# AUTOMATIC PARALLELIZATION OF SEQUENTIAL SPECIFICATIONS FOR SYMMETRIC MPSOCS

Fabrizio Ferrandi, Luca Fossati, Marco Lattuada, Gianluca Palermo, Donatella Sciuto and Antonino Tumeo
*Politecnico di Milano, Dipartimento di Elettronica e Informazione. Via Ponzio, 34/5, 20133 Milano, Italy. Ph: +39-02-2399-4009. Fax: +39-02-2399-3411*[*]
{ferrandi,fossati,lattuada,gpalermo,sciuto,tumeo}@elet.polimi.it

**Abstract:** This paper presents an embedded system design toolchain for automatic generation of parallel code runnable on symmetric multiprocessor systems from an initial sequential specification written using the C language. We show how the initial C specification is translated in a modified system dependence graph with feedback edges (FSDG) composing the intermediate representation which is manipulated by the algorithm. Then we describe how this graph is partitioned and optimized: at the end of the process each partition (cluster of nodes) represents a different task. The parallel C code produced is such that the tasks can be dynamically scheduled on the target architecture; this is obtained thanks to the introduction of start conditions for each task. We present the experimental results obtained by applying our flow on the sequential code of the ADPCM and JPEG algorithms and by running the parallel specification, produced by the toolchain, on the target platform: with respect to the sequential specification, speedups up to 70% and 42% were obtained for the two bebchmarks respectively.

**Keywords:** Partitioning, Clustering, Automatic parallelization, thread decomposition, compilers for embedded systems, MPSoCs, FPGA.

## 1. INTRODUCTION

The technology trend continues to increase the computational power by enabling the incorporation of sophisticated functions in ever-smaller devices. However, power and heat dissipation, difficulties in increasing the clock frequency, and the need for technology reuse to reduce time-to-market push towards different solutions from the classic single-core or custom technology. A solution that is gaining widespread momentum consists of exploiting the

[*]Research partially funded by the European Community's Sixth Framework Programme, hArtes project.

inherent parallelism of applications, executing them on multiple off-the-shelf processor cores. Having separate cores on a single chip allows better usage of the chip surface, reduces wire-delay and dissipation problems and it provides more possibilities to exploit parallelism.

Unfortunately, the development of parallel applications is a complex task. Parallel programming is largely dependent on the availability of adequate software tools and environments and developers must contend with problems not encountered during sequential programming, namely: non-determinism, communication, synchronization, data partitioning and distribution, load-balancing, heterogeneity, shared or distributed memory, deadlocks, and race conditions.

This work tries to overcome some of these problems by proposing an approach for automatic parallelization of sequential programs. It focuses on a complete design flow, from the high level sequential C description of the application to its deployment on a multiprocessor system-on-chip prototype. In a first phase the sequential code is partitioned in tasks with a specific clustering algorithm that works on the System Dependence Graph (SDG). In a second phase the resulting task graph is optimized and the parallel C code is generated. The backend can produce OpenMP compliant code for functional validation on the host machine and C code that is runnable on multiprocessor system-on-chip architectures with a minimal operating system layer, as required by an embedded system. In both cases a mechanism to generate dynamically schedulable tasks has been defined. Through run-time evaluation of boolean conditions it is possible to determine the specific execution flow of the program and if a specific tasks needs spawning. Thanks to the mechanism implemented, tasks will be created as soon as only the real data dependences of the specific execution flow are satisfied, leading to an efficient exploitation of the underlying hardware.

The remainder of this paper is organized as follows: Section 2 gives an overview of the current state of the art on automatic parallelization techniques; Section 3 presents the target architecture used to execute the benchmarks. Section 4 introduces the flow we implemented focusing on the different steps which compose the partitioning and merging phases. Section 5 shows the numerical results obtained by the parallelization of the JPEG and ADPCM encoding algorithms and finally Section 5 concludes the paper.

## 2. RELATED WORK

MultiProcessor systems are becoming common, not only in the high performance segment, but also in the consumer and embedded ones. Developing programs for these new architectures is not easy: the developer needs to correctly decompose the application in order to enhance its performance and to exploit the multiple processing elements at his disposal. Sometimes, it is bet-

ter to rewrite a sequential description rather than to try porting it on complex, and often very different, architectures. Several strategies to ease the life of the developers, through partially automatization of the porting process, have been proposed; they are mainly related to two different approaches. The first one uses problem-solving environments which generate parallel programs starting from high level sequential descriptions. The other relies on machine independent code annotations. Our work adopts the first approach.

