The original publication is available in Elsevier's Parallel Computing, DOI:10.1016/j.parco.2022.102895


# Compiler-Assisted, Adaptive Runtime System for the Support of OpenMP in Embedded Multicores

Spiros N. Agathos[a], Vassilios V. Dimakopoulos[b,*], Ilias K. Kasmeridis[b]

[a]*Swarm64 AS Zweigstelle Hive, Ullsteinstrasse 120, Neubau Turm C, Berlin, Germany*
[b]*Dept. of Computer Science and Engineering, University of Ioannina, Ioannina, Greece*

**Abstract**

The latest versions of OpenMP have introduced constructs for exploiting heterogeneous compute units alongside the main multicore CPU. The offloaded program portions (kernels) may generate parallelism within the target device by employing standard OpenMP constructs. However co-processors, especially embedded ones, often have limited resources to provide efficient OpenMP support. Designing an OpenMP infrastructure for such devices is a challenge and a usual design decision is to support OpenMP only partially.

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. 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, which usually offer additional performance benefits. The mechanism is based on preparatory compile-time kernel analysis which generates metrics regarding the OpenMP functionality present in each kernel. These are then fed to a mapper module which, given a set of rules, decides what the optimal runtime configuration is. Our proposal is demonstrated by a complete implementation on the popular Parallella-16 board, exhibiting consistently large size savings and significant performance gains.

*Keywords:* compilers, embedded systems, multicores, OpenMP, offloading, runtime systems

## 1. Introduction

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

Embedded systems provide computing capabilities to devices that are small, portable, autonomous, while 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 paradigm trend. For example, Parallella [1] combines a small number of ARM-based cores with a many-core coprocessor of up to 64 RISC cores. Another design example is the NVidia Jetson Xavier platform which consists of 8 ARM cores and 512-core Volta GPU and has been considered for automotive applications [2].

However, in order to exploit the computation capabilities of a heterogeneous system efficiently, significant programmer effort is required. The common case is to utilize 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. In the same line, programming models such as OpenCL [3] and CUDA [4] provide very efficient albeit rather primitive mechanisms for an application to take advantage of the hardware capabilities of GPUs and general purpose accelerators. In addition, requiring different code bases for the host CPU and the accelerator devices increases code complexity and decreases portability.

One of the biggest challenges in the current 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 an effective 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. Starting with version 4.0, OpenMP [5] 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

*Corresponding author
*Email addresses:* agathosspiros@gmail.com (Spiros N. Agathos), dimako@cse.uoi.gr (Vassilios V. Dimakopoulos), ikasmeridis@cse.uoi.gr (Ilias K. Kasmeridis)

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. This way the user's code implies the choice of an appropriately optimized RTS which may result in reduced executable sizes and/or faster execution times. Our technique is quite general and could 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. As a testbed platform, we use 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 [6] infrastructure was modified to analyze the kernels code and to select optimized runtime library versions according to the results of the analysis. Our experimentation with a plethora of application codes verified the benefits of our strategy, which in some cases resulted in more than 30% size and 90% execution time reductions.

The remainder of the paper is organized as follows: In Section 2 we give an overview of related work as well as the motivation behind this work. In Section 3 we present some background material on OpenMP and the extensions for device support; we also discuss the difficulties induced when developing an OpenMP RTS for devices. An overview of our adaptive RTS proposal is presented in Section 4. A concrete implementation is presented in Section 5. In particular Sections 5.1 and 5.2 present the kernel analysis procedure that produces a set of metrics and the way these metrics are utilized by the "mapper" module that provides the best RTS configuration. We then describe the prototype implementation of our methodology for the Parallella-16 board (Section 6) and evaluate the space and time efficiency of our proposal (Section 7). Section 8 summarizes and concludes this work.

## 2. Related Work and Motivation

OpenMP was considered as a possible model for accelerators or multicore embedded systems long before the introduction of its latest device extensions. In [7] Hanawa et al. evaluate the OpenMP model for the Renesas M32700, ARM/NEC MPCore, and Waseda University RP1 multicore embedded systems. Sato, Nakajima, Ojima and Hotta [8] implement OpenMP and report its performance on a dual M32R processor, which runs Linux and supports fully the POSIX execution model. Liu and Chaudhary [9] implement an OpenMP compiler for the 3SoC Cradle system, a system combining multiple RISC and DSP-like cores. In [10] Woo-Chul and Soonhoi discuss an OpenMP implementation that targets MPSoCs with physically shared memories, hardware semaphores, and no operating system. Chapman et al. [11] 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 [12] Burgio, Tagliavini, Marongiu and Benini present an OpenMP task implementation for a simulated embedded multicore platform inspired by the STHORM architecture. Notice that all the above works refer to older versions of OpenMP on a single processor. That is, the multicore CPU plays the role of the *host* in recent OpenMP 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 V4.0. Cabrera, Martorell, Gaydadjiev, Ayguadé and Jiménez-González in [13] propose extensions to provide a high level API for executing code on heterogeneous systems with FPGA-based accelerators. They provide constructs for offloading tasks to any of the available accelerators, along with runtime optimizations which try to hide the FPGA configuration time needed when a bitstream has to be loaded. In [14] Agathos, Dimakopoulos, Mourelis and Papadogiannakis 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 (device) sides in a seamless way, providing an interface similar to the current device model of OpenMP for offloading and executing OpenMP kernels on the MPSoC. Other directive-based approaches for offloading code onto attached devices include HMPP [15] and OmpSs [16]. It should be noted that in all these works, with the exception of [14], the offloaded portions of the code did not contain any OpenMP functionality.

OpenMP offloading is considered a significant high-level programming abstraction for heterogeneous systems. Sommer, Stock, Soliz-Vasquez and Kock [17] evaluate OpenMP, OpenCL

and CUDA and conclude that the high-level abstractions defined by the OpenMP standard allow for a very good programmer productivity, maintainability and portability while yielding competitive performance in benchmarks targeting heterogeneous embedded systems, with a bias towards automotive industry codes. OpenMP V4.5 is the main programming model in the software stack of the HERO heterogeneous embedded platform which is based on a hard ARM Cortex-A multicore host processor and RISC-V based accelerator cores, implemented as soft cores on an FPGA fabric [18]. OpenMP offloading constructs for heterogeneous systems are considered as an extension for the next version of the OmpSs model [19].

