MAPS: An Integrated Framework for MPSoC Application Parallelization

J. Ceng, J. Castrillon, W. Sheng,
H. Scharwächter, R. Leupers,
G. Ascheid, H. Meyr
Integrated Signal Processing Systems
RWTH Aachen University, Germany
maps@iss.rwth-aachen.de

T. Isshiki, H. Kunieda
Tokyo Institute of Technology
Tokyo, Japan
isshiki@vlsi.ss.titech.ac.jp

ABSTRACT

In the past few years, MPSoC has become the most popular solution for embedded computing. However, the challenge of programming MPSoCs also comes as the biggest side-effect of the solution. Especially, when designers have to face the legacy C code accumulated through the years, the tool support is mostly unsatisfactory. In this paper, we propose an integrated framework, MAPS, which aims at parallelizing C applications for MPSoC platforms. It extracts coarse-grained parallelism on a novel granularity level. A set of tools have been developed for the framework. We will introduce the major components and their functionalities. Two case studies will be given, which demonstrate the use of MAPS on two different kinds of applications. In both cases the proposed framework helps the programmer to extract parallelism efficiently.

Categories and Subject Descriptors

D.1.3 [**PROGRAMMING TECHNIQUES**]: Concurrent Programming - Parallel programming

General Terms

Design

Keywords

Embedded, MPSoC Programming, Software, Parallelization

1. INTRODUCTION

In the past few years it has become clear that Multiprocessor System-on-Chip (MPSoC) is the most promising way to keep on exploiting the high level of integration provided by the semiconductor technology and, at the same time, matching the constraints imposed by the embedded system market in terms of performance and power consumption. However, the MPSoC solution does not come for free. Instead, it poses a great amount of challenges that have to be addressed by designers (both on the hardware and on the software side). Out of those challenges, the problem of programming such parallel architectures in an efficient way counts as the biggest one.

To solve this problem, new programming models have been proposed, which allow programmers to describe the parallelism of an application explicitly in the source code, like OpenMP [1] and PThreads [2]. Usually, such solutions require the programmer to heavily modify the original source code with parallel constructs and at the same time ensure the correctness of parallel execution. Perhaps the most important lesson from the past few decades of parallel programming is that the complexity of parallel programming should be hidden from the programmers as far as possible [3]. It is our belief that with advanced compiler techniques a toolkit can take care of the complexities such as synchronization and memory consistency, rather than putting up burdens on programmers' shoulders.

Other new models of computation or languages have also been proposed, like *Kahn Process Networks* (KPNs) based [4], task graphs based [5] or domain specific models such as StreamIt [6]. In those cases, a complete rewrite of the application with a new language would be required. Therefore, no new parallel programming models/languages have been widely adopted in embedded MPSoCs so far. Software developers have been familiar with sequential programming like C for a long time. We believe that, instead of using a new language/computational model, an evolutionary approach of sequential modeling plus concurrency constructs will bring more efficiency in terms of parallelization effort.

Decades of writing applications using the C language have left us a huge amount of legacy codes for a wide range of applications, and those are often used as reference implementations for MPSoC software. In fact, around 85% of embedded system developers still use C/C++ [7]. It becomes common that a sequential C implementation needs to be parallelized so that the parallel efficiency of the MPSoC hardware platform can be exploited. Unfortunately, efficient compiler support for this purpose is not available until today. There is so far no "silver bullet" existing which can solve the problem fully automatically.

In this paper, we present our MPSoC Application Programming Studio (MAPS), which is an integrated framework developed for assisting MPSoC programmers with C application parallelization. Unlike a fully autonomous parallel compiler, MAPS provides a set of tools which guides the parallelization process. Starting from the sequential C code, the MAPS analysis facilities are able to provide high level profiling information so as to help the programmer to

DAC 2008, June 8–13, 2008, Anaheim, California, USA

analyze the application. In order to extract coarse-grained parallelism, we propose a novel granularity level on which the MAPS partitioning tools try to find task candidates. The programmer is provided with suggestions on how to partition the application. Besides, the programmer also has the freedom of modifying the generated results and creating his own tasks. The final parallelization result can be obtained from both machine analysis and human knowledge, which depends on how the programmer uses the MAPS tools. Moreover, our approach is orthogonal to MPSoC programming models. We have coupled MAPS with the TCT framework [8] [9] which utilizes a so-called *Tightly-Coupled-Thread* (TCT) programming model [10].

