Adaptive OpenMP Runtime System for Embedded Multicores

Spiros N. Agathos
Swarm64 AS Zweigstelle Hive
Ullsteinstrasse 120, Neubau Turm C
Berlin, Germany
agathosspiros@gmail.com

Vassilios V. Dimakopoulos
Dept. of Computer Science and Engineering
University of Ioannina
Ioannina, Greece
dimako@cse.uoi.gr

Abstract

The recent OpenMP device constructs enable application writers to utilize the host CPUs along with other attached computational resources, in an intuitive and productive manner. These `target`-related directives offload portions of the program code (kernels) to any of the available computational 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 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 for embedded multicores. We propose a compiler-assisted, adaptive runtime system organization, which generates application-specific support by implementing only the OpenMP functionality required each time. Full OpenMP support is 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. Our proposal is demonstrated by a complete implementation on the popular Parallella board.

1 Introduction

Contemporary systems, from plain PCs to high-performance supercomputers are heterogeneous in nature, including a mix of different processor and memory hierarchies within the same system. Responding to the ever increasing demand of end-user applications for computational power and multitasking, embedded systems have also joined the heterogeneous paradigm trend. However, in order to exploit the computation capabilities of a heterogeneous system efficiently, significant programmer effort is required. The common case is to use low-level SDKs in order to optimize portions of an application with respect to the specific hardware unit features. This poses significant challenges, even for expert programmers. Moreover, requiring different code bases for the host CPU and the accelerator devices increases code complexity and decreases portability.

Recently, OpenMP 4 [20], the de facto parallel programming model for shared memory systems, 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 parts in a unified and seamless way.

The new device extensions allow full OpenMP functionality within the regions of code executed by a selected device (also known as *kernels*). This 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 on 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 or 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 runtime system (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. Our technique is quite general and can be also utilized in the OpenMP runtime system executing on the host.

To the best of our knowledge this is the first time an adaptive, application-specific OpenMP runtime system is proposed. As such, we also present a concrete implementation of our ideas on the popular Parallella-16 board, a credit-card sized computer with two processors (a dual-core host and a 16-core accelerator). The OMPi OpenMP compiler [10] infrastructure was modified to analyze the kernels code and to generate optimized runtime library versions according to the results of the analysis. Our experimentation with a plethora of

application codes confirms the benefits of our strategy, which in some cases resulted in executable size reductions of more than 30%.

1.1 Related Work

OpenMP was considered as a possible model for accelerators or multicore embedded systems long before the introduction of its recent device extensions [6, 8, 16, 17, 21]. All these works refer to supporting older versions of OpenMP on a single processor. That is, the multicore CPU plays the role of the *host* in OpenMP 4 terminology and as such, they do not address the heterogeneous host/device execution model.

In order to provide a unified model for systems consisting of a host and a set of attached devices, extensions to OpenMP were proposed before the release of OpenMP 4. Cabrera et al in [7] propose extensions to provide a high level API for executing code on heterogeneous systems with FPGA-based accelerators. In [2] 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, providing an interface similar to the device model of OpenMP 4 for offloading and executing OpenMP kernels on the MPSoC. Other directive-based approaches for offloading code onto attached devices include HMPP [11] and OMPSs [13]. It should be noted that in all these works, with the exception of [2], the offloaded portions of the code did not contain any OpenMP functionality.

Support for OpenMP 4.0 devices is fairly limited both in the compiler and the device sides. Details of the offload procedure in the Intel ICC compiler are given in [19]. Preliminary support for the OpenMP `target` construct is also available in the ROSE compiler. Chunhua et al [9] discuss their experiences on implementing a prototype called HOMP on top of the ROSE compiler, which generates code for CUDA devices. Bertolli et al [4] 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 [18] 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, Agathos et al in [3] present the first implementation of the OpenMP 4.0 accelerator directives for the Parallella board [1], a credit-card sized multicore system consisting of a dual-core ARM host processor and a 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, based on compiler instrumentation.

2 The OpenMP Device Model

One of the key features of version 4.0 of the OpenMP API [20] is the introduction of a state-of-the-art, platform-agnostic model for heterogeneous parallel programming. 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. 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. A device data environment can be manipulated through `map` clauses which determine how the specified variables are handled within the data environment (`alloc`, `from`, `to` and `tofrom` map types).

Data movements between the host and the devices may be the cause of 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. 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 environments. 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.