While OpenMP specifications have recently reached V5.1, device support remains limited; the number of compilers adhering to the standard (in various degrees) increases slowly [20], but unfortunately there exists support for relatively few device types. Details of the offload procedure in the Intel ICC compiler are given in [21]. Preliminary support for the OpenMP `target` construct is also available in the ROSE compiler. Chunhua, Yonghong, de Supinski, Quinlan and Chapman [22] discuss their experiences on implementing a prototype called HOMP on top of the ROSE compiler, which generates code for CUDA devices. Bertolli et al. [23] 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 V4.0. In [24] the authors present their implementation of OpenMP V4.0 on a TI Keystone II, where they use the DSP cores as devices to offload code to. Finally, Agathos, Papadogiannakis and Dimakopoulos in [25] present the implementation of the OpenMP accelerator directives for the Parallella-16 board [1], a credit-card sized multicore system consisting of a dual-core ARM host processor and a 16-core Epiphany coprocessor. All 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.

### *2.1. Motivation*

The motivation behind this work comes from [25], which presents an OpenMP infrastructure for the Parallella-16 board where the Epiphany accelerator is treated as an OpenMP device. Supporting OpenMP on the device side entails a major effort due to the constraints and limited resources of the Epiphany chip. A full OpenMP runtime library was carefully designed but due to the sheer volume of OpenMP features and their inherent complexity, compromises were necessary at every level. For example, the implementation of tasking was necessarily as simple as possible; it was based on a single shared queue so as to minimize memory requirements and keep the queue in the fast local memory of the accelerator cores. As a result, it may not be the most performant configuration in high loads, while at the same time it deprives other OpenMP data structures of precious space. More details are given in Section 6.

One of our observations is that the majority of the kernels we have run on the board have been loop-based; they do not have any use for tasks. If *all* kernels were loop-based, there would be no need to implement tasking at all. Instead one would be able to utilize the memory space e.g. for supporting larger application datasets. However this is entirely hypothetical, since some applications do utilize tasking. To tackle this issue, in the course of this work we conceived a rather unorthodox approach: create two different OpenMP runtime libraries for the device, one that supports tasking and another one that does not; the latter has a much smaller footprint, leaving more space for application data. Generalizing the idea, we should be able to implement any number of different runtime library versions which provide different sets of OpenMP facilities.

Given such a set of libraries, the next problem is how to choose which library version to use each time. Clearly, the choice must be made by observing what OpenMP constructs a kernel uses and then selecting the most appropriate library version to accommodate them. In the next sections, we present the details behind the concepts as well as the way they can be implemented for any device, using static code analysis and a general selector mechanism.

## 3. The OpenMP Device Model

One of the key features in the latest versions of the OpenMP API [5] is the introduction of a state-of-the-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 which 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).

Fig. 1 shows a simple example where host function `dev_add` can be called to perform array addition at a device. The `target` region is executed on the device. Host variables `x`, `y` and `n` get transferred to the device due to the `map(to:)` clause while `res` will obtain its value from the device because of the `map(`

```
/* Host function that offloads to device */
void dev_vadd(int x[], int y[], int res[], int n)
{
  #pragma omp target map(to: x[0:n], y[0:n], n)\
                     map(from: res[0:n])
  {
    int i;
    #pragma omp parallel for
    for (i = 0; i < n; i++)
      res[i] = x[i] + y[i];
  }
}
```

Figure 1: Adding two vectors on a device.

`from:)` clause. The `[0:n]` notation specifies that only a section of `n` elements of the vectors are actually mapped, starting form element 0. The array addition gets offloaded and executed in parallel at the device, because of the `parallel for` OpenMP directive, utilizing its processing cores. When the calculation is completed, the value of `res` gets copied from the device to the host memory. Actual data transfers may not be needed if the host and the device share memory.

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.

### *3.1. 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. 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, name space, or class scope. 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 an 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 such as large amounts of shared memory, supported by caches and sophisticated cache coherency protocols, high bandwidth interconnects and a large set of hardware-assisted synchronization primitives. Moreover, they are equipped with an operating system accompanied with optimized low-level 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 aims at streaming applications or may be better suited to speed up matrix-based computations. Co-processors are synonymous to hardware diversity; 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 [21], a common characteristic of the various types of co-processors is that they offer a limited amount of resources. Hence, the challenges posed when designing an OpenMP runtime support system (RTS) for such devices depend on these resource limitations. The typical absence of a POSIX-like interface for manipulating threads may add design difficulties or considerable offloading costs regarding dynamic or nested parallelism. As is evident in [26, 27], supporting efficiently just the tasking primitives of OpenMP at the device side of an embedded accelerator is a major undertaking. 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 on 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-16 [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 [24, 22, 11, 26, 27]. 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 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

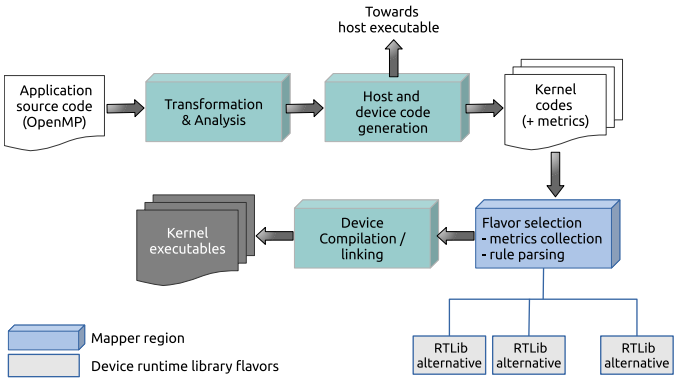


Figure 2: Overview of the proposed system for adaptive, kernel-specific OpenMP RTSs

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 [25, 23, 22].