The rest of this paper is organized as follows. In section 2 a summary of the related works is presented. Section 3 describes the main components of the MAPS framework. Section 4 presents two case studies to demonstrate the use of MAPS. Finally, section 5 concludes this work and gives insights into the future work.

2. RELATED WORKS

Since the dawn of parallel computing, there have been plenty of papers about automatic extraction of parallelism published. Compiler techniques for fine-grained *Instruction Level Parallelism* (ILP) are surveyed by Banerjee et al [11] while compiler techniques for fine-grained *Data Level Parallelism* (DLP, i.e. SIMD) are presented by Leupers in [12]. Our approach focuses on coarse-grained parallelism, but still, techniques for ILP would apply to the individual processors in an MPSoC. On the other side, general techniques for exposing coarse-grained parallelism such as those presented by Hall et al in [13] could also be integrated in our framework.

Several attempts to produce a partition of an application starting from a sequential implementation have been reported in literatures. Verdoolaege et al [14] and Harriss et al [15] derive KPNs from loops based on the COMPAAN compiler technology. The transformation applies only for applications in which conditions in the loops are affine combinations of the iterators. In that sense, they cannot process applications with dynamic behavior.

Karkowski et al [16] present an algorithm to implement pipelining from loops written in C. Their approach is similar to ours in that they also use a weighted statement control data flow graph. However, their analysis is performed solely on outermost loops and the partitioning approach is simple and does not handle hierarchy (e.g. nested loops and function calls).

Wiangtong et al [5] develop a HW/SW co-design environment targeting UltraSONIC reconfigurable system. Their approach employs a three-step algorithm, i.e., task clustering, partitioning and scheduling, to extract coarse-grained parallelism. However, their entry point is a *Directed Acyclic Graph* (DAG) representation of the application and manual efforts are needed for the conversion from high-level language to the DAG, which is non-trivial.

Ramaswamy et al [17] also present a similar approach in the network processor domain. They use run-time instruction traces to derive an *Annotated DAG* (ADAG) representation of the application as input to the later clustering and mapping processes. Compared to our approach, they profile on assembly level and their clustering results are difficult to be understood by designers as coarse-grained tasks.

Some works in the field of high performance computing are also related to our approach. This shows how the borders between general-purpose and embedded computing are becoming blurry. Ottoni et al [18] propose a *Decoupled Software Pipelining* (DSWP) approach as opposed to the thread level speculation which requires a lot of hardware support. They analyze the dependency graph and generate a DAG out of a loop by merging together the strongly connected components. Thereafter, they perform pipeline balancing in a first-bin-packing fashion breaking ties by selecting the solution that reduces the communication overhead. This approach is targeted for loops and it is not clear how they handle hierarchy of partitions or function calls. Besides, the user has to provide the time budget for the pipeline stage. Thies et al [19] present an approach to exploit pipelined parallelism from C applications with user annotations. In this work the user has to tell the compiler where the stages of the pipeline are. They use dynamic analysis to analyze the data flowing between pipeline stages. Bridges et al [20] present a work developed mostly in parallel to that in [19]. They present a case study showing the potential in pipeline execution. They use the DSWP [18] approach and extend C with constructs that allow a set of legal outcomes from a program execution instead of only one, these constructs give more room for parallelizing code. They do not present an automatic way for extracting threads though. All these approaches require the user to have a good understanding of the application, either to identify the threads by himself or to estimate the time for a pipeline stage.

3. MAPS FRAMEWORK

The MAPS framework consists of a set of tools which assist programmers with the parallelization of sequential C programs. The output of the framework are C source files with parallel constructs. Currently, MAPS supports the TCT programming model. Fig. 1 gives an overview of the MAPS tool flow with the TCT framework together.

