A General-Purpose Mapper Module for Adaptive OpenMP Runtime Support

Ilias K. Kasmeridis
Dept. of Computer Science and Engineering
University of Ioannina
Ioannina, Greece
ikasmeridis@cse.uoi.gr

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

Abstract

The latest versions of the OpenMP specifications have introduced constructs that enable programmers to utilize heterogeneous compute units alongside the main multicore CPU. They allow offloading specific regions of the program code to any of the available computational devices; the offloaded code may itself generate parallelism by employing suitable OpenMP constructs. While the concept seems ideal, co-processors and accelerators, especially embedded ones, often have limited resources or capabilities to provide efficient OpenMP support. Designing an OpenMP infrastructure for such devices can be a real challenge. A very effective solution has been proposed in the form of compiler-assisted, adaptive runtime support that is tailored to each specific application. In this work, we present a general-purpose mapper module, which has the ability to select the best runtime configuration given a) a set of metrics that profile the application and are obtained by compile-time analysis and b) a set of device-specific decision rules, which an implementor provides for a device, written in a customized language.

1 Introduction

Current computing systems are heterogeneous, with different processor and memory hierarchies co-existing in a single system. Following the trend of increasing computational power and multitasking capabilities, embedded systems have started to adopt similar organizations. However, the exploitation of the computation resources of each of the subsystems places significant effort on the programmers side. Low-level SDKs are employed to utilize specific hardware units, which usually entail restructuring portions of the application codes and, inevitably, increasing complexity.

Targeting heterogeneous systems, the OpenMP API [16] was recently augmented with directives that allow programmers to take advantage of the underlying system structure without resorting to low-level facilities. These directives manage the offloading of portions of the program code (known as *kernels*) from the main CPU onto any computational unit available, such as GPUs and accelerators, generically called *devices*. It is worth noting that the application source code remains unified, mixing the host CPU and device parts in an seamless fashion.

According to OpenMP specifications, kernels offloaded to any device may include OpenMP directives, exploiting multiple processing elements which may be available within the device. Consequently, each device must provide runtime support for OpenMP functionality. This, however, can be inefficient or even infeasible, especially in devices with limited resources; embedded multicores or multicore systems-on-chip are characteristic examples of devices with limited on-chip memory. The system designer thus faces the choice of whether to support OpenMP fully (albeit inefficiently) or partially (limiting program expressiveness and functionality).

In a previous work [2] we proposed a novel solution to the above problem. The concept is to provide compiler-assisted, adaptive OpenMP runtime support, tailored to each different kernel: kernels that make limited use of OpenMP directives maybe be accompanied by a small and fast OpenMP runtime infrastructure; more complex kernels can also be accommodated, albeit with possibly less efficient runtime support, depending on the available resources. Compiler analysis is employed in order to discover the level of OpenMP support required by each kernel; an appropriate runtime infrastructure should then be chosen to support optimally a particular kernel.

This work consummates our proposal by introducing a device-agnostic *mapper* module. The mapper is responsible for collecting the compiler metrics and making an automated decision about the optimal runtime configuration to provide the required OpenMP functionality to a kernel. Instead of implementing one runtime system for supporting OpenMP on a device, a multitude of runtime libraries (termed *flavors*) exist, each one implementing a specific subset of OpenMP functionality. Given the compiler analysis for a particular kernel, the role of the mapper is to select the most appropriate among the available runtime flavors.

Since such an action requires knowledge about the flavor features, the mapper is equipped with a flavor-selection language called MAL. Based on MAL, the device designer can describe the selection logic among the different flavors through a set of rules.

The rest of the paper is organized as follows. Section 2 gives a brief introduction to the device-related features of OpenMP. Section 3 summarizes the compiler-assisted, adaptive runtime support mechanism. Section 4 presents the concept of the mapper module and details its design; a case study is given in Section 5. Finally, Section 6 concludes this work.

