` library routine.

The structured block of the `parallel` construct determines the code that will be executed by each thread.

There is an implied barrier at the end of a parallel region. After the end of a parallel region, only the master thread of the team resumes execution of the enclosing task region.

If a thread in a team executing a parallel region encounters another `parallel` directive, it creates a new team, and it becomes the master of that new team. This is called nested parallelism, and there are runtime routines to disable support of nested parallelism, in which case the thread that encounters the `parallel` directive ignores it and runs the code sequentially.

2.3.1 Parallel Clauses

Just like most OpenMP directives, `parallel` directive supports a number of optional clauses that modify its behavior. These are:

- `if(`*scalar-expression*`)`

  If the clause exists, and it evaluates to false no threads are created and the structured block is executed sequentially.

- `num_threads(`*integer-expression*`)`

  Specifies the number of threads that will execute the parallel region.

- `default(shared | none)`

  If `default(none)` is selected, each variable that is referenced in the construct must have its data-sharing attribute explicitly determined by being listed in a data-sharing attribute clause (private/firstprivate/shared/copyin/reduction).

- `private(`*list*`)`

  The `private` clause declares one or more list items to be private to a thread.

- `firstprivate(`*list*`)`

  The `firstprivate` clause declares one or more list items to be private to a thread, and initializes each of them with the value that the corresponding original item has when the construct is encountered.

```c
#define NUM_STEPS 100000

int main () {
    int i;
    double x, pi, sum = 0.0, step = 1.0 / NUM_STEPS;

    #pragma omp parallel for reduction(+:sum) private(x) num_threads(8)
    for (i = 1; i <= NUM_STEPS; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    printf("Pi = %.10g", step * sum);
}
```

Program 2.2: Reduction example - Calculation of π

- `shared`(*list*)

  The `shared` clause declares one or more list items to be shared among the threads.

- `copyin`(*list*)

  The `copyin` clause provides a mechanism to copy the value of the master thread's threadprivate variable to the threadprivate variable of each other member of the team executing the parallel region.

- `reduction`(*operator* `:` *list*)

  The `reduction` clause specifies an operator and one or more list items. For each list item, a private copy is created in each implicit task, and is initialized appropriately for the operator. After the end of the region, the original list item is updated with the values of the private copies using the specified operator. An example of reduction usage can be seen in Program 2.2.

2.4 Task Construct

An OpenMP task is a unit of work scheduled for asynchronous execution by an OpenMP thread. Each task[1] has its own data environment and can share data with other tasks.

[1]Tasks are distinguished as explicit and implicit ones. The former are declared by the programmer

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

Program 2.3: Calculating Fibonacci With The Use Of `task`

A task may generate new tasks and can suspend its execution waiting for the completion of all its child tasks with the `taskwait` construct. Within a parallel region, task execution is also synchronized at barriers: OpenMP threads resume only when all of them have reached the barrier and there are no pending tasks. Tasks are also classified as tied and untied, depending on whether resumption of suspended tasks is allowed only by the OpenMP thread that suspended them or by any other thread. For details of the implementation of the `task` construct in the OMPi compiler refer to [6].

Program 2.3 presents a simple `task` example. Please note that the `fib()` function should be called from within a parallel region for the different specified tasks to be executed in parallel. Also only one thread of the parallel region should call `fib()` unless multiple concurrent Fibonacci computations are desired.

2.4.1 Task Clauses

- `if(`*scalar-expression*`)`

  If the clause exists, and it evaluates to false no new task is created, the affiliated code is instead executed immediately by the encountering thread.

- `final(`*scalar-expression*`)`

  A task for which execution is sequentially included in the generating task region.

with the `task` construct, while the latter are created by the runtime library, when a `parallel` construct is encountered; each OpenMP thread corresponds to a single implicit task.

That is, it is undeferred and executed immediately by the encountering thread. Any task generated by a final task is also final.

- `untied`

  By default, a task is tied and its suspended task region can only be resumed by the thread that started its execution. If the untied clause is present on a `task` construct, any thread in the team can resume the task region after a suspension.

- `mergeable`

  A task that uses the data environment of its parent task.

- `default(shared | none)`

  Same as in `parallel` (Section 2.3.1).

- `private(`*list*`)`

  Same as in `parallel` (Section 2.3.1).

- `firstprivate(`*list*`)`

  Same as in `parallel` (Section 2.3.1).

- `shared(`*list*`)`

  Same as in `parallel` (Section 2.3.1).

Chapter 3

OMPi

OMPi is a lightweight, open source OpenMP compiler and runtime system for C, developed by the Parallel Processing Group (PPG) [7] of University of Ioannina. It is a research compiler that is developed mainly through diploma, master and PhD thesis and publications of PPG's members. It's source code is freely available in OMPi's website [8]. OMPi strives to be portable, and has been tested and on a variety of operating systems such as Linux, Solaris, Irix and Windows (with the help of the Cygwin project).

OMPi started as a project in 2001, and was the first research compiler [9] to support Version 2.0 [10] of the specifications. A major update of the project yielded a novel modular runtime architecture in 2007 [11] and a new translator in 2008, with full support for OpenMP V.2.5 [12]. Support for OpenMP 3.0 [13] was available in version 1.1.0 (2010), which included an efficient tasking infrastructure [6]. The current target is to support version 4.0 [3] of the specification, with this thesis discussing the implementation of the device directives of OpenMP v4.0.

OMPi consists of a source to source compiler and several threading libraries. The compiler takes as input C code annotated with OpenMP directives, and outputs C99 code which is parallelized by using calls to OMPi's runtime system. There is a common API between the different runtime libraries. All the libraries can be used to parallelize the code, but each implements OpenMP threads in a different manner.

Some of the benefits of using OMPi include a high performance threading system under multicore platforms by leveraging the use of PSTHREADS library, which provides access to both kernel-level and user-level threads [14], efficient nested parallelism support [15], and a highly optimized implementation of OpenMP tasking suitable for the NUMA nature of modern multicore multiprocessors [16]. OMPi performs on par, and sometimes even better than commercial compilers. It also sports a modular runtime architecture that can be used to created custom runtimes / threading libraries tailored to specific needs.

3.1 The Compilation Process

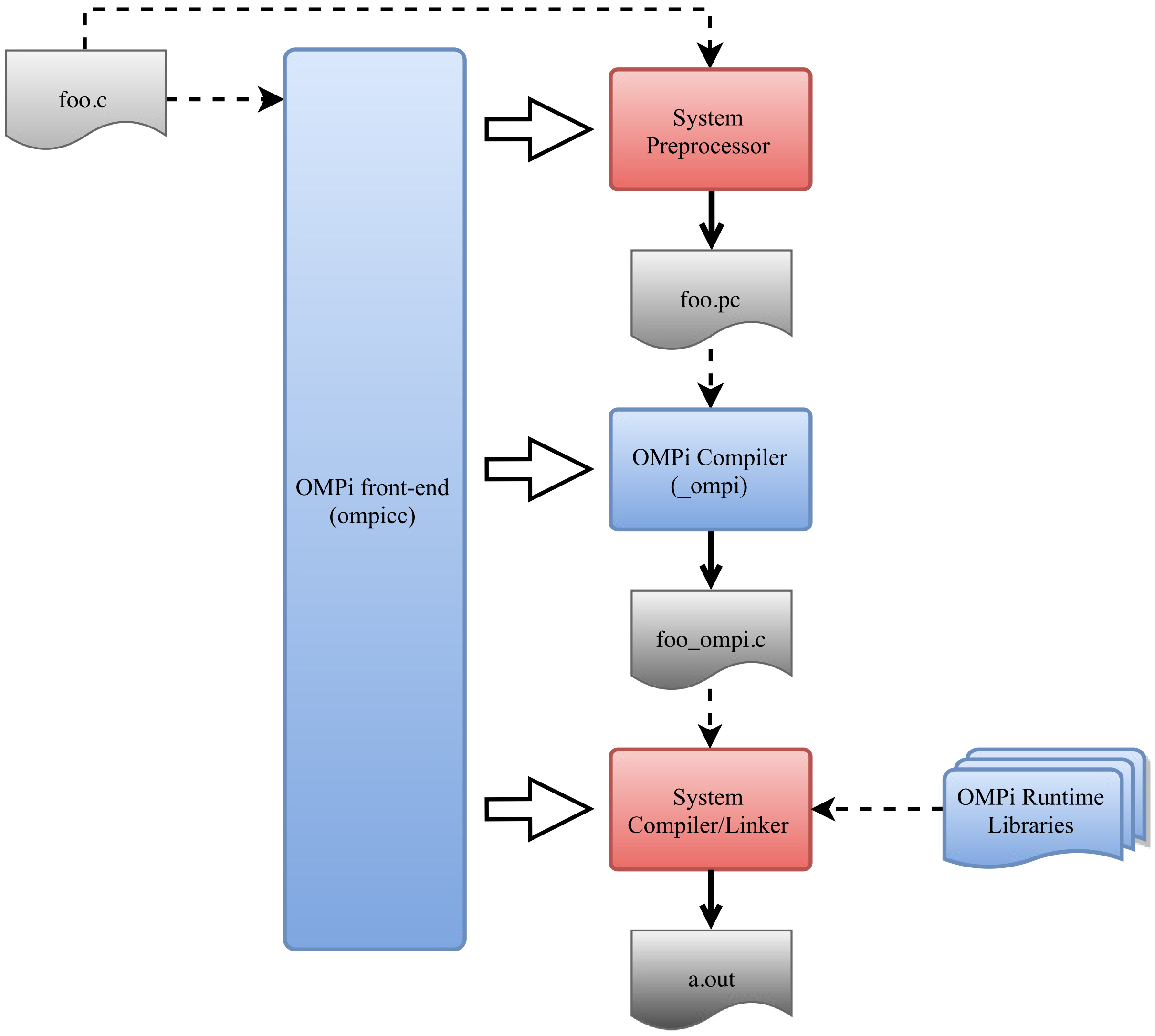


Figure 3.1: Compilation process

Compilation with OMPi is handled using a front-end called ompicc. It is responsible for calling the OMPi compiler and the various system tools[1].

Given a .c file ompicc first calls the system preprocessor on the file. A .pc file is produced which in turn is passed on to the OMPi compiler (_ompi). OMPi compiler parses the .pc file, and transforms any OpenMP directives into calls to functions in OMPi's runtime libraries, producing ans _ompi.c file. Finally the _ompi.c file is given to the system compiler and the system linker along with the selected OMPi runtime library to produce the executable file (a.out).

After the final executable has been produced, ompicc deletes the intermediate files (.pc, _ompi.c)[2], and restarts the process for the next input file, if there are any.

3.2 The Compiler

As mentioned, OMPi's compiler (_ompi) is a source to source compiler. It transforms C code annotated with OpenMP directives into plain C99 code with calls to the runtime system in order to achieve parallel execution of the produced program. The output of the compiler is then given as input to the system compiler which produces a platform-specific executable.

Other than the output being source code, _ompi's organization is just like other compilers. The input file is given to a lexical analyzer (scanner) that generates tokens which are then fed into a parser. The parser consumes the tokens and produces intermediate code. In _ompi the scanner is implemented using flex, the parser is implemented using bison, and the intermediate code is an abstract syntax tree (AST) (more on that in Section 4.2.1). During parsing, the symbol table is populated and identifiers are checked for undefined references and multiple definitions. After the parsing stage, the symbol table is cleared.

OMPi's symbol table is dynamic and scopes cause the addition and deletions of identifier symbols. This means that after parsing is complete, the symbol table has lost all its information about identifiers, scopes and declarations (only the global scope is still there and gets drained afterwards). In compiler terminology, OMPi's symbol table is "imperative", or non-persistent (non-functional) [17].

To speed up lookups the symbol table does not use the identifier directly as a key,

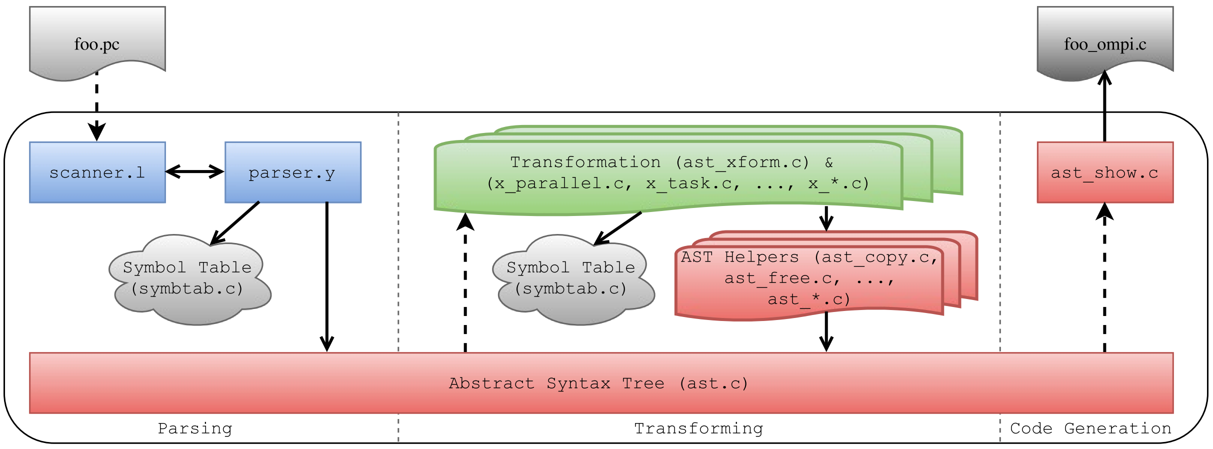
[1]Using -v option on ompicc we can get verbose information about each of the tools that are called during compilation.

[2]We can instruct ompicc to keep the intermediate files for debugging purposes by using -k or -K arguments.

instead a secondary hash table is used to pair each identifier with a number. That number is used in place of the identifier as key in the symbol table and everywhere in the AST. Along the hash table of the symbol table there is also a linked list where each new symbol is added. When a new scope is opened, a special symbol is inserted in the list to mark the beginning of the scope. When a scope ends we traverse the list backwards and remove all the entries from both the list and the hash table until we meet the special symbol, thus removing all the symbols that were declared in that scope.

After the parsing has finished it's time for the transformations to take place. Transformations manipulate the state of the AST by adding, removing and editing existing nodes of the tree. This occurs through a depth-first traversal of the AST. During the traversal any identifiers met in the nodes are added to the symbol table (and removed as soon as they get out of scope).OpenMP nodes are handled in a post-order fashion. This means that before transforming any OpenMP construct, the body of the construct has already been transformed and any nested directives have already been transformed. Transformation complexity ranges from very simple, for example a `barrier` construct (Section 2.1.3) is replaced by a call to the runtime function `ort_barrier()`, to extremely complex like for example the `parallel` construct which requires outlining (refer to Chapter 4).