The parallelization process supported by MAPS can be roughly divided into three phases: analysis, partitioning and code emission. At the beginning, two inputs are expected by MAPS, the sequential C code and the description of the target MPSoC platform. In the analysis phase, both static and dynamic approaches are applied in order to provide profiling information to the programmer and control/data flow information to the MAPS partitioning tools which search for parallel tasks in the target application. In the code emission phase, the resulting tasks will then be translated to parallelized C code. Here, the TCT framework appears as a back-end which is responsible for compiling the parallelized C code and simulating the result in order to evaluate the parallel performance. The user interface of the MAPS framework is built upon the C/C++ development tools of the Eclipse environment [21]. We have extended it with a set of plug-ins to control the MAPS tools conveniently and visualize the result intuitively.

Currently, MAPS focuses on the parallelization of C applications. Mapping coarse-grained tasks to *Processing Elements* (PEs) is not addressed. This is done by the TCT compiler. Besides, the information in the architecture description is now partially used in the framework for performance estimation. The partitioning algorithm is not fully aware of the difference between the processing elements in case of a heterogeneous MPSoC. In the following sections, we will discuss the major components in detail.

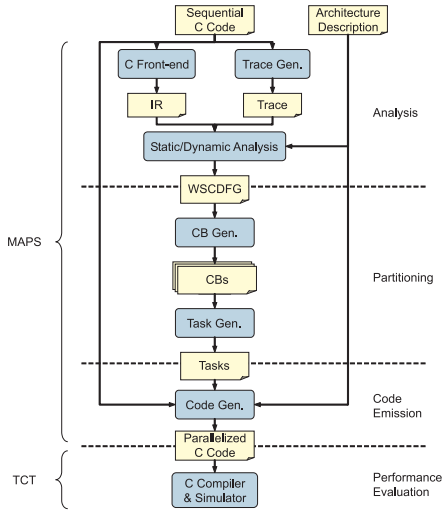

Figure 1: Overview

3.1 Architecture Description

As input to the MAPS framework, a description of the target MPSoC is required. For each type of PEs, there is a corresponding cost table defined. It gives out the estimated execution costs of different primitive operations used in the C language, e.g. addition and multiplication, in terms of clock cycles on the target processor. The inter-processor communication channels are modeled as logical channels between PEs. The transmission cost is calculated by the size of transferred data in bytes times the estimated clock cycles per transferred byte. The potential bus contentions are currently not considered in MAPS but modeled in the TCT framework accurately.

3.2 Profiling and Analysis

Within the MAPS framework, the first step to parallelize a sequential C program is to analyze the source code. This is necessary both for programmers who might not be familiar with the application and for the partitioning tool which searches for coarse-grained tasks automatically. For this propose, MAPS employs both static and dynamic analysis. The static control/data flow analysis module in MAPS works just like the same component in a traditional C compiler. It takes an *Intermediate Representation* (IR) of the program as input. Here, the LANCE compiler [22] front-end is used to generate the IR. As depicted in Fig. 1, the same source code is also used to create an instrumented application for generating the execution trace which provides dynamic information about the program. The application execution history is kept in the trace file in form of a sequence of basic block IDs which are assigned during instrumentation. Along with the basic block trace, all memory accesses performed through pointer dereference are also recorded in the file and later used by the dynamic data flow analysis module to characterize data dependencies caused by the usage of pointers. This is necessary, since the nature of C pointers makes it impossible to implement a 100% accurate static point-to analysis [23]. Therefore, we use the dynamic information which is accurate enough to drive the partitioning process. However for the binary code generation, the TCT back-end has a static point-to analysis engine in its C compiler, which is able to ensure correctness.

The estimation of the application execution cost is done based on the cost tables in the architecture description and the static/dynamic information from the IR and the trace. For example, the cost of an IR statement n_i can be calculated by $n_i = (IR\,Operation\,cost) * (Execution\,count)$. The required execution count can be derived from the trace file by counting the occurrence of the basic block containing the IR statement. A similar method has been applied in [24] and proved to provide enough accuracy for RISC like processing elements which is exactly the case of TCT.

