SOMA: An OpenMP Toolchain For Multicore Partitioning

Emanuele Ruffaldi
Scuola Superiore Sant'Anna
Pisa, Italy
e.ruffaldi@sssup.it

Filippo Brizzi
Scuola Superiore Sant'Anna
Pisa, Italy
fi.brizzi@sssup.it

Giacomo Dabisias
Scuola Superiore Sant'Anna
Pisa, Italy
g.dabisias@sssup.it

Giorgio Buttazzo
Scuola Superiore Sant'Anna
Pisa, Italy
g.buttazzo@sssup.it

ABSTRACT

Advancements in multicore platforms enabled the development of complex embedded systems incorporating algorithms that were typically executed on high-performance workstations. Although many solutions exist today for supporting software development on multicore platforms, they rarely take timing constraints into account. This work presents a toolchain aimed at guaranteeing real-time constraints into parallel OpenMP code. This toolchain, called SOMA (static OpenMP multicore allocator), uses code profiling for estimating the multicore timing requirements and produces a static schedule for a set of parallel tasks. The toolchain is implemented using the source-to-source translation capabilities of CLang. Performance results are provided on a computer vision application.

CCS Concepts

•**Computer systems organization** → *Real-time languages;*
•**Software and its engineering** → *Translator writing systems and compiler generators;*

Keywords

OpenMP; Real-time; Multicore; Clang; Scheduling; Refactorization; Profiling

1. INTRODUCTION

The last years have seen the transition from single core architectures towards multicore architectures in the desktop and server environments and lately also in small devices as smartphones, tablets and embedded platforms. High Performance Computing (HPC) has been able to follow this trend studying advanced algorithms and building libraries to exploit such an additional computational power.

On the other hand, real-time systems, despite their massive and increasing use in many application domains as computer vision, robotics, simulation, video encoding, lack a general methodology to exploit more than one single computing core. While in the literature there exist many examples of multi-core real-time scheduling algorithms, there is a limited availability of multicore tools in this area. This is a serious limitation considering that real-time application requirements are continuously growing.

This paper presents a framework aimed at supporting the parallelization of code while taking timing constraints into account. This objective is pursued by starting from the paradigm of the OpenMP [7] annotation and transforming the program in a way that it can support multicore static scheduling based on virtual partitioning [5].

The SOMA framework is organized as a toolchain that operates a source-to-source transformation for improving the schedulability of parallel code moving from the run-time best effort of OpenMP implementations to a profile based static scheduling. The input source code is first analyzed, profiled, and then refactored to generate a statically specified schedule that is executed by means of a run-time support.

In the practice of homogeneous multicore development there are several libraries or languages that allow taking advantage of the available computing power [19]. In the context of C/C++ development, OpenMP [7] allows to easily transform a sequential application to a parallel one without requiring an explicit thread management. The OpenMP approach is interesting also for research and it has been extended for supporting heterogeneity, as OpenMP 4 [4]. What is typically missing in OpenMP implementations is the management of timing constraints, for programming soft or hard real-time applications [20].

While the OpenMP run-time allocates tasks to a threading pool with one thread per core, this work proposes to analyze the workflow and timing of an OpenMP application for creating static schedules. Several theoretical studies and algorithms have been proposed in the literature for providing solutions for scheduling tasks on multi-core architectures [8], typically distinguished between global and partitioned approaches. In global algorithms the highest priority task is

SAC 2016,April 04-08, 2016, Pisa, Italy

DOI: http://dx.doi.org/10.1145/2851613.2851720

taken from a single queue and allocated to a core, while in partitioned schemes the task is first allocated to a core and then scheduled. This work follows the approach of virtual partitioning [5] that produces a partitioned schedule from the precedence graph of an application expressed as a direct acyclic graph.