Finally, after all the transformations end, the code generation phase starts, where the AST is traversed again using depth-first and a mix of pre-order, in-order and post-order visits, and export the nodes on a new file called [name of input file]_ompi.c.



foo.pc
foo_ompi.c
scanner.l
parser.y
Transformation (ast_xform.c) & (x_parallel.c, x_task.c, ..., x_*.c)
ast_show.c
Symbol Table (symbtab.c)
Symbol Table (symbtab.c)
AST Helpers (ast_copy.c, ast_free.c, ..., ast_*.c)
Abstract Syntax Tree (ast.c)
Parsing
Transforming
Code Generation

Figure 3.2: Compiler stages

Figure 3.2 lists the three stages of the compiler and the files involved in them. During the parsing, the scanner (scanner.l) and the parser (parser.y) work together in order to build the AST. They also use the symbol table (symtab.c) to identify any errors.

In the transformation stage the code in ast_xform.c traverses the AST looking for OpenMP directives, and transforms them using various helper files (named ast_*.c) such as ast_copy.c. Complex transformations, such as those involved in the tranformation of `parallel` (x_parallel.c) or `task` (x_task.c) constructs, are contained in their own modules (named x_[directive].c).

Finally the code in ast_show.c traverses the AST and converts its nodes back to C code, exporting it into a new file.

3.3 The Runtime System

The compiler and the runtime share a common API. The compiler replaces OpenMP directives with calls to the runtime, which is responsible for executing the user's program in parallel, by following the user's directions.

OMPi RunTime (ORT) consists of several libraries. The difference between them lies in the way OpenMP threads are utilized. a major design choice for the compiler was to not get tied to a particular threading library. Instead, it produces code that refers to abstract *execution entities* (EEs) which correspond to OpenMP threads. The runtime (ORT) implements the EEs by a multitude of interchangeable thread or process libraries, called EELibs (execution entity libraries).

Each of the EELibs is responsible for providing execution entities for ORT. The main EELib (ee_pthreads) is using POSIX threads as execution entities, but there are other EELibs such as ee_psthr which uses PSTHREADS [18] to utilize user-level threads as execution entities, and ee_process which implements each execution entity on a separate system process.

In addition to functions for execution entity creation, EELibs provide lock functions, which are used for protecting critical areas and for synchronization.

The rest of the runtime is shared between EELibs. For each OpenMP directive, one or more functions are provided by ORT, which leverage the abstract execution entities when necessary. Those are resolved into the actual execution entities during the linking of the library components.

Chapter 4

A Flexible Infrastructure for Outlining OpenMP Code

In order to implement some of the complex constructs of OpenMP OMPi uses outlining [19] (other OpenMP compilers also use this technique [20, 21, 22, 23, 24, 25]). Outlining is the process of extracting a block of code (in our case the constructs's *structured block*) into a new function and replacing the original block with necessary code in order to be able to call the new function e.g. Program 4.1

Simply extracting a block of code would not suffice. Usually the code references variables that where declared in a previous scope, so the new function would throw syntax errors for undefined variables. To make the new function syntactically correct, we have to declare a new variable for each corresponding variable that is being used in the outlined block and was declared in a previous scope.

Furthermore for the program to be semantically correct, the new variables must be initialized with the values the original variables had right before the outlined block, and any changes to them must be reflected back to the original variables. We accomplish this by creating a new struct, we will call it *data-struct* from now

```
{
    int i;
    for (i = 0; i < 10; i++) {
        ...
    }
}
```

(a) Original code

$\longrightarrow$

```
{
    newfunc1();
}
```

(b) Replacement code

```
void *newfunc1() {
    int i;
    for (i = 0; i < 10; i++) {
        ...
    }
}
```

(c) New function

Program 4.1: Outlining example.

on, and adding struct members for any variables that need some kind of initialization/synchronization (an example is given in Program 4.4). The type of each struct member (pointer or not) and the way it is initialized depends on the corresponding variable and any data-sharing attribute clause it appears in. Section 4.3 explains how each data-sharing attribute clause is treated.

When the OMPi project started only the `parallel` construct needed to outline its associated structured block. Version 3.0 of OpenMP introduced the `task` construct, whose implementation was also based on outlining. The initial implementation of the `task` construct in OMPi started as a direct copy of the transformation code used by `parallel`, therefore duplicating the outline code which was part of the transformation. Because of how strongly connected the code that did the outlining was with the tranformation of the `parallel` construct (and later with the `task` construct), implementing new constructs which required outlining of their structured block was not an easy task. That's why part of this thesis was to untangle and unify the code related to outlining into a new independent function.

4.1 The Outline Procedure

Before the outlining procedure begins, we call a function named `outline_OpenMP()` which handles OpenMP related code (construct/clauses). The function

- Parses the code, discovering all the used variables and categorizing them according to any OpenMP clauses they may appear, more information in Section 4.2.2.

- Marks any copyin variables,[1] treating them as “by ref” in `outline()`.
- Checks if a `default` clause is present in the directive and sets the appropriate default attribute handler (Section 4.3).
- Moves all implicit variables (variables that didn’t appear in any data-sharing clause) to the appropriate sets by using the default attribute handler.
- Removes the directive from the code.
- Finally, calls the `outline()` function, passing the used variables and the code block without the directive as parameters.

The behavior of the `outline()` function and the attribute handlers is controlled by several options. For example there are options that determine the name of the function, the name of the data-struct that will used for passing variable values, and any extra parameters that the new function may require. More information on the interface can be found in Section 4.4

The `outline()` function starts by calling all the attribute handlers (Section 4.3). Then it combines the struct members returned by the attribute handlers to create the data-struct. The data-struct along with variable declarations and initializations, and the body of the construct are combined together to form the new function. Finally data-struct initializations and a function call are placed where the original code used to be.

After the `outline()` function ends, control is restored to the `outline_OpenMP()` function. To ease debugging we insert a comment containing the OpenMP directive that was removed previously both before the replaced code and in the new function. Then, the new function is injected in the code, after the enclosing function, while a function prototype is injected before the enclosing function. Finally, if the directive was nested in a `target` directive, a copy of the function is stored in a list which is later used to generate the kernel file (for information on `target` directive refer to Section 5.2, and for information on its implementation in OMPi check Chapter 6).

4.2 Discovering Used Variables

As mentioned earlier in order for the outlined function to be syntactically correct, we have to discover all the variables used in the outlined structured block, which where

[1]copyin specific code lies in the `parallel` transformation code

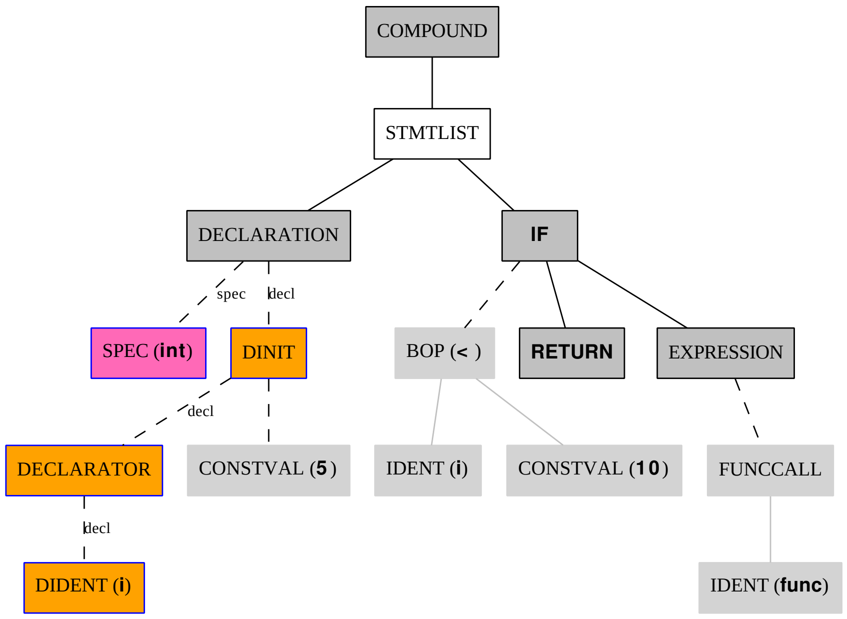
defined outside the outlined code, and define them in the generated function. Before we discuss how the code that finds those variables works, we should describe briefly the internal representation of the code in the OMPi compiler.

4.2.1 OMPi Internal Representation

The internal representation of OMPi is based on an abstract syntax tree (AST) [17]. Since OMPi is a source-to-source compiler with both its input and output consisting of C source code, it makes sense that an AST is not only used during the parsing of the input code, but also for the intermediate representation of the code. The leaves of the tree are C tokens as produced by the lexical analyser, and the nodes consist of both C/OpenMP tokens and parser rules. A small example can be seen in Figure 4.2.

```
1  {
2      int i = 5;
3      if (i < 10)
4          return;
5      else
6          func();
7  }
```

(a) C source



(b) AST

Program 4.2: Internal representation.

Transformations of code are applied directly on the nodes of the AST. For example in order to enclose in “static void ofunc1(int i) { ... }” the code from lines 2-6 of Program 4.2a we need to use the code in Program 4.3. Provided that we somehow set `body` in line 2 to point to the declaration of variable `i` from 4.2a, variable `function` will contain the code of a function named `ofunc1`, with one parameter (`i`), that calls function `func` if `i` is greater than or equal to 10. The code in line 3 traverses the tree upwards finding the parent of the node containing the declaration, and then selects the second child (the sibling of the declaration node).

```
aststmt function, body;
body = <find declaration node of i>;
body = body->parent->u.next;
function = FuncDef(
             Speclist_right(StClassSpec(SPEC_static), Declspec(SPEC_void)),
             Declarator(
               NULL,
               FuncDecl(
                 IdentifierDecl("ofunc1"),
                 ParamDecl(
                   Declspec(SPEC_int),
                   Declarator(
                     NULL,
                     IdentifierDecl(Symbol("i"))
                   )
                 )
               )
             ),
             NULL, body
           );
```

Program 4.3: Enclosing block into a new function.

Working with OMPi's IR may be hard for an uninitiated programmer, but the fact that it consists of one-to-one mapping between the nodes of the tree and C code makes transforming the code fast. Furthermore printing the AST is a simple depth-first traversal with each node printing its corresponding code accordingly.

4.2.2 Variable Analysis

Finding used variables in a block of code requires traversing the AST subtree that contains the code. The code for that is located in ast_vars.c. While traversing the tree if we encounter data-sharing clauses, we tag the involved variables in the symbol table. When we encounter a variable outside the directive we first need to check the scope it was declared in. If the variable was declared in a scope earlier than the scope the directive is in, then we need to store it somewhere so that we can apply the necessary transformations to the code later (Section 4.3), otherwise we can safely ignore it.

Used variables are stored in sets depending on the clauses they appeared in. For each attribute handler (Section 4.3) there is a corresponding set. There is also a set containing variables which didn't appear in any clause, and therefore are handled according to the rules of the directive.

When creating the outline procedure ast_vars.c was modified to be more generic. The original code used for analyzing run a hardcoded function whenever a C identifier was found. It was modified to run a user-supplied function (using a function pointer). By changing the function that is executed on identifiers we can manipulate them in any way we want, for example we can turn a set of variables into pointers.

Later on, it was further modified to use a new traversing code that applies user defined functions on all the nodes of the tree. That moved the actual traversing code from the file, and several other files that needed to traverse the AST, into a common file. Before this change, when a new type of node was added in the AST, it was necessary to change the code used for traversing the tree in several files which included their own traversing code.

4.3 Data-Sharing Attribute Handlers

Construct clauses dictate how variables should be treated in order for the code to produce the expected output. Simply creating a new variable for each used variable of the original code is not enough. To make the outlined code functional we have to be able to initialize variables at the beginning of the function and pass their values back to original variables.

Each used variable is inserted in a set, depending on the clause it appears in (Sec. 4.2.2). Each of these sets is processed by a different handler. If a variable does not appear in any clause, it has already been placed in the correct set, using the default handler associated with the construct that is being transformed, in `outline_OpenMP()` before `outline()` is even called. Therefore by the time the handlers are called all variables are in the correct sets.

Attribute handlers consist mainly of three parts. The first part handles the creation of the struct members that will be combined to form the data-struct, the second part handles code associated with the variables that is placed were the original code was (usually code that initializes data-struct members from the original variables), and finally the third part handles the code that is injected in the outlined function (usually declaring and initializing the new variables using the values passed with the help of the data-struct).

The handlers were divided like this because some of them share functionality and this allows us to use the same code on multiple handlers. For example, the part that

```
int i, j;
```

(a) Original variables

$\longrightarrow$

```
1 struct __shvt__ {
2     int i;
3     int (* j);
4 } _shvars;
```

(b) Generated data-struct

Program 4.4: Generating data-struct for passing values.

initializes a data-struct member using a variable's address is used by both the `by reference` handler (4.3.2) and the `reduction` handler (4.3.6). The part of data-struct member creation is handled by a single function that is shared by all handlers, with the only difference being that each handler chooses whether the member should be a pointer (Program 4.4b line 3) or not (Program 4.4b line 2) through a boolean parameter.

The following subsections describe how each handler works, and what each part entails. An example of a handler is shown in Program 4.5, for more examples the reader is referred to the Appendix.

4.3.1 Private Handler

Private variables are the simplest to handle. We only need to create a new local variable in the outlined function with the same type and size (in the case of arrays) as the original one. No member is inserted in the data-struct and no extra code is inserted in the place the original code was.

4.3.2 By Refence Handler

By reference variables are the variables which are shared between different threads and are implemented by adding a reference (a pointer) to the original variable in the data-struct (Program 4.5b line 4 and 4.5c line 4). The struct member is initialized with the address of the original variable (Program 4.5b line 6). A pointer is created in the new function, instead of simply a variable, which points to the original variable by copying the address which was placed in the data-struct (Program 4.5c line 9). All uses of the variable in the outlined function need to be modified to take into account that the variable is now a pointer (Program 4.5c line 12). We call this process pointerization.

