Compiler-Assisted OpenMP Runtime Organization for Embedded Multicores

Spiros N. Agathos[*] *Vassilios V. Dimakopoulos*
Department of Computer Science and Engineering
University of Ioannina
P.O. Box 1186, Ioannina, Greece, GR-45110
E-mail: {sagathos,dimako}@cse.uoi.gr

Technical Report TR-2016-1
April 1, 2016

Abstract

The recently introduced OpenMP device constructs open a whole new world for application writers, enabling them to easily utilize the host CPUs along with other attached computational resources, in an intuitive and productive manner. At the same time, multicore architectures have conquered the whole computing spectrum. General-purpose and embedded system alike integrate multicore CPUs and multicore co-processors or accelerators. The new OpenMP `target`-related directives offload portions of the program code (kernels) to any of the available devices; the kernels themselves can take advantage of the multiplicity of processing elements within the target device by employing OpenMP constructs. However, most co-processors or accelerators, especially embedded ones, have limited resources. This severely constrains the extend of OpenMP support that can be implemented within a device. A usual design decision is to only support OpenMP partially, in effect hindering the full exploitation of the device capabilities through a high-level programming model. In this work, we present a novel solution to this problem. We propose a compiler-assisted, adaptive runtime system organization, which generates application-specific support by implementing only the OpenMP functionality required each time. In particular, based on extensive compiler analysis, the offloaded kernels can be accompanied by a runtime library tailored to the needs of the given application. Full OpenMP support is thus available, if needed. However, in the usual scenario where kernels do not require complex OpenMP functionalities, our method can lead to dramatically reduced executable sizes and/or execution times. To demonstrate the potential of our proposal we present an implementation on the popular Parallella board.

[*]S.N. Agathos is supported by the Greek State Scholarships Foundation (IKY)

1 Introduction

Personal computers utilize multiple general-purpose cores in a socket, usually combined with a multicore graphics processor (GPGPU). On the other hand, high performance supercomputers are heterogeneous systems and benefit from the combination of multicore CPUs with specialized devices such as GPGPUs, DSPs and FPGAs, in order to accelerate a broad range of applications and also gain power savings. As a result, modern system architectures present a mix of different processor and memory hierarchies within the same system. At the same time the building blocks of such heterogeneous computing nodes are designed for different workload scenarios; multicore CPUs perform best in coarse grained tasks, while accelerators reach their computational potential in large scale data and fine grained vector processing.

The embedded systems market has been formed to satisfy the demand for devices that are small, portable, autonomous and also relying on limited energy resources. As a response to the ever increasing demand of end-user applications for computational power and multitasking, embedded systems have also joined the heterogeneous multicore area. One design paradigm for heterogeneous embedded system is Parallella [4] which combines a small number of ARM-based cores assisted by a many-core group of (up to 64) RISC cores. Another example is the STM STHORM [7] architecture that utilizes multi-cluster computation fabrics, where each cluster contains up to a few tens of simple processing elements communicating via low-latency, high-throughput interconnection and shared L1 memory.

However, in order to exploit the computation capabilities of a heterogeneous system efficiently, significant programmer effort is required. The common case is to utilize a low-level SDK in order to optimize an application with respect to the specific hardware features. This fact poses significant challenges, even for expert programmers. In the same line, programming models such as OpenCL [2] and CUDA [22] provide very efficient albeit rather primitive mechanisms for an application to take advantage of the hardware capabilities of GPGPUs. In addition, requiring different code bases for the host CPU and the accelerator devices increases code complexity, decreases its portability and also complicates the compiler and runtime system (RTS).

The biggest challenge in this era of multicore computing proliferation is to provide a programming model that enables the extraction of satisfactory performance while also keeping programmer productivity at high levels. OpenMP has proven to be a productive solution for parallel programming on shared memory systems. It became quite popular mainly due to the fact that it is a directive-based model which does not change the base language (C/C++/Fortran), making it quite accessible to mainstream programmers. Recently, OpenMP 4 [28] has come to embrace platforms based on a heterogeneous collection of processors, co-processors and accelerators; it has been augmented with new directives which allow offloading portions of the application code onto the processing elements of an attached *device*. One important and desirable characteristic of OpenMP is that the application blends the host and the device code

portions in a unified and seamless way. Although support for the OpenMP 4.0 device model has been slow to be adopted by both compiler and device vendors, it is gaining momentum.

The new device extensions allow full OpenMP functionality within the regions of code executed by a selected device (also known as *kernels*). This fact provides flexibility and ease of use regarding parallelization expressiveness. However, it requires an OpenMP infrastructure within the co-processor. In the general case, implementing such an infrastructure is a non-trivial task. Supporting the required functionality, which was originally designed for shared memory multiprocessors, can be a very difficult procedure due to limited resources. As a result, common approaches are to either provide partial OpenMP support (i.e. handle a subset of the directives in the device side) or implement full but simplified OpenMP facilities so as to avoid consuming the limited amount of resources. For example, in devices such as embedded multicores of multicore systems-on-chip (MCSoC), the small amount of on-chip memory and hardware synchronizers must accommodate both the OpenMP runtime libraries and the application code/data. This holds even in cases where particular application kernels do not make use of all the provided OpenMP functionality.

In this paper we propose a novel RTS organization designed to work with an OpenMP infrastructure which targets the aforementioned problems. Instead of having a single monolithic OpenMP RTS for a given device, we propose an adaptive RTS architecture which implements only the features required by a particular application. More specifically, the compiler analyzes the kernels that are to be offloaded to the device, and provides metrics which are later used to select a particular RTS configuration tailored to the needs of the application. This way the user's code implies the choice of an appropriately optimized RTS which may result to reduced executable sizes and/or faster execution times. For example, an OpenMP kernel which does not make use of explicit tasks can greatly benefit by a barrier which avoids time consuming tasking actions. Furthermore, the resulting specialized runtime library, does not need to include the tasking subsystem, leading to reduced executable size. Our technique is quite general and can be also utilized in the OpenMP runtime system executing on the host.

The remainder of the paper is organized as follows: In Section 2 we give an overview of related work while in Section 3 we present some background material on OpenMP and the new extensions for device support. In Section 3.1 we discuss the difficulties induced when developing an OpenMP RTS for devices. An overview of our adaptive RTS proposal is presented in Section 4. In particular Sections 4.1 and 4.2 present the kernel analysis procedure that produces a set of metrics and the way these metrics are utilized. We then describe a prototype implementation of our proposal in Section 5 and in Section 6 we present some evaluation measurements. Section 7 concludes this work.

2 Related Work

OpenMP was considered as a possible model for accelerators or multicore embedded systems long before the introduction of its device extensions. Liu and Chaudhary [23] implement an OpenMP compiler for the 3SoC Cradle system, a heterogeneous system with multiple RISC and DSP-like cores. Additionally in [17] double buffering schemes and data prefetching are proposed for this system. Sato et al [25] implement OpenMP and report its performance on a dual M32R processor, which runs Linux and supports fully the POSIX execution model. In [21] Woo-Chul and Soonhoi discuss an OpenMP implementation that targets MPSoCs with physically shared memories, hardware semaphores, and no operating system. Cabrera et al in [11] propose some OpenMP extensions to provide a high level API for executing code on FPGAs. They propose a hybrid computation model supported by a bitstream cache in order to hide the FPGA configuration time needed when a bitstream has to be loaded. In [19] Hanawa et al evaluate the OpenMP model for the Renesas M32700, ARM/NEC MPCore, and Waseda University RP1 multicore embedded Systems.

Furthermore, extensions to OpenMP have been proposed to enable additional models of execution for embedded applications. Gonzàlez et al [18] extend OpenMP to facilitate expressing streaming applications through the specification of relationships between tasks generated from OpenMP worksharing constructs. Carpenter et al in [12] propose a set of OpenMP extensions that can be used to convert conventional serial programs into streaming applications. Chapman et al [13] describe the goals of an OpenMP-based model for different types of MPSoCs that take into account non-functional characteristics such as deadlines, priorities, power constrains etc. They also present the implementation of the worksharing part of OpenMP on a multicore DSP processor. In [10] Burgio et al present an OpenMP task implementation for a simulated embedded multicore platform inspired by the STHORM architecture. Their system consists of doubly linked queues which store the tasks. They make use of task cut-off techniques and task descriptor recycling. In [5] Agathos et al present an implementation of OpenMP on the STHORM accelerator. The innovative feature of their design is the deployment of the OpenMP model both at the host and the fabric sides in a seamless way, which provides the programmer with an interface similar to the device model of OpenMP 4 for offloading and executing OpenMP kernels on the MPSoC.