The adaptation of an OpenMP application to a custom schedule requires code analysis and source-to-source transformation. These functionalities can be found in research frameworks such as the Mercurium [2] compiler of OMPSS [9] or the Rose Compiler [18]. For the present work, instead, the Clang compiler of the LLVM framework [16] has been chosen. Clang provides a standard-compliant front-end for the C, C++ and Objective-C programming languages, relying on LLVM as back-end. In addition it is designed to be highly compatible with GCC, having a similar command line interface and sharing many flags and options. Four main reasons guided this choice:

1. It has proven to be faster and less memory consuming in many situations[1];
2. It has a modular library based architecture allowing the programmer to easily embed Clang's functionalities inside its code;
3. It allows performing code analysis, information extraction and, most important, source-to-source translation, not available in GCC at the time of development;
4. It has a strong and usable implementation of the Abstract Syntax Tree (AST).

Given the previous premises, this work has the following contributions: (1) it proposes a scheduling mechanism for multicore application written in OpenMP and (2) presents a toolchain based on Clang for partitioning real-time OpenMP applications on multicore platforms.

The paper is structured as follows. Section 2 presents the related work. Section 3 illustrates the overall design of the system. Section 4 describes the scheduling algorithm. Section 5 presents the run-time support. Section 6 describes a case study and, finally, Section 7 states our conclusions.

2. RELATED WORK

There is some previously related work [14] which aims at scheduling directed acyclic task graphs onto a cluster of machines, taking into account execution anda data transfer times. In our case the the schedule is produced for a single parallel machine and no data transfer times are considerated since we are working in a shared memory system.

StarPu [1] is a nice example of a parallelization tool over heterogenous resources. It is a runtime layer that provides an interface unifying execution on accelerator technologies as well as multicore processors. It also provides a scheduler which can allocate task queues on single workers; some basic scheduling policies are also already predefined, but in our case we need a timing guarantee on the tasks deadline which is not present in this framework.

[1]http://clang.llvm.org/features.html#performance

Is is also important to mention RT-OpenMP [10] which is similar to SOMA, but it is more a theoretical work than a real implementation. In our case we tried to address a smaller OpenMP directive set to be able to use the Toolchain on actual C++ code.

An extention of OpenMP with new directives to support asynchronous parallelism has been develeped at the Barcelona Supercomputing Center (BSC) which is called OMPSS [9]. Asynchronous parallelism is enabled in OMPSS by the use data-dependencies between the different tasks of the program. The OpenMP task construct is extended with the in (standing for input), out (standing for output) and inout (standing for input/output) clauses to this end. They allow to specify for each task in the program what data a task is waiting for and signaling is readiness. OMPSS can also be understood as new directives extending other accelerator based APIs like CUDA or OpenCL. OMPSS has not been chosen because it is based on its own source-to-source translator and the development is limited to the BSC. The aim of this project instead is to interface the parallel capabilities to a community-driven system as Clang. This will allow to benefit from the main contributors of Clang and other plugins of Clang.

Lately appeared also automatic parallelization tools; these softwares allow to automatically transform a sequential code into an equivalent parallel one, like the Intel C++ compiler and Parallware [13]. Sadly none of these software implements the solutions provided by the literature to tackle the requirements of real-time system.

3. FRAMEWORK DESIGN

The SOMA toolchain employs source-to-source transformation as the foundation for hybrid static and run-time analysis. The analysis generates a static schedule for the threads to be parallelized over a multicore system obtained from the profiling of the target program. The result of the SOMA execution is a new source-code paired with a scheduling plan aimed at optimizing the execution on a machine with a specified number of physical cores.

The proposed transformation starts from an OpenMP annotated source: the code is transformed via source-to-source manipulation both for profiling instrumentation and for generation of the final program. OpenMP has been chosen because it has minimal annotation overhead, it is implemented in the GNU GCC compiler and being based on pre-processor directives allows ignoring pragmas when no OpenMP support is needed.

SOMA is organized as a toolchain in the sense that each step is supported by a separate tool that exchanges with the others XML files or instrumented source code, allowing the replacement or further analysis of the results.