## 4. Proposed System

In this work we propose a general methodology which can be utilized to offer flexible and adaptive OpenMP runtime support. The main idea is to depart from the common practice of having a fixed runtime library to support OpenMP on the device side; customized, adaptive runtime libraries should be selected, tailored to the requirements of the kernels. The goal is the development of an RTS organization which implements only the OpenMP features required by each particular application. That is, it results in an *application-specific* RTS *configuration*. For this to work, compiler assistance is required in order to determine the actual needs of every kernel. This is indeed 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 a 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 mechanism is shown in Fig. 2. The application code is fed to the compiler which performs the necessary transformations. Out of the unified application program, code generation produces code for the host, while `target` regions result in code (kernels) to be executed on the devices. Along with each kernel, a set of metrics gathered during its analysis are output. The metrics are then passed to a "mapper" module. The latter is responsible for choosing the most efficient runtime configuration for the given kernel metrics, out of a set of available runtime library alternatives.

In order for the above to work, knowledge about the kernel characteristics is necessary, so as to determine the level of required OpenMP support. Consequently, the first phase of the mechanism consists of detailed *kernel analysis*. An OpenMP kernel is a block of code enclosed lexically within a `target` construct. The actual kernel *region* additionally 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. 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. We leave the discussion of what the analysis options are for later, where an actual implementation of the proposed mechanism is presented in detail.

The kernel analysis performed by the compiler should result in a set of *metrics* that quantify the exact usage of OpenMP constructs within a particular kernel. Given the analysis results, the second phase of the mechanism chooses the most appropriate device runtime *flavor*, i.e. the RTS library alternative which is to be linked with the kernel code and provide the required OpenMP support. For example, an ideal case would be the existence of a flavor that supports just the needed constructs, nothing more, offering the smallest possible footprint, and thus benefit co-processor cases with small amounts of local memory.

In practice the RTS flavors could be a fixed set of pre-compiled libraries, selected to address specific classes of applications, as derived from typical use-case scenarios. Another possibility is to have on-the-fly parameterizable libraries. Because the default values of the runtime parameters may not suit all applications, tuning some parameters according to kernel characteristics and building different library variants can be proven beneficial.

The module that brings it all together, and is responsible for collecting the compiler analysis results and selecting the runtime flavor to employ is the *mapper*. The mapper should choose the most appropriate flavor so as to minimize the offered OpenMP functionality while at same time cover all kernel OpenMP requirements.

The above outline the general ideas behind our proposal. In the next section we present a full implementation of the mechanism. We discuss in detail both kernel analysis issues as well as the challenges in providing the envisaged mapper functionality.

## 5. Implementation in the OMPi Compiler

The OMPi compiler [6] is a lightweight OpenMP C infrastructure, composed of a source-to-source translator and a 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 V4.5 features including all the `target`-related device ones.

The compilation process for an accelerator-assisted program is shown in Fig. 3. 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

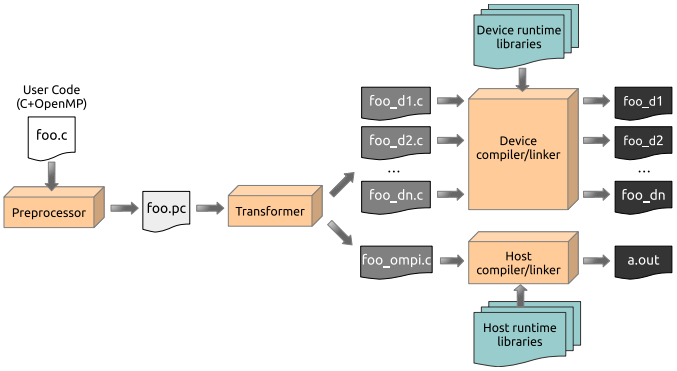


Figure 3: OMPi compilation chain

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 was incorporated. The intermediate files must carry the deduced metrics so as to guide the mapper. We modified the compiler and equipped it with new kernel analysis capabilities in order to derive the desired metrics, as described next.

### *5.1. Kernel Analysis*

The analysis of the kernels is done at a high level, before the actual code transformations. 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 the metrics currently gathered:

- The total number of OpenMP constructs
- The number of `parallel` constructs ($N_p^{(f)}$).
- The number of `for`, `sections` and `single` constructs.
- The number of constructs with `nowait` clauses ($N_{nw}^{(f)}$).
- The number of `ordered`, `atomic` directives.
- The number of `reduction` clauses ($N_{red}^{(f)}$).
- The number of `task` constructs ($N_t^{(f)}$).
- The number of explicit `barrier` directives.
- The types of loop schedules employed, through `schedule` clauses.
- The number of OpenMP ICV modifying clauses and routines.
- The maximum level of parallelism ($L_p^{(f)}$).

All the metrics except the last one count the constructs encountered in the function $f$ 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.

For the whole kernel $K$, the metrics are summed over all kernel functions. For example, the total number of parallel regions is given by:

$$N_p = \sum_{f\in K} N_p^{(f)}.$$

The only exception is the highest level of parallelism where the maximum over all functions is kept:

$$L_p = \max_{f\in K} L_p^{(f)}.$$

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 [28] and requires handling sizable data structures. If $N_{nw} = 0$, there is no need to implement it; a simple blocking barrier would be enough to support all worksharing regions.

As another example of the potential these metrics provide, if there are no parallel constructs in the kernel code ($N_p = 0$), most of the required OpenMP functionality is rather trivial and could be provided by a very small library. If, at the extreme case, there exist no OpenMP constructs, there is no need for an OpenMP RTS at all at the device side.

Regarding the example in Fig 1, the `target` region only generates a single level of parallelism, with the threads sharing the iterations of a `for` loop. Consequently, the derived metrics are as follows: $N_p = 1$, $L_p = 1$ and $N_{\text{for}} = 1$.

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. All metrics get enlisted following

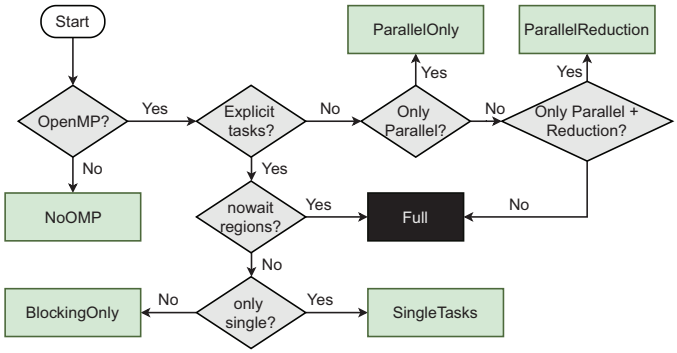


Figure 4: Decision flowchart example

a simple key-value format, one metric per line. The mapper gathers the metrics simply by parsing the top-section comment block of the kernel code.