Support for OpenMP 4.0 devices is fairly limited both in the compiler side and the device side. In fact, there are very few compilers that implement the `target` construct and the only device they support is the Intel Xeon Phi [20, 1]. Details of the offload procedure in the ICC compiler are given in [26]. Preliminary support for the OpenMP `target` construct is also available in the ROSE compiler. Chunhua et al [14] discuss their experiences on implementing a prototype called HOMP on top of the ROSE compiler, which generates code for CUDA devices. Bertolli et al [8] propose a method to coordinate threads in an NVIDIA GPU using a single kernel as opposed to multiple kernels; they also discuss how their methods could be implemented as

part of the LLVM compiler implementation of OpenMP 4.0. In [24] the authors present their implementation of OpenMP 4.0 on a TI Keystone II, where they use the DSP cores as devices to offload code to. Finally, Papadogiannakis et al in [29] present the infrastructure for device support in the OMPi compiler, placing special emphasis on the problem of data environment handling, while Agathos et al in [6] present the first implementation of the OpenMP 4.0 accelerator directives for the Parallella board [4], a credit-card sized multicore system consisting of a dual-core ARM host processor and a distinct 16-core Epiphany co-processor. All the these works either propose a partial OpenMP implementation or a monolithic full implementation, which may consume the limited system resources. This is in contrast to our proposal, where adaptive RTS configurations are utilized for different applications.

3 The OpenMP 4 Device Model

One of the key new features of version 4.0 of the OpenMP API [28] is the introduction of a state of art, platform-agnostic model for heterogeneous parallel programming. Multiple devices, as for example co-processors, graphical processors or accelerators, can be utilized to reduce the execution time and improve the energy efficiency of an application by utilizing the new device directives. The programmer simply marks portions of the (unified) source code to be offloaded to a particular device; the details of data and code allocations, mappings and movements are orchestrated by the compiler. The OpenMP device model requires that the target devices are connected to a host processor which is also considered a device. The program execution follows a host-centric model; it starts executing at the host side until one of the newly introduced constructs is met, which may trigger the creation of data environments and the execution of a specified portion of code on a given device.

In order to transfer data and control flow to a device, the `target` directive is used. This directive has an associated structured block representing the code (*kernel*) to be offloaded and executed directly on the device side. During the execution of the kernel the host task waits until the device finishes and returns back the control. Each `target` directive may contain its own data environment, that is a set of variables accessible in some way by both the host and the device, initialized when the kernel starts and freed when the kernel ends its execution. In the case where the `if` clause is used, and the condition evaluates to false then the target region will be executed by the host CPU instead of the chosen device.

Data movements between the host and the devices may be the cause for large delays during the launch or the completion of the kernels. In order to avoid repetitive creation and deletion of data environments, the `target data` directive allows the definition of a data environment which persists among successive kernel executions. When an `if` clause is used, and the condition is evaluated to false then the data environment is initialized in the host memory space. Furthermore, the programmer can use the `target update` directive between successive kernel offloads to selectively update data values that reside in the host and the device data envi-

ronments. Finally, the `declare target` directive specifies that the associated set of variables and functions are mapped to a device. In essence, the `declare`d variables are allocated in the global scope of the target device, and their lifetime equals the program execution time. The code of the `declare`d functions is compiled to produce device binaries accessible from the `target` regions.

A device data environment can be manipulated through `map` clauses within `target data` and `target` directives. These clauses determine how the specified variables are handled within the data environment. When an `alloc` map type is used, an uninitialized variable is defined, whereas with a `to` map type the variable is additionally initialized from the value of the corresponding host variable. If variable is mapped as `from` then an uninitialized device variable is defined; when the specified directive region finishes, the value of the device variable is copied back to the original host variable. Finally, if no type is specified or the type is `tofrom`, the variable has the characteristics of both `to` and `from` types.

Except for the aforementioned core constructs, the OpenMP device model includes additional directives that may be suited to better exploit particular device architectures. For example, the `teams` directive creates a given number of thread teams, where each team has a specified number of threads and the master thread of each team executes the associated code block. The `distribute` worksharing construct distributes the iterations of the loops across the master threads of all teams that execute the `teams` region. The combination of `target`, `parallel`, `teams` and `distribute` directives offers an effective way for exploiting the compute units of GPGPU-style accelerators.

A major characteristic regarding the kernels code is that they can utilize arbitrary OpenMP functionality, with no restrictions whatsoever (except that they cannot offload code to other devices). This implies that any code that adheres to v3.1 [27] of the specifications can potentially form a legal kernel. The only requirement is that all the global variables and functions accessed from within the kernel code must be declared in a `declare target` directive and reside at file, namespace, or class scope. Thus, the constructs for dynamically creating a team of threads, synchronizing them, sharing work among them (`for` loops, `sections`), using explicit tasking, even employing nested parallelism, are all allowed within a `target` region. This flexibility makes OpenMP a very powerful parallel programming model for taking advantage of all available compute resources of a heterogeneous system in a intuitive and efficient manner. Ideally any OpenMP program originally written for a shared memory system, can be easily offload some of its computationally intensive parts onto specialized hardware. On the other hand, to make all the above possible, the attached devices are effectively required to provide complete OpenMP support. Since most of these devices are not usually equipped with abundant resources, implementing full or efficient OpenMP support is not an easy task.

3.1 OpenMP on the Device Side

As discussed above, the OpenMP device model offers a great deal of flexibility regarding the constructs allowed within kernel codes. This in effect requires that a complete OpenMP RTS be present to support kernel execution. However, OpenMP was originally designed for shared memory multiprocessors. These machines include a large amount of shared memory supported by sophisticated cache coherent protocols, high bandwidth interconnections and usually offer a large set of hardware-assisted synchronization primitives (e.g. compara-and-swap, fetch-and-add, memory barriers). Moreover, these systems are equipped with an operating system accompanied with optimized low-level software libraries such as POSIX threads, for manipulating the execution units of the system.

On the other hand, embedded or attached accelerators have different architectures and are designed to serve different purposes. For example, the organization of some accelerators is targeted to the efficient execution of streaming applications; GPGPUs are better suited to speed up matrix-based computations; e.g. co-processors are synonymous to hardware diversity, since each manufacturer equips a product with specialized hardware modules and target a specific class of applications.

With some notable exceptions such as the Xeon Phi accelerator[26], a common characteristic of the various types of co-processors is that they offer a limited amount of resources. Hence, the challenges posed when implementing an OpenMP RTS for such devices depend on these resource limitations. The absence of a POSIX-like interface for manipulating threads may add design difficulties or considerable offloading costs regarding dynamic or nested parallelism. For example, most of the current native development tools for GPGPUs do not support nested parallelism. Lack of hardware synchronization primitives would add overheads in the cooperation of the accelerator cores, since software implementations of locks or barriers result to considerable delays. Arguably, one of the most important limitations is the size of the available memory; small private or shared memories at the co-processor cores impose restrictions regarding the kernel executable size and/or the actual application data. This is particularly pronounced in the absence of a fast global memory; the kernel code has to include the OpenMP RTS, further limiting the available memory space. The Epiphany accelerator used in the Parallella [4] is an example of an embedded accelerator with severely limited memory resources; each core is equipped with just 32KiB of fast local memory. While it can also access a larger 32MiB memory shared with the host processor its access times are almost an order of magnitude larger.

There are two approaches for supporting OpenMP on a device with limited resources:

Partial support Partial support of the constructs is a pragmatic solution that works in real world applications [24, 14, 13]. For example, there is no point in trying to implement an optimized tasking infrastructure for a GPGPU with limited local memory which lacks fine grain synchronization primitives. Instead, a careful implementation of a combined

construct such as `target teams distribute parallel for` is a desirable feature that exposes the computational power of this kind of hardware. Of course, partial support minimizes the expressiveness of the programming environment. The application code may have to be redesigned to match the availability of OpenMP constructs, a fact that also reduces code portability and re-usability.

Full support Some works in the bibliography [20, 1] choose to support OpenMP fully on the device side. This strategy provides a powerful tool for developing parallel applications based on a high level hardware abstraction. Nevertheless, the design of a complete OpenMP RTS is not a trivial task. Furthermore, the hardware limitations may lead to poor performance for some of the OpenMP constructs [6, 8, 14].

4 Proposed System

In this work we propose a general methodology which can be utilized to offer flexible and adaptive OpenMP RTS. The goal is the development of an RTS architecture which implements only the OpenMP features required by each particular application. That is, it results in an application-specific RTS configuration. This is possible because of a key observation: all kernel code must lie within a single source file. This enables a compiler to analyze the behavior of the kernel with respect to OpenMP constructs, through detailed interprocedular analysis. Thus, it can decide exactly what constructs are used, their nesting levels, the types of employed loop schedules, etc.