The parallelization process can be decomposed in several different steps. The initial specification needs to be parsed in an intermediate graph representation. This representation, which is used to explore the available parallelism, gets partitioned. After partitioning, an initial task graph is obtained. A task graph is a Directed Acyclic Graph (DAG) where each node describes a potential parallel code block. In a multiprocessor system it is necessary to allocate each task to one of the available processors. This allocation is usually realized through a two-step process: clustering and cluster-scheduling (merging). Many are the works related to the partitioning of the initial specification adopting specific intermediate representations. Among them, Girkar et al. [2] propose an intermediate representation, called Hierarchical Task Graph (HTG), which encapsulates minimal data and control dependence and which can be used for extraction of task level parallelism. Much of their work focuses on simplification of the conditions for execution of task nodes. Luis et al. [7] extend this work by using a Petri net model to represent parallel code, and they also apply some optimization techniques to minimize the overhead due to explicit synchronization.

Newburn and Shen [8], instead, present a complete flow for automatic parallelizazion through the PEDIGREE compiler. Their tool uses the Program Dependence Graphs (PDG) as intermediate representation and applies an heuristic to create overlapping inter-dependent threads. Their approach searches the PDG for control equivalent regions (i.e., groups of statements depending from the same control conditions) and then partition these region with a bottom up analysis. The resulting task graph is finally scheduled on subsets of processors of a shared memory multiprocessor architecture.

The clustering and merging phases have been widely discussed. Usually, these two phases are addressed separately. Well known deterministic clustering algorithms are dominant sequence clustering (DSC) by Yang and Gerasoulis [12], linear clustering by Kim and Browne [6] and Sarkar's internalization algorithm (SIA) [9]. On the other side, many researches explored the cluster-scheduling problem with evolutionary algorithms [4, 11]. An unified view is given by Kianzad and Bhattacharyya [5], who modified some of the deterministic clustering approaches introducing probability in the choice of elements for the clusters, and proposed an alternative single step evolutionary approach for both the clustering and cluster scheduling aspects.

Our approach starts from an intermediate representation which is not hierarchical like HTGs and PDGs, but instead flattens out all the dependence information at the same level. This creates bigger structures but gives the opportunity to extract more parallelism as it allows more complex explorations. Moreover, although our flow starts from the analysis of control-equivalent regions, we effectively partition them working on the data flow. This initial clustering is then optimized in order to create thread of homogeneous size, but we don't need any sort of static scheduling mechanism as we relie on dynamic start conditions.

## 3. TARGET ARCHITECTURE

The target architecture of the presented approach is a symmetric shared memory multiprocessor system-on-chip (MPSoC); we chose those systems because they are composed by many processing units, thus it is necessary to use applications composed by many concurrent tasks in order to exploit their processing power. We choose to test the produced code on a specific MPSoC prototype developed on Field Programmable Gate Array (FPGA), the CerberO architecture [10].

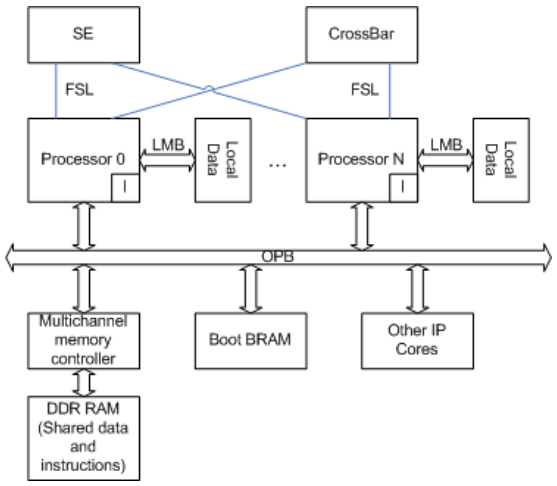


*Figure 1.* The target architecture of the presented approach

CerberO, shown in Figure 1, is realized connecting multiple Xilinx MicroBlaze softcores on a single shared Coreconnect On-Chip Peripheral Bus (OPB). Shared IPs and the controller for the shared external memory reside on the OPB bus. Each MicroBlaze in the system is connected to a local memory through the Local Memory Busses (LMB) for private data. The shared instructions and data segments of the applications reside on the (slower) external DDR mem-

ory, while private data are saved in the local memories. The addressing space of each processor is partitioned in two parts: a private part and a shared part. The private space is accessible only by the local processor, while the shared one is equally seen by all components of the system. Instructions are cached by each processor, while data are explicitly moved from the shared to the fast, low latency private memory.