By accessing the variables using a pointer all the threads manipulate the same memory, thus sharing changes between them. Also the original variables are auto-

```c
int a = 0;
#pragma omp parallel shared(a)
{
    ...
    a++;
    ...
}
```

(a) Original code

```c
int a = 0;
{
    struct __shvt__ { /* The struct */
        int (* a);
    } _shvars;
    _shvars.a = &a; /* Initialize data-struct members */
    /* Call the new function through the runtime */
    ort_execute_parallel(_thrFunc0_, (void *) &_shvars, ...);
}
```

(b) Replacement code

```c
static void * _thrFunc0_(void * __arg)
{
    struct __shvt__ { /* The data-struct */
        int (* a);
    };
    /* Cast the argument struct */
    struct __shvt__ * _shvars = (struct __shvt__ *) __arg;
    /* Create a new variable and initialize it using the data-struct */
    int (* a) = _shvars->a;
    { /* Below is the original code */
        ...
        (*a)++; /* The variable has been pointerized */
        ...
    }
}
```

(c) New function

Program 4.5: By reference transformation

matically updated whenever a thread changes them. This can cause race conditions, but in OpenMP it's the user's responsibility to prevent situations like that.

4.3.3 By Value Handler

The by value handler works on variables that are private to each thread, but need to be initialized with the value of the original variable. This includes `firstprivate` variables from `parallel` and `task` directives (check Section 2.3.1) and **to** variables from `target` directive (Section 5.2).

This handler behaves differently depending on the type of the construct. During `parallel` transformation it is treated as "**by name**". A pointer to the original variable is placed in the data-struct, which is then used to initialize a local variable in the new function. In this case the replacement code is the same as the one in by reference variables (Section 4.3.2). This approach has the benefit of accessing a local variable in the new function, and not using a pointer as in the case of `task` constructs.

Because execution of a `task` construct may be deferred until later, since tasks are not executed immediately like the `parallel` construct, the original variable could be modified by the time the task starts execution. Therefore we can't use a pointer in the data-struct, it has to contain a copy of the original variable. Also in the case of the `target` construct, using a pointer may not be possible. For these reasons both `task` `firstprivate` and `target` **to** variables are treated "**by copy**". A copy of the original variable is placed in the data-struct, which is used to initialize a variable in the new function. When the variable is non-scalar, in order to avoid wasting memory we use a pointer to the data allocated inside the data-struct. This has the downside of accessing the data through a pointer, as opposed to "**by name**" variables discussed earlier.

4.3.4 By Result Handler

Variables appearing in the `map` clause with map-type **from**, of a `target` construct need to return their value from the new function back to the original variable (information on the `map` clause are available in Section 5.3). The by-result handler creates a space for the new variable in the data-struct. If the variable is non-scalar then a pointer is used, pointing to the space allocated in the data-struct to avoid using twice as much memory. If it is scalar we simply create a local variable and copy its value at the end of the new function. Finally after the execution of the new function has ended, we copy the value from the data-struct back to the original variable.

4.3.5 By Value-Result Handler

By value-result handler is a combination of the "by value" an the "by result" handlers. It is used for the **tofrom** map-type of the `map` clause of `target` construct (Section 5.2), where we need to both initialize the variable in the new function, and return it's value back to the original variable after it finishes execution.

4.3.6 Reduction Handler

The reduction handler is more complicated. The replacement and the struct code is the same as in the "by reference" handler. But instead of using a pointer on the new function, we create a local variable. In the end of the function we inject code that updates the original variable using the address that is stored in the data-struct. The code has to be protected by locks to prevent race conditions, and depends on the type of the `reduction` clause (explained in Section 2.3.1). For example if the type of the reduction is "add" we simply add the value of the variable to the original variable. Besides that, the local variable has to be initialized according to the type of the reduction, e.g to 0 for addition, 1 for multiplication e.t.c.

4.4 Outline Interface

The `outline()` function has three parameters:

```
outcome_t outline(aststmt *b, outpars_t oo, set(vars) *usedvars);
```

The first one, *b*, is an AST statement node containing the code that will be outlined. During OpenMP outlining, it is usually a `compound` type node (code contained inside curly brackets {}), but could be anything from a single expression (e.g. *i++;*) to a selection (*if*, *switch*) or an iteration (*while*, *for*) statement. The statement should contain no OpenMP type nodes since the OpenMP nodes we are currently transforming have already be removed by `outline_OpenMP()` function (Section 4.1), and any OpenMP nodes contained inside the body of the construct have been transformed (as mentioned in Section 3.2).

The second parameter, *oo*, is a struct containing all the options that control the behavior of the `outline()` function, and the attribute handlers. Some of the options include the name the new outlined function will have, a function that will be used to call the outlined code e.g. in the case of `parallel` construct *ort_execute_parallel()* is used, and if "by-value" handler will act as a "by name" or "by-copy" handler.

Finally the third one, *usedvars*, is an array of sets (lists). Each set contains a different type of used variables, e.g. there is a set with all the "by reference" variables and another one for the "by name" variables, and is handled by a different attribute handler (Section 4.3).

The return type of the `outline()` function is also a struct. This struct contains pointers to various statements produced during the transformation. For example there are a pointers to the definition of the struct in both the new function and the replaced code, a pointer to the return statement of the new function, a pointer to the body of the new function, as well as pointers to the whole new function code and replacement code. These pointers can be used later in the code of the individual construct tranformation to add, modify or remove part of the generated code.

4.5 Outlining Parallel and Task Regions

The new outlining infrastructure was used to re-implement the `parallel` and `task` construct transformations. With the refined infrastructure implementing new OpenMP constructs is simple (in Table 4.1 we can see that both `parallel` and `target` transformation code were around a thousand lines, while they are only around two hundred line now), we just need to prepare the *outpars* struct of the outline with the proper options and call `outline_OpenMP()` (Section 4.1). For example the parameters used by `parallel` are:

```
static outpars_t oo =
{
    "",                      //+functionName
    "ort_execute_parallel", // functionCall
    NULL,                    //+extraParameters
    true,                    // byvalue_type (by name)
    false,                   //+global_byref_in_struct
    "__shvt__",              // structName
    "_shvars",               // structVariable
    NULL,                    // structInitializer
    xp_implicitDefault,      // implicitDefault function
    NULL                     // deviceexpr
};
```

These stay the same for each subsequent outline call during `parallel` outlining with the exception of three values which are filled in.

- *`functionName`*, is the name of the outlined function. In `parallel` it is always *`_thrFuncX_`*, where X is a unique number that is increased with each transformation.

- *`extraParameters`*, is filled with the parameters used to call the runtime function that will handle the execution of the outlined function (in this case the runtime function is called `ort_execute_parallel`). These parameters depend on the clauses of the directive.

- *`global_byref_in_struct`*, is a boolean which controls how how global by reference (shared) variables are accessed by the outlined code. When the construct that is being outlined is nested inside a `target` construct (Chapter 5), pointers for those global shared variables are inserted into the data-struct, otherwise the global shared variables are ignored during outlining and the outlined function access them directly. The `target` construct is executed on devices other than the CPU, and in most cases those devices don't have direct access to the main RAM.

File	Before	Additions	Deletions	After
x_parallel.c	1015	+119	-925	209
x_target.c	926	+121	-847	200
outline.c	0	+794	-0	794
Total	1941	+1034	-1772	1203

Table 4.1: Line changes during outlining refactoring according to Git.

The parameters used for `task` are similar in nature, with different names for the functions and the struct, and with *`byvalue_type`* being false. That is the option that controls how by value (Section 4.3.3) variables are handled.

After analysing the clauses in order to complete the struct, the transformation code calls:

```
op = outline_OpenMP(t, oo);
```

Here *`t`* is the statement that contains the construct we want to outline. After the function returns, *`op`* will hold pointers to various places in the transformed code, which can be used to add code specific to the current directive being transformed. For example `copyin` clause (Section 2.3.1) which is unique to `parallel` directive is not handled in `parallel`'s transformation after the completion of outlining with the help of *`op`*. Also during `task`'s transformation, depending on the presence of `if`, `untied` and `mergeable` clauses, the code may also get *inlined* by utilizing *`op`*.

Chapter 5

OpenMP 4 and Device Constructs

The OpenMP specification is actively being developed. Version 4.0 [3] of the API came out on July 2013, and version 5.0 is already in the works. Version 4.0 introduced many new and exciting features to the API. In section 5.1 we will take a brief look on what is new in Version 4.0, before presenting the device constructs in detail on the following sections. An example of how the device constructs work is given in Section 5.6.

5.1 New Features

Before focusing on the new device constructs here is a list of the notable additions in OpenMP v4.0:

- SIMD constructs were added to support SIMD (Single Instruction Multiple Data) parallelism.
- The `taskgroup` construct was added to support more flexible deep task synchronization.

- The `cancel` construct, the `cancellation point` construct and the `omp_get_cance-llation()` runtime routine were added to support the concept of cancellation.

- Device constructs and routines were added to support execution on devices. You can find more information for those in the sections that follow.

- The `proc_bind` clause and the `omp_get_proc_bind` runtime routine were added to the `parallel` construct to support thread affinity policies.

- Task dependencies were added through the new `depend` clause of the `task` construct.

- The `reduction` clause was extended and the `declare reduction construct` was added to support user defined reductions.

- The `atomic` construct was extended to support atomic swap with the `capture` clause, to allow new atomic update and capture forms, and to support sequentially consistent atomic operations with a new `seq_cst` clause.

5.2 Target

Among the new functionality that v4.0 of the OpenMP specification brought is the ability to execute code on devices other than the CPU. This was made possible by a set of new constructs such as the `target` construct. By using simple directives we can instruct the compiler to run portions of our code on a co-processor such as the Epiphany [26], an accelerator such as the Intel® Xeon Phi™ [27], a GPU (Graphics Processing Unit) [28] or even a DSP (Digital Signal Processor) [29].

The primary device construct is *`target`*, which instructs the compiler to offload the structured block associated with the directive on a device and execute it. When the target device is not available, the offloaded code is executed by the host device (the CPU). The syntax of the `target` directive is as follows:

```
#pragma omp target [clause[[,] clause]...] new-line
    structured-block
```

The `target` construct creates a device data environment just like `target data` (Section 5.3) and executes the construct on the target device. The encountering task waits for the device to complete the target region.

The supported clauses are:

- **device(***scalar-integer-expression***)**

  An integer indicating the id of the device where the construct will execute. If the is no `device` clause the construct is executed in the default device.

- **map(***[map-type :] list***)**

  A list of variables from the current task's data environment that will mapped to the device data environment associated with the construct. For each item in the list a corresponding new list item is created in the device data environment associated with the construct.

  The original and corresponding list items may share storage such that writes to either item by one task followed by a read or write of the other item by another task without intervening synchronization can result in data races.

  If a corresponding list item of the original list item is in the enclosing context device data environment, the new device data environment uses the corresponding list item from the enclosing device data environment. No additional storage is allocated in the new device data environment and neither initialization nor assignment is performed, regardless of the *map-type* that is specified. In simple words, if a variable has appeared in an enclosing `target data` directive that was targeting the same device it must be ignored the second time. This poses a challenge to the compiler, as the id of the targeted device is not known during compilation. Our approach to this is presented in Section 7.2

  Map-type can be any of the following:

  - **init**

    A new variable is created in the device data environment with undefined initial value.

  - **to**

    The corresponding variable is initialized with the value of the original variable upon entry to the region.

  - **from**

    On exit from the region the corresponding variable's value is assigned to each original variable. The initial value of the corresponding variable is undefined.

  - **tofrom**

    On entry to the region each new corresponding list item is initialized with the original list item's value and on exit from the region the corresponding list item's value is assigned to each original list item.

  If a *map-type* is not specified, the *map-type* defaults to **tofrom**.

- **if(***scalar-logical-expression***)**

  When an `if` clause is present and the `if` clause expression evaluates to *false*, the target region is executed by the host device.

Any variables referenced in a `target` construct that is not declared in the construct is implicitly treated as if it had appeared in a `map` clause with a *map-type* of **tofrom**.

5.3 Target Data

When a `target data` construct is encountered, a new device data environment is created, on the targeted device, containing any variables that appear in a `map` clause.

The new device data environment is constructed from the enclosing context device data environment, the data environment of the encountering task and any data-mapping clauses on the construct. The encountering task continues with the execution of the target data region.

The supported clauses are the same as `target` construct:

- **device(***scalar-integer-expression***)**

  An integer indicating the id of the device where the new device data environment will be located. If the is no `device` clause the new device data environment is created on the default device.

- **map(***[map-type :] list***)**

  Refer to Section 5.2

- **if(***scalar-logical-expression***)**

  When an `if` clause is present and the `if` clause expression evaluates to *false*, the new device data environment is created on the host device.

5.4 Declare Target

The `declare target` directive specifies that variables and functions are mapped to a device. The `declare target` directive is a declarative directive; it doesn't execute any code, it only marks the code for future reference. It appears in the global scope and its syntax is:

```
#pragma omp declare target new-line
    declarations-definition-seq
#pragma omp end declare target new-line
```

For each variable that is declared between the `declare target` and `end declare target`, a corresponding variable is created in the initial device data environment for all devices. If the declaration has an initialization then the corresponding variables get initialized to the same value, otherwise their value is undefined. The life span of the corresponding variables is the duration of the program. Any changes in corresponding variables during a `target` directive's execution persist among successive kernel executions. There is no implicit synchronization between the original variables and the corresponding ones. In order to update their values, the programmer needs to use the `target update` directive (Section 5.5).

In order to be able to call a function in a `target` directive's body, the function must have been declared in a `declare target` directive. Any functions called or global variables referenced in this function must have also been declared inside a `declare target` directive.

5.5 Target Update

The target update directive makes the corresponding list items in the device data environment consistent with their original list items, according to the specified motion clauses (**to**/**from**). It is used to synchronize variables that appear in a `target data` or a `declare target` directive. It may not appear inside a `target` directive. The supported clauses are:

- **device(***scalar-integer-expression***)**

  An integer denoting the id of the device where the corresponding variable resides. If the is no `device` clause the default device is used.

- **if(***scalar-logical-expression***)**

When an `if` clause is present and the `if` clause expression evaluates to *false* then no assignments occur.

- **to(***list***)**

  For each list item in a `to` clause the value of the original list item is assigned to the corresponding list item.

- **from(***list***)**

  For each list item in a `from` clause the value of the corresponding list item is assigned to the original list item.

5.6 Example of Device Directives

```c
int input = getInput(), implicit = ..., result;

#pragma omp target data map(to:input) map(from:result)
{
    int newvar = ...;

    #pragma omp target map(tofrom:input) /* The map of input is going to be ignored */
    {
        implicit += ...;
        result    = ... input;
    }

    #pragma omp target update from(result)
    printf("Intermediate result = %d", result);

    #pragma omp target map(alloc: newvar)
    {
        newvar  = ...;
        result += newvar;
    }
}
printf("Result = %d", result);
```

Program 5.1: Example of target directives.