The proposed system is shown in Fig. 1. The compiler is responsible for analyzing and transforming the code. It takes as input an OpenMP program with `target`-related constructs. The output is a set of files; the main one is to be executed on the host and the other files represent the kernels to be executed on the devices. Along with each kernel, a set of *metrics* which are gathered during its analysis are output. The metrics are passed to the *mapper*. The latter is responsible for choosing the most efficient runtime configuration for the given metrics. In order to do this, the mapper either selects one from a precompiled set of libraries or parametrizes appropriately one of them and builds it on the fly.

4.1 Analyzing a Kernel

The motivation behind our proposal stems from the observation that providing comprehensive OpenMP support for an attached device can be quite demanding both in terms of memory requirements and execution overheads, especially in devices with limited resources (e.g. embedded MPSoCs). Each kernel should be accompanied by a rather sizable runtime library in order to enjoy OpenMP support. However, most applications (typical kernels included) rarely need all of the OpenMP facilities. What if the compiler can decide on the subset of OpenMP

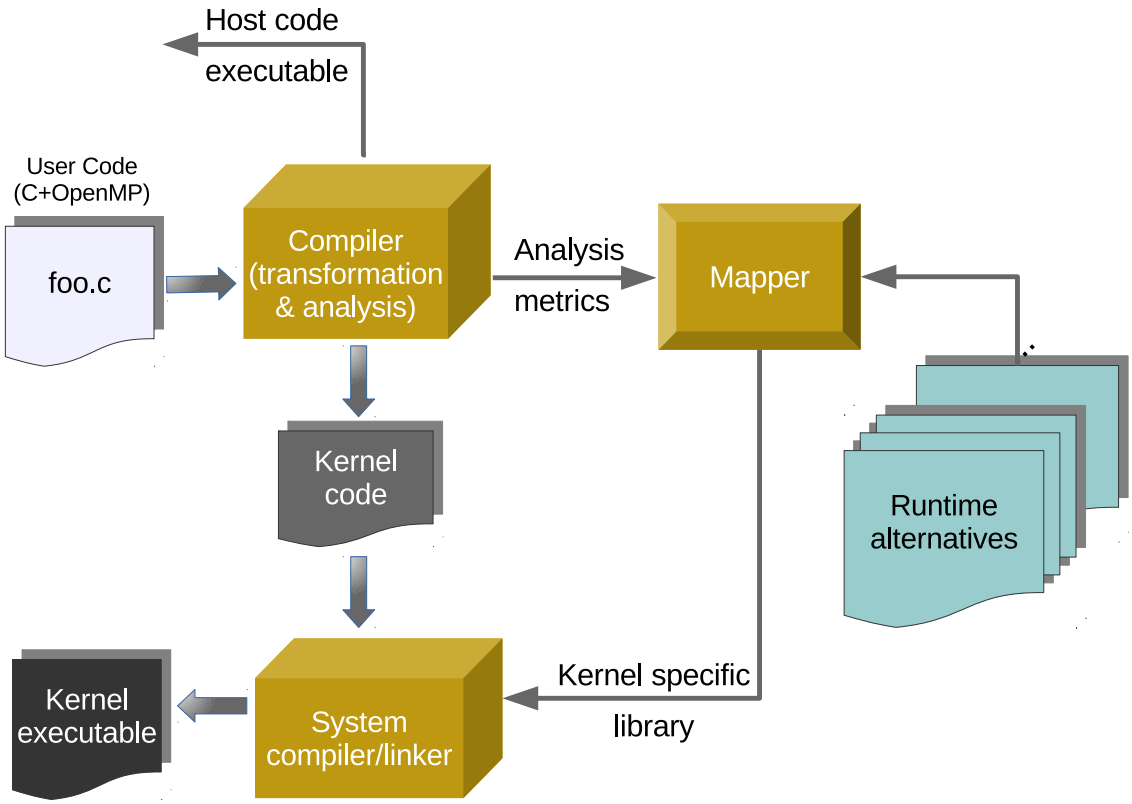

Figure 1: Overview of compiler assisted RTS

functionality that is necessary and just utilize a custom, kernel-specific runtime library to accompany it? The potential savings could be quite significant.

An OpenMP kernel is defined by a block of code enclosed lexically within a `target` construct. The actual kernel *region* includes any code in called routines. Such routines are defined within `declare target` constructs and are in fact offloaded with the kernel. The compiler has thus access to the whole kernel region and can employ inter-procedural analysis in order to analyze the entire dynamic extend of the kernel.

The compiler can build the call graph of each kernel and visit each of the called routines. Our thesis it that the compiler can then extract information about the employed OpenMP constructs (if any), and thus determine the actual OpenMP functionality that is necessary for the execution of each particular kernel. More often than not, a given kernel will not require the entire OpenMP functionality but a rather small portion of it. Given this information, the offloaded kernel can be accompanied by a suitable subset of the OpenMP runtime library, potentially decreasing the total offloaded footprint.

Here is a number of important conclusions, among others, that can be derived from the above code analysis:

- *OpenMP in kernel*: Decide whether OpenMP functionality is required at all. If no OpenMP directives are utilized and no OpenMP runtime functions are called, then there is no need to include OpenMP support.

- *Dynamic parallelism*: Determine whether the kernel spawns parallelism, through `parallel` directives, and if possible, their nesting levels and/or the total number of threads employed. The absence of parallelism can make the required runtime support rather minimal. Knowledge about the number of threads and the nesting levels can also tailor the corresponding runtime data structures to exact sizes.

- *Work-sharing regions*: A major portion of an OpenMP runtime library is devoted to the handling of worksharing regions. Knowledge about the exact types of worksharing constructs utilized by a kernel (`for`, `sections`, `single`) can slim down the necessary runtime support.

- *Explicit tasking*: Discover the presence of user-defined `task`s. If none is observed, the tasking subsystem of the runtime library is not needed at all. Supporting tasks is one of the most sophisticated assets of an OpenMP RTS, with significant overheads and memory requirements.

- *Internal control variables*: Decide whether the code makes use of OpenMP Internal Control Variables (ICVs), either by setting them or getting their values. Furthermore, it can be determined exactly which ICVs are being utilized. Storing and maintaining ICV values represents a major issue in an OpenMP support library. If, for example, ICV values are used only for retrieving information then a single copy of them (instead of repeating them in every task structure) is adequate for the execution of the whole kernel.

The above proposal can be extended to the general case of host OpenMP programs, not just kernels. The only obvious requirement, is that the application code must not refer to external routines, so that the compiler is in a position to perform full inter-procedural analysis and derive the above conclusions. In case where the program depends on external routines, the analysis response will declare inability to provide valid conclusions or metrics. Otherwise, the conclusions and a set of related metrics will be output to optimize the RTS used for the specific application.

4.2 Mapper: Utilizing Compiler Metrics

The set of metrics generated by the compiler are passed to the mapper module which is responsible for choosing the most appropriate runtime "flavor". In the case where the kernel does not make use of OpenMP directives, the RTS should only include basic features for enabling a single co-processor core to execute the serial code of the kernel. Thus, this RTS should include mainly host-specific functionalities, and will result to a minimal footprint library regarding the

device side. The reduced capabilities of the RTS include the co-processor initialization and finalization phases, as well as the code and data offloading. This minimal RTS may prove quite useful in systems where the parallelization capabilities can not be abstracted as a team of independent threads. Furthermore, there might be cases where dynamic creation of a parallel team on the device side is hard or sometimes impossible to implement. A workaround for this scenario is the utilization of a team of threads executing on the host side, that concurrently offload kernels (containing serial code) to a multicore co-processor.

The usual case, nevertheless, is where the kernel includes directives for creating a parallel team of threads that cooperatively execute a code block. The RTS library that is to be linked with the kernel code consists of some RTS-specific data along with the code implementing the required functionalities. In more detail the internal data are related to:

- the execution entities, represented mainly by some kind of thread abstractions and
- the implicit (or explicit) tasks executed by the threads

The smaller total footprint for the RTS library the more beneficial would be in the cases where the cores of a co-processor are equipped with small amount of local memory.

Additionally, the presence of extra functionalities within the library not needed by the kernel code, may lead to non-optimized execution times. A representative example of such functionality are the tasking extensions of the OpenMP barrier. Upon encountering a barrier construct, a thread has to wait until all its siblings reach the barrier and until all pending tasks of its team are executed. The implementation of former condition results to far less overheads when compared with the task executing part. Typically threads can be notified about the entrance of their siblings in the barrier through the use of a counter or a matrix of flags. Ensuring that all pending tasks are executed though, involves repeated snooping to shared queues and/or counters, adding substantial amount of overheads. Thus, avoiding the unnecessary functionalities may result to execution speed-up.

Implementing a general, full fledged RTS which is capable of offering OpenMP support is the typical approach in the bibliography. What we propose here is to utilize custom, application-driven RTSs, that only supply the functionality required by the particular application. In Fig. 1 the mapper is the module responsible for this: based on the application characteristics as depicted by the compiler-generated metrics, it optimizes the RTS by tuning its internal data and functionalities to best fit for the particular application.