1.1 Related Work

Devices were added to OpenMP in V4.0 of its specifications but OpenMP was considered as a possible programming model for accelerators or multicore embedded systems much earlier [7, 8, 13]. These works, however, refer to older versions of the standard and, most importantly, do not address heterogeneity. In [3] 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 V4.0 for offloading and executing OpenMP kernels on the MPSoC. Other directive-based efforts to offload kernel onto attached devices include OmpSs [12] and HMPP [11]. None of these works supports OpenMP functionality in the offloaded code.

While OpenMP specifications have recently reached V5.0, device support remains limited; there exist relatively few compilers supporting relatively few device types. The offloading process of the Intel compiler is described in [15], targeting the Xeon Phi coprocessor. Similarly, [5, 9] target NVIDIA GPUs, through HOMP, a prototype based on the ROSE compiler, and the LLVM compiler, correspondingly. In [14] 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 [4] propose an implementation of OpenMP device directives on the Parallella [1] board, which consists of a dual-core ARM processor and a 16-core Epiphany co-processor. In contrast to the above works which provide either partial OpenMP runtime support or a monolithic full implementation, our proposal utilizes compiler-assisted, adaptive runtime system configurations.

2 OpenMP Devices

Starting with version 4.0, OpenMP [16] offers a platform-agnostic model for heterogeneous parallel programming. The programmer can offload arbitrary portions of the application code to any available device. All the low-level details related to data and code transfers are orchestrated by the compiler and implemented by device-specific runtime libraries. The host processor (i.e. the main CPU) executes the application code until a device-related construct is met, whereby execution is switched to a specified device, by creating a new data environment and offloading the associated portion of code to the corresponding compute unit.

The `target` directive is used to mark the code region (kernel) which will get offloaded, along with its data environment. The latter is defined through `map` clauses and can be manipulated with further directives such as `target data`, `target update`, `declare target` and others.

A key feature of offloaded kernels is that they may employ unrestricted OpenMP functionality, with the exception of offloading code to other devices. Parallelism-generating, tasking and worksharing constructs such as `parallel`, `task`, `for`, etc. can all exist within a `target` region. This flexibility makes OpenMP a very powerful parallel programming model, 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.

The aforementioned functionality is possible only if the devices themselves offer complete OpenMP support. However, while OpenMP was originally designed for systems with abundant resources, embedded or attached accelerators have different architectures and are designed to serve different purposes. With some notable exceptions such as the Xeon Phi accelerator [15], 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 runtime system (RTS) for such devices depend on these resource limitations. 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] board 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.

As a consequence, a common approach is to provide only *partial* OpenMP support on a device with limited resources [8,9,14], i.e. implement a subset of the API. Clearly, partial support reduces expressiveness; the application code may have to be redesigned to match the availability of OpenMP constructs, which also reduces code portability. Some compilers (e.g. GCC) strive to support OpenMP *fully* on the device side, providing a powerful, high level parallel pro-

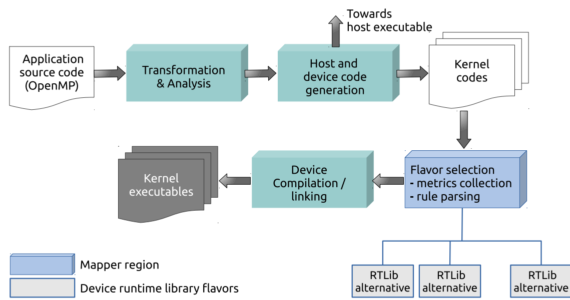

Figure 1: Toolchain process for adaptive runtime support.

gramming abstraction. Nevertheless, the design of a complete OpenMP RTS is not a trivial task for arbitrary devices. Furthermore, hardware limitations may lead to poor performance for some of the OpenMP constructs [4, 5, 9].

3 Compiler-Assisted Adaptive Runtime Support

In this section, we give an overview of the concept of a compiler-assisted adaptive runtime support mechanism for devices [2]. 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. For this to work, compiler assistance is required in order to determine the actual needs of every kernel.

The mechanism is depicted in Fig. 1. The application code is fed to the compiler which performs the necessary transformations. Out of the unified application code, code generation produces code for the host, while `target` regions produce code (kernels) for the devices. For each kernel, one of the alternative runtime libraries (called *flavors*) is selected and along with the kernel itself forms the final kernel executable, using device-specific compilation and linking tools.