Let's take a look on an example, to help us comprehend device directives easier. The example in question is shown in Program 5.1. Our example starts with a few variable declaration, two of which contain an initialization. In line 3 we reach our first `target data` directive. Looking at the clauses of the directive two variables are created on the default device (since there is no `device` clause). The first one, *`input`*, has a *map-type* of **to**. What this means is that the space allocated on the device will be initialized with the current value of the original variable. This is usually used for input variables which wont change during our kernel execution. We can still modify the value inside the offloaded kernel, but the value wont be returned to the original variable (unless we update it explicitly). The other one, *`input`*, has a **from** *map-type*, therefore the new variable on the device wont be initialized. At the end of the `target data` (after line 21), the value of the original variable will be updated to equal the value of the corresponding variable.

The body of the `target data` directive is executed by the host, whatever thread was executing before line 3 will continue to execute line 4 and onwards. That means that the new variable declared in line 5, is created on the host and not on a device.

Moving on to the first `target` directive, we notice that the input variable, which was mapped earlier as a **to**, appears again in a `map` clause. According to the rules of the map clause, this "remaping" will be ignored. Lines 8-11 will be executed on the targeted device, during which time the host thread will pause execution. In line 9 variable *`implicit`* is modified. Because it did not appear in any `map` clause it is treated as a **tofrom** variable, which means that the corresponding variable starts with the same value as the original variable, and any changes to it will be reflected on the original variable, after the termination of the kernel. Also note that the life time of the corresponding variable lasts only for the duration of the `target`, as soon as it ends there is no corresponding variable, and using it on a new `target` will result in a new corresponding variable being allocated. Line 10 modifies *`result`* which was already on the device from the `target data` in line 3.

After execution of the target region, the value of the original *`implicit`* variable is updated, and the host thread resumes execution. At this point, if we print *`result`* variable, we will (probably) see garbage data. Original and corresponding variables are not kept in sync, they are only modified when we enter/exit the region that creates the corresponding variable, therefore at line 12 original *`result`* variable still has not been initialized. If we want to print the value it currently holds on the device, we must first update the original variable with the value of the corresponding one. This is done with the `target update` on line 13. After that we can safely print the variable.

But if we wanted to print the *`result`* variable here, what was the point of adding it in `target data` of line 3, when we could just add it in `target` of line 7? The answer is simple, the variable is still on the device, and we don't need to reallocate space for it on the next `target` directive that also accesses and modifies it. The `target` in line 16 contains an **alloc** variable, *`newvar`*. The corresponding variable will have no initial value, and the original variable wont be modified after the end of the kernel execution. If the **alloc** was on a `target data` we would still be able to synchronize the original and the corresponding variables, but in this case there is no way to synchronize them, since `target update` may not appear inside a `target` and the corresponding variable ceases to exist after the `target` ends.

Finally if we printed *`result`* immediately after the end of the `target` directive of line 16 (after line 20 but before the end of the block of line 21), the result would be the same as in line 14. As stated earlier the original *`result`* variable is not updated until after line 21 where the structured block of the `target data` directive ends.

Chapter 6

Implementation of Device Constructs in OMPi

This chapter will analyse the implementation of the device constructs, which were discussed in Sections 5.2 through 5.5, in OMPi compiler. Only the transformations of the compiler will be explored here, for the changes to the runtime system the reader is referred to Chapter 7.

During the transformation of programs with device constructs, several files are generated, one for each `target` directive encountered plus one with the main code. Each of those files contains the outlined function of the corresponding `target` directive, and the declared functions and variables from `declare target` directives. The main contains, in addition to the host code, all the outlined functions from `target` directives, since the host may be called to execute any of them. The organization of the generated files is depicted in Fig. 6.1.

6.1 Target Data

We will use a simple example to demonstrate the transformation procedure. Consider the code in Program 6.1a which contains a target nested inside a target data. The

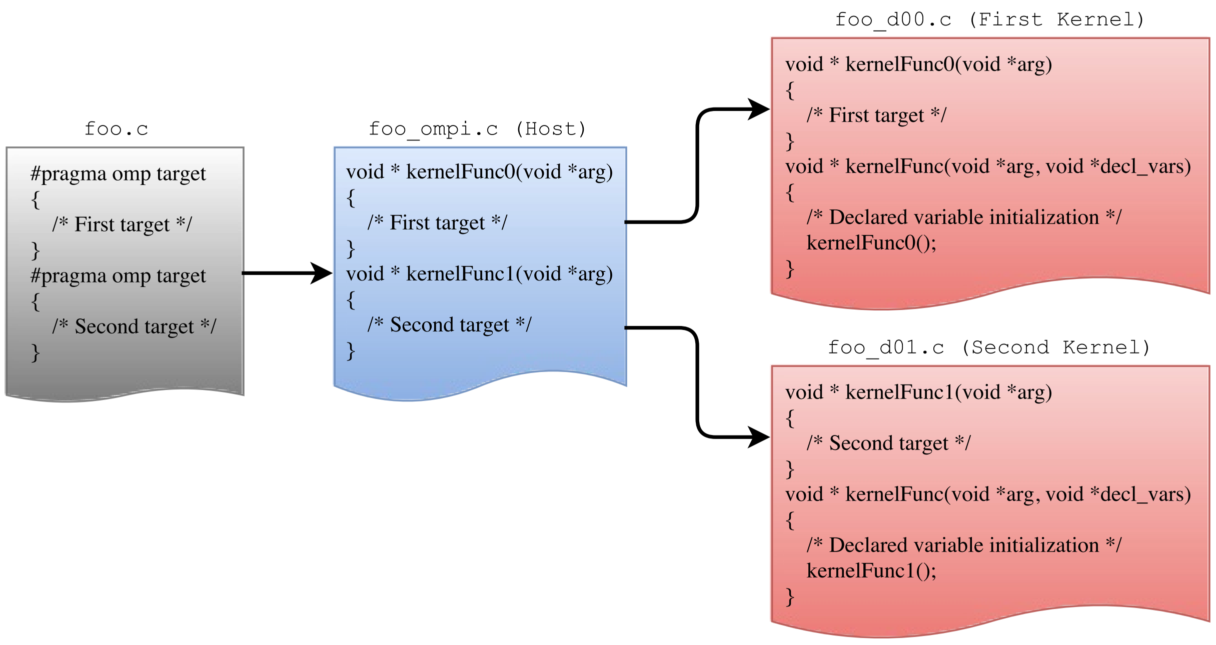

foo.c
#pragma omp target
{
/* First target */
}
#pragma omp target
{
/* Second target */
}
foo_ompi.c (Host)
void * kernelFunc0(void *arg)
{
/* First target */
}
void * kernelFunc1(void *arg)
{
/* Second target */
}
foo_d00.c (First Kernel)
void * kernelFunc0(void *arg)
{
/* First target */
}
void * kernelFunc(void *arg, void *decl_vars)
{
/* Declared variable initialization */
kernelFunc0();
}
foo_d01.c (Second Kernel)
void * kernelFunc1(void *arg)
{
/* Second target */
}
void * kernelFunc(void *arg, void *decl_vars)
{
/* Declared variable initialization */
kernelFunc1();
}

Figure 6.1: Files produced during compilation when code contains `target` directives

result of the transformation is given in Program 6.1b. When transforming the `target data` directive, a new device data environment is created by injecting a `start` and an `end` call, before and after the body of the directive (Program 6.1b, lines 3 and 36). If the directive is nested in another `target data` directive, the data environment of the latter is passed in the `start` call to let the runtime have access to the parent environment (Program 6.1b, line 10). For each variable, depending on the map type, calls are inserted at the beginning and end of the body.

- **alloc**

  An `_alloc_tdvar` call is inserted at the start of the body.

- **to**

  An `_init_tdvar` call is inserted at the start of the body.

- **from**

  An `_alloc_tdvar` call is inserted at the start of the body, and a `_finalize_tdvar` call is inserted at the end of the body (Program 6.1b, lines 12 and 28).

- **tofrom**

  An `_alloc_tdvar` call is inserted at the start of the body (Program 6.1b, lines 4–5), and a `_finalize_tdvar` call is inserted at the end of the body (Program 6.1b, lines 34–35).

```
#pragma omp target data map(x,y)
    #pragma omp target map(from: x) device(2)
        x=y=z=1;
```

(a) Original code

```
{                             // Start target data
    _devid = -1;              // Default device
    _ddenv = _start_ddenv(NULL, _devid, ..);
    _init_tdvar(&x, sizeof(x), _ddenv, _devid);
    _init_tdvar(&y, ..);

    {                         // Start target
        _devid = 2;           // Requested device
        _ddenv_prev = _ddenv;
        _ddenv = _start_ddenv(_ddenv_prev, _devid, ..);

        _alloc_tdvar(&x, ..); // Ignored if default device is 2
        _init_tdvar(&y, ..);  // Ditto

        struct __dd__ {
            int (* x);
            int (* y);
            int z;
        } * _devdata = _devdata_alloc(_devid, sizeof(struct __dd__));
        _devdata->x = get_vaddress(&x, ..); // Request address @device
        _devdata->y = get_vaddress(&y, ..);
        _devdata->z = z;      // Optimized

        ort_offload_kernel(_kernelFunc0_, _devdata, ..); // Kernel code

        z = _devdata->z;

        _finalize_tdvar(&x, _ddenv);
        _finalize_tdvar(&y, _ddenv);

        _end_ddenv(_ddenv);
    }                         // End target

    _finalize_tdvar(&x, _ddenv);
    _finalize_tdvar(&y, _ddenv);
    _end_ddenv(_ddenv);
}                             // End target data
```

(b) Transformed code

Program 6.1: Target/target data transformation.

According to the specifications, if a variable that appears in a `map` clause, already exists in the device data environment, no new space should be allocated and no assignments should occur. Since the `device` clause is an arbitrary expression and there is no restriction on the `device` clause of nested `target data` directives, the compiler is unaware of the devices that a variable has already been mapped on. For example, there could be a `target data` that adds variable a in device 1 and then another `target data` with the same variable that does not contain a `device` clause, and therefore the new device data environment will be allocated on the default device which, however, can be changed during execution. So during compilation there is no way to know if the two `target data` directives are targeting the same device or not, and whether we should allocate new space or use the existing one during the second directive. Some of the other compilers which added support for the device directives did not take into account that the code may contain nested device data environments, thus producing wrong results. We were the first ones to identify the issue and present a general solution [30].

Our solution to this subtle problem is to delegate most of the required functionality to the runtime. In particular, we inject calls to the runtime for creating, initializing and finalizing environments with variables that appear in `map` clauses, regardless of whether they have appeared or not in an enclosing `target data` directive, and let the runtime handle it, as described in Section 7.2. In Program 6.1b, this occurs for variables `x` and `y` in lines 12–13 and 28–29.

The code inside the construct is left untouched during the transformation of the `target data` directive, and will be executed by the host thread when it reaches the region. That is not visible in our example, because the `target data` contains only a `target` which is also transformed (in fact the transformation of the `target` construct precedes that of the `target data`).

6.2 Target

Each `target` directive is outlined similarly to `parallel` and `task` directives, albeit with different handlers for the variables. We create a new data environment, as if the construct was a `target data` one (lines 8–13 in Program 6.1b). As stated in Section 6.1, the compiler can't tell what device a `target data` or a `target` directive is addressing. That means that any variable referenced in the body of the directive may already be in the device, if it was part of a previous `target data` directive. For that reason all variables that have already appeared in any enclosing data environment are "inserted" in the new one (we make the call to the runtime let it decide if they should be inserted or not, more information on the runtime's mechanism are given in Section 7.2). Pointers to these variables are then placed in the *devdata*-struct, which will be passed to the outlined function. The pointers are initialized using runtime calls to get the address of the variables on the device space (lines 20–21 of Program 6.1b).

As an optimization, if a variable does not appear in any enclosing `target data` directive, space for the variable is created directly within the *devdata*-struct, instead of using a pointer (Program 6.1b line 22 and 26). In this case the handler used for dealing with the variable depends on the map-type:

- **alloc** is treated as a private variable.
- **to** uses "**by copy**" handler.
- **from** uses "**by result**" handler.
- **tofrom** uses "**by value-result**" handler (in the example, variable *z* is treated as **tofrom**, since that is the default way to treat variables that didn't appear in a `map` clause).

When outlining a `target` region, we store a copy of the outlined function along with any other outlined functions that may occur during the transformation of its body (e.g. when having a `parallel` region inside the `target`), in a global list. After the main transformation phase of the code, we use the information stored in that list to produce *kernel* files, one for each `target` construct.

In the example in Program 6.1, the kernel body (`x=y=z=1;`) has been moved to an outlined function `_kernelFunc0_()`; the actual execution of the kernel occurs in line 24, where the runtime call `ort_offload_kernel(...)` is given the function name and the *devdata*-struct.

6.3 Declare Target

During the transformation of an OpenMP program, whenever the compiler encounters a `declare target` directive, it stores any contained functions, function prototypes and variables in appropriate lists, and also marks them as *declared* in the symbol table. The body of the directive is left as is during the main transformation phase of the code. While transforming `target` directives, any variables that were declared within a `declare target` body are left intact.

The transformation for the `declare target` constructs occurs after the normal transformation phase ends, by which point all the declared variables have already been marked. The code surrounded by the `declare target` and `end declare target` directives is left unaltered in the main file (the file containing the code that the host

```c
#pragma omp declare target
int x, y = 42;
#pragma omp end declare target

void somefunc() {
    #pragma omp target ...
    {
        x = y;
    }
}
```

(a) Original code

```c
static void __register_global_decl(); // Register function prototype
void (* __register_global_decl_p)(void) = __register_global_decl; // Pointer to register func

void somefunc() {
    { /* #pragma omp target  */
        int _devID = -1;              // Default device
        ...
        struct __decl_struct {        // The struct used for declared variables
            int (* x);
            int (* y);
        } * _decl_data;
        if (__register_global_decl_p) // This function pointer is nullified after the first run
            ort_call_decl_reg_func(&__register_global_decl_p);
        _decl_data = ort_devdata_alloc(sizeof(struct __decl_struct), _devID);
        _decl_data->x = (int (*)) ort_get_declvar(&x, _devID); // Request address @device
        _decl_data->y = (int (*)) ort_get_declvar(&y, _devID);
        ort_offload_kernel(_kernelFunc0_, _dev_data, _decl_data, ...);
    }
}

static void __register_global_decl()  // The registration function
{
    static const int init_y = 42;     // Local static variable holding the initial value
    ort_register_declvar(&x, sizeof(x), (void *) 0); //Register var without initialization
    ort_register_declvar(&y, sizeof(y), &init_y);    //Register var with initialization
}
```

(b) Host code

Program 6.2: Declare target: host code

executes). The changes done to the variables and functions are only exported into the kernel files. The runtime though has to know about the declared variables in order to allocate memory for them in the devices. This is done in the host code.