Possible realizations of specialized runtime libraries include:

- A fixed set *pre-compiled* libraries. The set of libraries is selected to target specific classes of applications, as derived from typical use-case scenarios. For example, there can exist a library that does not provide tasking support. Another possibility would be a trimmed down library that only supports a selected work-sharing construct (e.g. `for` loops). The mapper then undertakes the task of mapping the provided kernel metrics to the set of

available libraries; the most appropriate one should be selected so as to minimize the offered OpenMP functionality while at same time covering all kernel requirements.

- A set of on-the-fly *parameterizable* libraries. Because not all applications can benefit from the default values of the runtime parameters, the mapper can choose to tune some parameters according to kernel characteristics and build different library flavors. For example, if the team sizes are known, the barrier data structures can be tuned to service the specific number of threads. Of course, parametrization requires recompiling and thus the custom libraries are built at the compile-time of the application.

The mapper combines the metrics with all possible library configurations to provide the optimized library. The larger the number of the pre-compiled libraries/parametrized RTSs, the better the decisions taken by the mapper for the result libraries that will support the application kernels.

5 Implementation in the OMPi Compiler

The OMPi compiler [15] is a lightweight OpenMP C infrastructure, composed of a source-to-source translator and a flexible, modular RTS. OMPi is an open source project and targets general-purpose SMPs and multicore platforms. It adheres to V3.1 of the OpenMP specifications, while also supporting a number of V4.0 constructs. In particular, all the `target`-related device constructs, the cancellation constructs, and the `taskgroup` construct are already supported.

The compilation process for an accelerator-assisted program is shown in Fig. 2. The compiler takes as input C code with OpenMP directives, and after the pre-processing and transformation steps, it outputs a multi-threaded C file for executing on the host and another set of intermediate files, one for each kernel (i.e. one for each `target` region in the user program). Every intermediate file has been augmented with calls to the RTS of the corresponding device. In the last stage, the intermediate files are compiled with the appropriate system compiler in order to provide the final executables. In order to implement the proposed mechanism, this last stage is where the mapper module should be inserted. The intermediate files must carry the deduced metrics so as to guide the mapper. Finally, the compiler must be equipped with new kernel analysis capabilities in order to derive the desired metrics.

5.1 Kernel Analysis

The analysis of the kernels is done at a high level. The whole program is represented by an abstract syntax tree. Upon encountering an OpenMP `target` node, the compiler analyzes its body and follows the chain of routine calls (if any) in order to discover the OpenMP functionality required by this particular kernel. To avoid visiting a routine multiple times (since

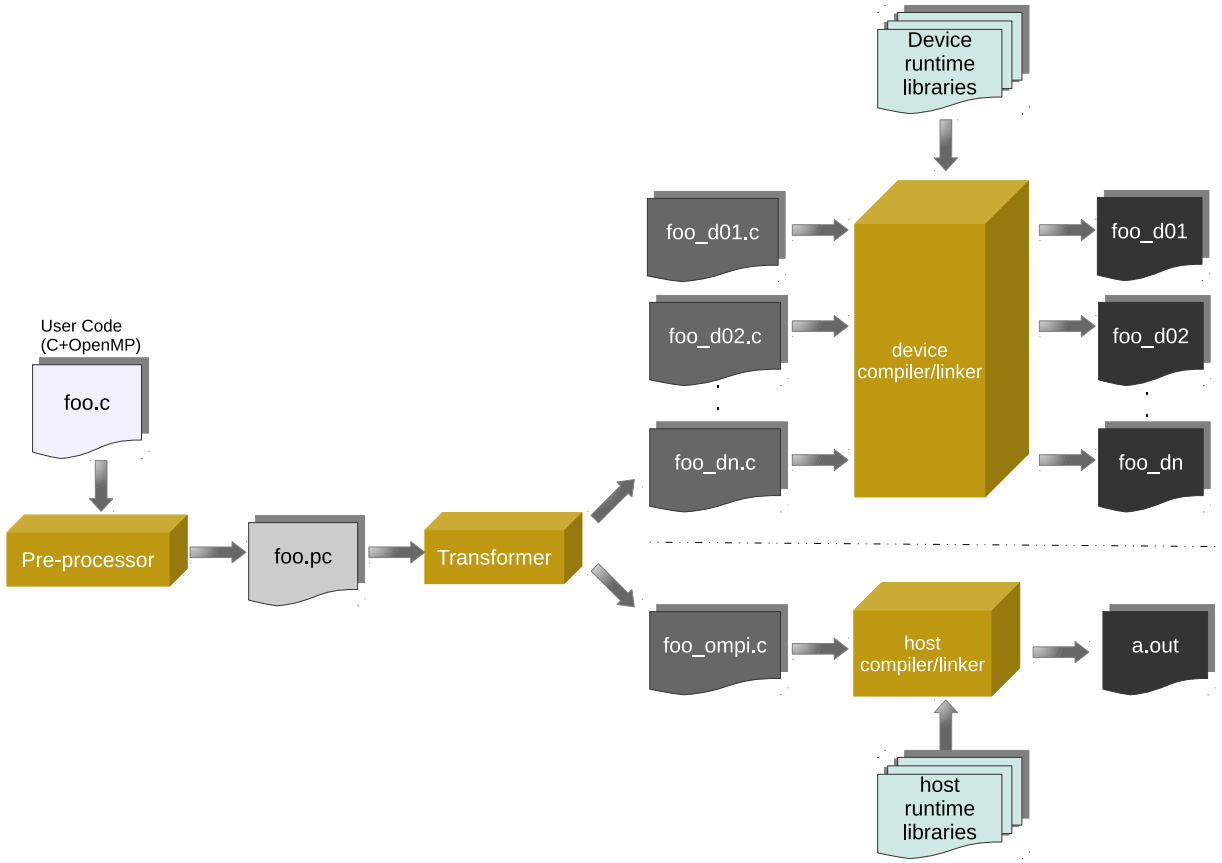

Figure 2: OMPi compilation chain

it may be called by multiple kernels), all routines defined within `declare target` regions are analyzed before any other program transformations. The compiler constructs the call graph and traverses it; for each visited function f, the following are some of the metrics currently gathered:

- The total number of OpenMP constructs
- The number of first-level (non-nested) `parallel` constructs ($N_p^{(f)}$).
- The number of `for` loop ($N_l^{(f)}$), `sections` ($N_s^{(f)}$) and `single` ($N_i^{(f)}$) constructs; a counter for the number of constructs with `nowait` clauses is also maintained ($N_{nw}^{(f)}$).
- The number of `task` constructs ($N_t^{(f)}$).
- The number of explicit `barrier` directives ($N_b^{(f)}$).
- The maximum level of parallelism ($L_p^{(f)}$).

All the metrics except the last one count the constructs encountered in the function itself. The parallelism nesting level is determined from the function and all the functions called by it as follows: If a function g is called by f at nesting level $l_{f\to g}$, then the nested parallelism level

for this particular call is given by $l_{f\to g} + L_p^{(g)}$. The maximum parallelism level observed for function f is given by:

$$L_p^{(f)} = \max_{g \text{ called by } f} \left\{ l_{f\to g} + L_p^{(g)} \right\}.$$

Consequently, if for example $L_p^{(f)} = 1$, there may be no need to add support for nested parallelism to a kernel that calls function f. If the compiler detects recursion, this particular metric is disabled.

To maximize performance, OMPi allows overlapping worksharing regions whereby each thread of a team may proceed independently to a following worksharing region, as long as the previous one contains a `nowait` clause. The mechanism is quite complex [30, 31] and requires handling of sizable data structures. If $N_{nw}^{(f)} = 0$, there is no need to implement it; a simple blocking barrier would be enough to support all worksharing regions. On the other hand, if for all functions f called by a kernel, $N_{nw}^{(f)} = N_l^{(f)} + N_s^{(f)} + N_i^{(f)}$, and $N_b^{(f)} = 0$, there may not be a need to implement a barrier mechanism at all.

The gathered metrics are used at every encounter of a `target` tree node during code transformations. Before actually transforming the construct, its body is analyzed in a similar way as above, and the metrics are combined with the precomputed ones for every function called from the kernel. The final set of metrics are stored in a table and the compiler proceeds to the transformation of the kernel body. During code generation, the computed metrics for each `target` construct are embedded into the corresponding kernel file as C language comments, for communicating them to the mapper.