The analysis module in MAPS is able to provide programmers with high level profiling information like call graph with functions weighted by execution costs and cost information of each source line. For the partitioning tools in the next phase, the information is summarized in so-called *Weighted Statement Control Data Flow Graphs* (WSCDFGs). In the next section we will discuss in detail how the WSCDFG is defined and why it is used for driving coarse-grained task generation.

3.3 CB and WSCDFG

One key issue in the process of discovering coarse-grained parallelism in C programs is to find a suitable granularity on which parallel tasks should be created. At this point, it comes naturally to the question why not to use *Basic Blocks* (BBs) or functions as the granularity level just like a traditional compiler does. Our observation is that both are not suitable for the purpose of MAPS. BBs and functions are all defined based on the control relationship of statements:

- A BB is defined as *"a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end"* [25].
- A function is simply defined as subroutine which has its own stack.

In practice, it is easy to find cases where BBs and functions are too big or too small for constructing parallel tasks. For example, a BB containing a sequence of time consuming function calls is not a good candidate for a task. The programmer might need to break it down into several small tasks in order to get performance gain. For this reason we propose a novel concept called *Coupled Block* (CB) as the smallest unit for coarse-grained task generation. The design of CB considers two criteria. First, a CB must be *schedulable*. That means, we cannot group arbitrary parts of the application into one CB, which is not schedulable. Second, a CB should be internally *tightly coupled by data-dependence*. This is mainly for the purpose of reducing inter-CB communication which consequently reduces the potential inter-processor communication that is often expensive in MPSoCs.

Since a CB can be larger or smaller than BBs, the search for CBs must start from the most atomic element in compiler, the IR statement. In our case, this is the LANCE three-address-code. Based on IR statements, WSCDFGs are first constructed on function level by the MAPS static/dynamic analysis module before CB generation.

WSCDFG Definition

Definition 1. *A function's WSCDFG is a directed graph defined by a tuple* $G = (V, CE, DE, CW, DW, N)$ *where the elements are defined as follows:*

- V is the set of compiler IR statement nodes whose weights constitute the set N.
- CE is the set of control-flow edges whose weights constitute the set CW

- DE is set of data-flow edges whose weights constitute the set DW
- $cw_{ij} \in CW$ is the weight of the control edge ce_{ij}. It is equal to the number of times the control path from node v_i to v_j is taken.
- $dw_{ij} \in DW$ is the data edge weight of de_{ij}. It is equal to the size of all dependent data between node v_i and v_j. Here, the data dependence information is provided by both the static data flow analysis and the dynamic point-to information extracted from the trace.
- $n_i \in N$ is the weight of the IR statement node v_i. It can be calculated by:
  $n_i = (IR\ operationcost) * (Execution\ count)$
  The IR operation cost is derived from a processor specific cost table in the architecture description, and the execution count is provided by the trace file.

An example of WSCDFG is shown in Fig. 2.

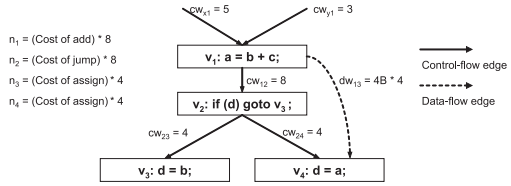

Figure 2: An Example of WSCDFG

CB Definition

DEFINITION 2. *A Coupled Block (CB) $G' = (V', CE', DE', CW', DW', N')$ is a sub-graph of WSCDFG G which satisfies the following properties:*

- $\exists v_{entry} \in V'$: v_{entry} *dom* $v_i \forall v_i \in V'$
- $\exists v_{exit} \in V'$ v_{exit} *postdom* $v_i \forall v_i \in V'$
- $\forall v_i \in V', \frac{w_1 \cdot \sum_{cw_{ij} \in CW'} cw_{ij} + w_2 \cdot \sum_{dw_{ij} \in DW'} dw_{ij}}{D(v_i, V')} > T$
  where $D(v_i, V'_x) = d_{CFG}(v_i, v_x) + w_3 \cdot |V'_x|$ with v_x one of the graph centers of G'_x, and $d_{CFG}(v_i, v_x)$ the distance between v_i and v_x on the control flow graph derived from G [1]. w_1, w_2, w_3 and T are tunable parameters (according to the architecture and the application)

These conditions stand for the above mentioned CB design criteria; the first two conditions ensure CB schedulability whereas the last one enforces CB to be tightly coupled by data-dependence. Besides, the term $D(v_i, V'_x)$ is introduced to enforce locality of clusters (distance computation) and perform a *ratio-cut*-like reduction (cardinality of set V'_x).