The high-level structure of the toolchain is depicted in Figure 1 and includes the following steps: (1) Code Analysis, (2) Instrumentation for profiling, (3) Profiling, (4) Schedule generation, (5) Instrumentation for execution and (6) Run-time support.

In the figure boxes represent an operation while ellipses data such as C++ source code or XML files. The left branch of

the graph is devoted to profiling, first presenting the instrumentation and then profiling execution based on a given hardware configuration and application specific input data. The right branch is devoted to the instrumentation for execution producing the final application. The execution step at the bottom uses the computed schedule.

3.1 Code Analysis

The starting point of the toolchain is the OpenMP annotated code where OpenMP pragmas provide hints for parallelization. Although the current tool has been developed to analyze C++ source code, it can easily be modified to analyze C code, or even to annotation schemes different from OpenMP.

The first phase of the toolchain is the analysis of the source code with the aim of generating a graph-based representation of the program that is further analyzed by the following phases. This phase is obtained by analyzing the C++ Abstract Syntax Tree (AST) as it is made available from the compiler after the semantic phase. From the AST all the relevant information of each OpenMP pragma are extracted and inserted in the graph structure. Then the graph is translated into an XML description.

The graph produced from the first stage of the code analysis is associated to the code structure and it consists of a tree of OpenMP pragmas that are rooted at the functions that contain at least a pragma. Some pragma statements do contain other statements such as parallel, section or task, and also for loops have been analyzed for extracting the variable used and the type of loop.

This graph provides a partial view of the program execution due to two aspects. The first aspect is the code execution flow that is induced by the pragmas: the tree graph obtained from the code analysis has to be converted into an execution flow that explains how the parallel tasks branch and join thanks to the OpenMP barriers. A step of the code analysis takes the code graph and produces the execution graph for further analysis.

The second aspect is related to the fact that with the static analysis it is not possible to identify the execution paths and functions invoked by the program inside a given parallel task. This is the main reason why SOMA uses a hybrid approach: the profiling task is not only computing the timing but also the function dependencies at run-time.

In terms of implementation the source-to-source transformation for profiling and analysis are based on Clang. At the time of the preparation of this work OpenMP was present in a branch of Clang and it supported the parsing of OpenMP. A patch has been added to support custom clauses inside pragma declarations. This patch allows to add specific information to each task such as deadline, activation time and period.

3.2 Visualization

For debug purposes a tool for visualization has been developed. This tool processes the model graphs and produces visual representations thanks to the Graphviz software. Figure 2 shows the original code parallel structure after the profiling, displaying for each component execution time statistics.

3.3 Profiling

To produce a good schedule the framework needs information about the tasks, in particular their computation times. The best way of getting this information is to profile at runtime the sequential version of the input code. In general OpenMP profiling has been investigated with probing as in ompP [12], or via proposed changes to the API [15]. For the present work a custom profiler has been implemented measuring each pragma block and the function call dependences. By using the same source-to-source tool for profiling and final execution instrumentation, the custom profiler implementation is compact in the implementation, tailored to the regions of interest for SOMA, and more efficient than parsing extensive log files from other tools.

To be profiled the code needs to be instrumented; in the original code, calls to a run-time support are added to calculate the execution time of each task, track the caller id of each function and store them in a log file. Tasks can be nested in each other, hence the outermost tasks computation time includes the computation times of all its sub-tasks; in other words, the sum of all the tasks' computation times could exceed the total computation time of the program. To prevent this problem a method has been designed to enable each task to keep track of the computation time of its children in order to obtain its effective computation time. This method also allows to keep track of the caller identity of each pragma and function, which is always either another pragma or function.

At the end the profile log contains for every task the following information: (1) the total time of the task, from when it was activated since it terminates; (2) the time of all its nested tasks; (3) the identifier of the pragma or function that called the task; (4) in the case of For task, the number of iterations.

