Critical Path Method Based Heuristics for Mapping Application Software onto Heterogeneous MPSoCs

Nikolina Frid and Vlado Sruk
University of Zagreb, Faculty of Electrical Engineering and Computing, Zagreb, Croatia
nikolina.frid@fer.hr, vlado.sruk@fer.hr

Abstract – In this paper the authors propose new heuristics for automation of software partitioning and mapping onto heterogeneous multiprocessor System-on-Chip (MPSoC) platform – Longest Parallel Path mapping algorithm (LPP). In contrast with traditional approach to solving this NP-complete problem – the Integer Linear Programming (ILP), our method uses a modified version of Critical Path Method with additional heuristics that rely on greedy approach. The algorithm performs one-to-many mapping of application to platform with minimizing the overall execution time of the application as the main objective. Our experiments with generic application model and several different platform layouts show that the proposed algorithm provides an efficient mapping scheme enabling significant execution speedup. In addition, the comparison with another greedy mapping algorithm shows that LPP algorithm exploits available task level parallelism better.

Keywords: Design Space Exploration, Software partitioning and mapping, Task scheduling, Critical Path Method

I. INTRODUCTION

Modern embedded systems are above all ubiquitous: they are found everywhere - from simple electronic products, communication and entertainment devices to expensive and complex systems (car, plane). They are getting more and more complex, and have to be able to perform advanced computation and meet special requirements: real-time response, maximum performance, minimum size and weight, low power consumption. This leads to development of new generation of embedded systems – MPSoCs which must often be very heterogeneous to meet all requirements with minimal cost.

Designing this type of systems is a challenging and time consuming task. Automation of design process would significantly lower cost and time-to-market. However, because of MPSoC versatility and heterogeneity, this is easier said than done. Many different approaches exist and many algorithms have been developed but this issue is far from being solved yet.

Scheduling and mapping software to hardware on heterogeneous platforms is NP – complete problem, and finding near optimal solution in reasonable amount of time is challenging [8].

Traditionally this problem is solved using Linear Programming model [4], but high complexity of this approach and extremely high computational requirements significantly increase design time. There have been several attempts to solve this problem, including greedy deterministic [7][6] and evolutionary non-deterministic approaches [11]. Greedy approach is the most common approach and delivers the final mapping solution much faster than evolutionary approach but always gives a suboptimal solution.

The Longest Parallel Path Mapping Algorithm (LPP), proposed in this paper, aims to present another possible approach to solving task mapping problem on heterogeneous MPSoC architectures.

II. DESIGN SPACE EXPLORATION

Design Space Exploration (DSE) is a process to find out near-optimal system architectures for a given application, considering the design constraints and objectives. It involves task partitioning and mapping, architecture and processing element selection, and performance estimation before a hardware prototype is built [2]. The process starts with application specified using a model of computation, description of architecture instance and a set of constraints for the overall system. Those inputs are fed to mapping engine which performs partitioning and mapping of software to hardware. Finally, a performance analysis is conducted to determine whether the output design meets given constraints. If the result is positive the production phase begins, if not modifications to application and/or architecture are made

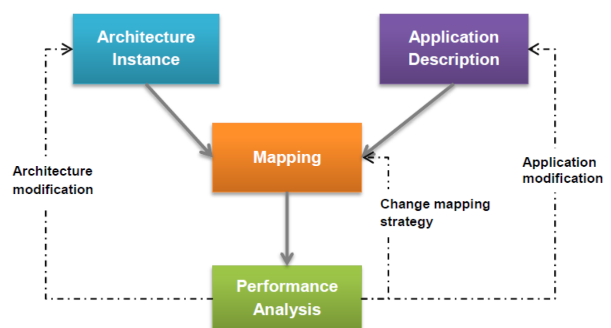

Figure 1. Design Space Exploration Flow

and the whole process is repeated. It is also possible to alter the mapping strategy and check if it produces better outcome. Figure 1 illustrates the described DSE flow.

A. Related Work

There is no commercial tool available to automate the whole DSE process. However, a tool is regarded as a DSE tool if it enables the system designer to alter the target architecture and task mapping easily, and it provides the estimated performance. Several tools exist that provide a general modelling and mapping framework to allow DSE [2]. Most prominent is a commercial solution - CoFluent Studio from CoFluent Design, a SystemC-based system-level design tool that supports DSE through Y-chart modelling. The designer specifies separate models for the application and the platform and combines them at the mapping stage. Then, the resultant design is evaluated by simulating the transactional SystemC code that is automatically generated. VisualSim from Mirabilis Design is a system-level design tool that supports diverse simulation capabilities in a single framework. Both algorithm simulation and architecture simulation can be performed at various abstraction levels. By changing the mapping information and architecture parameters, the user can explore the design space efficiently. However, in each of these tools task mapping must be performed manually.