An example of CB in a graphical representation taken from the MAPS IDE is shown in Fig. 3. On the left-hand side the source code is displayed while on the right-hand side the CBs in the WSCDFG are shown. For example, the top CB node shown in the right-hand side contains the LANCE IR statements for the source lines `114,115`. The control-flow edges are annotated in the graph with the execution counts. The edge annotated with `100` between the two nodes is the control-flow edge as the nodes are in the loop executed `100` times. The data-flow edges are shown in the format *Name: Bytes * Counts*. The `array2_36` data-flow edge means that there is a *def-use* relationship between the nodes, i.e., `array2_36` is defined in the top node and used in the node below. Moreover, this data-flow edge has been counted `100` times because of the loop.

[1] In order to compute the distance, the direction of the edges on the control flow graph is not considered.

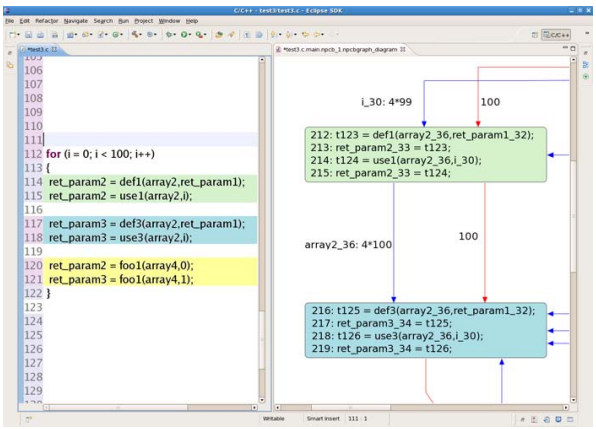

Figure 3: An Example of CB in the WSCDFG

3.4 CB Generation

For generating the CB partition, a heuristic algorithm inspired from data mining and machine learning techniques (mostly based on DBSCAN [26]) was developed. The algorithm starts by finding highly connected components in the sense of DBSCAN from the WSCDFG (details on the definition of the similarity measure and computation of free parameters are beyond the scope of this paper). Thereafter, the clusters are modified to meet the constraints described in section 3.3. The free parameters of the algorithm are computed automatically from the application itself. The complexity of the algorithm is governed by the Floyd algorithm for computing the similarity which is known to be $O(|V|^3)$.

Several iterations of the algorithm are performed to agglomerate fine-grained CBs into coarse-grained ones. The iterative process will stop after a user-defined number of iterations. Since different applications have variable internal control/data characteristics, the optimum iteration number varies from one application to another. A performance measurement derived from either estimation or execution is required here to determine the best iteration number. Fig. 4 shows an example on how the granularity of CBs increases through two iterations of the algorithm - small CBs are grouped into larger ones during this process.

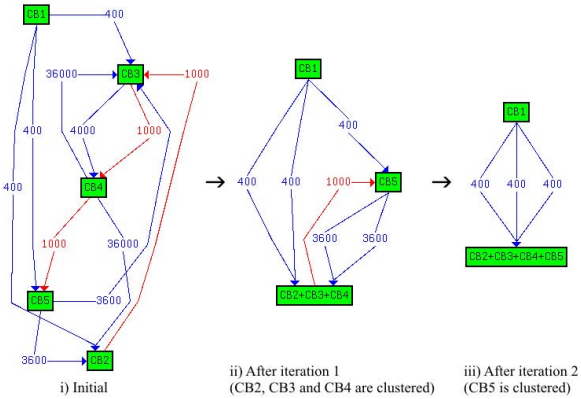

Figure 4: An Example of Iterative Clustering

3.5 Task Generation

In the MAPS framework, tasks are defined as contiguous C code blocks which are supposed to run in parallel on the target platform. Compared to a CB, a task is actually less constrained concerning its internal control and data relationship as given in section 3.3. In this sense, the generated CBs

can be considered as tasks proposed by the MAPS framework. There is a module in MAPS which creates tasks one-to-one from CBs automatically. Nevertheless, the MAPS framework is not supposed to be a fully automatic parallel C compiler, it still keeps the flexibility for the programmer to create or modify tasks. The final tasks could be a combination of CB generated tasks and user defined tasks.