2.1 OpenMP on the Device Side

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 of the OpenMP specifications can potentially form a legal kernel. Thus, the constructs for dynamically creating a team of threads, 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 easily offload some of its computationally intensive parts onto specialized hardware.

To make all the above possible, the attached devices are effectively required to provide complete OpenMP support. However, OpenMP was originally designed for shared memory multiprocessors, i.e. systems with abundant resources

(memory, caches, OS). 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 aims at streaming applications or may be better suited to speed up matrix-based computations. Co-processors are synonymous to hardware diversity, since each product is equipped with specialized hardware modules and targets a specific class of applications.

With some notable exceptions such as the Xeon Phi accelerator [19], 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. 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 [1] 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 practical situations [8,9, 18]. For example, there is no point in trying to implement an optimized tasking infrastructure for a GPGPU which lacks fine grain synchronization primitives. 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 [14, 15] 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 [3, 4, 9].

3 Proposed System

In this work we propose a general methodology which can be utilized to offer flexible and adaptive OpenMP runtime support. 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

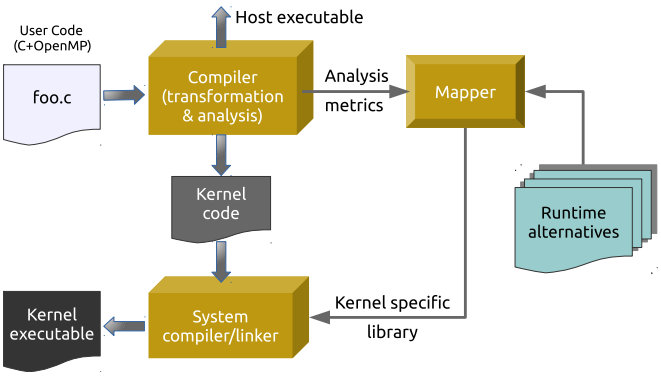

Figure 1: Compilation system for adaptive, kernel-specific OpenMP RTSs

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 interprocedural 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* 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.

3.1 Kernel Analysis

An OpenMP kernel is 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 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. For example, kernel analysis may reveal that there are no `task` constructs utilized, thus there is no need for the RTS to provide a mechanism for the support of explicit tasking.

3.2 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". The RTS library which is to be linked with the kernel code consists of some RTS-specific data along with the code implementing the required functionalities. The internal data are related to:

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

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

Implementing a general, full fledged RTS which is capable of offering OpenMP support is a 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 the particular application.

Possible realizations of specialized RTS libraries include:

- A fixed set of *pre-compiled* libraries. The set of libraries is selected to address 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, the barrier data structures can be tuned to service a specific number of threads, if this information is known at compile time. Of course, parametrization requires recompiling and thus custom libraries need to be built at the compile-time of the application.

4 Implementation in the OMPi Compiler

The OMPi compiler [10] 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 OpenMP V3.1 specifications, while also supporting a number of V4.0/V4.5 constructs including the `target`-related device ones.

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

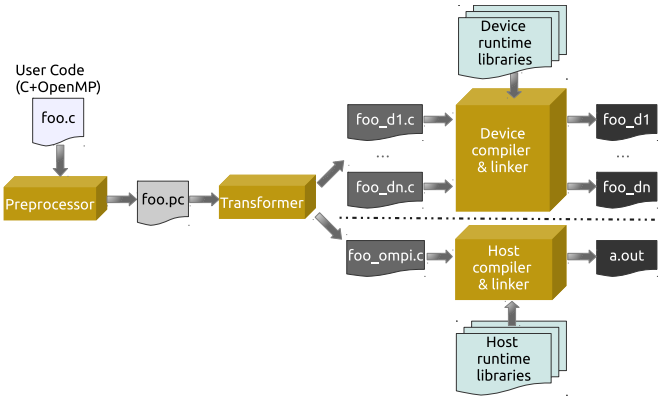

Figure 2: OMPi compilation chain

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. To implement the proposed mechanism, this last stage is where the mapper module is inserted. The intermediate files must carry the deduced metrics so as to guide the mapper. Finally, we modified the compiler and equipped it with new kernel analysis capabilities in order to derive the desired metrics.

4.1 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 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 `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.

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 is 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 passing them to the mapper.