The previously instrumented code is executed several times. At each iteration the algorithm produces, for each function and pragma, their execution time and, in case of a *#pragma omp for*, the number of executed cycles. This data is gathered during each iteration and then the mean value of the execution time, executed loops and variance for each node is saved in a log file.

4. SCHEDULING

The problem of finding the best possible schedule on a multicore architecture is known to be an NP hard problem. Given N tasks and M computing cores, the problem consists of creating K, possibly lower than M, execution flows in order to assign each task to a single flow. Each flow represents a sequential execution that needs to be allocated to a core. To find a good solution a recursive algorithm has been developed which, by taking advantage of a search tree, tries to explore all possible solutions, pruning "bad" branches as soon as possible.

To limit the running time of the algorithm due to the high dimension of the search space, a timer has been added to

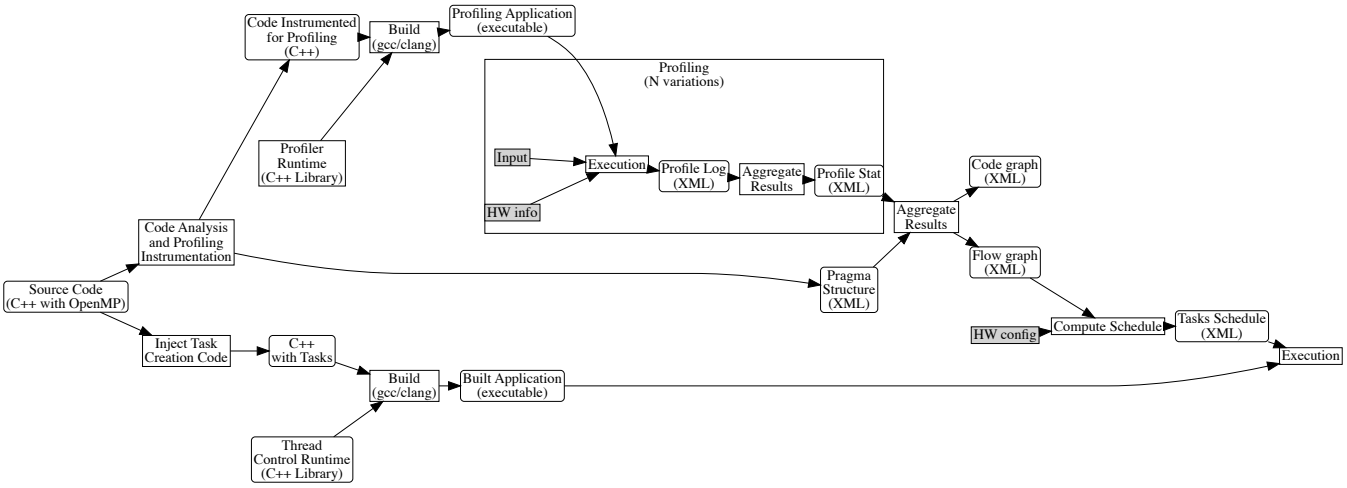


Figure 1: High-level structure of the SOMA toolchain: the diagram shows data elements (dashed) and operation (solid) of the toolchain. On the left the source code is entered, then the top branch deals with the instrumentation for profiling, profiling execution (in the large block), then preparation of the scheduling. The lower branch prepares the C++ code for the new scheduling by organizing the code in tasks, and finally the schedule plan (in XML) is passed to the new executable.