3.6 Code Emission

In general, the code emission part of the MAPS framework takes the definition of tasks and the sequential C code as input and generates parallelized C code. Currently, the TCT programming model is supported. The transition from sequential to parallel is done by inserting the TCT `THREAD` scope annotations to the code blocks defined as tasks. Fig. 5 shows the conversion of a piece of pseudo C code from sequential form to threaded form.

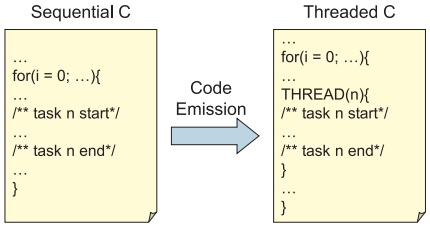

Figure 5: Code Emission

3.7 TCT Back-end

As mentioned previously, the MAPS framework currently uses the TCT framework as back-end for binary code generation, instruction-set simulation and parallel performance profiling [10]. Its target PE is a RISC processor inside an MPSoC [8] [9] where 6 PEs are connected with a full crossbar. The PEs also connected with their host-CPU through an AHB bus. Each PE has a dedicated module for high speed communication (4-6 cycles setup, 4Bytes/cycle burst transfer, 1 cycle PE-to-PE latency). The TCT SW toolkit was originally developed for this MPSoC, but it can also model an arbitrary number of PEs. The toolkit includes:

- **TCT Compiler** which takes C codes with `THREAD` annotation as input, analyzes inter-procedural dependence flows including pointer dereferences, inserts thread communication instructions for thread activation, data transfer and data synchronization, and finally generates partitioned thread executable binaries. Its concurrent execution model is constructed upon a hierarchy of functional pipelining and task parallelism for a fully distributed memory system which guarantees race-free, deterministic behavior. Current TCT Compiler assumes one thread per processor, i.e., no multi-threading capability is exploited.
- **TCT Simulator** which consists of a set of cycle-accurate instruction-set simulators for parallel thread execution. PE communication on the full crossbar interconnect is also modeled at cycle accuracy.
- **TCT Trace Scheduler** which is a fast performance estimation tool. It uses an abstract computation/ communication model to quickly report the estimated execution time with accuracy of few % error from the TCT Simulator.

In the complete exploration flow, the role of the TCT framework is to examine the speedup brought by the MAPS discovered parallelism. The programmer gets important feedback like thread schedule, sequential/parallel performance, and other characteristics. Based on this information, the programmer can decide whether further improvement is necessary to explore the potential of the parallel platform.

4. CASE STUDY

In this section, some initial results are presented to demonstrate the applicability and efficiency of our MAPS framework to parallelize C applications. We have carried out two case studies with the following objects in mind:

1. Demonstration of applicability of the parallelization methodology of the MAPS framework.
2. The efficiency of C application parallelization using the MAPS framework.

4.1 JPEG Encoder

We have taken a JPEG encoder application as our first case study. The JPEG encoder code has been taken from the TCT toolkit, which is modified from the original source by [27]. A manual partitioned JPEG encoder version is included in the TCT tools which achieves `9.43x` speedup [2] using `19` threads. Since currently each processor runs only one thread, the efficiency of this partition is `49.6%` [3]. The same source code (without manual threading) is used as input to our MAPS framework and the generated results are compared to the one of manual threading. In order to show the efficiency of our framework, a user with knowledge on the TCT backend and little knowledge on JPEG encoder is assumed here. The steps followed by this user to get a final partition of the JPEG application are reported in the following and the summary is given in Table 4.1.