Host Code.

A program may consist of more than one source file. If there was only one source file, registering the declared variables would be easy, we could simply add their registration in *main()*, which is the first function that is called. Each of the source files can contain `declare target` and `target` directives, and the runtime needs to know about the declared variables in each of them. Unfortunately `C` doesn't have constructors like `C++`, or anything else that runs at the startup of a program, that could be used for registrations/initializations. The GCC compiler has a mechanism for doing something like that but it is not standard `C` and using it in OMPi would limit OMPi's portability.

As a workaround a new static function is created in every source file for registering the declared variables during runtime (Program 6.2b lines 21–26). To avoid initializing devices and allocating memory if they are not used, we delay the actual memory allocations until the first time each device is used. This creates a problem with variable declarations that include initializations, because the corresponding variables must also be initialized with the same value, and by the time the memory allocations take place the original variables may not contain the initial values anymore. To deal with that, for each initialized variable, a separate static variable is created inside the new function, using the same initializer, and passing its address to the runtime (Program 6.2b line 23).

The above function is called whenever a `target` or `target update` directive which uses one of the declared variables is encountered (Program 6.2b line 13). This guarantees that the runtime has registered all these variables before they are actually used. In addition, precautions are taken so that the operation is completed only once and is not subject to concurrent invocations (Program 6.2b lines 2 and 12–13). Finally, the declared variables used in a `target` region, are placed in a separate struct and are given to the runtime at offload time (Program 6.2b lines 14–17).

Kernel Code.

As mentioned previously, for each `target` directive we produce a separate kernel file. The kernel file starts with the declared function prototypes (Program 6.3b line 1), followed by the variables declared inside `declare target` constructs, which are converted into pointers (lines 2–3 of Program 6.3b). Before exporting declared functions and the outlined function of the `target` directive, we transform them, replacing any occurrences of the declared variables by pointers. During this transformation, we also check to make sure that all the variables and function calls used, have been declared inside a `declare target` constructs.

Moreover, a wrapper function is created, which is the first function called by the runtime library of the device (Program 6.3b lines 18–29). The wrapper serves two purposes; first it initializes the pointers of the declared variables from the struct passed in the offload function (lines 20–26), and second it acts as a single point of entry for the runtime, calling the actual outlined kernel function (line 28), whose name depends on the number of `target` directives present in the original source

```c
#pragma omp declare target
int x, y = 65535;
void declared_function() {
    x = y;
}
#pragma omp end declare target

void somefunc() {
    #pragma omp target
        declared_function();
}
```

(a) Original code

```c
 1  void declared_function(); // Declared function prototype
 2
 3  int (* x) ;               // Declared variable
 4  int (* y) ;               // Declared variable
 5
 6  void declared_function()  // Declared function
 7  {
 8      (*x) = (*y);
 9  }
10
11  static void * _kernelFunc2_(void * __arg) // The outlined function
12  {
13      { /* #pragma omp target  -- body moved below */
14          declared_function();
15      }
16  }
17
18  void _kernelFunc_(void *__dev_data, void *__decl_data) // Wrapper function
19  {
20      struct __decl_struct  {
21          int (* x) ;
22          int (* y) ;
23      } * _decl_data;
24      _decl_data = (struct __decl_struct  *) __decl_data;
25      x = devrt_get_dev_address(_decl_data->x, sizeof(*(_decl_data->x)));
26      y = devrt_get_dev_address(_decl_data->y, sizeof(*(_decl_data->y)));
27
28      _kernelFunc2_(__dev_data);
29  }
```

(b) Kernel code

Program 6.3: Declare target: kernel code

code. This is the first function executed on a device, by the device's runtime.

6.4 Target Update

Transformation of `target update` directives are pretty straightforward. Each directive is replaced by one or more runtime calls. Depending on the type of the motion clause, either an `ort_read_tdvar()` or an `ort_write_tdvar()` function call is inserted for each variable that appears in the motion clauses. For example:

```
#pragma omp target update from(x, y) to(z)
```

Will result in:

```
ort_read_tdvar(x);
ort_read_tdvar(y);
ort_write_tdvar(z);
```

Chapter 7

Runtime Support for Devices

In order to support the device constructs, new libraries were added in OMPi Run-Time (ORT) called *device modules*, one for each device type. These are loaded dynamically during program execution. A module can contain more than one device, and is divided into two parts, one for execution on the host (*host part*) and the other for execution on the actual device (*dev part*). The latter is usually packaged and offloaded with the kernels.

Additionally the old runtime, now called host runtime, was enriched with three new files. The first one, target.c implements the API used by the compiler for manipulating devices and is presented in Section 7.1. The second, dataenv.c, contains a mechanism for handling device data environments. More information is given in Section 7.2. Finally modules.c handles dynamic loading of modules and is analysed in Section 7.3.

7.1 Additions to ORT API

Target.c consists of the of functions used for the implementation of device directives. Calls for any of those functions may be injected by the compiler in the code of the host. These functions do not actually interact with the devices, they merely check the targeted device id, and then call the appropriate function from the targeted module. The API consists of the following functions:

- **`ort_devdata_alloc/free()`** used for allocating/deallocating the structs used for passing variable values for both *mapped* and *declared* variables. They are injected during the transformation of `target`.
- **`ort_start/end_target_data()`** used by `target data` and `target` to create and remove device data environments.
- **`ort_alloc/init/finalize_tdvar()`** add and "remove" variables on device data environments. These interact with dataenv.c, to make sure the variables do not already exist on the targeted device, and are used by both `target data` and `target` transformations.
- **`ort_read/write_tdvar()`** used by `target update` to move data from or to device data environments.
- **`ort_get_vaddress()`** which returns the address of a corresponding variable on a device data environment, or a handler which can be used on the device to retrieve the address.
- **`ort_offload_kernel()`** which starts execution of kernels.

Function `ort_alloc_tdvar()` first checks if the variable is on the data environment of the targeted device. If it is, it does nothing, otherwise it creates a corresponding variable on the data environment. Function `ort_init_tdvar()` is a combination of `ort_alloc_tdvar()` and `ort_write_tdvar()`.

Finally `ort_finalize_tdvar()` checks if the variable was added in the current device data environment, not in an enclosing (one created by a previous `target data`) environment, and calls `ort_read_tdvar()`. This assures that "no assignments are prerformed" according to the specification, in the case of a variable appearing in a `map` clause while it already has a corresponding variable in the enclosing device data environment. The deallocation of the memory allocated by `ort_alloc_tdvar()` happens during the call of `ort_end_target_data()`.

7.2 Data Environment Handling

The host device needs an indexing mechanism in order to store and retrieve information regarding the variables that constitute the data environments. This information

```
1   int a, b;
2
3   #pragma omp target data map(a)          // DE1
4       #pragma omp parallel num_threads(2)
5       {
6           int c;
7           #pragma omp target data map(b)   // DE2$^{(1)}$, DE2$^{(2)}$
8               #pragma omp target map(a, c) // DE3$^{(1)}$, DE3$^{(2)}$
9                   ....
10      }
```

Program 7.1: Nested device data environments created by a team of threads.

is used when the host accesses these variables for reading or writing, for example during the mapping of a variable or before/after the execution of a kernel, when the `target update` directive is used. This information is also used during the initialization phase of a kernel. Because of the arbitrary nesting of data environments, and the possibility of multiple devices, the indexing cannot be statically handled at compile time.

To allow the host to have fast access to information regarding the variables involved in a data environment, we employ a novel mechanism based on typical separately-chained hash tables (HT). It works approximately as a functional-style compiler symbol table [17], albeit operated by the runtime. A sequence of nested data environments in the source program produces a dynamic sequence of HTs with entries for the mapped variables. The information stored on each entry includes the id of the device which the mapping refers to, and a pointer to the actual storage space of the variable. The hash function is given as input the original variable address combined with the targeted device number and returns the corresponding bucket. Collisions are handled through separate chaining.

To make comprehension easier, we will present the mechanism through two illustrative examples; detailed analysis follows right after. In Program 7.1 we show a code snippet where in line 3 a data environment (DE1) is created with the mapping of variable a. Then a team of two threads is created on the host device and each thread defines a separate data environment (DE$2^{(1)}$ and DE$2^{(2)}$) which includes the variables a and b. Finally, each thread offloads a code block, while at the same time creates a new data environment (DE$3^{(1)}$ and DE$3^{(2)}$) which includes the variables a and c. All the mappings in the example refer to the default device.

The sequence of the HTs for the above example is given in Figure 7.1. The result

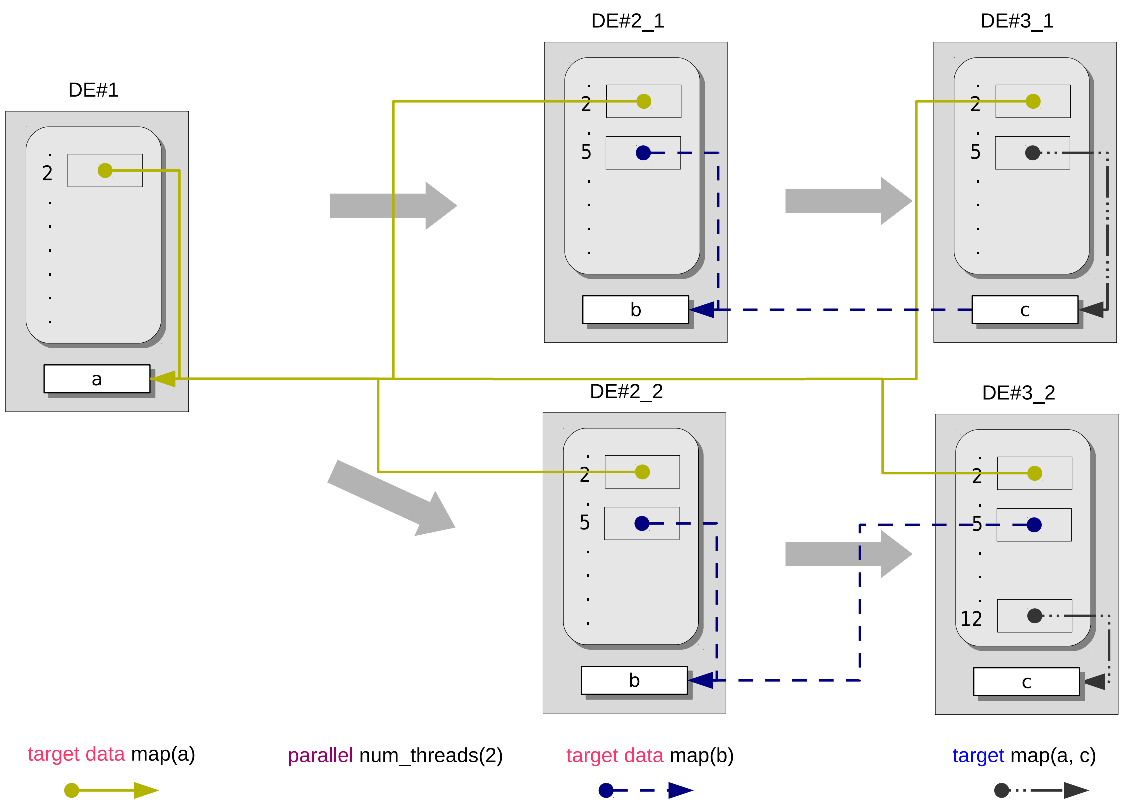


Figure 7.1: Hash table sequence for code in Program 7.1. *Solid yellow, dashed blue and mixed black arrows denote allocations and definitions made at lines 3, 7 and 8, respectively.*

of line 3 is the creation of DE1 which stores only one entry, that for variable a. The creation of the thread team does not affect the indexing mechanism, and both threads can access the data of DE1 without the need of locks. Line 7 defines DE$2^{(1)}$ and DE$2^{(2)}$ (one for each thread); this causes the creation of new HTs, both of which are created as copies of the previous HT. As a result, each thread retains access to the enclosing data environment. The mapping of variable b adds a new entry for DE$2^{(1)}$ and DE$2^{(2)}$ and the addition of a pointer from the HT to this entry. Similarly, in line 8 we have the creation of new HTs that handle DE$3^{(1)}$ and DE$3^{(2)}$, respectively, as copies of the previous two HTs. In the case of the second thread, variable c is hashed onto bucket 12, which is a free bucket. In contrast, variable c of the first thread caused a collision (hashed onto bucket 5), resulting in a chain between the variables c and b.

An additional complication arises by the possible presence of multiple devices, where a data environment may be nested within another environment that refers to a different device, as can be seen in the code of Program 7.2. In this example, in line 3, a data environment is defined for device 1 (DE4), with variable d. Then, DE5 is created in line 4 for device 2 that includes variables d and e. Finally, DE6 is created

```
int d, e, f;

#pragma omp target data map(d) device(1)        // DE4
    #pragma omp target data map(d, e) device(2) // DE5
        #pragma omp target map(f) device(1)     // DE6
        {
            e = d++; // implicit mapping of variable e
            ...      // in device 1
        }
```

Program 7.2: Nested device data environments created for two different devices.

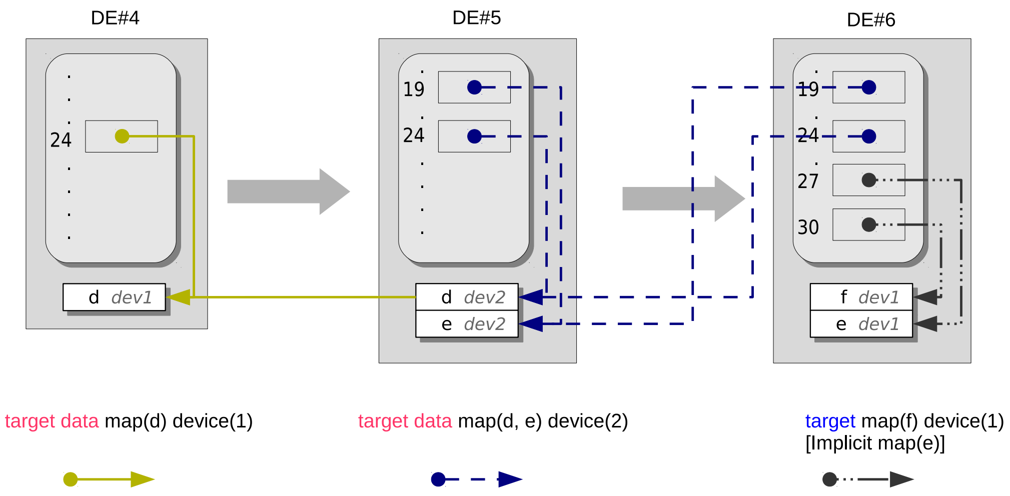


Figure 7.2: Hash table sequence for code in Program 7.2. *Solid yellow, dashed blue and mixed black arrows denote allocations and definitions made at lines* 3*,* 4 *and* 5*, respectively.*

in line 5 for device 1 and a block of code is offloaded for execution. Notice that d is already mapped through DE4 while there is an implicit mapping for variable e on device 1. The corresponding sequence of HTs is shown in Figure 7.2. In DE4 there is only one entry for d on device 1. The HT for DE5 starts with a copy of DE4; entries for variables d and e are then added in buckets 19 and 27, correspondingly. Notice that because d now refers to device 2, the hash function returns a different bucket than the one used for device 1. Finally, DE6 starts as a copy of DE5 and has two entries added for variables f and e. Since d is already present in the data environment of device 1, no further actions are required.