Most prominent non commercial DSE tools include The Modular Performance Analysis (MPA) toolbox from ETH Zurich, which is a performance analysis tool based on real-time calculus, and Daedalus framework from University of Amsterdam, which has the capability of automatic mapping but the process takes several hours to finish [13]. However, much effort is put into further research and development of these tools and it is reasonable to expect that future improvement will resolve current issues.

B. Task partitioning and mapping

The key step in design space exploration process is partitioning application software into separate tasks and deciding where to allocate each task on the target platform. Due to heterogeneity of both application and platform this problem requires an NP-complete class algorithm to find the optimal solution. This means that the solution cannot be reached in polynomial amount of time. A faster alternative is using some form of specialized heuristics based on greedy strategy. This way a suboptimal solution is obtained much faster, but heuristics are usually rather specialized and are applicable to a smaller set of problems.

III. Problem Defintion

The main problem can be summarized as: finding the near optimal solution to mapping of a generic type of embedded application to a heterogeneous multiprocessor platform in a reasonable amount of time.

The rest of this chapter gives a more detailed problem specification.

A. Application Model

Due to versatility of embedded systems, embedded applications come in a wide variety of forms. For the purpose of evaluation of the proposed algorithm a generic application model is constructed with certain assumptions and constraints in mind.

First, it is assumed that application has a flow from a single point of entry to a single point of exit and inside the application flow there are no cycles at all. However, the application as a whole can be cyclic, meaning that after exit point it starts all over again from start. Second, some level of functional parallelism in application is implied because otherwise the discussion about mapping parts of application to different processors is pointless, unless a certain task requires a certain type of special HW accelerator.

The application is modelled using a parallel model of computation and represented as a Directed Acyclic Graph (DAG), $G = (T, E)$. For each vertex $t \in T$ which represents one task, we define set of edges $E_t \subseteq E$ connected to t that represent communication between tasks. Figure 2 depicts an example application graph.

Vertex weight, given by a weight function:

$$w_t : T \to \mathbb{R} \quad (1)$$

stands for the number of operations executed in the process (in Mops) and the edge weight, given by a weight function:

$$w_e : E \to \mathbb{R} \quad (2)$$

is the amount of data transacted (in KB), both as reported by a profiler.

B. Architecture Model

The MPSoC hardware platform is also modelled as a directed graph $H = (P, F)$, each vertex $p \in P$ represents a processing element. Vertex weight is the computation

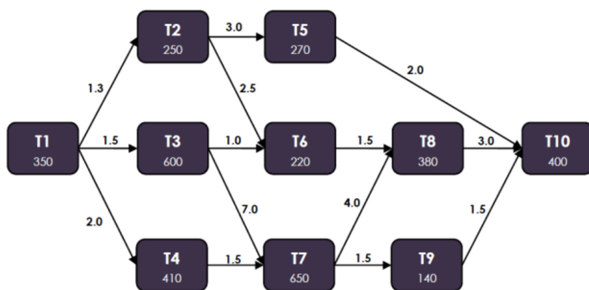

Figure 2. Application representation

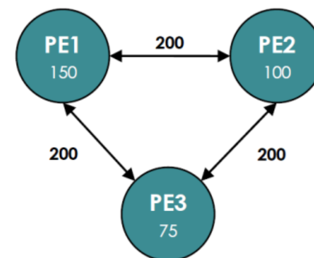

Figure 3. MPSoC platform representation

speed in Mops/s and is given by a weight function:

$$u_p : P \to \mathbb{R} \quad (3)$$

Each edge represents connection between PEs and weight is the throughput in kBps given by a function:

$$u_f : F \to \mathbb{R} \quad (4)$$

Figure 3 depicts an example platform graph with three different processing units.

In this model, processing elements differ from each other strictly by raw computational power, i.e. it is assumed that each PE is capable of performing all types of operations equally efficient.

It is also assumed that each processing element has its own private memory and communicate via interconnect network.

C. *Mapping Problem*

Given the application and architecture models, the mapping problem can be defined as assigning application components - tasks to architecture components in a way that the overall execution time required to execute the application is minimized. The final solution has to be found under the constraints that all the tasks are assigned a PE and one task is assigned only to one PE.