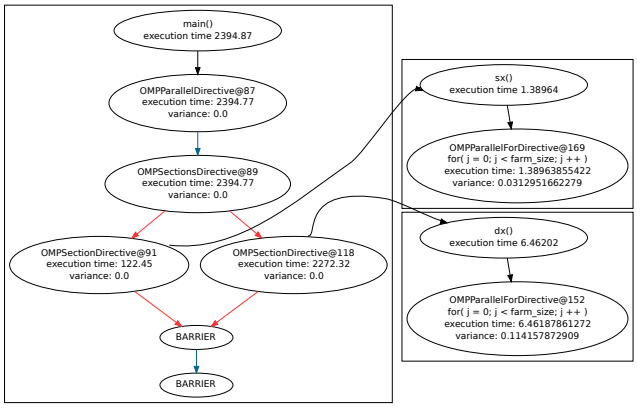


Figure 2: Parallelism graph. It is possible to see the dependencies of the parallel nodes jointly with their execution time, variance and function call hierarchy.

stop the computation after a certain amount of time given as input. At each level of the search tree a single task is considered; the algorithm inserts the task in each possible flow, and, if the partial schedule is feasible, this process continues until all the tasks have been considered. To check if the partial solution is feasible the algorithm calculates the cost of the actual solution and compares it with the best solution found so far, checks that the number of created flows is less than a predefined number and that the timer has not expired; if all these requirements are met, the branch continues its execution, otherwise it is pruned.

Once all the tasks are considered and all the requirements are fulfilled, the actual solution is compared with the optimal found so far and, in case updated, the best solution found is compared with the current one and possibly updated. Two solutions are compared using a simple heuristics that considers as a cost the computation time of tasks, flows and set of flows present in the solution. Given this metric a solution is better than another if it has a lower cost. Having a low flow cost means that the flows are well balanced; it is also important to notice that the algorithm is working in a breadth-first manner so that the number of flows is conservative, meaning that the lowest possible number is used to find the best solution.

There is a small variation of the algorithm when a task containing a *#pragma omp for* is encountered. In this case the algorithm tries to split the for loop as much as possible creating new tasks which are added to the task list.

It is worth noticing that solution could in principle not be schedulable, since the algorithm does not take care of precedence relations, but tries only to find the cheapest possible allocation. This could happen both if the algorithm checks all solutions finding the best one or if the algorithm terminates due to the timer limitations. In the second case it is possible to rerun the algorithm in order to check more solutions and possibly get a new schedule since the tasks are taken at random from a list.

To check the feasibility of a solution a second algorithm has been implemented following a modified version of the parallel Chetto&Chetto algorithm [6]. This algorithm works in two phases: the first one sets the deadline for each task, while the second one sets its arrival time. To set the deadlines, the algorithm sets the deadline of all the task with no predecessors to the deadline given in input; after that it recursively sets the deadline of all tasks having all their successors deadline set as the minimum of the difference between the computation time and the deadline of the successor.

In the second phase the algorithm sets the arrival time of every tasks with no predecessors to zero; after that it recursively sets the arrival time of all tasks, having the arrival time of all predecessors set as the maximum between all the

arrival time of the predecessors belonging to the same flow and the deadline of all the tasks that are assigned to a different flow. This is due to the following fact: let τ_j be a predecessor of τ_i, written as $\tau_j \rightarrow \tau_i$, with arrival time a_i and let F_k be the flow τ_i belongs to. If $\tau_j \in F_k$, then the precedence relation is already enforced by the previously assigned deadlines so it is sufficient to ensure that task τ_i is not activated before τ_j. This can be achieved by ensuring that:

$$a_i \geq a_i{}^{prec} = \max_{\tau_j \rightarrow \tau_i, \tau_j \in F_k} \{a_j\}.$$

If $\tau_j \notin F_k$, we cannot assume that τ_j will be allocated on the same physical core as τ_i, thus we do not know its precise finishing time. Hence, τ_i cannot be activated before τ_j's deadline d_j, that is:

$$a_i \geq d_i{}^{prec} = \max_{\tau_j \rightarrow \tau_i, \tau_j \notin F_k} \{d_j\}.$$

The algorithm checks that all the deadlines and arrival times are consistent and, if so, produces a scheduling.

It is important to notice that there is no guarantee that the produced schedule is the best possible one. The only guarantee is that is will fulfill all the timing deadlines. A parallel version of this algorithm has also been developed in order to check more solutions at the same time.