4.2 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 CPU, 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 CPU which is used as a 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 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 cores are possible. In Epiphany-III the chip is pinned on a 4×4 submesh of the virtual 64×64 mesh whose northwest coordinates are (32, 8), as shown in Fig. 3. The chip has four eLinks that may be used 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 Epiphany core (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.

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 [3]. It 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 two parts 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.

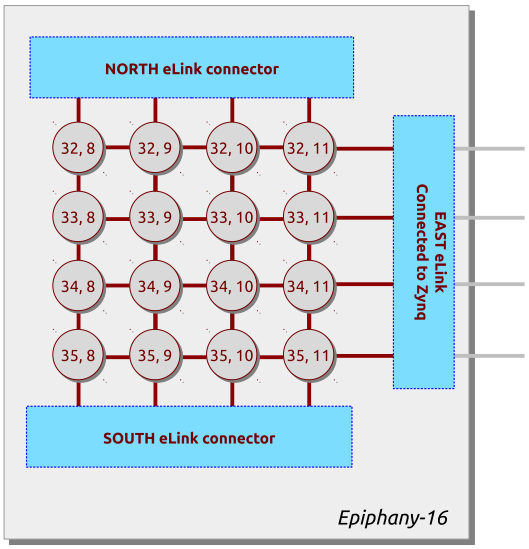

Figure 3: The Epiphany co-processor

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 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 largely independent routines so that the system linker can attach only the necessary ones with each kernel. However, the complex relations between the internal data structures and the routines usually 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 [3]. Furthermore, because dynamic memory allocation is not supported at the eCORE level, the RTS must reserve in advance enough local space to cover the worst case. Consequently, the actual local memory left for pure application data is well below the 32 KiB available.

4.3 Runtime Flavors

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:

- *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 becomes more than doubled if a second level of parallelism is to be supported. Supporting more than two levels is pointless.
- *Worksharing*. The OpenMP worksharing constructs (`single`, `for`, `sections`) may have different combinations of `nowait`, `reduction`, `schedule`, `collapse` and `ordered` clauses. Supporting all of them requires data structures with a large memory footprint. In practice, typical applications do not utilize all possible variations. As a result, supporting specific combinations of the above constructs and clauses may potentially benefit some kernel cases.
- *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.

Based on the above analysis of the original RTS, we designed and implemented manually 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 RTSs is as follows:

(1) *NoOMP*. This RTS does not support any OpenMP directives within the kernel; eCOREs execute 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 is an extension of the previous one, and implements the `reduction` clause.

(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 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 for kernels with no explicit tasks. The rest of the OpenMP functionality (e.g. worksharing, synchronization, etc) is present.

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

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

(11) *TasksNoICVs*. This RTS provides support for teams of eCOREs that can create explicit tasks. It is assumed that per-task ICVs are kept unmodified and thus can be omitted from the task descriptor of all but the initial task.

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

The above set does not cover all possible use cases, i.e. it 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 for all possible kernels.

In all cases, the RTS routines were carefully re-implemented to offer only the required support. Barrier routines constitute a characteristic 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*, *TasksNoICVs* and *Full* there is no tasking support and consequently barriers were simplified to handle only thread synchronization.

The mapper imports the set of metrics provided by the compiler and uses them in order to choose the most appropriate RTS flavor to be linked with a kernel. The mapper is designed to work exclusively with the specific device (Epiphany-III). 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. The mapper operation can be summarized in two steps: First, it reads the metrics generated by the compiler and decides the RTS flavor to be used; then, it parametrizes (if needed) the chosen RTS and compiles its sources to provide the final binary of the library.

Here is an overview of the decision making mechanism of the first step: 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 worksharing constructs.

5 Evaluation

To evaluate our proposed method, we used the Parallella-16 SKU-A101020 board, which comes with standard peripheral ports such as USB, Ethernet, HDMI, GPIO, etc. and is equipped with a dual-core ARM Cortex A9, which is the *host* and an Epiphany-III 16-core co-processor, considered as our *device*. All common programming tools are available for the ARM host processor. For the Epiphany, a Software Development Kit is available (eSDK), which includes a C compiler and runtime libraries for both the host (eHAL) and the co-processor (eLIB). We used eSDK v5.13.9.10 which includes the GCC and E-GCC compilers for the host and the Epiphany executables respectively.

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` eSDK tool to examine the produced ELF object files.