Synchronization mechanisms are provided through a dedicated module, the Synchronization Engine (SE), connected to each processor through the Fast Simplex Link (FSL) point to point connections. The SE is a configurable, centralized hardware lock/barrier manager that permits atomic accesses to shared memory locations and shared peripherals.

On top of the CerberO architecture, we developed a thin operating system (OS) layer that permits to dynamically schedule and allocate threads.

# 4. PARALLELIZATION

This section introduces our flow to automatically produce multi-threaded programs using as input a sequential specification written in the C language. The toolchain manages the partitioning of the sequential description and generates code that can be dynamically scheduled on the target architecture using specific boolean task start conditions.

Figure 2 provides an overview of the entire flow. The sequential C code is compiled with a slightly modified version of the *GNU C Compiler* (GCC) version 4.0 and the internal structures generated by the compiler are dumped. From these structures, our tool suite *PandA* creates an abstract representation of the program in terms of several graphs describing data and control dependence. The C to C partitioning algorithm works on a modified system dependence graph (SDG). We define the SDG as a graph in which both data dependence and control dependence are represented for each procedure. This graph gets clustered and optimized, and the resulting task graph is then converted back in parallel C code by the specific backend. This parallel C code can finally be compiled for the target MPSoC architecture.

## FSDG Creation

In this phase of the process *PandA* parses the dump of the intermediate representation of the modified *GCC* 4.0 compiler and creates a data structure containing all the relevant information expressed in the initial specification. From the version 3.5/4.0 of the GCC compiler the front-ends parse the source language producing GENERIC trees, which are then turned into GIMPLE. GENERIC and GIMPLE are language independent, tree based representations of the source specification [1]. Although GIMPLE has no control flow struc-

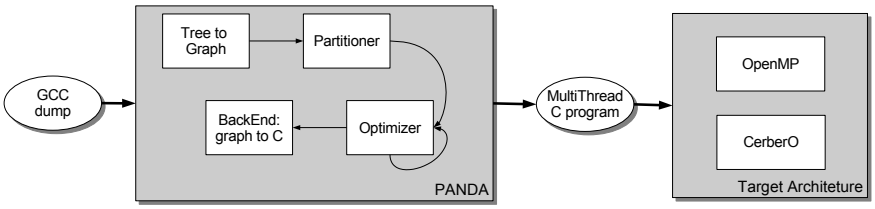


*Figure 2.* Overview of the toolchain

tures, GCC also builds the control flow graph (CFG) to perform language independent optimizations.

The GCC analysis and the GIMPLE parsing correspond to the first step performed by the PandA framework to analyze the input sequential program; several graph structures describing control and data dependencies in the program are then derived. Each graph is a different view of the dependencies among the operations of the initial specification.

In particular, the proposed partitioning algorithm executes a post-processing analysis on the dependencies represented by a SDG [3] extended by introducting feedback edges (FSDG). A sample FSDG is shown in Figure 3. Vertices are statements (round and square boxes) or predicate expressions (rhombus boxes). Grey solid edges represent data dependencies and describe flow of data between statements or expressions. Black edges represent control dependence, and express control conditions on which the execution of a statement or expression depends. Black dashed edges represent both control and data dependencies. Grey dashed edges represent feedback edges, which makes possible to distinguish nodes belonging to loops from other nodes. Finally, the entry node represents the entry to the procedure. It is worth noting that all the loops are converted in do-while loops, since this code transformation simplifies the management of the exit condition. This graph also allows the recognition of control-equivalent regions, which are groups of nodes that descend from the same condition (True or False) of a father predicate node.