5.2 A Concrete Target: The Epiphany Accelerator

The Parallella-16 board is a popular 18-core credit card-sized computer equipped with two processing modules; the main (host) CPU, a dual-core ARM Cortex V9 processor, and the 16-core Epiphany chip which is used as a co-processor. Each Epiphany core (eCORE) has 32 KiB of fast local memory. The board has 1 GiB of DDR3 RAM, addressable by both the host CPU and the Epiphany providing a 32 MiB portion to be used as *shared memory*. More details about the hardware are provided in Section 6. Currently OMPi supports most of the device directives of OpenMP and is the first compiler to support the Epiphany accelerator of the Parallella board. Here we present the key aspects of the original RTS for the Epiphany; more details can be found in [6, 29]. The Epiphany module consists of two parts; the first is executed at the host space and is used for controlling and accessing the Epiphany device. The second part is executed by the Epiphany cores and provides support of OpenMP within the device side.

The communication between the ARM and the Epiphany eCOREs occurs through the shared memory portion of the system RAM. The eCOREs do not execute any operating system and there is no provision for creating and handling dynamic parallelism within the Epiphany chip. Because only the host can activate and control the eCOREs, when offloading a kernel it chooses

the first idle core and loads the precompiled kernel object file to it for immediate execution. Dynamic parallelism in a kernel is achieved with support from the host; the core that meets a `parallel` region contacts the host and requests the activation of a number of cores. A copy of the same kernel is then offloaded to the newly activated cores.

The limited local memory of the device cores makes it impossible to fit sophisticated OpenMP RTS structures alongside the application data. The original RTS started as a customized version of the host OpenMP runtime library, carefully trimmed so as to minimize its memory footprint. The coordination among the participating eCOREs occurs through structures stored in the local memory of a team's master core. The synchronization mechanisms (locks and barriers) are customized versions of those provided by the native libraries. The tasking infrastructure is based on a simple blocking shared queue which is also stored in the local memory of the team's master eCORE, for speed. On the other hand, the corresponding task data environments for each task are stored in the slower shared memory area, due to space requirements.

This original RTS was used as a basis for the design of a set of adjustable RTSs, each one specialized for a certain type of kernels. For the rest of the text we will refer to the original RTS as the *Full* RTS. It is built as a Linux static library, and is linked with each offloaded kernel. It is organized as a collection of a largely independent routines so that the system linker can attach only the necessary routines with each kernel. However, the complex relations between the internal data structures and the runtime routines force the linker to include sizable portions of the library. As a result, the *Full* RTS has a relatively large footprint, even when it accompanies an effectively empty kernel [6]. Furthermore, because dynamic memory allocation is not supported at the eCORE level, the RTS must reserve enough local memory space to cover the worst case. As a result, the actual local memory left for pure application data is well below the 32 KiB available.

Our strategy for implementing the proposed mechanism was to create different library flavors, aiming to minimize the library footprint. In particular, based on detailed analysis of the runtime organization, we identified three parts that contribute the most both because of the size of the involved routines and the size of the required data structures:

1. Dynamic parallelism. A substantial amount of data and routines are needed in order to support dynamic parallelism within a kernel. In particular, beyond the data structures needed for controlling parallel team members, extra room is necessary for communicating with the host processor. Furthermore, the thread synchronization mechanisms, especially the barrier, consume additional memory space. All this is more than doubled if a second level of parallelism is to be supported.

   All this infrastructure can be discarded if the kernel does not contain a `parallel` directive. Hence we developed an RTS variant which does not support dynamic parallelism creation. Although the original design can support arbitrary levels of nested parallelism,

there is no practical use for more than two levels of parallelism on 16 cores. Consequently, we designed two versions of the RTS, one supporting exactly one level of parallelism and another supporting two levels.

2. Worksharing. The OpenMP worksharing constructs (`single`, `for`, `sections`) may have different combinations of `reduction`, `schedule` (with the various schedule types), `collapse`, `ordered` and `nowait` clauses. Supporting all of them requires data structures with large memory footprint. In practice, typical applications do not utilize all possible variations. As a result, we developed a set of RTSs that support specific combinations of the above constructs and clauses.

3. Tasking. The tasking infrastructure for the Epiphany is the module with the largest memory requirements. The required functionalities include fine grain synchronization so most of the runtime data must be stored in local memories; in particular they are stored in the local memory of the team's master eCORE. This means that the local memory of one eCORE stores the tasking data of all eCOREs. Because all eCOREs are candidates for team masters, preallocated tasking structures must be present in the local memories of all eCOREs. Furthermore, barrier synchronization is charged with task execution duties which impact overall performance. To optimize the support for applications which do not utilize tasks, we developed RTS flavors with no tasking subsystem. In addition, these flavors implement lighter/faster versions of the barrier mechanism.

6 Evaluation

6.1 Environment

To evaluate our proposed method we used the Parallella-16 SKUA101020 board. The board comes with standard peripheral ports such as USB, Ethernet, HDMI, GPIO, etc. and is equipped with two processing modules; the main (host) processor, which is a dual-core ARM Cortex A9 with 32 KiB L1 cache per core and 512KiB shared L2 cache, built within a Zynq 7010 SoC and an Epiphany-III 16-core co-processor. The former runs Linux and uses virtual addresses while the latter does not have an OS and uses a flat, unprotected memory map. The Epiphany-III has 16 cores (eCOREs) and a peak performance of approximately 25 GFLOPS (single-precision) with a maximum power dissipation of less than 2 Watt. 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.

A closer look at the architecture of the Epiphany reveals a 64×64 mesh interconnect, so in theory systems up to 4096 eCOREs are possible. On the Epiphany III the chip is pinned on a 4×4 submesh of the virtual 64×64 mesh whose north-west coordinates are (32, 8), as shown in Fig. 3. The chip has four eLinks (west, east, north and south), that may be used

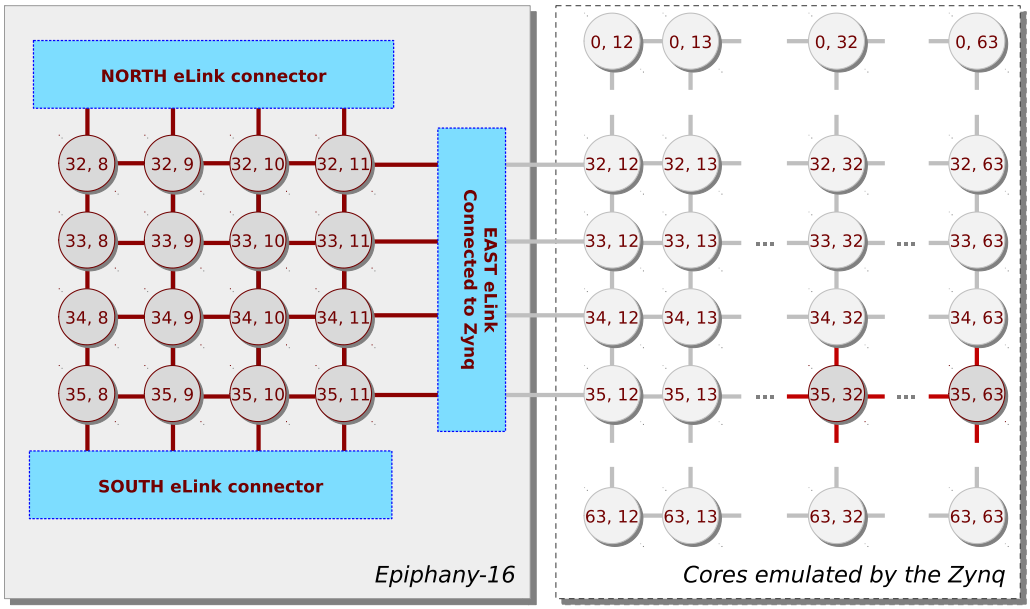

Figure 3: The Epiphany co-processor

to interconnect it with other chips. In the Parallella board version, the west eLink is inactive and the east eLink is connected to the Zynq host. Each eCORE is a 32-bit superscalar RISC processor, capable of performing single-precision floating point operations, and owns 1 MiB of the total address space, which is addressable by all cores. However it comes with just 32 KiB of local scratchpad memory; in addition it is equipped with two DMA engines. All memories are available through regular load/store instructions by all eCOREs.

All common programming tools are available for the ARM host processor. For the Epiphany, a Software Development Kit (eSDK [3]) is available, which includes a C compiler and runtime libraries for both the host (eHAL) and the co-processor (eLIB). For the results presented here we used eSDK v5.13.9.10 which includes the GCC and E-GCC compilers for the host and the Epiphany executables respectively.

6.2 A Detailed Breakdown of the RTS of OMPi

The original RTS support for the Epiphany [6] was designed to provide full OpenMP support, under the constraint of the limited memory resources. The first step towards designing a set of adjustable RTSs was to analyze the original runtime and understand the impact each component has. The purpose of this procedure was to discover in detail byte sizes of all different data structures and the corresponding functionalities they support. The results of this analysis guided the design of distinct RTSs, specialized to different kernel scenarios.