In the first step (*step 1*) the user uses MAPS to partition the JPEG application directly. By analyzing the call graph, MAPS finds the hot spots of the application and performs partitioning on the most costly functions. (By using the visualization tool, the user can also specify the functions to be partitioned. The user does not do that here though.) The initial automatic partition gives `3.61x` speedup with an efficiency of `22.58%`.

By examining the graphical execution traces provided by the TCT tool, it is easy to find out that another `THREAD` annotation is needed to obtain a hierarchical pipeline (*step 2*) which is an architectural feature of TCT. After the second step, the speedup and the efficiency are already reasonably good, `5.48x` and `32.3%` respectively.

Also from the traces, a small modification is very easy to be recognized - a processor is rarely used. The user can remove this processor by subtracting a `THREAD` annotation around the two consecutive `ReadOneLine` function calls to improve the efficiency further (*step 3*).

After 3 iterations, the user has an already parallelized version of the JPEG application that provides `5.48x` speed up with an efficiency of `34.3%`. The aforementioned steps are carried out in around two hours. Our tool produces partitions in exactly the same functions that were chosen for the manual partition by TCT. Furthermore, `10 THREAD` annotations out of the `12` annotations generated by using our tool are located exactly in the same places as in the manual partition.

[2] $Speedup = \frac{Clock\ Cycles\ of\ the\ Sequential\ Execution}{Clock\ Cycles\ of\ the\ Parallel\ Execution}$

[3] $Parallel\ Efficiency = \frac{Speedup\ of\ Parallelization}{Number\ of\ Processors} \times 100\%$

Step	Speedup	No. of PEs	Parallel Efficiency
1	3.61x	16	22.58%
2	5.48x	17	32.3%
3	5.48x	16	34.3%
manual	9.43x	19	49.6%

Table 1: Summary of JPEG Encoder Parallelization by MAPS

By interacting with our tool, the user only fails to reproduce the manual partition inside the `BLK8x8` function, which is the cause for the speedup gap between the manual and the semi-automatic partition. This is mainly due to the complex data dependencies between different executions of this function across different iterations. A possible solution to overcome this problem is to extend the data dependency analysis, e.g. by taking the loop distance information into account.

4.2 ADPCM Decoder

To test our framework's applicability, we have studied another test-case - *Adaptive Differential Pulse Code Modulation* (ADPCM) decoder whose source code was obtained from [28]. TCT currently has a simple assumption that executing each thread always requires one separate processing element. That implicitly means that the code blocks not in the sequential order will not be located into the same processing element. A small source code modification in the ADPCM decoder, which moves the step to find the new index value to a later stage of execution, has been made to be in favor of it.

ADPCM is in nature not friendly to be parallelized algorithmically and no manual partition within the TCT toolkit is available. There are many references using ADPCM as a benchmark but most of them focus on improving the ILP. In [18] ADPCM decoder was analyzed along with a variety of benchmarks on 2 Itanium duo cores achieving a speedup of `1.1x` with the "best manually directed" approach. The MAPS toolkit is used to partition this application and a result with three `THREAD`s is generated which achieves a speedup of `1.28x` on `4` processors giving an efficiency of `32%`. This automatic partition is obtained within half an hour.

5. CONCLUSIONS

Within this paper, we have presented MAPS, an integrated framework aiming at parallelization of C applications. Main components of the framework have been discussed in detail. A novel granularity level has been proposed for the discovery of coarse-grained parallelism in C. The framework has been coupled with the TCT platform. The parallelism exploration is accelerated in a way that the programmer can now quickly get information about where and how to parallelize.

In future, we will further improve the partitioning algorithm in order to extract more parallelism automatically. Moreover, we also envision better support for heterogeneous MPSoC including task mapping and PE-aware partitioning.

6. REFERENCES

[1] The OpenMP Specification for Parallel Programming. *http://www.openmp.org*.

[2] Standard for Information Technology - Portable Operating System Interface (POSIX). Shell and utilities. IEEE Std 1003.1-2004, The Open Group Base Specifications Issue 6, section 2.9. *IEEE and The Open Group*.