The first set of tests included a modified version of the EPCC microbenchmark suite [5] where their basic routines are offloaded through `target` directives. These benchmarks are intended for measuring the overheads of specific constructs; we utilized them to exhibit possible size benefits for the produced kernels. We present a representative sample of results pertaining to the following benchmarks: `barrier`, `for` with `static` schedule, `critical`, `single`, `for` with the `ordered` clause, and locks.

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 [12]. 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 experimentation concluded with two more complex applications. The first one is the well-known Conway's Game of Life which is one of the available Parallella code examples. The original code is rather simplistic and refers to a 4×4 field of cells. We implemented a more sophisticated version, which is based on an $16 \times C$ field, parallelized with OpenMP. The program code is offloaded as a `target` region, with the initial field array residing in shared memory. Each core starts by bringing its assigned field row along with the next and the previous one, to its local memory for speed. From then on, it operates exclusively on local data. The value of C (the number of columns) depends on the available local space. For the *Full* RTS we were able to fit fields with $C = 184$ columns which is the value used in our experiments. However, it is worth noting that from the size reductions possible when optimized runtimes are employed, we managed to experiment with fields of up to $C = 950$ columns.

Table 1: Kernel analysis

Application	Computed metrics
Mandelbrot	$N_p = 1, N_b = 1, L_p = 1$
Pi calculation	$N_p = 1, N_{red} = 1, L_p = 1$
Game of Life	$N_p = 1, N_f = 1, N_b = 4, L_p = 1$
NQueens	$N_p = 1, N_t = 1, L_p = 1$

Table 2: Elf sizes (bytes)

Application	Full RTS	Optimized RTS		Reduction
Empty kernel	8648	2252	(*NoOMP*)	73.96%
Mandelbrot	13724	9620	(*ParallelOnly*)	29.90%
Pi calculation	12744	8864	(*ParReduction*)	30.45%
Game of Life	15412	11320	(*ForStatic*)	26.55%
NQueens	20908	19148	(*TasksNoICVs*)	8.42%
EPCC-barrier	12316	8268	(*ParallelOnly*)	32.87%
EPCC-for-static	14744	10992	(*ForStatic*)	25.45%
EPCC-critical	13184	9420	(*ParCritical*)	28.55%
EPCC-single	12768	8944	(*SingleOnly*)	29.95%
EPCC-ordered	14704	10992	(*ForOrdered*)	25.24%
EPCC-locks	12932	8716	(*ParallelOnly*)	32.60%

Finally, we consider the Mandelbrot deep zoom application which calculates a Mandelbrot set and zooms in and out up to $10500\times$ at six predefined points, generating 204 frames per zoom point. The source code for this application is provided as a performance exhibition example included with the eSDK. We have parallelized it using OpenMP [3].

We summarize the kernel metrics reported by the compiler for the 4 applications in Table 1. Each EPCC microbenchmark contains 1 `parallel` region ($N_p = 1$) with an additional OpenMP construct (which the microbenchmark measures) and a single level of parallelism ($L_p = 1$). The `ordered` one contains an additional `for` region ($N_f = 1$).

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 represents the reduction percentage with respect to *Full* RTS. A quick glance reveals significant improvements in 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 4.3 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 only the `parallel` and `task` directives. In addition, no OpenMP internal control variables are modified in the user code. Con-

sequently, the *TasksNoICVs* runtime library was linked with the kernel. For the EPCC-based kernels the mapper employed the RTSs (2) and (4) through (7) according to kernel directives; the final result exhibits 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 requires separate host and Epiphany programs with a total of 301 lines of code, the OpenMP program lies in a single file with 198 lines.

Due to limited room, we only focus on size results here. However, we also note that the different RTSs result in different OpenMP construct overheads which in turn manifest themselves in different kernel execution times. In particular, we have measured $\approx 7\%$ reduction in execution time for the π calculation kernel and an impressive $\approx 69\%$ reduction for the Game of Life as compared to the *Full* RTS.

6 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 as compared to the original, monolithic RTS. We are currently optimizing the runtime library flavors so as to produce even smaller and faster kernels. while also we examine whether new metrics can be added to our code analysis. We are also generalizing our idea to provide application-specific runtime support for the main (host) part of traditional OpenMP programs.

References

[1] Adapteva, *Parallella Reference Manual*, Sept. 2014.