In Table 1 we present the sizes of the most important runtime data structures. Notice that these represent only the eCORE-resident parts; additional data structures are kept in the (slower) shared memory and are of no interest here. The RTS of OMPi utilizes two fundamental

Table 1: Data sizes in the original RTS

Data structures	**Size in bytes**
EECB data	1440 (1 active region)
Essential	48
Worksharing	≥ 80
No-wait regions	$8 + 64 \times$ (# active regions)
Loops	32
Static loops	4
Ordered	28
Sections	4
Tasking data	1312
Task descriptor	72
ICV data	32 per task
Reductions	16 per eCORE
Critical	16 per eCORE
User defined locks	176 per eCORE
Nested parallelism	88(+1088 in SM) per level

descriptors: the thread and the task descriptor. The former is named EECB (execution entity control block) and holds all the information needed by an OpenMP thread to execute a code region and to coordinate with sibling or child threads. The later holds the data required for the execution of a specified task, either implicit or explicit. As seen in Table 1, the sizes of these entities have the biggest impact on the total footprint of the RTS.

Not all bookkeeping data are actually needed for the execution of every kernel. The idea is to trim down these structures to save local memory, while at the same time satisfy the real needs of a kernel. The essential data require 48 bytes per EECB while the data for worksharing constructs are 80 bytes. A closer look at each worksharing construct reveals that the loop construct occupies almost half of the space. A performance-oriented but memory-consuming feature of the original runtime, is the ability to allow multiple active worksharing regions, whereby each thread of a team may proceed independently to a following worksharing region, as long as the previous one contains a `nowait` clause. If up to n overlapping regions are supported, an additional $n \times 64$ bytes per EECB are necessary.

The space needed for a task descriptor is 72 bytes. In the current RTS all the task-related structures of all eCOREs must be stored in the EECB data structure of the master eCORE. Because all eCOREs are eligible as team masters, the same space must be provided in all eCOREs. This results in a total 1312 bytes per EECB for the whole tasking mechanism. This demonstrates the potential for reducing the memory footprint in the case where the an application does not use explicit tasking. The size of the necessary ICVs is measured to be 32 bytes per task. If the

kernel does not modify any of their values (e.g. there are no calls to `omp_set_xxx()` routines), then it could be possible for the RTS to use only one copy of the ICVs for all tasks. In such a case, up to 12256 bytes could be saved in total (in the local memories of all 16 eCOREs).

Synchronization between the eCOREs is relatively cheap, since 16 bytes per core are needed for the `reduction` and `critical` constructs. Due to lack of dynamic memory allocation, the original runtime pre-allocates space for 8 user-defined locks. This mechanism requires 176 bytes in the local memory of each eCORE, even if a kernel uses no locks at all. Finally, considering parallelism levels, the data needed for each supported nesting level occupies a significant amount of memory. Each additional level needs 88 extra bytes in the local memory (plus more than 1KiB on the shared memory).

We should make two important observations at this point. First, except for the data structures, there is the corresponding code that handles them, so removing unnecessary data structures has the beneficial side-effect of decreasing the size of the library code. Second, slimmer code usually means faster code. Although in this section we concentrated on minimizing the library sizes, we also expect to have some performance gains for free. Additional performance gains are possible by redesigning the employed algorithms. For example, if the tasking subsystem is removed from the equation, a significantly faster barrier implementation is possible, which avoids polling for tasks to execute.

6.3 Implementation of Runtime Flavors

Based on the above analysis of the original RTS, we redesigned and implemented 12 different runtime flavors. Each flavor is a modified version of the original, trimmed to support a limited number of constructs. For each flavor we removed the unnecessary internal data structures and modified all routines respectively. The set of the different RTSs are as follows:

(1) *NoOMP*. This RTS does not support any OpenMP directives within the kernel. Each eCORE can execute only sequential code.

(2) *ParallelOnly*. This RTS provides the mechanism for an eCORE to form and deform a parallel team. No other OpenMP functionality is supported.

(3) *ParReduction*. This RTS is an extension of the previous one, which implements the `reduction` clause on a list of variables.

(4) *ParCritical*. This RTS extends (2) and allows only the `critical` synchronization construct between the eCOREs of a parallel team.

(5) *ForStatic*. This is the *ParallelOnly* RTS where the team members can also utilize the `for` worksharing construct. Only the `static` schedule is supported. No other worksharing constructs are offered.

(6) *ForOrdered*. This alternative extends the previous one by adding the ability to utilize the `ordered` clause of the `for` directive.

(7) *SingleOnly*. Here we extend the *ParallelOnly* flavor by supporting only the `single` worksharing construct.

(8) *NoTasks*. We developed this RTS to optimize the support of kernels with no explicit tasks. The rest of the OpenMP functionality (e.g. workharing, synchronization, etc) is present.

(9) *BlockingOnly*. This is an almost complete OpenMP RTS but the support for `nowait` worksharing regions has been disabled in order to reduce the footprint of the related RTS structures.

(10) *NoTasksBO*. We added a variation of the *BlockingOnly* flavor where the tasking support has been removed.

(11) *SingleTasks*. This RTS provides support for creating teams of eCOREs that can create explicit tasks and can workshare only through the `single` construct. This is based on a common pattern where one thread generates tasks and the others consume them.

(12) *Full*. This is the original RTS.

The above set does not cover all possible use cases, i.e. does not include all possible combinations of OpenMP constructs. Instead it was guided by common sense for supporting usual application scenarios. Anyway, our goal is to prove the potential of the proposed mechanism, and not to derive all possible runtime flavors targeted for all possible kernels.

Furthermore, all RTS routines were carefully trimmed to implement only the required functionality. Barriers constitute an important example. In a complete OpenMP runtime system a barrier has to synchronize team threads and also act as a task scheduling point. In all flavors but *BlockingOnly*, *SingleTasks* and *Full* there is no tasking support and consequently barriers were simplified to handle only thread synchronization.

The RTSs are parametrized so as to offer better adaption to specific kernels. This adjustment occurs during the library compilation time, based on the metrics produced by the compiler. Specifically, in all flavors but *NoOMP* we parametrize the number of parallel nesting levels that the library supports. For the *NoTasks* and *Full* RTSs there is an additional parameter that sets the number of maximum active `nowait` worksharing regions. Furthermore for the RTSs (8), (9), (10) and (12) we can limit the number of user-defined locks that the library can support.

6.3.1 Choosing the Runtime Module

The mapper imports the set of metrics provided by the compiler and uses them in order to choose the most appropriate RTS module to be linked with a kernel. In our implementation the

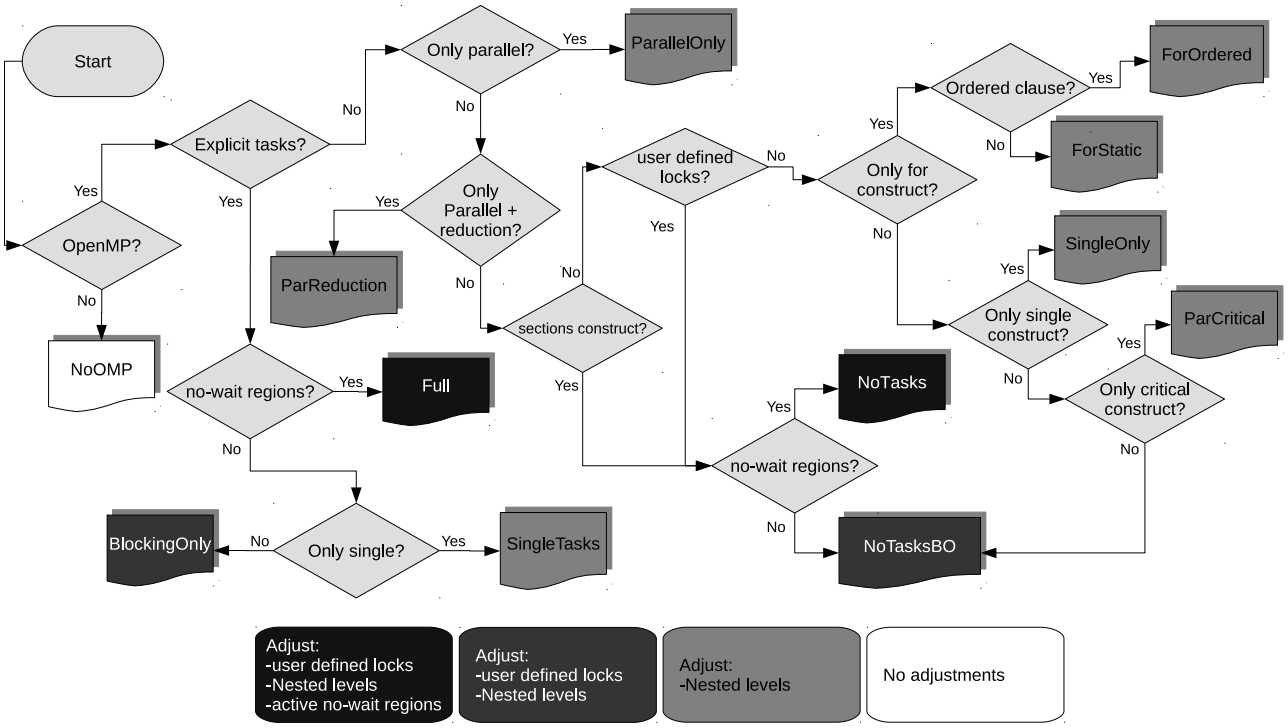

Figure 4: Mapper decision flow

mapper is designed to work with the specific device (Epiphany). This means that the mapper is aware of the characteristics of the 12 RTS flavors described in the previous section and maps the compiler metrics onto them. A different approach would be to utilize a configurable, device-agnostic mapper. Such a mapper can adapt its decision process according to the capabilities of each device. This is achievable through additional information taken from a meta-data file that describes the features of the available RTSs. The implementation of such a universal mapper would offer the advantage of providing a single mechanism that can deal with different types of devices and runtime flavors, but it is beyond the scope of this work.