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* 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. It first builds the call graph of the kernel and then visits 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.

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 optimized flavor may support just the needed constructs, offering the smallest possible footprint, and thus benefiting co-processor cases with small amounts of local memory. The runtime flavors could be a fixed set of precompiled 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 prove 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.

3.1 Implementation in the OMPi Compiler

In [2], an initial version of the mechanism was implemented in the context of the OMPi compiler [10], a lightweight OpenMP C infrastructure, composed of a source-to-source translator and a flexible, modular RTS. The compiler takes as input C code with OpenMP directives, and after the pre-processing and transformation steps, it outputs a multi-threaded C file for executing on the host and another set of intermediate files, one for each kernel. Every intermediate file is augmented with calls to the RTS of the corresponding device. In the last stage, the intermediate files are compiled with the appropriate system compilers in order to provide the final executables.

The compiler was modified to perform the required kernel analysis. It operates at the abstract syntax tree level and produces the metrics below as the result of the analysis:

- The total number of OpenMP constructs.
- The number of `parallel`, `for`, `sections`, `single` and `task` constructs.
- The number of explicit `barrier` directives.
- The maximum level of parallelism.

As a first and important contribution of the current work, we extended the analysis scope of the compiler, and we now offer the following additional metrics:

- The total number of `reduction` clauses.
- The total number of `ordered` directives.
- The total number of `atomic` directives.
- The total number of `nowait` clauses.
- The loop schedules used, through `schedule` clauses.
- Whether explicit locking is employed or not; this is based on detecting calls to the corresponding OpenMP runtime routines.
- Whether any Internal Control Variables (ICVs) are accessed, through OpenMP runtime calls.

The prototype in [2] did not implement a general mapper module; there was only support for one specific device, and a fixed set of runtime flavors. The selection was based on a hard-coded logic that was only applicable to that particular device and those particular runtime flavors. In this work, as our second and most important contribution, we present the design of a general, novel mapper module, which works for arbitrary devices and arbitrary runtime flavors.

4 A General, Device-Agnostic Mapper

In this section we introduce a general mapper design which is able to accommodate any device and any set of runtime flavors. To make the discussion concrete, we have also implemented a fully working mapper module in the OMPi compilation chain. As shown in Fig. 1, 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.
- Keep 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.

Since OMPi is a source-to-source compiler, the kernel code generation produces source code files, ready to be compiled and linked by device-specific tools. We pass the metrics from the compiler to the mapper by embedding them as comments at the top of the output kernel files. All metrics presented in Section 3 get enlisted following 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.

Because different devices may have different sets of runtime flavors with different levels of OpenMP support, there

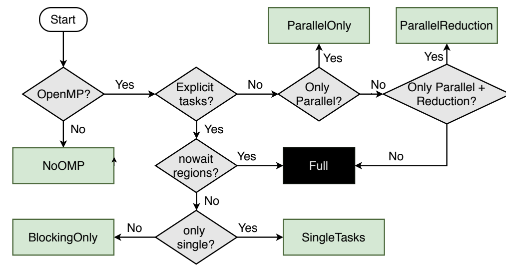

Figure 2: Decision flowchart example

does not exist a single set of rules that always gives the optimum mapping. Hence, the conception of a mapper with an intelligent, always-optimum selection algorithm is rather pointless. As a result, we chose to design a general mapper module that can accommodate device-specific selection. In other words, one has to instruct the mapper 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 decision logic is delegated to the device developer; she is the one that specifies *how* to find the best runtime flavor based on the kernel metrics. The decision process should be 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. 2 (details below). Decision nodes (diamonds) query some kernel metric and based on its value, transfer the process to other nodes, until a flavor node is reached (rectangle) and that specific flavor is chosen as the most appropriate. The device developer provides such a decision flowchart for her device. The mapper is, then, responsible for traversing it for every kernel, using the kernel's metrics to navigate among the decision nodes.