## Partitioning algorithm

The partitioning phase uses the FSDG as defined above as its input. The first step of the algorithm analyses feedback edges and generates partitions of nodes from the initial FSDG: one for each loop and one grouping all the nodes not in loops. Thanks to this procedure, parallelism can be extracted also for

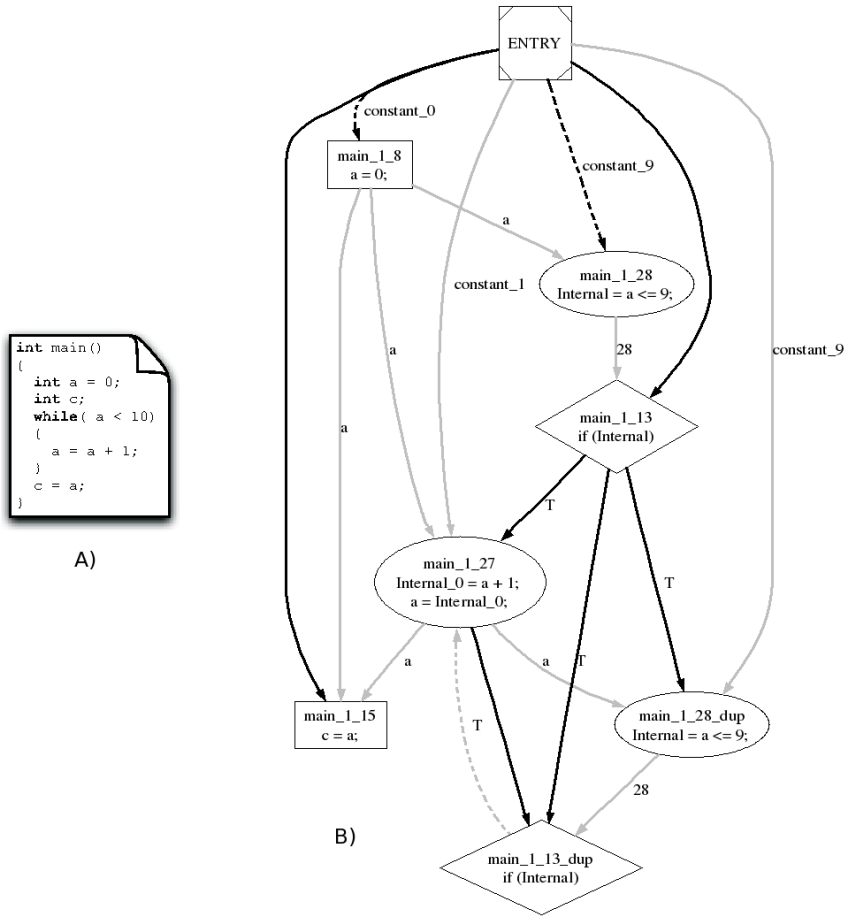


*Figure 3.* Example of an FSDG graph

the code inside the loops, instead of considering loops as atomic entities. The following steps of the algorithm consider these partitions separately, one at a time, and try to extract only the parallelism contained in each of them.

After identifying the loop partitions, the algorithm performs an analysis of the control edges in order to recognize the *control-equivalent* regions. Statement nodes descending from the same branch condition (True or False) of a predicate node are grouped together, forming a single control-equivalent region, as in Figure 4, A. The procedure runs as long as all the predicate nodes are analysed.

Since the nodes in each control-equivalent region can only be data dependent among each other, each region is a potential candidate to form a parallel

task. For each control-equivalent region data dependence analysis is then executed, grouping together nodes that are inter-dependent and, hence, must be serially executed, like in Figure 4, B. The analysis starts from a generic node in a control-equivalent region with a depth-first exploration. A node is added to the cluster being formed if it is dependent from one and only one node or if it is dependent from more than one node, but all its predecessors have already been added to the current cluster. Otherwise, the cluster is closed and the generation of a new set starts. These operations are iterated until all the nodes in the control-equivalent partition are added to a set.

The final result of this procedure is a clustered FSDG graph, which represents a potential task graph, Figure 4, C. Each partition of the clustered graph represents a single basic block of instructions, with none, or minimal inter-dependence. Clusters that not depend on each others represent blocks of code that can potentially execute in parallel. Edges among clusters express data dependences among blocks of code, thus the data represented by in-edges of a partition must be ready before the code in that partition can start.