For n tasks (n=|T|) and m processing elements (m=|P|), where $n \geq m$, the cost of mapping of task $i := t_i$ to processing element $k := p_k$ can be defined as the time required to complete all of required computation and communication.

$$T = T_{comp} + T_{comm} \quad (5)$$

Computation time is calculated as:

$$T_{comp}(i,k) = \frac{w(t_i)}{u(p_k)} \quad (6)$$

Communication time is the sum of duration of communication between task i and all predecessor tasks mapped to a processing element different that k (communication cost between two tasks on the same PE is zero). The cost of communication between two tasks i and j mapped to processors k and l respectively is expressed in (7).

$$T_{comm}(i,j) = \frac{w(t_i,t_j)}{u(p_k,p_l)} \quad (7)$$

In order for a task to begin execution, all its predecessors must finish execution. After the predecessor j finishes execution, it would take $T_{comm}(i,j)$ time before the task i can use the results. This has to be calculated for all predecessors of task i and the maximum such value is the time when task i can start execution. The time the task i takes to execute $T_{comp}(i)$ depends on the PE it is assigned to.

IV. Longest Parallel Path Mapping Algorithm

In order to solve this resource allocation problem, a static scheduling algorithm [14] has been developed based on Critical Path Method [5] with additional modifications to develop fast and efficient heuristics. This algorithm's objective is to minimize the overall execution time of the application.

For each task the earliest and the latest possible start time governed by data dependencies can be defined. If both times for a certain process are equal that process is a *critical process*. A set of critical nodes (processes) and edges between them from source to finish form a *critical path*.

The mapping process, as shown on Figure 4, consists of four stages. In the first stage, first of all processing elements are sorted descending by their computation speed. Afterwards, using a modified version of Dijkstra algorithm [12], the longest path is identified and mapped to the fastest processor. As long as there are free PEs, the algorithm identifies the second critical path, the third etc. and maps all tasks in the path to the fastest free PE. Thus, communication costs between tasks on the same path are minimized.

Next, a preliminary schedule is made for mapped tasks, which enables the search for free gaps in PE task list where unmapped tasks will be mapped in stage 3.

Stage 3 comes in two versions. In the first one, mapping of remaining tasks starts with PEs *sorted ascending* by computation speed. For each unmapped task a time frame is calculated - it begins when all of task's mapped predecessors have finished and ends before the first mapped successor has to start. Starting with the slowest PE, mapper checks if it is free during time frame and fast enough to compute the task in time; if

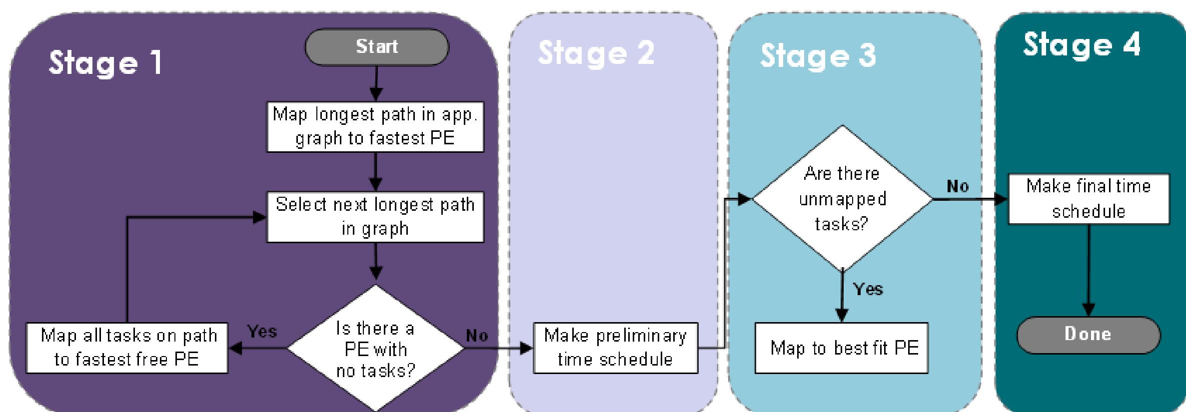

Figure 4. Longest Parallel Path algortihm flow

it is not, the overflow for that task and that PE (expressed as time) is stored for later evaluation.

In the second version, PEs are *sorted descending* by computation speed thus search for candidate PE starts with fastest processor.

When the fit PE is found the task is mapped to that PE – a perfect fit, and in case when there is no perfect fit the task is mapped to PE with least overflow.