4.1 Mapper Language

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 syntax of a MAL rule file is quite familiar, similar to JSON or Python lists and dictionaries. The file consists of two components. The first one, `flavors`, is a list of all the available runtime flavors for the device in question. The second component, `nodes`, is a list of rules that concern the gathered metrics and represent the decisions. 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

```
# The available runtime flavors
flavors = [
    NoOMP, ParallelOnly, ParallelReduction,
    BlockingOnly, SingleTasks, Full
],
# Decision nodesure
nodes = [
     checkomp = { has(openmp),
                  true: checktasks,
                  false: NoOMP },
   checktasks = { num(task),
                  > 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 3: A MAL rule file for the flowchart in Fig. 2.

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 can be easily understood through the example rule file in Fig. 3, which represents the decision flowchart of Fig. 2. 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 `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. 2 and are declared in the top `flavors` section in the rule file (Fig. 3). 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.

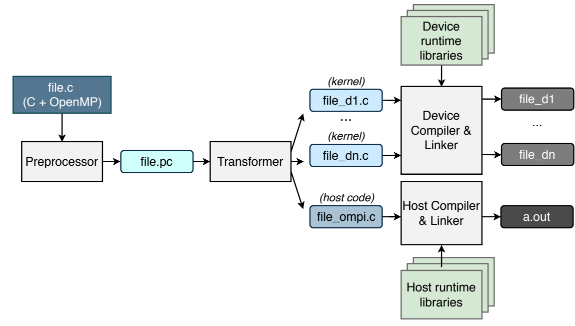

Figure 4: OMPi compilation chain.

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*).

4.2 Implementation Details

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 [10] compilation chain, which is shown in Fig. 4. OMPi consists of a source-to-source compiler (transformer) and a sophisticated runtime support system. After the syntax analysis stage, the application source code is stored using an abstract syntax tree representation. All transformation steps as well as the kernel analysis we described previously operate on the abstract syntax tree. The code generation phase, then, produces intermediate files, ready to be compiled by the host/device system compiler. The final step utilizes, again, system tools to link the host and kernel program with the required runtime libraries, yielding the final executables.

Based on Fig. 1, 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 compilers; the mapper is then able to traverse the decision rules for each kernel and each requested device, and select the most

appropriate runtime flavor, based on the embedded kernel metrics. This flavor is given to the device linker to link against the kernel file.

5 A Case Study

To demonstrate our mechanism we use Parallella [1], a popular 18-core credit card-sized computer with two processing modules; the main CPU, a dual-core ARM Cortex A9 (built within a Zynq 7010 SoC), and an Epiphany-III 16-core CPU which is used as a co-processor. The ARM and the Epiphany use a 32 MiB portion of the system RAM as *shared memory*. The Epiphany-III chip contains a 4×4 mesh of cores. 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 chip address space, which is addressable by all cores. However it comes with just 32 KiB of fast local scratch-pad memory. All memories are available through regular load/store instructions by all eCOREs.

OMPi is the first compiler to support the Epiphany accelerator as an OpenMP device [4]. The limited local memory of the device cores makes it impossible to fit sophisticated OpenMP RTS structures alongside the application data. 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). In [2] this original (*Full*) RTS was used as a basis for a set of 12 different RTSs, each one optimized for a certain type of kernels. Each flavor is a modified version of the original, trimmed to support a limited number of constructs. For each flavor all unnecessary internal data structures were removed and all routines were modified accordingly.