### 5.2. Mapper

As shown in Fig. 2, the mapper has a central role in the proposed mechanism. In particular, for each kernel, the mapper must:

- Collect the metrics which resulted from the kernel analysis performed by the compiler.
- Include information about the set of available runtime flavors for each device.
- Decide which flavor is the most appropriate for the given kernel, and a specific device.

In other words, given a kernel and a device, the mapper must *map* the metrics to the best available runtime flavor.

Because different devices may have different sets of runtime flavors with different levels of OpenMP support, a single set of rules does not always give the optimum mapping. Hence, we chose to design a general mapper module that can instructed how to make the optimum flavor choice based on the analysis metrics, for each distinct device. For each device, the mapper must be aware of the set of available flavors. In addition, it must be able to make an intelligent decision among them for each kernel. In our system, the device developer decides *how* to find the best runtime flavor based on the kernel metrics. The decision process is encoded as a set of *rules* which check specific metrics and wind up to the correct optimal choice. The mapper, consequently, is given a set of decision rules; it chains them using the compiler metrics until it reaches a final flavor decision.

The selection process can be represented by a flowchart such as the one given in Fig. 4 (details below). Decision nodes (diamonds) query some kernel metric and based on its value, transfer to other nodes, until a flavor node is reached (rectangle) and that specific flavor is chosen as the most appropriate.

In order for the device developer to specify the decision rules and the mapper to utilize them, we designed a custom language for rule files, called MAL. Its syntax is simple and generic, allowing decisions based on exported metrics. The

```
# The available runtime flavors
flavors = [
    NoOMP, ParallelOnly, ParallelReduction,
    BlockingOnly, SingleTasks, Full
],
# Decision nodes
nodes = [
     checkomp = { has(openmp),
                  true: checktasks,
                  false: NoOMP },
   checktasks = { num(tasks),
                  > 0: checknowait,
                  = 0: checkparall },
  checknowait = { has(nowait),
                  true: Full,
                  false: checksingle },
  checksingle = { hasonly(tasks,single),
                  true: SingleTasks,
                  false: BlockingOnly },
  checkparall = { hasonly(parallel),
                  true: ParallelOnly,
                  false: checkreduct },
  checkreduct = { hasonly(parallel,reduction),
                  true: ParallelReduction,
                  false: Full }
]
```

Figure 5: A MAL rule file for the flowchart in Fig. 4.

syntax of a MAL rule file is quite familiar, similar to JSON or Python lists and dictionaries. The file consists of two sections. The first one, `flavors`, is a list of all the available runtime flavors for the device in question. The second section, `nodes`, is a list of rules that process the gathered metrics and represent the decision logic. Specifically, each rule follows a dictionary syntax and consists of:

1. A query statement related to a specific metric. Available queries are: `has`, `hasonly` and `num`. Queries are in the form of one-parameter functions. A `has` (`hasonly`) query checks whether (only) the specified metric exists in the analysis results; hence the outcome can be either `true` or `false`. For a `num` query, the outcome is an integer representing the value of the specified metric.
2. A set of adjacent (follow-up) rules, conditioned on the result of the query statement. The condition is either the truth status of the `has`/`hasonly` query or a comparison which involves a relational operator and an integer for `num` queries. The outcome of the query is compared to the integer through the relational operator and if the condition holds, the corresponding follow-up rule is scheduled to be checked next.

The MAL grammar is given in the Appendix but can be easily understood through the example rule file in Fig. 5 which represents the decision flowchart of Fig. 4. Anything after a hash (#) is considered a comment and the rest of the line is ignored.

In the example, there exist 6 different runtime flavors, named *NoOMP*, *Full*, *BlockingOnly*, *SingleTasks*, *ParallelReduction* and *ParallelOnly*. The first should be utilized if a kernel makes no use of OpenMP functionality while the second one is for kernels which embed complex OpenMP constructs. A common case is to only utilize a `parallel` construct with or without

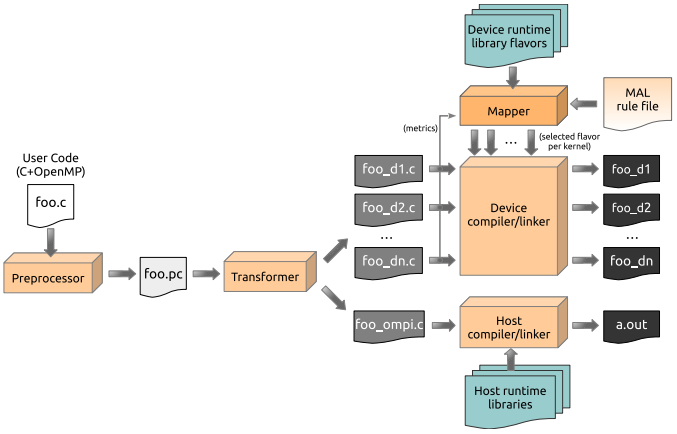


Figure 6: Modified OMPi compilation chain with integrated mapper

`reduction` clauses, giving rise to the last two optimized flavors. The *SingleTasks* flavor tries to capture task-based parallelism where a single thread creates all the tasks, while *BlockingOnly* fits the rest of the cases.

The flavors are shown as green rectangles in Fig. 4 and are declared in the top `flavors` section in the rule file (Fig. 5). The decision logic is given in the `nodes` section of the rule file and correspond to the diamond nodes in the flowchart. The starting node is the first one listed, hence the decision process always begins with the *checkomp* node. The node consists of a query statement (`has(openmp)`) which queries the "openmp" analysis metric that indicates whether the kernel utilizes OpenMP at all. If the outcome is true, the next node to be visited is node *checktasks*; otherwise the next node is *NoOMP* which happens to be a flavor node, terminating the decision process and promoting it as the most suitable flavor.

If the *checktasks* node is visited, the query `num(task)` checks how many `task` regions are present in the kernel; if none exists, there is a transfer to the *checkparall* node which may lead to lighter runtime flavors. For example, if `parallel` is the only OpenMP construct present, then a flavor that provides support only for creating a team of threads is the optimal choice (*ParallelOnly*).