7.2.1 Details of the Mechanism.

The data handling mechanism is operated by the host, resides in the host address space, and is independent from any attached devices. The HTs allow for efficient variable insertion and look-up operations. Each time a nested data environment is created, a new HT is initialized as a copy of the HT used by the enclosing data environment, as in functional-style symbol tables; destruction of the data environment requires a single memory deallocation for the corresponding HT. If a team of threads is created within a `target data` region, separate HTs are created, when needed, for each thread; this completely eliminates mutual exclusion problems.

In addition, we employ a further optimization. When initializing a new data environment for a particular device, the compiler informs the runtime about the *maximum possible number of variables added* to the environment. This is calculated statically during the analysis of the program; an exact number is not possible to derive because the devices the environments refer to are given by full expressions and can only be calculated at runtime, in the general case. In practice, only variables not already mapped on this device are actually inserted in the HT. Because of this information, the runtime is able to acquire memory for the HT *and* the entries in a single allocation request (this explains why in Figures 7.1 and 7.2 the HTs and the corresponding entries lie within the same rectangle). This also has the desirable side-effect of increased data locality, as the hash table and the entry information reside on the same memory block.

The space requirements of the presented mechanism have an upper bound of $O(L \cdot K + n)$, where L stands for the maximum number of alive data environments, K is the size of a hash table (derived from the static compiler information discussed above) and n is the total number of variables that are mapped in all data environment definitions. In a program that defines a total of E data environments, exactly $\Theta(E)$ of memory allocations and deallocations are made, which include the memory required for the HT and the memory used for the entries. With a load factor at most equal to 1, the average time for an insertion or a lookup is $\Theta(1)$.

7.3 Device Modules

An abundance of computational units, such as GPUs, DSPs and general or special purpose accelerators can be found in today's computer systems, which can be used to aid the computations done by the CPU. Each of them has a different way of being invoked. Therefore each type of device requires a unique approach from the runtime.

To be able to support more than one type of device, a different library, called *device module*, is implemented for each one of them.

A module consists of two parts. The first one runs on the host device and is responsible for the communication of the host with the devices. It is loaded dynamically during program execution. The other one runs on the device, and is a full implementation of an OpenMP runtime, designed to operate within the device. Furthermore each module library may handle more than one device, provided that the devices are of the same type.

7.3.1 Host

The availability of each device may change between compilation and execution time, especially if the system where execution takes place is not the same as that of the compilation. Hardcoding calls to each module in the host runtime is thus precluded, as they will fail if the machine that executes the application has a different device configuration. Instead, the host part of each module is implemented as a dynamic library which is loaded during execution time. This is done though the use of POSIX API's *dl* calls (`dlopen()`, `dlsym()`, `dlclose()`).

During the initialization of ORT, which is done as soon as `main()` starts, the runtime searches for known modules in the current folder of execution, in the folder OMPi's libraries reside, and in the system's default dynamic library folder. If it finds a module, it uses `dlopen()` to open it and calls function `hm_get_num_devices()` to query the number of available devices supported by the module. This is done using "lazy" binding in order to minimize the initialization time, and `dlclose()` is called shortly afterwards.

ORT keeps track of the number of devices, and assigns a global id to each one based on the order of detection. For example if there are two devices available from the first module, then device ids 1 and 2 are assigned to that module. If the second module has three devices, those devices are assigned global ids 3, 4 and 5 and so on.

The first time a device from a module is used, we make a regular `dlopen()` call when opening the module, and use `dlsym()` to get get access to all the functions, which are stored in a struct in ORT. Then function `hm_initialize()` is called from the module, to initialize the device, along with the local id of the device in the module. For example, if we want to initialize device 4 from the previous example, we request for the second device in the module. The module is initialized only once, when the first device in it is initialized. If for any reason a device fails to initialize, that device's id is mapped to the host device.

The functions available in each module for communicating with the device are more or less the same as the ones in target.c (Section 7.1). There are functions to allocate and free memory, move data to and from the device, get the address of a variable as seen by the device and finally offload a kernel. The complete interface is as follows:

- `hm_get_num_devices()` which is used during initialization of the host runtime to determine the number of devices available by each module.
- `hm_print_information()`, a function used by the compiler frontend to display information on the available devices.
- `hm_initialize()` which is called by the host runtime to initialize a device, the first time its id apears in any device directive.
- `hm_finalize()` that frees any allocated resources when the program execution ends.
- `hm_offload()` which handles the offloading of a function to a device. This is called from `ort_offload_kernel()` (Section 7.1), and is the function that starts execution of a kernel file.
- `hm_dev_alloc()` which allocates space for use on the device. This space can be on a local memory of the device, or on memory shared by the host and the device, or even on the host memory if the device has direct access to it. The actual location of the allocated space depends on the module, and it is either the fastest or the largest memory available that both the host and the devices can access. This is called by `ort_devdata_alloc()` and `ort_alloc_tdvar()` (Section 7.1).
- `hm_dev_free()` that frees memory allocated by `hm_dev_alloc()`, and is called from `ort_devdata_free()` (Section 7.1). It is also called from the mechanism tha handles device data environments which is explained in Section 7.2 during the call of `ort_end_target_data()` (Section 7.1).
- `hm_fromdev()` and `hm_todev()` which are called by `ort_read_tdvar()` and `ort_write_tdvar()` respectively (Section 7.1). They are used to synchronize the values of an original and a corresping variable.
- `hm_get_dev_address()` which is called by `ort_get_vaddress()` and returns the address of a corresponding variable on the device, or a handler which can be used during the kernel execution to retrieve the actual address.

7.3.2 Device

The device part of a module, which is denoted as *devrt*, is a full implementation of an OpenMP runtime. This makes supporting new device types a highly involved task. The only missing OpenMP parts are the device constructs, which can only be used when running on the host, since targeting a device from within another device is not allowed.

The *devrt* is linked and offloaded with each kernel. As we mentioned earlier it's not possible to determine what device is being targeted during compilation. Therefore when compiling the program, each kernel code is compiled with each selected/available module's *devrt*. For example if we have a program with three `target` constructs and there are two available modules during compilation, 6 kernel executables will be created, one for each pair of construct-module, along with the executable of the host. Let's say that the first module uses *.cl* for its kernels suffix, and the second module uses *.srec* as a suffix. If the source file is named foo.c, the produced kernels will be: foo_d00.srec, foo_d00.cl, foo_d01.srec, foo_d01.cl, foo_d02.srec, foo_d02.cl.

7.4 Available Modules

At the time of writing there are two modules available in OMPi. The first one uses the Epiphany accelerator as a `target` device. The Epiphany is available as a co-processor on the Parallella board. The other module is based on OpenCL framework, and is still in the early stages of implementation. The is also a pseudo-module, which was created to aid us during testing. It does not use any real device, each "device" is a new process and utilizes *`mmap`* for shared memory between the host and the "devices". Although it's not useful as a module in its the current state, it could provide the basis of a module for devices that use new processes to execute kernels.

7.4.1 Epiphany Accelerator

The Parallella-16 board [31] is a popular 18-core credit card-sized computer and comes with standard peripheral ports such as USB, Ethernet, HDMI, GPIO, etc. The computational power of the board comes from its two processing modules. The main (host) processor is a dual-core ARM Cortex A9 with 512 KiB shared L2 cache, built within a Zynq 7010 or 7020 SoC. The other is an Epiphany 16-core chip which is used as a co-processor. The board has 1 GiB of DDR3 RAM, addressable by both the ARM CPU and the Epiphany. The former runs Linux OS and uses virtual addresses,

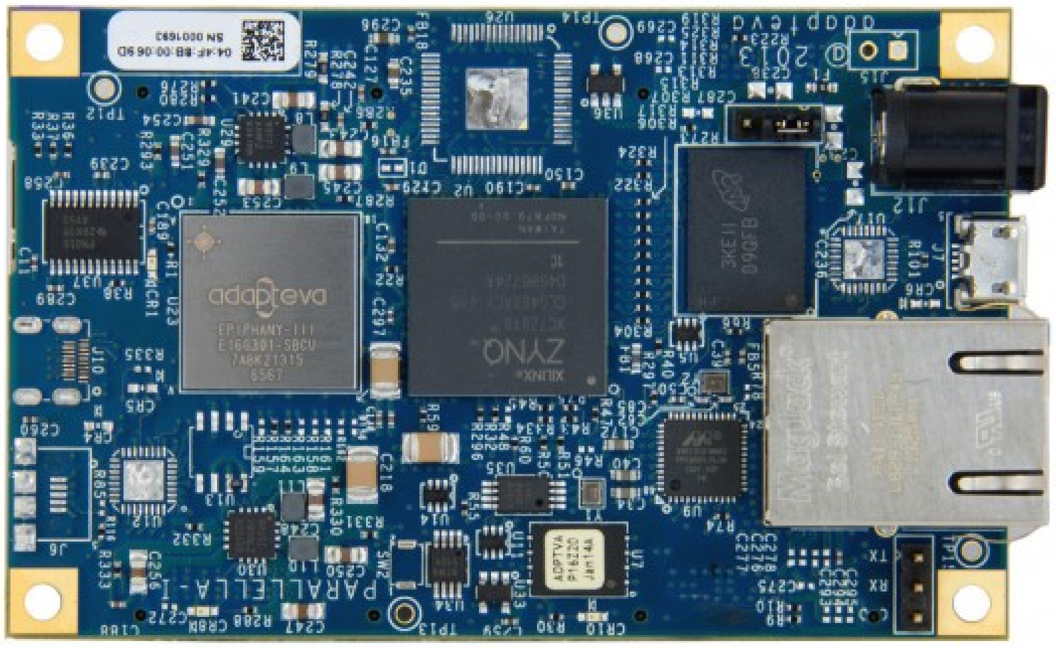


Figure 7.3: Parallela board

whereas the latter does not have an OS and uses a flat, unprotected memory map.

Two versions of the Epiphany co-processor are actually available: the Epiphany-16 (with 16 cores and a 4×4 mesh NoC) and the Epiphany-64 (with 64 cores and an 8×8 mesh). Although our discussion here holds for both versions, we refer mostly to the first one since it is the one widely available. Each Epiphany core (eCORE) is a 32-bit superscalar RISC CPU, clocked at 600 MHz, capable of performing single-precision floating point operations, and equipped with 32 KiB local scratchpad memory and two DMA engines. The ARM and the Epiphany use a 32 MiB portion of the system RAM as *shared memory* which is physically addressable by both of them. All common programming tools are available for the ARM host processor. For the Epiphany, a Software Development Kit (eSDK [32]) is available, which includes a C compiler and runtime libraries for both the host (eHAL) and the Epiphany (eLIB). Furthermore, OpenCL is provided by the COPRTHR SDK [33]. The latter also provides a threading API similar to POSIX.

Details on the runtime implementation of the Epiphany accelerator as an OpenMP 4.0 module, which is build upon the eSDK, are available in [26].

Host Module

The communication between the Zynq and the eCOREs occurs through the shared memory portion of the system RAM. The shared memory is logically divided in two sections: The first section is a fixed size of 4 KiB, and is used transparently by OMPi for kernel coordination and manipulation of parallel teams created within the Epiphany. The second part is used for storing the kernel data environments and part of the tasking infrastructure of the Epiphany OpenMP library.

In order to be able to control the eCOREs independently through eHAL calls, the initialization phase creates 16 workgroups, one for each of the available Epiphany cores and puts them to the idle state for energy and thermal efficiency. For offloading a kernel, the first idle core is chosen and the precompiled kernel object file is loaded to it for immediate execution. Due to the high overheads of the eSDK when offloading kernels to different workgroups, we developed an optimized low-level offload routine to assist the creation of OpenMP teams. We support multiple, independent kernels, executing concurrently within the Epiphany. Because the current version of eHAL does not provide a way for an eCORE to notify directly the host for kernel completion, a special region of the shared memory is designated for synchronization with the e COREs.

Devrt

Supporting OpenMP within the device side presents many challenges due to the lack of dynamic parallelism and the limited local memory of each eCORE. Regarding the former, when a kernel is offloaded to a specific eCORE, the core executes its sequential part until a `parallel` region is encountered. Because only the host can activate other eCOREs, the master core contacts the host, requesting the activation of a number of cores. A copy of the same kernel is then offloaded to the newly activated cores. During the parallel code execution all synchronization between the cores occurs through their fast local memories. When the region completes, the cores return to the idle, power saving state, while the master core informs the host thread about the termination of the parallel team.

The small scratchpad memory makes it impossible to fit sophisticated OpenMP runtime structures alongside the application data. To support the worksharing constructs of the OpenMP, the infrastructure originally designed for the host was trimmed down in order to minimize its memory footprint; this is linked and offloaded with each kernel. The coordination among the participating eCOREs utilizes structures stored in the local memory of the team master. The eSDK provides mechanisms for locks and barriers between the eCOREs but they assume that the synchronized cores belong to same workgroup. Since in our runtime each eCORE constitutes a different workgroup, we were forced to modify all these mechanisms. Finally, the tasking infrastructure is based on a blocking shared queue stored in the local memory of the master eCORE, while the corresponding task data environments are stored in the shared memory.