The rationale behind the first version of Stage 3 mapping is that it is hardest to find a task that can be mapped to slowest PEs without disrupting entire schedule. On the other hand, mapping one task to slow PE usually means that it will consume the most of the available time frame and leave little chunks of free processor time that are not big enough for any other task to fit in. On the contrary, fast processors require much less time to finish the computation and it is probable that when a task is assigned in a free time frame, enough room will be left for at least one more task to be fitted in the same time frame.

The final schedule is calculated in the last stage of the mapping process.

Based on the algorithm discussed above, Automatic Mapping Tool has been developed in Java programming language and preliminary results of conducted simulations are discussed in the next chapter.

V. Experimental Results & Future Work

In this chapter we present the results of performance testing of LPP algorithm, make comparison with results obtained by using another greedy algorithm – LPT [7] and propose directions for future development.

A. *Case Study*

The performance of proposed mapping algorithm, in terms of total execution time estimation, was examined on nine different platform configurations for two generic applications. The configuration for each test case is described in Table 1. The numbers represent PE speed in Mops, and in each case communication costs between two processes on different PEs are the same.

TABLE I. Platform Configurations for Test Cases

	1PE [Mops]	2PE [Mops]	3PE [Mops]
Configuration 1	150	150, 150	150, 150, 150
Configuration 2	100	150, 100	150, 100, 100
Configuration 3	75	150, 75	150, 100, 75

1) The LPP algorithm was tested for both versions of Stage 3 (ascending and descending order). The execution times estimation of mapping generic application, represented by a DAG on Figure 2, are shown on chart on Figure 5. In the same figure we compare to the results obtained using LPT algorithm. LPP – ascending version results

In all nine test cases LPP outperforms LPT by up to 130 percent. It is also important to note that LPP algorithm, in case of platform with two fast PEs (both 150 Mops), reaches optimal solution: total duration is equal to critical path duration, and LPT can never reach that situation. In that case LPP algorithm provides speedup of 1.56 times compared to a sequential solution. Even more interesting is that in Configuration 3, when two fast PEs (both 150Mops) are replaced by a set of PEs by a third and a half slower ('3PE' in Table 1) the speedup is still high - 1.5 times compared to sequential solution.

It is also important to note that time complexity of LPP algorithm is $T(n) = O(n \log n)$ which is a significant improvement compared to complexity of LPT algorithm which is $T(n) = O(n^3)$ [7].

2) LPP – descending version results

The results for the second version of LPP (Stage 3 PE search in descending order) are exactly the same as in the previous case. A possible explanation for this is that other features of the algorithm govern task scheduling in later stages. For example, the first stage of the algorithm works in a way that it puts most load to fastest processors which later have little room to host additional tasks. On the other hand slower processors are left with much more idle time after the preliminary scheduling and have a higher

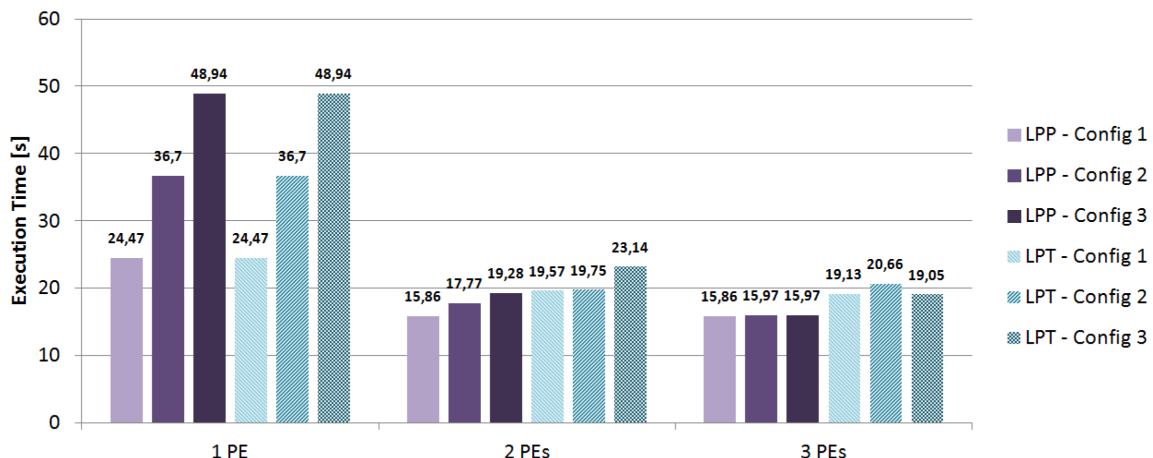

Figure 5. Total execution time comparison

possibility to contain a gap in scheduling that is big enough to fit a new task without overflow. Another possible reason for this, in case when all processors are left "crowded" after the preliminary scheduling, and overflow is almost iminent, is that faster processors will have less overflow (can compute the task faster) and will be filled with new incoming tasks first, regardless of the order of processor list traversal.