The operation of our mapper can be summarized in two steps:

- During the first step, it reads the metrics generated by the compiler and decides the actual RTS flavor to be used.
- In the second step, it parametrizes (if needed) the chosen RTS and compiles its sources to provide the final binary of the library.

An overview of the decision making mechanism of the first step is presented in Fig 4. RTS (1) is chosen to accompany kernels which do not include OpenMP constructs. Based on the tasking metrics, RTSs (9), (11) or (12) are used when tasks are present; the actual choice depends on the type of worksharing regions observed. If no explicit tasks are used RTSs (2)-(8) and (10) are candidates. The decision is driven by the presence of `parallel`, `reduction` and other worksharing constructs and clauses.

6.4 Results

For our experiments we use as a reference the *Full* RTS (12) with the default parameters, and compare it with the optimized RTSs resulting from the combination of the kernel analysis and the mapper selection. The kernels were compiled with "-O3 -funroll-loops" flags and we used the `e-size` tool of the eSDK to examine the produced ELF object files.

The first set of tests included a modified version of the EPCC microbenchmark suite [9] where their basic routines are offloaded through `target` directives. These benchmarks are intended for measuring the overheads of specific constructs; in addition we utilized them to exhibit possible size benefits for the produced kernels. From the whole set, we selected the benchmarks related to `for` with `static` schedule, `critical`, `single` and `for` with the `ordered` clause.

Next, we implemented three simple applications. The first one is the scenario of a kernel which does not include any OpenMP functionality at all. In practice, this is an empty kernel containing only one assignment instruction. The second one is the iterative computation of $\pi = 3.14159$, based on the trapezoid rule with $2,000,000$ intervals, and using an OpenMP kernel which spawns a parallel team of 16 threads. The third application is a modified version of the NQueens task benchmark, taken from the Barcelona OpenMP Tasks Suite [16]. 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. We present the results for $N = 12$ queens, and a cut-off value of 2, where a total of 144 tasks are produced.

Our last experiment was the Mandelbrot deep zoom application which calculates a Mandelbrot set and zooms in and out up to $10500\times$ at six predefined points. Each image frame is written directly to the frame buffer of the Parallella board (with a resolution of 1024×768), resulting in an impressive colorful video. The full traversal generates 204 frames per zoom point. The source code for this application is provided as an example included with the eSDK in order to exhibit the real time performance possibilities of the Epiphany chip. We have parallelized it using OpenMP [6]. The kernel statically distributes the calculation among the 16 cores; each core calculates the colors for a region of the image and writes the values to the frame buffer. At the end of each frame, all cores are synchronized through an OpenMP barrier.

6.4.1 Size Results

In Table 2 we present the sizes in bytes of the resulting object files when our mechanism is employed. Each application is linked with an appropriate optimized RTS as selected by the mapper. For comparison we show the corresponding sizes without applying our mechanism (i.e. the *Full* RTS is linked with the kernels). The last column of table represents the reduction percentage with respect to *Full* RTS case. A quick glance reveals significant improvements in

Table 2: Elf sizes (bytes)

Application	Full RTS	Optimized RTS	Reduction
Empty kernel	8228	2252	72.63%
Mandelbrot	13156	9620	26.88%
Pi calculation	11972	8864	25.96%
NQueens (tasks)	20908	19704	5.76%
EPCC-for-static	14176	10944	22.80%
EPCC-critical	12560	9320	25.80%
EPCC-single	12200	8900	27.05%
EPCC-ordered	14192	10952	22.83%

all cases.

For the special case of a kernel with no OpenMP directives the mapper clearly utilized the *NoOMP* RTS, listed as (1) in Section 6.3 and the savings were almost 6 KiB, freeing precious space in local memories for the eCOREs to fit more application data. For the case of the Mandelbrot application the chosen RTS was the *ParallelOnly* one, which provides only functionalities for creating and synchronizing a parallel team. This resulted in object file smaller by 3 KiB.

The kernel for the calculation of π creates a team of eCOREs that share evenly the workload. The code utilizes the `reduction` clause to combine the partial results. Therefore, the mapper selected the *ParReduction* RTS, which resulted in a savings of 3 KiB. The NQueens application utilizes the `parallel`, `single` and `task` directives. Consequently, the *SingleTasks* runtime library was linked with the kernel. In the cases of the EPCC-based kernels the mapper employed the RTSs (4) through (7) according to kernel directives; the final result exhibits memory savings in excess of 3 KiB.

For completeness, we note that the eSDK versions of the Empty kernel and the Mandelbrot application gave object files with sizes 2248 and 4728 bytes, respectively. Obviously, one cannot compare these with what an OpenMP compiler produces, since the lower-level eSDK API lacks most of the functionality provided by OpenMP. However, we consider important the fact that when OpenMP is not utilized in a kernel of the application, OMPi does not introduce any bloat to the executable (just 6 bytes). Furthermore, the productivity benefits should be clear. For example, while the eSDK version of the Mandelbrot application required separate host and Epiphany programs with a total of 301 lines of code, the OpenMP program was written in a single file with 198 lines.

6.4.2 Timing Results

Following the size results, we compare the execution performance of the optimized kernels with that obtained when the *Full* RTS is employed. Starting with the OpenMP overheads, in

Table 3: EPCC overheads (μsec)

Kernel	Full RTS	Optimized RTS	Reduction
EPCC-for-static	72.65	19.85	72.68%
EPCC-critical	2.17	1.55	28.57%
EPCC-single	83.72	14.92	82.18%
EPCC-ordered	4.70	4.66	0.85%

Table 3 we present timing results for the EPCC microbenchmarks. As mentioned previously, we modified the original suite by having their basic routines offloaded through `target` directives. Time measurements were taken from the host side, after carefully subtracting any offloading costs. These timings, shown in microseconds, corroborate our intuition on the performance benefits of the specialized RTSs. Improvements up to 82% are observed. The noticeable cases are those of `single` and `for` with `static` schedule. The reason is mostly the optimized barrier; in contrast to the *Full* RTS, the runtimes chosen by the mapper contain barriers with no tasking extensions. We get borderline improvements in the case of `for` with an `ordered` clause, because in both scenarios the loop iterations are executed in a serial manner and the eCOREs perform their synchronization through a shared variable, stored in the (slow) shared memory.

The execution times (in sec) regarding the other applications are given in Table 4. The 0.1 sec of the empty kernel is due to the way the Parallella handles execution on the Epiphany and is a performance burden that any offloaded kernels must bare (even eSDK-based ones). Regarding the Mandelbrot application, most of the execution time is spent on actual calculations, and the OpenMP overheads constitute a rather negligible quantity. Nevertheless, the optimized RTS results offers some minimal speed gains. The same holds for the NQueens kernel. In addition, accesses to the shared memory area which stores the tasks data environments have impact to the total execution time. Finally, a significant improvement of 7% is observed in the kernel that calculates π. The reason behind this is that the optimized runtime does not support tasks. Therefore, it utilizes the lighter barrier which has no tasking extensions. In fact, the barrier flavors are the only algorithmic optimization we implemented in the various RTSs. We expect to get even better performance if other portions of the OpenMP infrastructure are written from scratch, specialized for each different RTS.

We also report that the eSDK version of the Mandelbrot application runs in 26.76 sec; it is approximately 11% faster than our version. We consider this very encouraging, considering that the original is a hand-optimized, bare-metal code, while we only have a general-purpose OpenMP infrastructure prototype which still has room for optimizations.