Fig. 5 contains the decision flowchart for the 12 flavors. Our first action was to encode the flowchart as a rule file using MAL; this resulted in 190 lines of code, which was used to automate the mapper decisions. Our next concern was to demonstrate that the mapper did indeed succeed to reach the optimal decisions. We used the same set of applications as in [2] which included the EPCC microbenchmarks [6], as well as other applications such as the calculation of π, the calculation of all solutions of the N-queens problem, and the well-knwon Game of Life, which simulates an evolution process.

For a trivial application with an empty kernel, the mapper chose the *NoOpenMP* flavor, due to the absence of OpenMP instructions. The π calculation performs numerical integration based on the trapezoidal rule and uses a parallel team, with threads summing their partial results using a reduction clause. Therefore, the automatically selected flavor was *ParallelReduction*. To solve the N-queens problem, `parallel` and `tasks` directives are used, so the mapper has determined that the appropriate version is *TasksNoICVs*. The Game of Life uses only `parallel` and `for` constructs with barrier synchronization, so the most suitable flavor is *ForStatic*. Each of the EPCC tests utilizes a parallel region and one specific construct. The mapper successfully mapped the appropriate flavor for each one of them.

Table 1: Executable kernel sizes (bytes)

Application	Full RTS	Adaptive RTS	Reduction
Empty kernel	8648	2252	73,95%
Pi	12744	8864	30,44%
Nqueens	20908	19148	8,41%
Game of life	15412	11320	26,55%
EPCC-Barrier	12316	8268	32,86%
EPCC-Ordered	14704	10992	25,24%
EPCC-Critical	13184	9420	28,54%
EPCC-Static For	14744	10992	25,44%
EPCC-Single	12768	8944	29,94%

Table 1 (in agreement with [2]) demonstrates the benefits of our scheme. For each of the applications we report the final size of the kernel object file after the appropriate flavor was chosen by the mapper and got linked in. We also report the size without activating the mechanism, i.e. simply utilizing the original (*Full*) runtime library. It can be easily seen that the reductions are quite significant in every case.

6 Conclusions

In this work we present a novel mapper module, a central piece that completes the idea of compiler-assisted adaptive OpenMP runtime support. The mapper has the ability to automatically select among different runtime library flavors to accompany a kernel, given a set of metrics that profile the application and are obtained by compile-time analysis, and a set of device-specific decision rules which an implementor provides for a device. For the latter, we introduce a custom language (MAL) that is able to capture the logic of a decision flow diagram using a concise syntax. The mechanism is general and applicable to any OpenMP device. We are working in expanding the expressiveness of MAL, adding support for more devices, and examining the possibility of parametrized flavors, where parts of a flavor can be tuned at compile time.

References

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

[2] S. N. Agathos and V. V. Dimakopoulos, "Adaptive OpenMP Runtime System for Embedded Multicores," in *16th Int'l Conference on Embedded and Ubiquitous Computing (EUC-2018)*, Bucharest, Romania, Oct. 2018, pp. 174–181.

[3] S. N. Agathos, V. V. Dimakopoulos, A. Mourelis, and A. Papadogiannakis, "Deploying OpenMP on an embedded multicore accelerator," in *Proc. of*

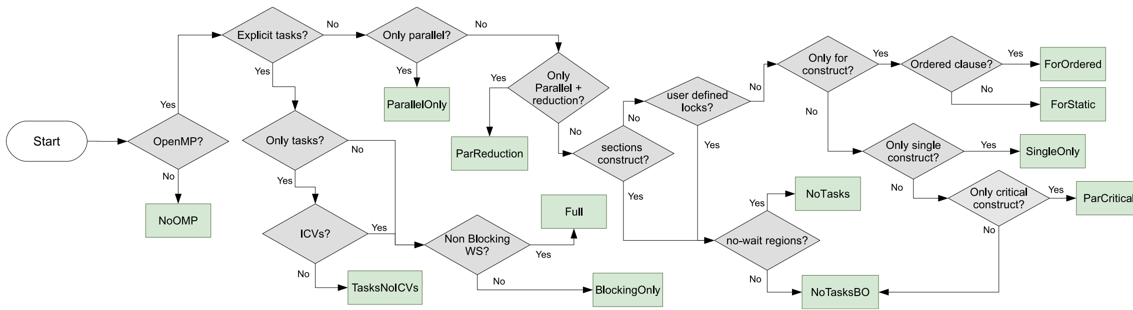

Figure 5: The full decision flowchart of the case study.

SAMOS'13, 13th Int'l Conf. on Embedded Computer Systems: Architectures, MOdeling and Simulation, Samos, Greece, Jul. 2013, pp. 180–187.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[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 Graphics Processing Units*, Boston, USA, Oct. 2007.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] OpenMP ARB, "OpenMP Application Program Interface V4.5," Nov. 2015.