The MAL grammar, is simple enough to allow for recursive-descent parsing. After a MAL rule file for a specific device gets parsed, it is stored internally as a graph using an adjacency list representation. This internal graph is traversed every time the mapper has to decide on the most suitable flavor for a kernel that targets this particular device.

The implementation of the mapper has been integrated within the OMPi compilation chain, as is shown in Fig. 6. Following the architecture shown in Fig. 2, in order to deploy the mapper module it was necessary to modify the last stage in the OMPi compilation chain. In particular, the intermediate kernel files are fed to the mapper before given to the device compiler. The mapper extracts the embedded metrics and uses them to traverse the decision rules for each kernel. It is then able to select the most appropriate runtime flavor; this flavor is given to the device linker to link against the kernel file.

## 6. A Case Study: The Epiphany Accelerator

As a concrete demonstration of the proposed compiler-assisted, adaptive runtime architecture, we used the well known Parallella-16 [1], a popular 18-core credit card-sized board equipped with two processing modules; the main CPU, a dual-core ARM Cortex A9 with 32KiB 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 32MiB 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 $64 \times 64$ mesh interconnect, so in theory systems up to 4096 cores are possible. In Epiphany-III the chip is pinned on a $4 \times 4$ submesh of the virtual $64 \times 64$ mesh. The chip has four eLINKs that may be used to interconnect it with other chips. In the Parallella-16 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 1MiB of the total address space (that is the maximum physical memory that can be integrated by design), which is addressable by all cores. However, in the current version, each core comes with just 32KiB 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.

### 6.1. The Full RTS

OMPi was the first compiler to support the Epiphany accelerator as an OpenMP device [25]. The RTS in [25] 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.

The original RTS was carefully designed so as to minimize its memory footprint, while supporting OpenMP fully (albeit inefficiently, using the much slower shared memory region for key structures). The limited local memory of the device cores makes it impossible to fit sophisticated OpenMP RTS structures alongside the application data. 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

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

Table 1: Data sizes in the original RTS

task are stored in the slower shared memory area, due to space requirements.

The above 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 [25]. 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.

The original runtime support for the Epiphany 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 the size of all different data structures and the corresponding functionalities they support. The result 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 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. As seen in Table 1, the sizes of these entities have the biggest impact on the total footprint of the RTS.

Not all those data are actually needed for the execution of every kernel. The instinctive 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, and demonstrates the potential for reducing the memory footprint in the case where an application does not use explicit tasking. The size of the necessary internal control variables (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.

### *6.2. Implementation of 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 because of both 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 common OpenMP worksharing constructs (`single`, `for`, `sections`) can have different combinations of `reduction`, `schedule`, `collapse`, `nowait` 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 hosts 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 a set of new 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 total 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) *ParAtomic*. This RTS extends (2) and allows only the `atomic` synchronization construct between the eCOREs of a parallel team.
(5) *ParCritical*. This RTS extends (2) by allowing only the `critical` synchronization construct between the eCOREs of a parallel team.
(6) *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.
(7) *ForOrdered*. This extends the previous one by adding the ability to utilize the `ordered` clause of the `for` directive.
(8) *ForWSOnly*. This is the *ForOrdered* RTS where the team members can utilize any of the three loop schedules (`static`, `dynamic`, `guided`) as well as the `ordered` clause.
(9) *SingleOnly*. Here we extend the *ParallelOnly* flavor by supporting only the `single` worksharing construct.
(10) *NoTasks*. We developed this RTS for kernels with no explicit tasks. The rest of the OpenMP functionality (e.g. worksharing, synchronization, etc) is present.
(11) *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.
(12) *NoTasksBO*. We added a variation of the *BlockingOnly* flavor where the tasking support has been removed.
(13) *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.
(14) *TasksICVs*. This RTS extends the previous one by supporting per-task ICVs; each task descriptor has a private copy of the ICVs which is inherited from the parent task, and which the task is allowed to modify using the appropriate routines.
(15) *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. In any case, our goal here is to prove the potential of the proposed mechanism, and not to derive all possible runtime flavors for all possible kernels.

Furthermore, 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*, *TasksICVs* 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. Fig. 7 shows the complete decision flowchart for the 15 flavors. Without going into great detail, here is a quick overview of the decision making process: RTS (1) is chosen to accompany kernels which do not include OpenMP constructs. Based on the tasking metrics, RTSs (11) and (13) through (15) are used when tasks are present; the actual choice depends on the type of worksharing regions observed and whether per task ICVs are needed. If no explicit tasks are used RTSs (2)-(10) and (12) are candidates. The decision is driven by the presence of `parallel`, `reduction`, `critical`, `atomic` and worksharing constructs.

The flowchart of Fig. 7 was coded as a rule file using MAL. The rule file contains the 15 flavors and 16 decision nodes corresponding to the diamond nodes of the flowchart. All in all, this resulted in 200 lines of code, which were used to automate the mapper decisions for every compiled kernel.