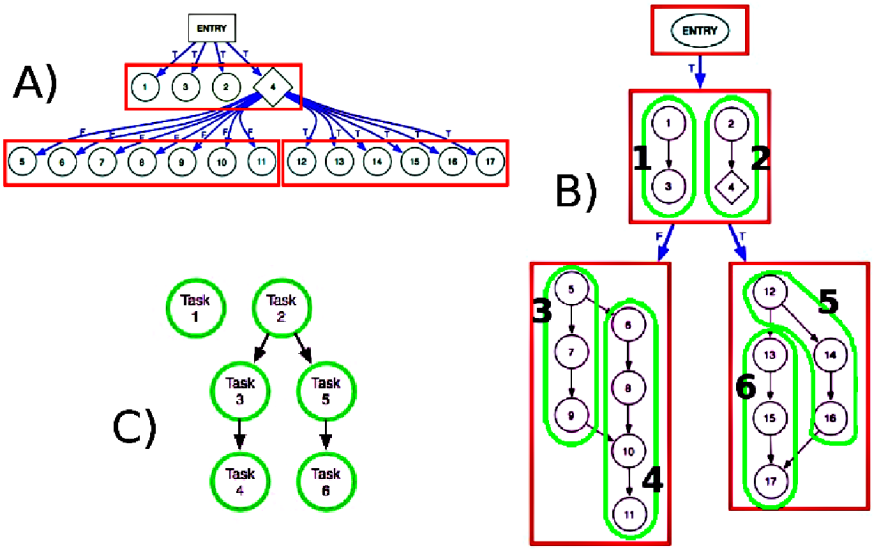


*Figure 4.* Example of clustering: A) Control equivalent regions gets analyzed, B) Control Equivalent Regions gets partitioned, C) Corresponding Task Graph

## Optimizations

Even if the prototyping platform has just a small operating system layer with little overhead due to thread management, shared memories must be atomically accessed in order to create the task structures and verify which processing elements are free. The partitioning phase, explained in Section 3, tends to produce too many small clusters, with the effect that the overhead of task management could be higher than the advantages given by concurrent execution. Thus, we decided to introduce an optimization phase aimed at grouping clusters together. Two different techniques are used: optimizations based on control dependencies and optimizations based on data dependencies.

**Control** structures, such as the `if` clause, must be executed before the code situated beneath them, otherwise we would have speculation: it seems, then, reasonable to put the control statement and the instructions which depend on it in the same task, in particular if there are no other nested control clauses. Another optimization consists in grouping the `then` and `else` clauses in the same cluster: they are mutually exclusive, the parallelism is not increased if they are in separate clusters. If a control clause does not get optimized, as it could happen if there are many nested control statements, it is replicated in each one of the clusters containing control dependent instructions. This means that the only type of dependencies remaining between different clusters are *data dependencies*, making the subsequent management of the scheduling easier.

**Data** dependent clusters can be joined together to form a bigger cluster; the candidates for joining are those clusters containing a number of nodes (instructions) smaller than $n$; this number roughly represents the overhead due to the management of a task. When a small cluster is encountered, the algorithm tries to join it with the successor (say that a task *b* is the successor of *a* if *b* has a control or data dependence on *a*); this operation is successfully carried out when all the data dependences on edges exiting from *a* have the same target cluster *b*. These steps are repeated until no more clusters are joined or no more clusters smaller than $n$ exist.

## Task creation

This part of the partitioning flow is used to translate the final clustered FSDG into specific data structures effectively representing the tasks, their input/output parameters and the relationships among them. The first step consists in the identification of the task variables: the edges coming from the special node `ENTRY` are associated with global variables; the edges entering in a cluster represent the input parameters of the task, while the outgoing edges the output

parameters. Finally, the edges whose both source and destination nodes are contained in the same cluster form the local variables of the task. Note that, of course, it may happen that two different edges are associated with the same variable: in this case just one variable is instantiated.

The next operation consists in the computation of the start conditions for each task; these conditions enable *dynamic scheduling* of the tasks. Through dynamic scheduling the application can decide at runtime if a task should start or not. Consider, for instance, that the application needs to start a task such as the one in Figure 5. The arrows pointing out of each variable indicate that they use a value produced by other tasks. To safely start the new task the application should wait that `a`, `b` and `c` are written by the preceding tasks. Hence the tool chain would have to determine a static scheduling and organize the execution flow in order to guarantee that all the preceding tasks have correctly written their values to cover for each possible execution path. Actually not all three variables are used together, using `a` and `b` implies that `c` is not used and vice-versa. A static scheduling would limit the exploitable parallelism, since the new tasks could not be overlapped either to the tasks producing `a` and `b` or to the task producing `c`. However, when the value of the condition `C1` is determined, it is decided which branch is going to be executed: in case the *else* branch is taken, the new tasks only need to wait for the task which produces `c`, otherwise for the tasks which produce `a` and `b` (this, of course, works if `a`, `b` and `c` are produced by different tasks). So, if we allow the application to precompute the value of the condition, it could fully take advantage of the support for dynamic scheduling of the target architecture and, effectively, start the new task as soon as only the true data dependencies of the specific execution flow are satisfied.