Experiments

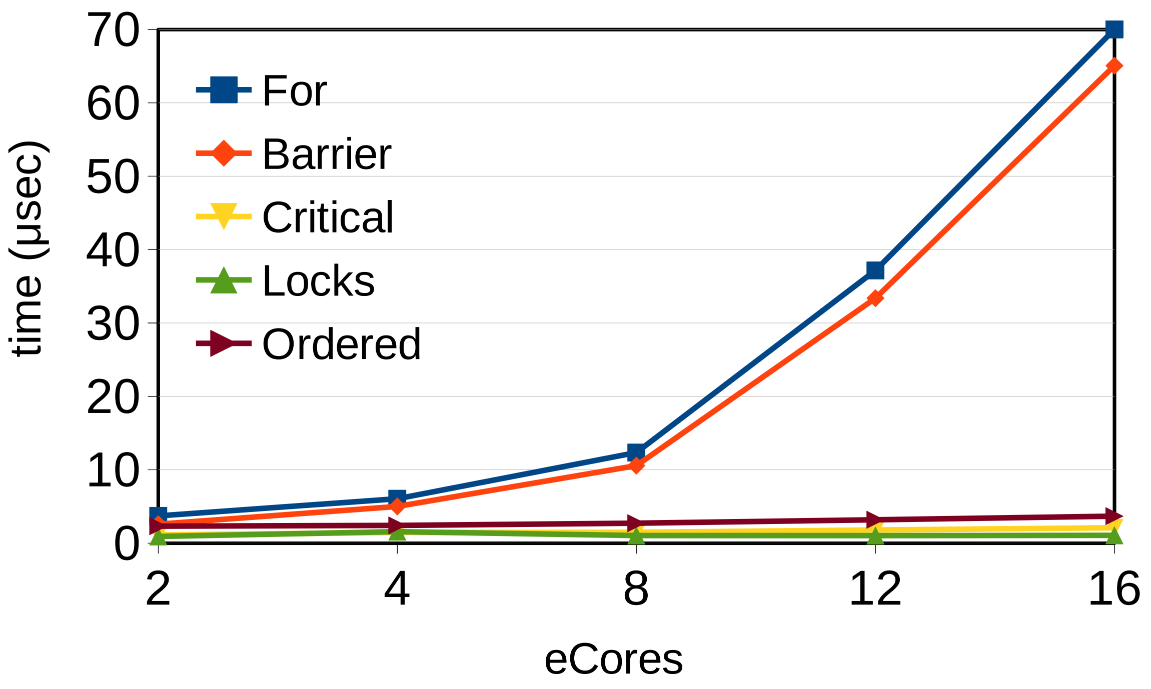


Figure 7.4: EPCC synchronization results

We have conducted a number of tests in order to measure the efficiency of our offloading mechanisms alongside the space and timing performance of the OpenMP runtime within the Epiphany accelerator. Our board is the Parallella-16 SKUA101020 and we use eSDK 5.13.9.10. To examine the memory overhead of our Epiphany-resident runtime, we created a set of simple OpenMP programs to compare with the size of the kernels produced when the native eLIB is used. In the case of an empty kernel, containing only a single assignment OMPi incurs a 4.5 KiB overhead as compared to the kernel created by the native eLIB. In other scenarios which involved a team of 16 cores and complex OpenMP functionality, up to 10 KiB more than a similar eLIB-based kernel were needed. We are currently optimizing the runtime memory footprint even more.

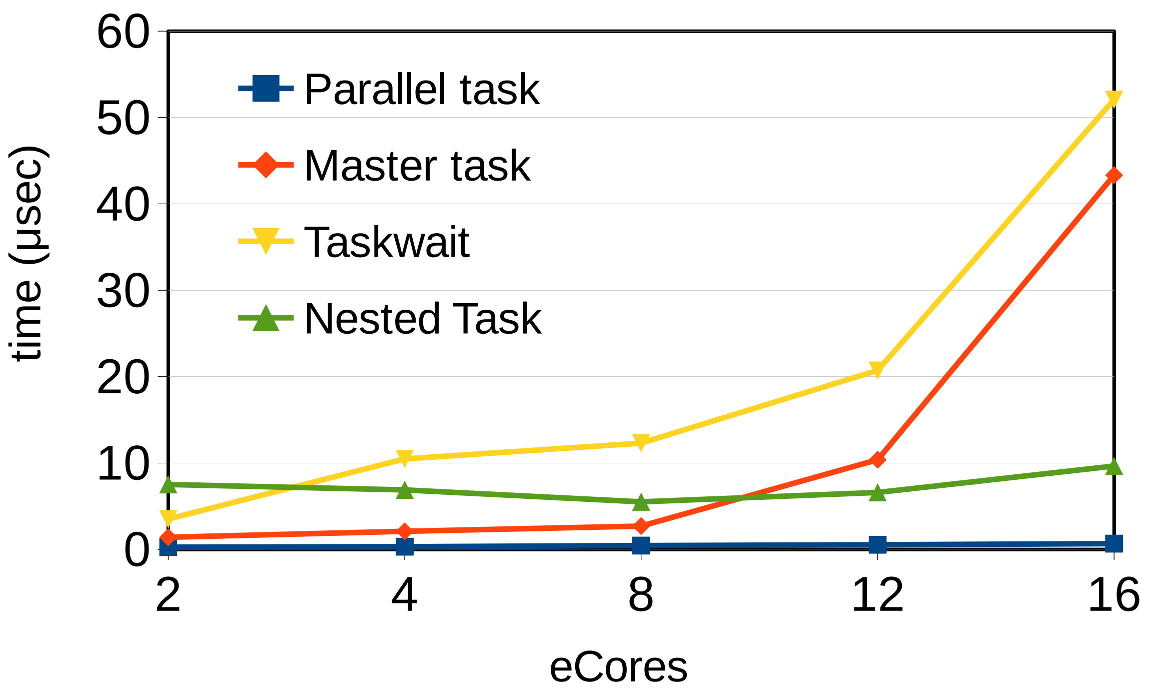


Figure 7.5: EPCC tasking results

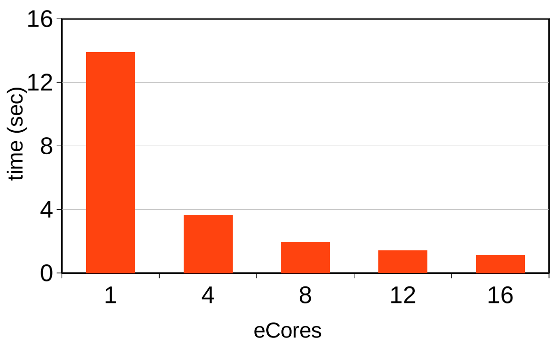


Figure 7.6: Pi computation

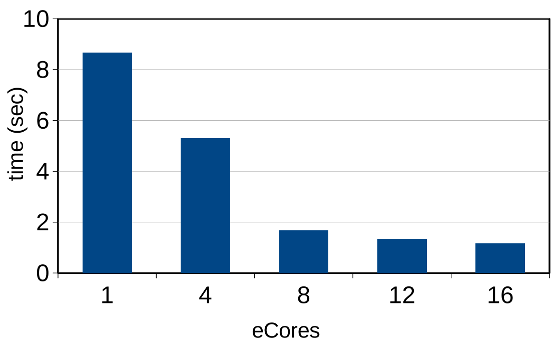


Figure 7.7: Nqueens(12)

In order to measure the OpenMP construct overheads within the Epiphany, we created a modified version of the EPCC microbenchmark suite [34] where their basic routines are offloaded through `target` directives. In Fig. 7.4 we plot a sample of the synchronization benchmark results. While most results are quite satisfactory, our initial prototype employs a non-optimized barrier, which also has a direct impact on the overheads of the `for` construct. We are currently optimizing its behavior. Sample results for the tasking benchmark are given in Fig. 7.5. The noticeable cases are those of the Taskwait and Master task tests. The contention on our simple lock-based shared task queue is quite high in these tests and shows up vividly in the case of 16 threads. Our implementation can be further improved, but at this point we strive mostly for functional correctness.

Finally, we include performance results for a few simple OpenMP applications. In Fig. 7.5 we plot timing results for a typical iterative computation of $\pi = 3.14159$, based on the trapezoid rule with 2,000,000 intervals, and using a kernel which spawns a parallel team of 1 to 16 threads. While the scalability is almost ideal, the serial execution is quite slow; the reason is that the Epiphany does not support double-precision numbers natively and the eCORE floating point unit does not implement division. In Fig. 7.7 we present the performance of a modified version of the Nqueens task benchmark, taken from the Barcelona OpenMP Tasks Suite (BOTS) [35]. This application computes all solutions of the n-queens placement problem on an $n \times n$ chessboard, so that none of the queens threatens any other. Due to the severe memory limitations of the Epiphany, we considered the manual cut-off version of the benchmark, where the nested production of tasks stops at a given depth. The figure shows the timing results for 12 queens, and a cut-off value of 2 (144 tasks produced). As it can be seen, with the addition of eCOREs we obtain an almost linear speedup.

7.4.2 OpenCl

Open Computing Language, more commonly known as OpenCL, is a framework for writing programs that can execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators and provides task-based and data-based parallelism.

When implementation of an OpenCL module begun, it targeted OpenCL 1.2 [36] devices. Unfortunately due to several restrictions on version 1.2 of the framework, supporting OpenMP on the device was close to impossible. The module got to the point of executing kernels on OpenCL enabled devices, but implementing an OpenMP runtime on them was not a viable option.

One of the problems encountered was the lack of global variables. This means that, disregarding the fact that some runtime functions may need global variables to function correctly, implementing `target data` would not be an easy or efficient task. One possible solution would be to have a struct holding all the "global" variables, and pass it around in every function call. Something like that though would require changing all the functions of the kernel code by the compiler.

Another issue were OpenCL's *address space qualifiers*. In OpenCL every variable definition has to be prefixed by an address space qualifier, such as *global* or *constant*, which determine where the variables are going to be allocated. The problem is that one cannot for example have a pointer that is in *local* address space point to a variable allocated in *global* address space. Depending on where each variable is allocated, there should be multiple versions of *declared* (through `declare target`) functions, each one with different address space qualifiers, depending on where the function gets invoked from.

Finally the lack of dynamic parallelism (i.e. creating teams of threads dynamically within the kernel) would have been an issue in implementing the `parallel` construct. This signifficant problem has been a subject of active research. The solutions proposed include: a) stopping the current running kernel when you meet a `parallel`, and b) starting a new kernel that would run on several computational units, or using a `switch case` construct to differentiate where the master thread runs, and where all threads should run [37]. However implementing any of those would require significant effort and they probably wont work in all cases. Furthermore it is questionable how efficient any of these proposed solutions would be.

For all the above reasons, we made a strategic decision to only address version 2.0 [38] of the OpenCL standard. OpenCL 2.0 adds support for program-scope (global) variables, introduces a generic address space, which removes the need for address space qualifiers on function parameters and adds nested parallelism. At the time only AMD had support for version 2.0 of the specification, for their GPUs. Recently Intel released a new SDK adding OpenCL 2.0 support Intel Core M processors and more implementations are sure to emerge.

Additionally it includes shared virtual memory, which means that a variable allocated in the system's RAM can be directly access by a kernel. Unfortunately our test system did not support *fine-grained* shared virtual memory at the time of writing, meaning that in order to use shared virtual memory the allocations must be done using a special function. When support for *fine-grained* shared virtual memory is implemented, it will be possible to skip memory allocations and data movements, by directly accessing the original variables from the device.

Currently, the module supports only plain kernels that don't contain any OpenMP constructs. The move to OpenCL 2.0 allowed full support of the `declare target` construct. The *devrt* of the module is still being designed, so unfortunately we don't have any data to present.

Chapter 8

Conclusions/Future Work

OpenMP is a widely used framework for shared memory programming. Its simple and incremental approach allows developers to easily parallelize their code. This has led to a wide adoption of OpenMP into a variety of commercial and research compilers. The most significant directive is `parallel` which spawns multiple threads to execute its body. OpenMP 3.0 expanded its applicability to not only loop-based parallelism but also task-based parallelism which encompasses irregular, recursive and pointer-based applications, by the introduction of the `task` construct, which creates work units that can be executed by any "free" thread.

OpenMP 4.0 brought a lot of exciting new constructs, and the device-related ones are probably the most promising. They allow programmers to create programs that can utilize the available hardware of a system, simply by surrounding the code they want to offload with the `target` directive. Along with that comes `target data` directive which allows allocating memory to the targeted hardware that persists between successive kernel executions, `declare target` that marks functions that can be used on the device and also can allocate variables with global scope and `target update` which allows updating the variables created by `target data` and `declare target`.

Unfortunately there are some drawbacks in the way device constructs are defined. Due to the fact that code executing on a device has access to nearly all OpenMP constructs, a full OpenMP runtime must be available in each device. Not all devices have the same power and capabilities, so making an efficient runtime for a device may be hard or even impossible as a result of the limited resources available on it. Also the lack of a way to run device specific code hampers the ability to take advantage of the true potential of the device. Finally there is no standard way to determine information about the available devices during runtime, except from learning the number of available devices.

8.1 Current Status of the OMPi Compiler

OMPi an open source OpenMP compiler which started back in 2001 and is being developed by Parallel Processing Group (PPG) of University of Ioannina, currently supports version 3.1 of the specification. It is available from PPG's website at `paragroup.cs.uoi.gr`. A special pre-compiled version is also available for the Parallella board. A new version will be released soon incorporating our changes, and also bring support for `task` dependencies and the `cancel` directive. In this thesis we:

- Redesigned outlining, to make it easy to use for new directive transformations. Outlining is a technique used to move a block of code into a new function, while the original code is replaced by a function call. Outlining is widely used by compilers to implement OpenMP directives, including OMPi. The new implementation is independent of the `parallel` and `task` transformations. We unified those two implementations and replaced them with an autonomous method, that can be used to outline any block of code.
- We used our new outlining infrastructure to implement the `target` construct. The body of the construct gets outlined into a new function which is placed both in the main code that gets executed by the host, and in the kernel code that is executed by the targeted device.
- We were the first to create a module for the Parallella Board and implement full OpenMP on the Epiphany accelerator.

8.2 Future Work

Although the Epiphany module is feature complete, there is still room for improvement. The runtime can be further optimized size-wise and performance-wise. The compiler could possibly help in this direction by careful analysis of the code and better code generation. It could also provide some insight to the runtime on what the kernel contains, allowing it to make better decisions, for example where to place the data.

Another improvement would be having a *resident kernel*, instead of creating a new one each for each `target`. A kernel would be offloaded onto the device, and stay there for the duration of the program, doing nothing in-between `target` executions. Because of the long overhead times we observed when offloading to the Epiphany device, this technique could theoretically increase performance. On the negative side more resources will be needed, since the code of all the kernel files would be loaded for the duration of the program.