5. EXECUTION

This section discusses the instrumentation and run-time support for the program execution.

5.1 Instrumentation for execution

The framework needs to be able to isolate each task and execute it in the thread specified by the schedule; to do so new lines of code are added in the original code to transform the old pragmas in a collection of atomic independent concrete tasks.

In this phase the functions are not considered as tasks and they will not be affected by the instrumentation. This is due to the fact that functions have no parallel semantic themselves and they can be simply executed by the tasks that invoke them, without affecting the semantic and improving the efficiency. The idea of this phase is to transform each pragma block of code into a function that will be called by the designated thread. One possibility is to take the code of the pragma, remove it from the function where it is defined and put it in a newly generated function; this way may be feasible with Clang but it is very complicated because of the presence of nested pragmas.

The other possibility, used in the framework, is to exploit, once again, the property of the pragmas to be associated with a scope. In C++ it is possible to define a class inside a function if the class is contained in a scope. By exploiting this feature each pragma statement has been enveloped inside a new local class declaration; in particular, it constitutes the body of a function defined inside the new class that also embeds a reference to all the variables used in the task but declared outside.

In the case of a *for* pragma, the framework needs to perform some additional modifications of the source code. Usually a *for* is split into more threads in the final execution so that the *for* declaration has to be changed to allow the iterations to be scattered between different threads. Two variables are added to the *for* declaration: an identifier used to distinguish the different threads and the number of threads concurring in the execution of the *for*. After the definition of the class, at the end of the scope, the framework adds a piece of code that instantiates an object of the created class and passes it to the run-time support. The object will be collected by the designated thread that will invoke the custom function that contains the original code, running it.

5.2 Run-time support

The aim of the run-time is to instantiate and manage the threads and to control the execution of the tasks. In particular it must allocate each task on the correct thread and must grant the precedence constraints between tasks. The run-time must have a very low execution overhead in order to satisfy all the task's timing constraints. For this reason the run-time does no time consuming computations and all its allocation decisions are made based on what is written in the schedule. All the heavy calculations to decide the tasks allocation have been already done by the schedule algorithm before the program execution and the produced schedule is taken as input by the program. Figure 3 shows the structure of the run-time support.

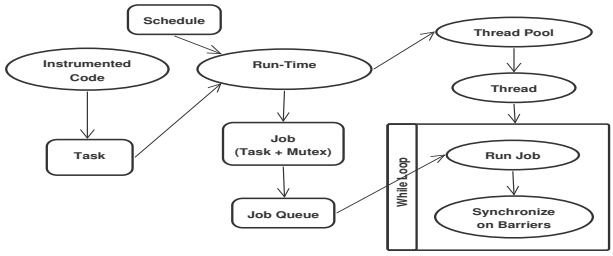


Figure 3: Run-time execution structure. The run-time support takes as input the schedule and a list of jobs to execute corresponding to the task previously obtained. It spawns the threads and allocates for each of them a queue where each job is inserted.

6. CASE STUDY

The test application consists of a face recognition algorithm in OpenCV that analyzes two videos which resembles the stereoscopic view of a person. To detect the faces a cascade multiscale detector has been used, a haar Detector. This algorithm uses Haar features and AdaBoost technique to detect the faces [17]. The application tries to recognize each face in each video frame and then prints a circle in the frame to locate it. The test videos have been produced in three different formats: 480p, 720p and 1080p; in each video two different people move around in a closed environment. For each different video format a separate schedule has been produced, given that the execution time of the tasks can fluctuate heavely when the data size changes. If