Taking these considerations into account, we enabled our Task Creator to generate, for each task, a specific start condition related to the runtime execution path of the application. A valid start condition for a task has to satisfy the following points: (1) a task must be started only once (i.e. only one of the predecessors of the task can start it) and (2) all the parameters necessary for a correct execution of the task must have already been computed when the condition evaluates to true. To enforce the first point we use a simple boolean variable which is set to true when the task starts; the second point is much more complicated since, depending on the execution path which will be taken, different variables are used.

To generate the start condition, the algorithm initially explores all the input parameters belonging to the task to be started. Parameters used in the same control region (i.e. all in the true or false branch) are put in an `and` condition (all of them must be ready if that control region is going to be executed). All the resulting “and” expressions (one for each control region) are joined by an “or” operator.

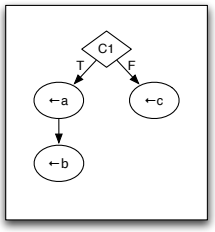


*Figure 5.* A task, depending on the particular execution flow, may need different parameters.

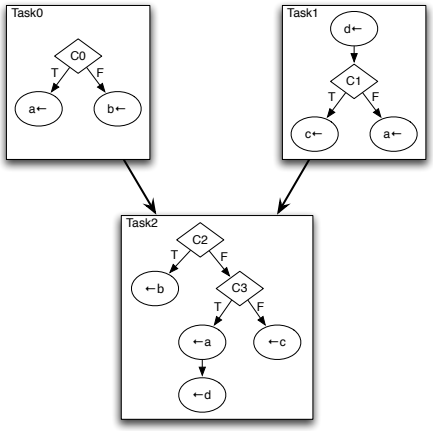


*Figure 6.* Typical sequence of tasks: depending on the value of *C2* and *C3*, Task2 uses different parameters

The second step consists in the exploration of the preceding tasks, the ones which have an edge entering in the current task, looking for the locations where the parameters, input to the task to start, are written. In case there are more control flows from which a parameter is written, all the corresponding paths are joined in an "or" expression. Using the diagram in Figure 6 as example, we would have the condition for *Task2*:

$$C2 \cdot \gamma\left(b\right) + \neg C2 \cdot \left[C3 \cdot \left(\gamma\left(a\right) \cdot \gamma\left(d\right)\right) + C3 \cdot \gamma\left(c\right)\right] \tag{1}$$

$\gamma\left(x\right)$ identifies all the possible paths, in the preceding tasks, which compute $x$; in the example: $\gamma\left(a\right) = C0 \cdot T0 + \neg C1 \cdot T1$, where $T0$ and $T1$ are boolean variables equal to true if Task0 and Task1 has already ended. After computing all the necessary $\gamma\left(x\right)$ functions we need to complete the start condition by indicating that Task2 can start only if it has not started yet, so we put Equation 1 in "and" with $\neg T2_s$ which is true if Task2 has already started. Since the resulting condition is usually long, but contains many redundant elements, we use BDDs (Binary Decision Diagrams) to reduce its complexity. The condition is inserted at the end of both Task0 and Task1: the one which ends last will evaluate the condition to `true` and it will actually launch the execution of Task2.

The last phase of the flow is the generation of the final parallel C code starting from the representation of the tasks created in the previous steps. This work is done by a specific C Writer backend, that needs to produce code in a syntax compliant with the primitives supported by the target architecture. The

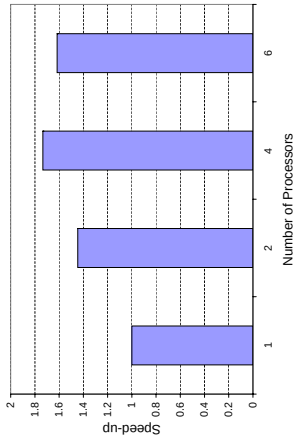


*Figure 7.* ADPCM performance

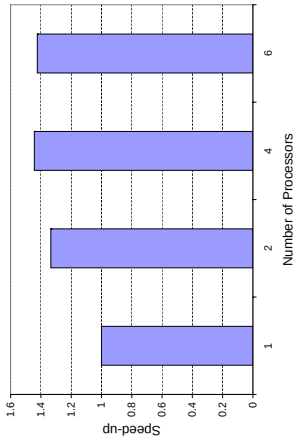


*Figure 8.* JPEG performance

backend can produce both OpenMP compliant code for functional validation and code runnable on the target platform for final implementation.