B. *Future development*

In future research, the LPP algorithm should be improved so it could tackle the following two issues: support for application with loops and recursions, and support for MPSoC platforms that have a more specialised type of components which are more suitable for a certain type of operation (e.g. floating point operations) – *mapping affinity problem.*

Another direction of development could address DSE from another point – giving recommendation for the platform architecture by selecting the most suitable configuration from a set of available components.

VI. CONCLUSION

In this paper we have given a short overview of Design Space Exploration, as a vital concept for modern heterogeneous multiprocessor embedded system, and related work. Further, we have addressed the issue of lack of quality mapping algorithms – crucial for automation of the entire DSE process by presenting new greedy strategy based on critical path method.

The presented LPP algorithm provides efficient automatic mapping strategy which exploits available task level parallelism in the given application. Conducted case study shows that the algorithm is able to schedule tasks on heterogeneous platforms with variable processor performance and achieve speedup very close to optimal solution in polynomial amount of time. In comparison to another acknowledged greedy strategy, LPP algorithm shows significant performance improvement – up to 130 percent. However, some issues concerning exploiting data level parallelism available in application and higher level of "platform awareness" remain and will be dealt in future research.

REFERENCES

[1] D. D. Gajski, S. Abdi, A. Gerstlauer and G. Schirner, *Embedded System Design. Modeling, Synthesis and Verification*. Ed. New York: Springer, 2009.

[2] H. Park, H. Oh and S. Ha, "Multiprocessor SoC Design Methods and Tools," *IEEE Signal Processing Magazine*, vol. 26, pp. 72 – 79, November 2009.

[3] W. Wolf, "Hardware and Software Co-design," in *High-Performance Embedded Computing. Architecures, Applications and Methodologies.* Ed. San Francisco, CA: Elsevier, 2007.

[4] J. Lin, A. Srivatsay, A. Gerstlauer, and B. L. Evans, "Heterogeneous multiprocessor mapping for real-time streaming systems," in *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Prague, Czech Republic, 2011, pp. 1605 – 1608.

[5] W.H. Kohler, "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems," in *IEEE Transactions on Computers*, Vol C-24, pp. 1235 - 1238, December 1975.

[6] K. Vivekanandarajah, and S. K. Pilakkat, "Task Mapping in Heterogeneous MPSoCs for System Level Design," in *Proceedings of 13th IEEE International Conference on Engineering of Complex Computer Systems,* Belfast, UK, 2008, pp. 56 – 65.

[7] *V.* Zadrija and V. Sruk, "Mapping algorithms for MPSoC Synthesis" *in Proceedings of MIPRO 2010, 33rd International Convention,* Opatija, Croatia, 2010, pp. 624 - 629.

[8] D. Pierre, *Optimization Theory With Applications*, ser. Dover Books on Mathematics Series. Dover Publications, 1986.

[9] R. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, ser. Series of Books in the Mathematical Sciences. W. H. Freeman, 1979.

[10] J. Hopcroft, R. Motwani, and J. Ullman, *Introduction to automata theory, languages, and computation*, ser. Addison-Wesley series in computer science. Addison-Wesley, 2001.

[11] M. Palesi, T. Girvagis,. "Multi-Objective Design Space Exploration Using Genetic Algorithms", *Proceedings of the Tenth International Symposium on Hardware/Software Codesign (CODES),* 2002, pp. 67 – 72.

[12] R. Sedgewick, K. Wayne, Algorithms, 4th Edition, http://algs4.cs.princeton.edu/home/, December 2013.

[13] M. Thompson, H. Nikolov, T. Stefanov et al., "A Framework for Rapid System-level Exploration, Synthesis, and Programming of Multimedia MPSoCs", *Proceedings of the 5th International Symposium on Hardware/Software Codesign CODES+ISSS'07, Salzburg, Austria, 2007*, pp. 9-14.

[14] N. Frid, V. Sruk. "Longest Parallel Path Mapping Algorithm for Heterogeneous MPSoCs", ACM womENcourage Europe Conference, March 2013, accepted for publication.