Table 4: Application kernels execution times (sec)

Kernel	Full RTS	Optimized RTS	Reduction
Empty Kernel	0.10	0.10	0%
Mandelbrot	30.05	30.00	0.16%
Pi calculation	0.28	0.26	7.14%
NQueens (tasks)	1.81	1.81	0%

7 Conclusions and Future Work

In this work we present a novel RTS organization that is able to produce specialized and optimized OpenMP support, tailored to the needs of each particular application. The compiler performs a detailed inter-procedural analysis of the `target` kernel regions and calculates a set of metrics depicting the kernel behavior with respect to OpenMP functionality. These metrics are fed to a mapper mechanism which decides on the most appropriate runtime library flavor to employ, and parametrize it according to the functionality requirements. As a result, each kernel offloaded to a device is accompanied by an optimized kernel-specific runtime library that is able to provide exactly the OpenMP features required.

We have implemented our ideas on the Parallella-16 board, in the context of the OMPi compiler. Our experiments show dramatic decrease in kernel sizes and execution times as compared to the original, monolithic RTS. We are currently working on three directions. First, we are optimizing the runtime library flavors even more so as to produce even smaller and faster kernels. Second, we work on implementing similar OpenMP support for other platforms. Third, we examine whether new metrics can be added to our code analysis and the potential impact they may have on further code optimization.

References

[1] GCC 5 Release Series. `https://gcc.gnu.org/gcc-5/changes.html`.

[2] OpenCL specification. `http://www.khronos.org/opencl/`.

[3] Adapteva. Epiphany SDK reference Manual. Sept. 2013.

[4] Adapteva. Parallella Reference Manual, Sept. 2014.

[5] Spiros N. Agathos, Vassilios V. Dimakopoulos, Aggelos Mourelis, and Alexandros Papadogiannakis. Deploying OpenMP on an embedded multicore accelerator. In *Proc. of SAMOS'13, 13th International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation*, pages 180–187, Samos, Greece, Jul. 2013.

[6] Spiros N. Agathos, Alexandros Papadogiannakis, and Vassilios V. Dimakopoulos. Targeting the Parallella. In *Proc. of Euro-Par 2015, 21st International European Conference on Parallel and Distributed Computing*, pages 662–674, Vienna, Austria, 2015.

[7] Luca Benini, Eric Flamand, Didier Fuin, and Diego Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Proc. of DATE'12, Design Automation and Test in Europe*, pages 983–987, Dresden, Germany, Mar. 2012.

[8] Carlo Bertolli, Samuel F. Antao, Alexandre E. Eichenberger, Kevin O'Brien, Zehra Sura, Arpith C. Jacob, Tong Chen, and Olivier Sallenave. Coordinating GPU Threads for OpenMP 4.0 in LLVM. In *Proc. of LLVM-HPC '14, LLVM Compiler Infrastructure in HPC*, pages 12–21, New Orleans, Louisiana, Nov. 2014.

[9] Mark J. Bull. Measuring Synchronisation and Scheduling Overheads in OpenMP. In *Proc. EWOMP '99, the 1st European Workshop on OpenMP*, pages 99–105, Lund, Sweden, Sept. 1999.

[10] Paolo Burgio, Giuseppe Tagliavini, Andrea Marongiu, and Luca Benini. Enabling Fine-Grained OpenMP Tasking on Tightly-Coupled Shared Memory Clusters. In *Proc. of DATE 13, Design Automation and Test in Europe*, Grenoble, France, Mar. 2013.

[11] Daniel Cabrera, Xavier Martorell, Georgi Gaydadjiev, Eduard Ayguadé, and Daniel Jiménez-González. OpenMP extensions for FPGA accelerators. In *Proc. of SAMOS'09, 9th International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation*, pages 17–24, Samos, Greece, Jul. 2009.

[12] Paul Carpenter, David Rodenas, Xavier Martorell, Alex Ramirez, and Eduard Ayguadé. A streaming machine description and programming model. In *Proc of SAMOS '07, 7th International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation*, pages 107–116, Samos, Greece, Jul. 2007.

[13] Barbara Chapman, Lei Huang, Eric Biscondi, Eric Stotzer, Ashish Shrivastava, and Alan Gatherer. Implementing OpenMP on a high performance embedded multicore MPSoC. In *Proc. of IPDPS '09, IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, Rome, Italy, May 2009.

[14] Liao Chunhua, Yan Yonghong, Bronis R. de Supinski, Daniel J. Quinlan, and Barbara M. Chapman. Early Experiences with the OpenMP Accelerator Model. In *Proc. of IWOMP 2013, 9th International Workshop on OpenMP, OpenMP in the Era of Low Power Devices and Accelerators*, pages 84–98, Canberra, Australia, Sept. 2013.

[15] Vassilios V. Dimakopoulos, Elias Leontiadis, and George Tzoumas. A portable C compiler for OpenMP V.2.0. In *Proc. EWOMP 2003, the 5th European Workshop on OpenMP*, pages 5–11, Aachen, Germany, Sept. 2003.

[16] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguadé. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *Proc. of ICPP '09*, pages 124–131, Vienna, Austria, Sept. 2009.

[17] Feng Liu and Vipin Chaudhary. Extending OpenMP for Heterogeneous Chip Multiprocessors. In *Proc. of ICPP 2003, 32nd International Conference on Parallel Processing*, pages 161–, Kaohsiung, Taiwan, Oct. 2003.

[18] Daniel González, Eduard Ayguadé, Xavier Martorell, and Jesus Labarta. Exploiting pipelined executions in OpenMP. In *Proc. of ICPP 2003, 32nd International Conference on Parallel Processing*, pages 153–160, Kaohsiung, Taiwan, Oct. 2003.

[19] Toshihiro Hanawa, Mitsuhisa Sato, Jinpil Lee, Takayuki Imada, Hideaki Kimura, and Taisuke Boku. Evaluation of multicore processors for embedded systems by parallel benchmark program using openmp. In *Proc. of IWOMP '09, 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, pages 15–27, Dresden, Germany, June 2009.

[20] Intel Corporation. User and Reference Guide for the Intel C++ Compiler 15.0, OpenMP* Support.

[21] Woo-Chul Jeun and Soonhoi Ha. Effective OpenMP Implementation and Translation For Multiprocessor System-On-Chip without Using OS. In *Proc. of ASP-DAC '07, 12th Asia and South Pacific Design Automation Conference*, pages 44–49, Yokohama, Japan, Jan. 2007.

[22] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors, Second Edition: A Hands-on Approach.* Morgan Kaufmann, MA 01803, USA, Dec. 2012.

[23] Feng Liu and Vipin Chaudhary. A Practical OpenMP Compiler for System on Chips. In *Proc. of WOMPAT 2003, Workshop on OpenMP Applications and Tools*, volume 2716, pages 54–68, Torondo, Canada, June 2003.

[24] Gaurav Mitra, Eric Stotzer, Ajay Jayaraj, and Alistair P. Rendell. Implementation and Optimization of the OpenMP Accelerator Model for the TI Keystone II Architecture. In *Proc. of IWOMP 2014, the 10th International Workshop on OpenMP, Using and Improving OpenMP for Devices, Tasks, and More*, pages 202–214. Salvador, Brazil, Sept. 2014.

[25] Mitsuhisa Sato Yoshihiro Nakajima Yoshinori Ojima Yoshihiko Hotta. OpenMP Implementation and Performance on Embedded Renesas M32R Chip Multiprocessor. In *Proc. of EWOMP '04, 6th European Workshop on OpenMP*, pages 37–42, Oct. 2004.

[26] Chris J. Newburn, Rajiv Deodhar, Serguei Dmitriev, Ravi Murty, Ravi Narayanaswamy, John Wiegert, Francisco Chinchilla, and Russell McGuire. Offload Compiler Runtime for the Intel Xeon Phi^{TM}Coprocessor. In *Proc. of IPDPS Workshops, 27th IEEE International Parallel and Distributed Processing Symposium*, pages 1213–1225. Boston, USA, May. 2013.

[27] OpenMP ARB. OpenMP Application Program Interface V3.1, Jul. 2011.

[28] OpenMP ARB. OpenMP Application Program Interface V4.0, Jul. 2013.

[29] Alexandros Papadogiannakis, Spiros N. Agathos, and Vassilios V. Dimakopoulos. OpenMP 4.0 Device Support in the OMPi Compiler. In *Proc of IWOMP 2015, 11th International Workshop on OpenMP, Heterogenous Execution and Data Movements*, pages 202–216, Aachen, Germany, Oct. 2015.

[30] G.C. Philos, V.V. Dimakopoulos, and P.E. Hadjidoukas. A Runtime Architecture for Ubiquitous Support of OpenMP. In *Proc. ISPDC 2008, 7th international Symposium on Parallel and Distributed Computing*, pages 189–196, Krakow, Poland, June 2008.

[31] G. Zhang, R. Silvera, and R. Archambault. Structure and algorithm for implementing OpenMP workshares. In *Proc. of WOMPAT '04, 5th Workshop on OpenMP Applications and Tools*, Houston, TX, USA, May 2004.