## 5. EXPERIMENTAL EVALUATION

The effectiveness of our partitioning flow was verified running the parallelized programs on the CerberO architecture. Execution time on different runs, using a different number of processors was measured. When analyzing the results, it must be taken into consideration that the execution time is affected not only by the degree of parallelism extracted, but also by the overhead introduced by the thread management routines of the underlying OS layer. For this reason, in order to perform a fair comparison, the original sequential program was slightly modified in order to execute it on top of the OS layer.

Figure 7 shows the speedup obtainable by running the ADPCM algorithm on a different number of processors. The maximum speedup is obtained when four processors are used, with a performance roughly 70% higher than the results on the single processor platform. With more than four processors the execution speed starts lowering. This behavior is due to the fact that the parallelized program contains at most four threads which have no inter-dependences and can, hence, run in parallel; using more processors does not increase the parallelism, but it does increase the synchronization overhead. The speed ups obtained are similar to the average results shown in [8] but, while its authors exploit also fine grained ILP parallelism, our target architecture adopts much simpler processor cores.

Figure 8 represents the speedup obtained by parallelizing part of the JPEG algorithm. The most computationally intensive kernels of the algorithm, the

RGB-to-YUV color space conversion and the 2D-DCT, have been parallelized. The RBG-to-YUV and 2D-DCT algorithms account for the 70% of the execution time of the sequential code on our target platform, while the remaining 30% comprises the reading of the input and the writing of the output image which are not parallelizable. As for the ADPCM algorithm, the maximum parallelism extracted by our tool chain is four, so using more than four processors leads to performance degradation for unnecessary synchronization operations and more load on the shared bus and memories of the target architecture. For the whole JPEG algorithm the maximum speedup reached is 42%.

## 6. CONCLUDING REMARKS

This paper presented our design flow for automatic parallelization of a sequential specification targeting a homogeneous MPSoC prototype. The main contributions of this paper can be summarized as follows: (1) it proposes a complete design flow from sequential C to parallel code runnable on homogeneous multiprocessor systems, (2) it describes a partitioning algorithm to generate parallel code transforming all control dependencies in data dependencies from sequential C and, finally, (3) it introduces a dynamic task scheduling model with specific start conditions to spawn the parallel threads of the application, without requiring complex operating system support by the target architecture. The flow has been applied to several standard applications, and the results obtained with the ADPCM and the JPEG algorithms on a MPSoC prototype on FPGA show promising levels of parallelism extracted, with speedups up to 70% and 42% respectively.

## REFERENCES

[1] GCC, the GNU Compiler Collection. http://gcc.gnu.org/.

[2] M. Girkar and C.D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):166–178, March 1992.

[3] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Not.*, 39(4):229–243, 2004.

[4] E.S.H. Hou, N. Ansari, and H. Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 5:113–120, 1994.

[5] V. Kianzad and S.S. Bhattacharyya. Efficient techniques for clustering and scheduling onto embedded multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 17(17):667–680, July 2006.

[6] S.J. Kim and J.C. Browne. A general approach to mapping of parallel computation upon multiprocessor architectures. In *Int. Conference on Parallel Processing*, pages 1–8, 1988.

[7] J.P. Luis, C.G. Carvalho, and J.C. Delgado. Parallelism extraction in acyclic code. In *Parallel and Distributed Processing, 1996. PDP '96. Proceedings of the Fourth Euromicro Workshop on*, pages 437–447, Braga, January 1996.

[8] C.J. Newburn and J.P. Shen. Automatic partitioning of signal processing programs for symmetric multiprocessors. In *PACT '96*, 1996.

[9] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, 1989.

[10] A. Tumeo, M. Monchiero, G. Palermo, F. Ferrandi, and D. Sciuto. A design kit for a fully working shared memory multiprocessor on FPGA. In *Proceedings of ACM GLSVLSI'07 – Great Lakes Symposium on VLSI*, March 11-13 2007.

[11] W.K. Wang. Process scheduling using genetic algorithms. In *IEEE Symposium on Parallel and Distributed Processing*, pages 638–641, 1995.

[12] T. Yang and A.Gerasoulis. Dsc: scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5:951–967, 1994.