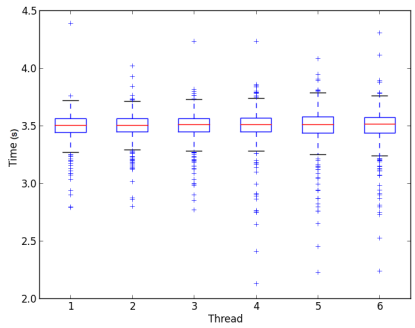
the execution flow is not deterministic the profiler will catch this behavior if execute nough times; a mean execution time is produced in order to take into account that possibility. With this method only small variations in the execution can be safely captured, but since the objective of this work is to work with real-time jobs, we can assume that there are no meaningful code branches. Figure 2 shows the parallel structure of the application together with the profiler information. This is not a hard-real time application for two reasons: first because in the case of missing a deadline no serious consequences occurs but mainly because the execution time for the delivering of an image depends on the data and it is not fixed. This last property is actually interesting because it allows to see how the framework manages and control non deterministic behavior.

The performance of SOMA has been analyzed in comparison to an OpenMP implementation with a varying number of available cores and problem size. The GNU GCC OpenMP implementation has been chosen as a reference due to the lower quality of the CLang implementation at testing time. The tests have been executed on a machine with an Intel Core i7-3930K, a 6-core CPU running at 3.20 GHz, with 12MB cache and 24GB RAM running Linux Ubuntu 13.04, Clang 3.2 and gcc 4.8.1.

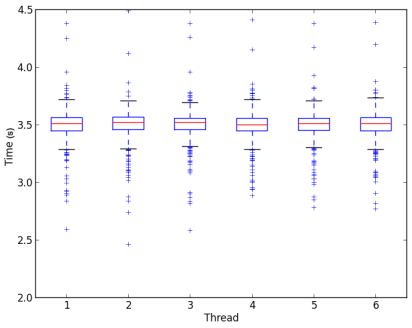
Table 1 presents the completion time of the various applications, while Table 2 shows their service time (gap between the delivery of a parsed image) with the variance. All these numbers have been calculated as an average of five identical executions.

The completion value of the OpenMP and SOMA executions are almost the same and this is a good result, meaning that the proposed framework does not introduce more delay than OpenMP, which represents the state of the art of parallel programming. What is even more important and interesting is that the variance of the service time in the SOMA scenario is always the lowest. This is a very important achievement for real-time requirements because it means that this approach can guarantee real-time constraints.

The considerations stated above are also confirmed by the comparison between 4(b) and 4(b) where service times are shown for each concurrent thread of the two applications. In the SOMA case we have not only a lower variance for each thread, but also that the mean service time varies less between different threads and the physiological outliers are more compact towards the mean value.



(a) OpenMP



(b) SOMA

Figure 4: Boxplot of the service time of each thread of the application for the 1080p image. Execution time on the Y axis, while number of threads (cores) in the X axis.

7. CONCLUSIONS

This presented a framework for automatic generation of parallel real-time code from an annotated sequential one. Code is manually annotated with OpenMP pragmas to specify parallel sections and their dependencies and is then inserted in the framework's toolchain. In a first step, the source code is profiled and information about each parallel component is extracted and used to create a custom, machine specific, scheduler. The annotated code is then instrumented to be able to run it with the new schedule. The framework has been tested with a soft real-time computer vision application demonstrating its capability of satisfying real-time requirements. The source code will be published online together with the paper.

There are some known issues in this framework when dealing with complex code structure and parallel components. The solution to this problems will be to substitute OpenMP with a custom set of pragma comments to be able to handle more scenarios and to add all kind of desired information inside the code. Anyhow OpenMP has been very helpful to test the feasibility of the approach. As a future work, we plan to extend the framework for handling hard-real time applications and source code with more sophisticated parallel structures. Examples of applications scenarios are the ones involving high-rate human-robot haptic interaction coupled with simulations [11, 3], or the case of autonomous driving vehicles in which real-time sensor fusion units are integrated with intention recognition modules for risk assessment: in this case there are real-time requirements for producing outputs within a bounded delay, and, at the same time, the structure of the computation is non-trivial due to the number of sensors and behavior models.