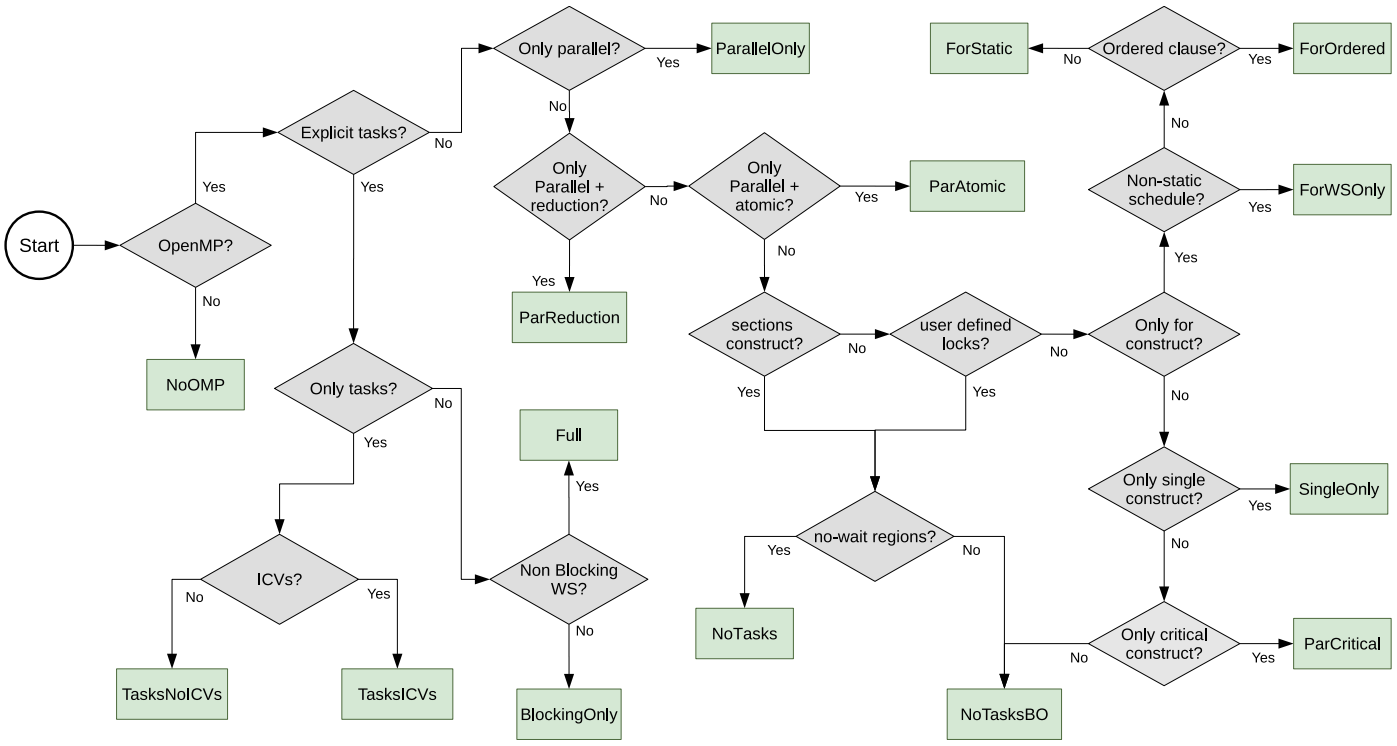


Figure 7: The full decision flowchart of the case study.

## 7. 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 (15) 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 [29] where their basic routines are offloaded through `target` directives. These benchmarks are intended for measuring the execution time overheads of specific constructs. In this study we additionally utilized them to exhibit possible size benefits of the produced kernel object files. 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 four 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 application 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 other two applications are task-based, taken from the Barcelona OpenMP Tasks Suite [30], One is the NQueens application which 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. The last one is the Sort application which sorts an 1D array by splitting its elements in halves, recursively sorting each part and then merging them with a parallel merge algorithm. The application uses parallel tasks and resorts to sequential sorting when the number of processed elements becomes too small. Here we consider an array of 1,048,576 random integers using a cutoff value of 2048.

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-16 code examples. The original code is rather simplistic and refers to a $4 \times 4$ 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

| Application | Computed metrics |
|---|---|
| Mandelbrot | $N_p = 1, N_b = 1, L_p = 1$ |
| Pi calculation | $N_p = 1, N_{\text{red}} = 1, L_p = 1$ |
| Game of Life | $N_p = 1, N_f = 1, N_b = 4, L_p = 1$ |
| NQueens | $N_p = 1, N_t = 1, L_p = 1$ |
| Sort | $N_p = 1, N_t = 9, N_{\text{single}} = 1, L_p = 1$ |

Table 2: Kernel analysis

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. At the end of each round each core updates the data of the appropriate neighbors, taking advantage of Epiphany's fast interconnection network for on-chip remote writes. 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.

The last experiment included the Mandelbrot deep zoom application which calculates a Mandelbrot set and zooms in and out up to 10500× at six predefined points. Each image frame is written directly to the frame buffer of the Parallella-16 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, as a way to exhibit the real time performance possibilities of the Epiphany chip. We have developed a parallel version of the code using OpenMP [25], where the kernel statically distributes the calculation among the device's 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.

We summarize the kernel metrics reported by the compiler for the 4 applications in Table 2. All of them contain a single level of parallelism ($L_p = 1$) resulting from one `parallel` region ($N_p = 1$). The NQueens and Sort applications are the only ones to utilize tasking ($N_t \geq 1$).

Table 2 does not include the the EPCC microbenchmarks since each one of them contains a `parallel` region ($N_p = 1$) plus 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 ($N_f = 1$).

### *7.1. Size Results*

In Table 3 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 utilized the *NoOMP* RTS, listed as (1) in Section 6.2

| 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% |
| Sort | 18308 | 16408 | (*BlockingOnly*) | 10.38% |
| 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% |

Table 3: Executable kernel sizes (bytes)

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 4 KiB.

The kernel for the calculation of $\pi$ 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 3.5 KiB. The Game of Life creates a parallel team of threads which share iterations of a static for loop, hence the `ForStatic` flavor was chosen as optimal, and 4 KiB smaller size is needed for the object file. The NQueens application utilizes only the `parallel` and `task` directives. In addition, no OpenMP internal control variables are modified in the user code. Consequently, the *TasksNoICVs* runtime library was linked with the kernel. Sort requires the *BlockingOnly* flavor since it also uses a `single` construct. For the EPCC-based kernels the mapper employed the RTSs (2), (5) through (7) and (9) 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.