Another line of future work is clearly the support of more devices. In particular, the completion of the OpenCL 2.0 module is of top priority. A possible candidate for a feature module is the Intel® Xeon Phi$^{\text{TM}}$ coprocessor. Xeon Phi is currently the only device supported by Intel's compiler and gcc, and a module targeting Xeon Phi would provide as with a common ground on which we can compare our implementation. Unfortunately we lack access to a machine with such a coprocessor, and even if we acquired a Xeon Phi, chances are that it would not work on the machines we have available because it requires special hardware to operate on (there have been a lot of reports with incompatibilities on non-Intel motherboards).

Bibliography

[1] H. Sutter and J. Larus, “Software and the concurrency revolution,” *Queue*, vol. 3, pp. 54–62, September 2005.

[2] R. V. Van Nieuwpoort, G. Wrzesińska, C. J. H. Jacobs, and H. E. Bal, “Satin: A high-level and efficient grid programming model,” *ACM Trans. Program. Lang. Syst.*, vol. 32, pp. 9:1–9:39, March 2010.

[3] OpenMP A.R.B., “OpenMP Application Program Interface Version 4.0,” July 2013.

[4] Adapteva, “The Parallella Board.” https://www.parallella.org/.

[5] OpenMP Architecture Review Board, “OpenMP Application Program Interface Version 3.1,” July 2011.

[6] S. N. Agathos, P. E. Hadjidoukas, and V. V. Dimakopoulos, “Design and Implementation of OpenMP Tasks in the OMPi Compiler.,” in *Proc. PCI ’11, 15th Panhellenic Conference on Informatics* (P. Angelidis and A. Michalas, eds.), (Kastoria, Greece), pp. 265–269, IEEE, September 2011.

[7] Parallel Processing Group, Dept. of Computer Science & Engineering, University of Ioannina. http://paragroup.cs.uoi.gr/wpsite/.

[8] OMPi webpage. http://paragroup.cs.uoi.gr/wpsite/software/ompi/.

[9] V. V. Dimakopoulos, E. Leontiadis, and G. Tzoumas, “A portable C compiler for OpenMP V.2.0,” in *roc. EWOMP 2003, 5th European Workshop on OpenMP*, (Aachen, Germany), pp. 5–11, September 2003.

[10] OpenMP Architecture Review Board, “OpenMP Application Program Interface Version 2.0,” March 2002.

[11] G. Philos, V. Dimakopoulos, and P. Hadjidoukas, “A Runtime Architecture for Ubiquitous Support of OpenMP,” in *Proc. ISPDC 2008, 7th Int’l Symposium on Parallel and Distributed Computing*, (Krakow, Poland), pp. 189–196, June 2008.

[12] OpenMP Architecture Review Board, “OpenMP Application Program Interface Version 2.5,” May 2005.

[13] OpenMP Architecture Review Board, “OpenMP Application Program Interface Version 3.0,” May 2008.

[14] P. E. Hadjidoukas, V. V. Dimakopoulos, and G. C. Philos, “Exploiting fine-grain thread parallelism on multicore architectures,” *Scientific Programming*, vol. 17, pp. 309–323, November 2009.

[15] P. E. Hadjidoukas and V. V. Dimakopoulos, “Nested Parallelism in the OMPi OpenMP/C Compiler,” in *Euro-Par 2007 Parallel Processing* (A.-M. Kermarrec, L. Bougé, and T. Priol, eds.), vol. 4641 of *Lecture Notes in Computer Science*, pp. 662–671, Springer Berlin Heidelberg, 2007.

[16] S. Agathos, N. Kallimanis, and V. Dimakopoulos, “Speeding Up OpenMP Tasking,” in *Euro-Par 2012 Parallel Processing* (C. Kaklamanis, T. Papatheodorou, and P. Spirakis, eds.), vol. 7484 of *Lecture Notes in Computer Science*, pp. 650–661, Springer Berlin Heidelberg, 2012.

[17] A. W. Appel, *Modern Compiler Implementation in C*. Cambridge: Cambridge University Press, 1999.

[18] PSThreads, a high-performance user-level threads library.
http://paragroup.cs.uoi.gr/wpsite/software/psthreads/.

[19] J.-H. Chow, L. E. Lyon, and V. Sarkar, “Automatic Parallelization for Symmetric Shared-memory Multiprocessors,” in *Proceedings of the 1996 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON ’96, pp. 76–79, IBM Press, 1996.

[20] M. Sato and S. Satoh and K. Kusano and Y. Tanaka, “Design of OpenMP Compiler for an SMP Cluster,” September 1999.

[21] Brunschen, Christian and Brorsson, Mats, “OdinMP/CCp - a portable implementation of OpenMP for C,” *Concurrency - Practice and Experience*, vol. 12, no. 12, pp. 1193–1203, 2000.

[22] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta, "Nanos Mercurium: a Research Compiler for OpenMP," in *European Workshop on OpenMP (EWOMP'04). Pp*, pp. 103–109, 2004.

[23] X. Teruel and P. Unnikrishnan and X. Martorell and E. Ayguade and R. Silvera and G. Zhang and E. Tiotto, "OpenMP tasks in IBM XL compilers," in *Proc. of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, 2008.

[24] Free Software Foundation, "GCC, the GNU compiler collection." http://www.gnu.org/software/gcc.

[25] LLVM Compiler Infrastructure. http://llvm.org/.

[26] S. Agathos, A. Papadogiannakis, and V. Dimakopoulos, "Targeting the Parallella," in *Euro-Par 2015: Parallel Processing* (J. L. Träff, S. Hunold, and F. Versaci, eds.), vol. 9233 of *Lecture Notes in Computer Science*, pp. 662–674, Springer Berlin Heidelberg, 2015.

[27] C. Newburn, R. Deodhar, S. Dmitriev, R. Murty, R. Narayanaswamy, J. Wiegert, F. Chinchilla, and R. McGuire, "Offload Compiler Runtime for the Intel® Xeon Phi^{TM}Coprocessor," in *Supercomputing, 28th International Supercomputing Conference, ISC 2013* (J. Kunkel, T. Ludwig, and H. Meuer, eds.), vol. 7905 of *Lecture Notes in Computer Science*, pp. 239–254, Leipzig, Germany: Springer Berlin Heidelberg, September 2013.

[28] Liao Chunhua, Yan Yonghong, Bronis R. de Supinski, Daniel J. Quinlan, Barbara M. Chapman, "Early Experiences with the OpenMP Accelerator Model.," in *OpenMP in the Era of Low Power Devices and Accelerators, 9th International Workshop on OpenMP, IWOMP 2013* (a. M. S. M. Rendell, Alistair P. Chapman M Barbara, ed.), vol. 8122 of *Lecture Notes in Computer Science*, (Canberra, ACT, Australia), pp. 84–98, Springer, September 2013.

[29] G. Mitra, E. Stotzer, A. Jayaraj, and A. Rendell, "Implementation and Optimization of the OpenMP Accelerator Model for the TI Keystone II Architecture," in *Using and Improving OpenMP for Devices, Tasks, and More, 10th International Workshop on OpenMP, IWOMP 2014* (L. DeRose, B. de Supinski, S. Olivier, B. Chapman, and M. Müller, eds.), vol. 8766 of *Lecture Notes in Computer Science*, pp. 202–214, Salvador, Brazil: Springer International Publishing, September 2014.

[30] A. Papadogiannakis, S. N. Agathos, and V. V. Dimakopoulos, *OpenMP: Heterogenous Execution and Data Movements: 11th International Workshop on OpenMP, IWOMP 2015, Aachen, Germany, October 1-2, 2015, Proceedings*, ch. OpenMP 4.0 Device Support in the OMPi Compiler, pp. 202–216. Cham: Springer International Publishing, 2015.

[31] Adapteva, "Parallella Reference Manual," September 2014.

[32] Adapteva, "Epiphany SDK reference Manual," September 2013.

[33] Brown Deer Technology, LLC, "COPRTHR® (CO-PRocessing THReads®) API Reference," 2014.

[34] J. M. Bull, "Measuring Synchronisation and Scheduling Overheads in OpenMP," in *Proc. of 1st EWOMP, European Workshop on OpenMP*, (Lund, Sweden), pp. 99–105, September 1999.

[35] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguadé, "Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP," in *38th Int'l Conference on Parallel Processing (ICPP '09)*, (Vienna, Austria), pp. 124–131, IEEE Computer Society, September 2009.

[36] Khronos OpenCL Working Group, "The **OpenCL** Specification Version: 1.2," November 2011.

[37] C. Bertolli, S. F. Antao, A. E. Eichenberger, K. O'Brien, Z. Sura, A. C. Jacob, T. Chen, and O. Sallenave, "Coordinating GPU Threads for OpenMP 4.0 in LLVM," in *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, LLVM-HPC '14, (Piscataway, NJ, USA), pp. 12–21, IEEE Press, November 2014.

[38] Khronos OpenCL Working Group, "The **OpenCL** Specification Version: 2.0," November 2013.

Appendix

Examples of data-sharing attribute handlers:

Private Handler

```
int a = 0;
#pragma omp parallel private(a)
{
    ...
    a++;
    ...
}
```

Original Code

```
int a = 0;
{
    /* Call the new function through the runtime */
    ort_execute_parallel(_thrFunc0_, (void *) 0, ...);
}
```

Replacement Code

```
static void * _thrFunc0_(void * __arg)
{
    /* Create a new variable */
    int a;
    { /* Below is the original code */
        ...
        a++; /* Note that since the variable wasn't initialized the result is undefined */
        ...
    }
}
```

New Function

By Value Handler - By Name

```
int a = 0;
#pragma omp parallel firstprivate(a)
{
    ...
    a++;
    ...
}
```

Original Code

```
int a = 0;
{
    struct __shvt__ { /* The struct */
        int (* a);
    } _shvars;
    _shvars.a = &a; /* Initialize struct members */

    /* Call the new function through the runtime */
    ort_execute_parallel(_thrFunc0_, (void *) &_shvars, ...);
}
```

Replacement Code

```
static void * _thrFunc0_(void * __arg)
{
    struct __shvt__ { /* The struct */
        int (* a);
    };
    /* Cast the argument struct */
    struct __shvt__ * _shvars = (struct __shvt__ *) __arg;

    /* Create a new variable and initialize it using the struct */
    int a = *(_shvars->a);

    { /* Below is the original code */
        ...
        a++;
        ...
    }
}
```

New Function

By Value Handler - By Copy

```c
int a = 0;
#pragma omp task firstprivate(a)
{
    ...
    a++;
    ...
}
```

Original Code

```c
int a = 0;
{
    struct __taskenv__ { /* The struct */
        int a;
    } * _tenv;
    /* Allocate memory for the struct */
    _tenv = (struct __taskenv__ *) ort_taskenv_alloc(sizeof(struct __taskenv__));

    _tenv->a = a; /* Initialize struct members */

    /* Call the new function through the runtime */
    ort_new_task(_taskFunc0_, (void *) _tenv, ...);
}
```

Replacement Code

```c
static void * _taskFunc0_(void * __arg)
{
    struct __taskenv__ { /* The struct */
        int a;
    };
    /* Cast the argument struct */
    struct __taskenv__ * _tenv = (struct __taskenv__ *) __arg;

    /* Create a new variable and initialize it using the struct */
    int a = _tenv->a;

    { /* Below is the original code */
        ...
        a++;
        ...
    }
}
```

New Function

By Result Handler

```c
int a;
#pragma omp target map(from: a)
{
    ...
    a = 1;
    ...
}
```

Original Code

```c
int a;
{
    struct __dev_struct {
        int a;
    } * _dev_data;
    /* Allocate memory for the struct */
    _dev_data = (struct __dev_struct *) ort_devdata_alloc(...);
    /* Call the new function through the runtime */
    ort_offload_kernel(_kernelFunc0_, (void *) _dev_data, ...);
    /* copy back the results */
    a = _dev_data->a;

    ort_devdata_free(_dev_data, __ompi_devID);
}
```

Replacement Code

```c
static void * _kernelFunc0_(void * __arg)
{
    struct __dev_struct { /* The struct */
        int a;
    };
    /* Cast the argument struct */
    struct __dev_struct * _dev_data = (struct __dev_struct *) __arg;

    int a; /* Create a new variable */

    { /* Below is the original code */
        a = 1;
    }
    /* copy back the results */
    _dev_data->a = a;
}
```

New Function

Reduction Handler

```c
int a = 0;
#pragma omp parallel reduction(+: a)
{
    ...
    a = 1;
    ...
}
```

Original Code

```c
int a = 0;
{
    struct __shvt__ { /* The struct */
        int (* a);
    } _shvars;
    _shvars.a = &a; /* Initialize struct members */
    /* Call the new function through the runtime */
    ort_execute_parallel(_thrFunc0_, (void *) &_shvars, ...);
}
```

Replacement Code

```c
static void * _thrFunc0_(void * __arg)
{
    struct __shvt__ { /* The struct */
        int (* a);
    };
    /* Cast the argument struct */
    struct __shvt__ * _shvars = (struct __shvt__ *) __arg;

    /* Create a new private variable and initialize it according to the reduction type */
    int a = 0;
    { /* Below is the original code */
        ...
        a = 1;
        ...
    }
    /* Surround the reduction code with a lock */
    ort_reduction_begin(&_paredlock0);
    /* Update the reduction variable */
    *(_shvars->a) += a;
    ort_reduction_end(&_paredlock0);
}
```

New Function

Index

Author's Publications

- S.N. Agathos, V.V. Dimakopoulos, A. Mourelis, A Papadogiannakis, "Deploying OpenMP on an Embedded Multicore Accelerator", in Proc. SAMOS XIII, 13th Int'l Conference on Embedded Computer Systems: Architectures, MOdeling, and Simulation, Samos, Greece, July 2013, pp. 180–187

- S.N. Agathos, A. Papadogiannakis, V.V. Dimakopoulos, "Targeting the Parallella", in Proc. Euro-Par 2015, 21st Int'l Conference on Parallel and Distributed Computing, Vienna, Austria, Aug. 2015, pp. 662–674

- A. Papadogiannakis, S.N. Agathos, V.V. Dimakopoulos, "OpenMP 4.0 Device Support in the OMPi Compiler", in Proc. IWOMP 2015, 11th International Workshop on OpenMP, Aachen, Germany, Oct. 2015, pp. 202–216

Short Vita

Alexandros Papadogiannakis was born in Chania, Greece in 1988. He was admitted at the Computer Science Department of the University of Ioannina in 2006. He received his B.Sc. degree in Computer Science in 2012 and is currently a postgraduate student at the Department of Computer Science & Engineering of the University of Ioannina. He is a member of the Parallel Processing Group since 2010. His research interests include Parallel Processing and Operating Systems.