8. REFERENCES

[1] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and*

	Sequential	OpenMP		Soma	
	$T_{seq}[s]$	$T_c(6)[s]$	$\epsilon(6) = \frac{T_{seq}}{6T_c(6)}$	$T_c(6)[s]$	$\epsilon(6) = \frac{T_{seq}}{6T_c(6)}$
480p	750	133	0.94	134	0.93
720p	3525	627	0.94	629	0.93
1080p	8645	1536	0.94	1539	0.94

Table 1: Completion time of the programs at the variation of the video quality.

Quality	Sequential		OpenMP		Soma	
	mean	variance	mean	variance	mean	variance
480p	0.2823	0.1984	0.3034	0.0016	0.3065	0.0005
720p	1.3263	0.1968	1.4257	0.0125	1.4117	0.0065
1080p	3.2524	0.1567	3.4901	0.0256	3.4990	0.0185

Table 2: Average service time in seconds for the various video qualities and the different algorithms. OpenMP and SOMA applications both executed on six threads.

Experience, 23(2):187–198, 2011.

[2] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta. Nanos mercurium: a research compiler for openmp. In *Proceedings of the European Workshop on OpenMP*, volume 8, 2004.

[3] M. Bergamasco, S. Perotti, C. Avizzano, et al. Fork-lift truck simulator for training in industrial environment. In *Emerging Technologies and Factory Automation, ETFA. The 10th IEEE International Conference on*, 2005.

[4] O. A. R. Board. Openmp application program interface 4.0, 2013.

[5] G. Buttazzo, E. Bini, and Y. Wu. Partitioning real-time applications over multicore reservations. *Industrial Informatics, IEEE Transactions on*, 7(2):302–315, 2011.

[6] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on software engineering*, (10):1261–1269, 1989.

[7] L. Dagum and R. Enon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[8] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (CSUR)*, 43(4):35, 2011.

[9] A. Duran et al. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[10] D. Ferry, J. Li, M. Mahadevan, K. Agrawal, C. Gill, and C. Lu. A real-time scheduling service for parallel tasks. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 261–272. IEEE, 2013.

[11] A. Frisoli, L. F. Borelli, C. Stasi, M. Bellini, C. Bianchi, E. Ruffaldi, G. Di Pietro, and M. Bergamasco. Simulation of real-time deformable soft tissues for computer assisted surgery. *International Journal of Medical Robotics and Computer Assisted Surgery*, 1(1):107–113, 2004.

[12] K. Fürlinger and D. Skinner. Performance profiling for openmp tasks. In *Evolving OpenMP in an Age of Extreme Parallelism*, pages 132–139. Springer, 2009.

[13] H. Gomez-Sousa, M. Arenaz, O. Rubinos-Lopez, and J. A. Martinez-Lorenzo. Novel source-to-source compiler approach for the automatic parallelization of codes based on the method of moments. In *Antennas and Propagation (EuCAP), 2015 9th European Conference on*, pages 1–6. IEEE, 2015.

[14] U. Hönig, W. Schiffmann, and L. Rechnerarchitektur. A comprehensive test bench for the evaluation of scheduling heuristics. In *Proc. 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS04)*, pages 437–442, 2004.

[15] M. Itzkowitz et al. An openmp runtime api for profiling. *OpenMP ARB as an official ARB White Paper available online at http://www. compunity. org/futures/omp-api. html*, 314:181–190, 2007.

[16] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[17] R. Lienhart and J. Maydt. An extended set of haar-like features for rapid object detection. In *Image Processing. 2002. Proceedings. 2002 International Conference on*, volume 1, pages I–900. IEEE, 2002.

[18] D. Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.

[19] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel. Mapping on multi/many-core systems: survey of current and emerging trends. In *Proceedings of the 50th Annual Design Automation Conference*, page 1. ACM, 2013.

[20] R. Vargas, E. Quinones, and A. Marongiu. Openmp and timing predictability: a possible union? In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 617–620. EDA Consortium, 2015.