#### *7.1.1. 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 4 we present timing results for the EPCC microbenchmarks. As mentioned previously, we modified the original suite

| Kernel | Full RTS | Optimized RTS | Reduction |
|---|---|---|---|
| EPCC-for-static | 58.43 | 6.55 | 88.79% |
| EPCC-critical | 2.17 | 1.55 | 28.57% |
| EPCC-single | 45.61 | 1.96 | 95.70% |
| EPCC-ordered | 4.70 | 4.66 | 0.85% |
| EPCC-barrier | 15.17 | 2.28 | 84.97% |

Table 4: EPCC overheads ($\mu$sec)

| Kernel | Full RTS | Optimized RTS | Reduction |
|---|---|---|---|
| Empty Kernel | 0.25 | 0.21 | 16% |
| Mandelbrot | 30.05 | 30.00 | 0.16% |
| Pi calculation | 0.27 | 0.26 | 3.7% |
| Game of Life | 2.08 | 1.37 | 34.16% |
| NQueens (tasks) | 1.82 | 1.81 | 0.6% |
| Sort (tasks) | 0.51 | 0.50 | 2% |

Table 5: Application kernels execution times (sec)

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 95% are observed. The noticeable cases are those of `single`, `for` with `static` schedule and `barrier`. 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 regarding the other applications are given in Table 5. Notice that a 0.1 sec delay is always present due to the way the Parallella-16 handles execution on the Epiphany, and is a performance burden that any offloaded kernels must bare (even eSDK-based ones). The 16% reduction in the empty kernel running time is due to the significantly smaller size of the RTS linked in the kernel executable, which has to be distributed to the eCORE(s). 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 and the Sort kernels. In addition, accesses to the shared memory area which stores the tasks data environments have an impact to the total execution time.

Finally, a significant improvement of 3.7% is observed in the kernel that calculates $\pi$ and an even more impressive reduction of 34.16% in the Game of Life. 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 redesigned 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.

## 8. Conclusions and Future Work

In this work we present a novel compiler-assisted, adaptive RTS organization that is able to produce specialized and optimized OpenMP support for offloading devices, tailored to the needs of each particular application.

The general idea is to build multiple device runtime libraries (flavors), each offering different levels of OpenMP functionality support. For a given application, the compiler performs a detailed inter-procedural analysis of all `target` kernel regions and calculates a set of metrics characterizing the kernel behaviors with respect to OpenMP functionality. These metrics are fed to a mapper mechanism which decides on the most appropriate runtime library flavor to employ. 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. This results in size-optimized executables, with clear potential for serious performance benefits.

Given the set of metrics that profile the application and are obtained by compile-time analysis, the mapper has the ability to automatically select among different runtime libraries using a set of device-specific decision rules which an implementer provides for a device. To this end, we introduce a custom language (MAL) which is able to capture the logic of a decision flow diagram using a concise syntax.

We have implemented our ideas on the Parallella-16 board, in the context of the OMPi compiler. Our experiments show dramatic decrease in both kernel sizes and execution times, as compared to the original, monolithic RTS.

The proposed mechanism is quite general and applicable to any OpenMP device. We are mainly working in three directions: (a) expanding the expressiveness of MAL, (b) adding support for more devices (especially GPUs), and (c) examining the possibility of parameterized flavors, whereby parts of a flavor can be tuned at compile time. We finally plan to expand our compiler analysis to support the ideas of [27, 31] in order to further optimize the requirements of the tasking subsystem.

## Appendix A. MAL Grammar

In this Appendix we provide the full grammar of the Mapper language for rule files (MAL), in Backus-Naur form (BNF).

⟨*S*⟩ ::= ⟨*flavor-list*⟩ ‘**,**’ ⟨*node-list*⟩

⟨*flavor-list*⟩ ::= **flavors** ‘=’ ‘`[`’ ⟨*flavor*⟩ ‘`]`’

⟨*flavor*⟩ ::= ⟨*name*⟩ ‘**,**’ ⟨*flavor*⟩
| ⟨*name*⟩

⟨*node-list*⟩ ::= **nodes** ‘=’ ‘`[`’ ⟨*nodes*⟩ ‘`]`’

⟨*nodes*⟩ ::= ⟨*node*⟩ ‘,’ ⟨*nodes*⟩
| ⟨*node*⟩

⟨*node*⟩ ::= ⟨*name*⟩ ‘=’ ‘{’ ⟨*node-body*⟩ ‘}’

⟨*node-body*⟩ ::= ⟨*bool-query*⟩ ‘,’ ⟨*bool-adj-list*⟩
| **num** ‘(’ ⟨*metric*⟩ ‘)’ ‘,’ ⟨*num-adj-list*⟩

⟨*bool-query*⟩ ::= **has** ‘(’ ⟨*metric-list*⟩ ‘)’ | **hasonly** ‘(’ ⟨*metric-list*⟩ ‘)’

⟨*bool-adj-list*⟩ ::= ⟨*bool-edge*⟩ ‘,’ ⟨*bool-edge*⟩

⟨*bool-edge*⟩ ::= ⟨*boolean*⟩ ‘:’ ⟨*name*⟩

⟨*num-adj-list*⟩ ::= ⟨*num-edge*⟩ ‘,’ ⟨*num-adj-list*⟩
| ⟨*num-edge*⟩

⟨*num-edge*⟩ ::= ⟨*relop*⟩ integer ‘:’ ⟨*name*⟩

⟨*name*⟩ ::= ‘"’ charstr ‘"’ | charstr

⟨*metric-list*⟩ ::= ⟨*metric*⟩ ‘,’ ⟨*metric-list*⟩ | ⟨*metric*⟩

⟨*metric*⟩ ::= charstr

⟨*boolean*⟩ ::= **true** | **false**

⟨*relop*⟩ ::= ‘>’ | ‘<’ | ‘=’ | ‘>=’ | ‘<=’ | ‘!=’

A “charstr” is any string which contains alphanumeric characters, “-” or “_”, and is used for the names of flavors, nodes and metrics. The metrics currently exported by the compiler and recognized by MAL are given in the following table.

| Metric name | Quantity measured |
|---|---|
| `parallel` | number of `parallel` constructs |
| `max_par_lev` | maximum level of parallelism |
| `single` | number of `single` constructs |
| `for` | number of `for` constructs |
| `schedstatic` | number of `for` constructs with static schedule |
| `scheddynamic` | number of `for` constructs with dynamic schedule |
| `schedguided` | number of `for` constructs with guided schedule |
| `sections` | number of `sections` constructs |
| `tasks` | number of `task` constructs |
| `reduction` | number of constructs with a reduction schedule |
| `ordered` | number of `for` constructs with the ordered clause |
| `uncritical` | number of unnamed `critical` constructs |
| `criticals` | number of named `critical` constructs |
| `atomic` | number of `atomic` constructs |
| `nowait` | number of constructs with a nowait clause |
| `hasextraicvs` | true if ICVs-changing functions are called |
| `haslocks` | true if user-defined locks are present |
| `openmp` | true if OpenMP functionality is present |

## References


[1] Adapteva, Parallella Reference Manual (Sept. 2014).