[2] S. N. Agathos, V. V. Dimakopoulos, A. Mourelis, and A. Papadogiannakis, "Deploying OpenMP on an embedded multicore accelerator," in *Proc. of SAMOS'13, 13th Int'l Conf. on Embedded Computer Systems: Architectures, MOdeling and Simulation*, Samos, Greece, Jul. 2013, pp. 180–187.

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

[4] C. Bertolli et al, "Coordinating GPU Threads for OpenMP 4.0 in LLVM," in *Proc. of LLVM-HPC '14, LLVM Compiler Infrastructure in HPC*, New Orleans, Louisiana, Nov. 2014, pp. 12–21.

[5] M. J. Bull, "Measuring Synchronisation and Scheduling Overheads in OpenMP," in *Proc. EWOMP '99, the 1st European Workshop on OpenMP*, Lund, Sweden, Sept. 1999, pp. 99–105.

[6] P. Burgio, G. Tagliavini, A. Marongiu, and L. 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.

[7] D. Cabrera, X. Martorell, G. Gaydadjiev, E. Ayguadé, and D. Jiménez-González, "OpenMP extensions for FPGA accelerators," in *Proc. of SAMOS'09, 9th Int'l Conf. on Embedded Computer Systems: Architectures, MOdeling and Simulation*, Samos, Greece, Jul. 2009, pp. 17–24.

[8] B. Chapman et al, "Implementing OpenMP on a high performance embedded multicore MPSoC," in *Proc. of IPDPS '09, IEEE Int'l Symposium on Parallel and Distributed Processing*, Rome, Italy, May 2009, pp. 1–8.

[9] L. Chunhua, Y. Yonghong, B. R. de Supinski, D. J. Quinlan, and B. M. Chapman, "Early Experiences with the OpenMP Accelerator Model." in *Proc. of IWOMP 2013, 9th Int'l Workshop on OpenMP*, Canberra, Australia, Sept. 2013, pp. 84–98.

[10] V. V. Dimakopoulos, E. Leontiadis, and G. Tzoumas, "A portable C compiler for OpenMP V.2.0," in *Proc. of EWOMP 2003, the 5th European Workshop on OpenMP*, Aachen, Germany, Sept. 2003, pp. 5–11.

[11] R. Dolbeau, S. Bihan, and F. Bodin, "HMPP: A Hybrid Multi-core Parallel Programming Environment," in *Proc. GPGPU 2007, Workshop on General Purpose Processing Using GPUs*, Boston, USA, Oct. 2007.

[12] 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 *Proc. of ICPP '09*, Vienna, Austria, Sept. 2009, pp. 124–131.

[13] A. Fernández et al, "Task-Based Programming with OmpSs and Its Application," in *Euro-Par 2014 International Workshop, Revised Selected Papers, Part II*, Porto, Portugal, Aug 2014, pp. 602–613.

[14] Free Software Foundation, "GCC 5 Release Series," https://gcc.gnu.org/gcc-5/changes.html.

[15] Intel Corporation, "User and Reference Guide for the Intel C++ Compiler 15.0, OpenMP* Support." [Online]. Available: https://software.intel.com/en-us/node/522679

[16] W.-C. Jeun and S. 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 Conf.*, Yokohama, Japan, Jan. 2007, pp. 44–49.

[17] F. Liu and V. Chaudhary, "A Practical OpenMP Compiler for System on Chips," in *Proc. of WOMPAT 2003, Workshop on OpenMP Applications and Tools*, vol. 2716, Torondo, Canada, June 2003, pp. 54–68.

[18] G. Mitra, E. Stotzer, A. Jayaraj, and A. P. Rendell, "Implementation and Optimization of the OpenMP Accelerator Model for the TI Keystone II Architecture," in *Proc. of IWOMP 2014, the 10th Int'l Workshop on OpenMP*, Salvador, Brazil, Sept. 2014, pp. 202–214.

[19] C. J. Newburn et al, "Offload Compiler Runtime for the Intel Xeon Phi Coprocessor," in *Proc. of IPDPS Workshops, 27th IEEE Int'l Parallel and Distributed Processing Symposium*, Boston, USA, May. 2013, pp. 1213–1225.

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

[21] M. Sato, Y. Nakajima, Y. Ojima, and Y. Hotta, "OpenMP Implementation and Performance on Embedded Renesas M32R Chip Multiprocessor," in *Proc. of EWOMP '04, 6th European Workshop on OpenMP*, Oct. 2004, pp. 37–42.