[3] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic Parallelism Requires Abstractions. *SIGPLAN Not.*, 42(6):211–222, 2007.

[4] L. Thiele, I. Bacivarov, W. Haid, and K. Huang. Mapping Applications to Tiled Multiprocessor Embedded Systems. In *ACSD 07*, pages 29–40, 2007.

[5] T. Wiangtong, P. Cheung, and W. Luk. Hardware/Software Codesign: a Systematic Approach Targeting Data-intensive Applications. *IEEE Signal Processing Magazine*, 22(3):14-22, 2005.

[6] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. *SIGOPS Oper. Syst. Rev.*, 36(5):291–303, 2002.

[7] Embedded Software Stuck at C. *http://www.eetimes.com/news/design/showArticle.jhtml?articleID=202102427*.

[8] M. Z. Urfianto, T. Isshiki, A. U. Khan, D. Li, and H. Kunieda. A Multiprocessor System-on-Chip Architecture with Enhanced Compiler Support and Efficient Interconnect. In *IP-SOC 2006, Design and Reuse, S.A.*, 2006.

[9] M. Z. Urfianto, T. Isshiki, A. U. Khan, D. Li, and H. Kunieda. A Multiprocessor SoC Architecture with Efficient Communication Infrastructure and Advanced Compiler Support for Easy Application Development. *IEICE Trans. Fundamentals*, Vol.E91-A,No.4, Apr. 2008.

[10] T. Isshiki, M. Z. Urfianto, A. U. Khan, D. Li, and H. Kunieda. Tightly-Coupled-Thread Model: A New Design Framework for Multiprocessor System-on-Chips. In *Design Automation Symposium (Japan)*, 2006.

[11] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic Program Parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.

[12] R. Leupers. Code Selection for Media processors with SIMD Instructions. In *DATE '00*, pages 4–8. ACM, 2000.

[13] M. H. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Detecting Coarse-Grain Parallelism Using an Interprocedural Parallelizing Compiler. In *Supercomputing '95*, page 49. ACM, 1995.

[14] S. Verdoolaege, H. Nikolov, and T. Stefanov. Improved Derivation of Process Networks. In *ODES-4*, 2006.

[15] T. Harriss, R. Walke, B. Kienhuis, and E. Deprettere. Compilation from Matlab to Process Networks Realized in FPGA. In *Design Automation of Embedded Systems*, volume 7, 2002.

[16] I. Karkowski and H. Corporaal. Design of Heterogeneous Multi-Processor Embedded Systems: Applying Functional Pipelining. In *PACT '97*, page 156. 1997.

[17] R. N. Weng and T. Wolf. Application Analysis and Resource Mapping for Heterogeneous Network Processor Architectures. In *in Proc. of Third Workshop on Network Processors and Applications (NP-3)*, pages 103–119, Feb. 2004.

[18] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *MICRO 38*, pages 105–118. IEEE Computer Society, 2005.

[19] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs. In *MICRO-40*, 2007.

[20] M. J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. I. August. Revisiting the Sequential Programming Model for Multi-Core. In *MICRO-40*, 2007.

[21] Eclipse C/C++ Development Tooling. *http://www.eclipse.org/cdt*.

[22] R. Leupers, O. Wahlen, M. Hohenauer, T. Kogel, and P. Marwedel. An Executable Intermediate Representation for Retargetable Compilation and High-Level Code Optimization. In *SAMOS*, 2003.

[23] M. Hind. Pointer Analysis: Haven't We Solved This Problem Yet? In *PASTE '01*, pages 54–61. ACM Press, 2001.

[24] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr. A SW Performance Estimation Framework for Early System-Level-Design Using Fine-Grained Instrumentation. In *DATE '06*, pages 468–473, 2006.

[25] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[26] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *KDD 96*, pages 468–473, Aug. 1996.

[27] Independent JPEG Group. *http://www.ijg.org/*.

[28] DSPStone. *http://www.ert.rwth-aachen.de/Projekte/Tools/DSPSTONE/dspstone.html*.