[2] L. Sommer, A. Koch, OpenMP device offloading for embedded heterogeneous platforms - work-in-progress, in: Proc. EMSOFT 2020, International Conference on Embedded Software, 2020, pp. 4–6.

[3] Khronos OpenCL Working Group, The OpenCL 3.0 Specification (Sept. 2020).

[4] D. B. Kirk, W.-m. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, 3rd ed., Morgan Kaufmann, MA 02139, USA, 2017.

[5] OpenMP ARB, OpenMP Application Program Interface V5.1 (Nov. 2020).

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

[7] T. Hanawa, M. Sato, J. Lee, T. Imada, H. Kimura, T. Boku, Evaluation of multicore processors for embedded Systems by parallel benchmark program using OpenMP, in: Proc. IWOMP ’09, 5th Int’l Workshop on OpenMP, Dresden, Germany, 2009, pp. 15–27.

[8] M. Sato, Y. Nakajima, Y. Ojima, Y. Hotta, OpenMP implementation and performance on embedded Renesas M32R chip multiprocessor, in: Proc. EWOMP ’04, 6th European Workshop on OpenMP, 2004, pp. 37–42.

[9] F. Liu, V. Chaudhary, A practical OpenMP compiler for system on chips, in: Proc. WOMPAT 2003, Workshop on OpenMP Applications and Tools, Vol. 2716, Torondo, Canada, 2003, pp. 54–68.

[10] W.-C. Jeun, S. Ha, Effective OpenMP implementation and translation for multiprocessor system-on-chip without using OS, in: Proc. ASP-DAC ’07, 12th Asia and South Pacific Design Automation Conf., Yokohama, Japan, 2007, pp. 44–49.

[11] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, A. Gatherer, Implementing OpenMP on a high performance embedded multicore MPSoC, in: Proc. IPDPS ’09, IEEE Int’l Symposium on Parallel and Distributed Processing, Rome, Italy, 2009, pp. 1–8.

[12] P. Burgio, G. Tagliavini, A. Marongiu, L. Benini, Enabling fine-grained OpenMP tasking on tightly-coupled shared memory clusters, in: Proc. DATE 13, Design Automation and Test in Europe, Grenoble, France, 2013.

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

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

[15] R. Dolbeau, S. Bihan, F. Bodin, HMPP: A hybrid multi-core parallel programming environment, in: Proc. GPGPU 2007, Workshop on General Purpose Processing Using Graphics Processing Units, Boston, USA, 2007.

[16] A. Fernández, V. Beltran, X. Martorell, R. M. Badia, E. Ayguadé, J. Labarta, Task-based programming with OmpSs and its application, in: Proc. Euro-Par 2014 International Workshop, Revised Selected Papers, Part II, Porto, Portugal, 2014, pp. 602–613.

[17] L. Sommer, F. Stock, L. Solis-Vasquez, A. Koch, Using parallel programming models for automotive workloads on heterogeneous systems – a case study, in: Proc. PDP 2020, 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, Västers, Sweden, 2020, pp. 17–21.

[18] A. Kurth, A. Capotondi, P. Vogel, L. Benini, A. Marongiu, HERO: An open-source research platform for HW/SW exploration of heterogeneous manycore systems, in: Proc. ANDARE ’18, 2nd Workshop on Autotuning and Adaptivity Approaches for Energy Efficient HPC Systems, 2018.

[19] A. J. Peña, X. Martorell, OmpSs+OpenACC/OpenMP interoperability, Tech. Rep. EPEEC Project, Deliverable 3.1 (March 2019).

[20] J. M. Diaz, K. Friedline, S. Pophale, O. Hernandez, D. E. Bernholdt, S. Chandrasekaran, Analysis of OpenMP 4.5 offloading in implementations: Correctness and overhead, Parallel Computing 89 (2019) 102546.

[21] 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, 2013, pp. 1213–1225.

[22] L. Chunhua, Y. Yonghong, B. R. de Supinski, D. J. Quinlan, B. M. Chapman, Early experiences with the OpenMP accelerator model., in: Proc. IWOMP 2013, 9th Int’l Workshop on OpenMP, Canberra, Australia, 2013, pp. 84–98.

[23] C. Bertolli et al, Coordinating GPU threads for OpenMP 4.0 in LLVM, in: Proc. LLVM-HPC ’14, LLVM Compiler Infrastructure in HPC, New Orleans, Louisiana, 2014, pp. 12–21. doi:10.1109/LLVM-HPC.2014.10.

[24] G. Mitra, E. Stotzer, A. Jayaraj, 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, 2014, pp. 202–214.

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

[26] G. Tagliavini, D. Cesarini, A. Marongiu, Unleashing fine-grained parallelism on embedded many-core accelerators with lightweight OpenMP tasking, IEEE Transactions on Parallel and Distributed Systems 29 (9) (2018) 2150–2163.

[27] A. Munera, S. Royuela, E. Quiñones, Towards a qualifiable OpenMP

framework for embedded systems, in: Proc. 2020 Design, Automation Test in Europe Conference Exhibition (DATE), 2020, pp. 903–908.

[28] G. Philos, V. Dimakopoulos, P. Hadjidoukas, A runtime architecture for ubiquitous support of OpenMP, in: Proc. ISPDC 2008, 7th Int'l Symposium on Parallel and Distributed Computing, Krakow, Poland, 2008, pp. 189–196.

[29] M. J. Bull, Measuring synchronisation and scheduling overheads in OpenMP, in: Proc. EWOMP '99, the 1st European Workshop on OpenMP, Lund, Sweden, 1999, pp. 99–105.

[30] A. Duran, X. Teruel, R. Ferrer, X. Martorell, E. Ayguadé, Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP, in: Proc. ICPP '09, Vienna, Austria, 2009, pp. 124–131.

[31] V. Kumar, A. Sbîrlea, A. Jayaraj, Z. Budimlić, D. Majeti, V. Sarkar, Heterogeneous work-stealing across CPU and DSP cores, in: Proc. 2015 IEEE High Performance Extreme Computing Conference (HPEC), 2015, pp. 1–6.