Hierarchy-Aware Mapping of Pipelined Applications

Kyoungwon Kim
Center for Embedded Computer Systems
University of California Irvine
Irvine, CA, 92697
kyoungk1@uci.edu

Daniel D. Gajski
Center for Embedded Computer Systems
University of California Irvine
Irvine, CA, 92697
gajski@uci.edu

ABSTRACT

This paper presents hierarchy-aware mapping of a pipelined application to a given heterogeneous platform. The applications we target are executed in a pipelined manner and can be captured with a Model of Computation (MoC) with explicit representations of pipelined execution. Our optimization goal is to minimize execution time.

It is crucial to balance the pipeline stages in the given MoC, where each stage consists of a set of hierarchical tasks, each is decomposed into either over which state transitions depending on data are defined or which run in parallel. Previous works have not considered such complex hierarchy in a general MoC; They assumed that each iteration of the execution of the application is represented with an acyclic directed graph and that a stage can be decomposed only into one or more parallel tasks. Therefore, the tasks tended to be coarse-grained since a task with complex hierarchy can be hardly partitioned more. Our approach partition each hierarchically described task and may map the partition separately to balance the stages. The case study is performed with randomly generated platforms and streaming applications: Canny Edge Detector and JPEG. We compare our Hierarchy-Aware mapping to the hierarchy-unaware mapping found through an exhaustive search. Hierarchy-Aware mapping decreases execution time on average by 23.3%.

Categories and Subject Descriptors

[**Hardware/software co-design**]

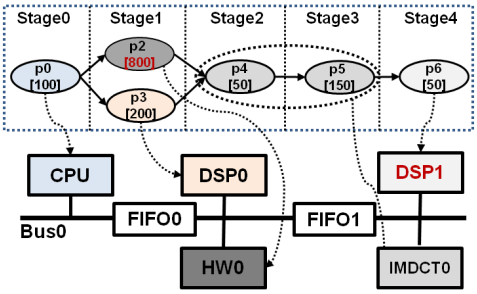
1. INTRODUCTION

The enormous growth in the design complexity of embedded systems has been led by the ever increasing functional and the non-functional constraints that embedded systems have to meet. Any design methodology should address productivity. Major semiconductor roadmaps have asserted the need to raise the level of abstraction to the electronic system level (ESL) [1]. In this sense, many ESL design methodologies and tools start the design process by specifying the applications with powerful models of computation (MoCs) and nonfunctional requirements. In such design approaches, automation in mapping is becoming critical since mapping is conducted in early design stages where there is a wide design space and embedded systems are already very complex. On the other hand, pipelined execution of the given application on a heterogeneous platform is a practical solution for certain application domains such as streaming applications [2]. The MoC in which such applications are captured may have explicit representations of the pipeline. Mapping techniques must exploit pipeline parallelism.

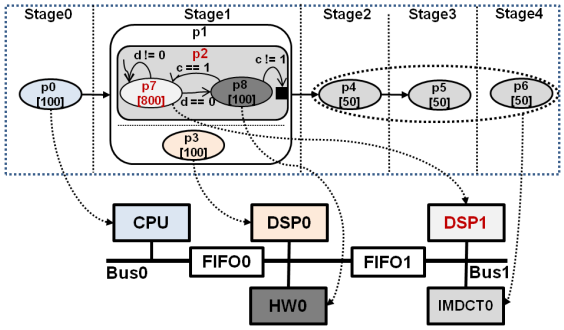
This paper describes mapping of a pipelined application while optimizing execution time. The applications we target are executed in a pipelined manner and can be captured in an MoC with explicit representations for pipelined execution. Such an MoC can be decomposed into one or more *pipeline stages* or, interchangeably, *stages*. Each stage consists of a set of sequential and/or parallel tasks, each of which may also be decomposed, recursively, into sequential/parallel tasks. Balancing the delays of the pipeline stages, the aim of which is to make the execution time of each stage approximately the same, significantly impacts the execution time of the system. Ideally, the execution time of the system is close to that of the stage with the longest delay.

Our contribution is, by being aware of hierarchy, to balance pipeline stages. Previous works [3] [4] [5] [6] [7] have assumed that hierarchy in the MoC is flattened. However, in general MoCs, a stage can have a complex hierarchy that does not easily allow such flattening. For example, in Program State Machines (PSMs) [8], a stage can be decomposed into ten program-states, over which are defined very complex state transitions depending on the data. Hierarchy-Aware mapping we present saves additional PEs by merging consecutive small stages, re-assigning the saved PEs to the large and hierarchically described stages, repartitioning the large stages and carrying out a remapping of the stages.

The rest of this paper is organized as follows. Section 2 compares previous work to the proposed work. Section 3 explains the application models and platforms. The problems are formalized in Section 4. In Section 5, the algorithms and objective functions for Hierarchy-Aware mapping are explained. Section 6 presents the case study, where Hierarchy-Aware mapping is compared to exhaustive hierarchy-unaware mapping. Section 7 offers our conclusions.



(a) Hierarchy-Unaware Mapping



(b) Hierarchy-Aware Mapping

Figure 1: Pipeline-Aware Mapping with and without Consideration of Hierarchy

2. PREVIOUS WORK

It is a challenging problem to pipeline-aware map an MoC to homogeneous or heterogeneous platforms. Over the past few years, a great deal of papers have appeared in the literature try to solve the problem. Lin et al [3] suggested pipeline-aware mapping of an application that is based on heuristics using Strength Pareto Evolutionary Algorithm (SPEA) II [9] and Integer Linear Programming to find Pareto optimal solutions in terms of throughput, latency and cost. Gordon et al [4] devised heuristics to optimize throughput in mapping of a streaming application to the target platform. However, these works require the application to be captured in Synchronous Data Flow (SDF) models. SDF is used for data-oriented applications only. A general MoC such as a PSM is hard to reduce to SDF.

Javaid and Parameswaran [5] conducted multiobjective optimization based on ILP and the heuristics the authors suggested. Optimal configurations for the given, reconfigurable processor-based platform are explored followed by mapping. Benoit et al [6] [7] established the complexity of pipeline-aware mapping problems and suggested heuristics if the time complexity is NP-hard.

Figure 1 shows the difference between Hierarchy-unaware mapping (i.e. [5], [6] and [7]) and the proposed Hierarchy-aware mapping. Hierarchy-unaware mapping may merge small consecutive stages (e.g. p4 and p5) and/or map parallel tasks in a stage separately (e.g. p2 and p3). In addition to that, the proposed work merges small stages more actively (e.g. from p4 through p6), saves PEs (e.g. DSP1) and assigns the saved PEs to large stages (e.g. Stage1). Following that, each of the large stages is divided into several smaller pieces and mapped separately. The goal of the procedures is a better balance among pipeline stages.

3. APPLICATION MODEL AND PLATFORM

The entire MoC is a pipeline, which can be decomposed into one or more *pipeline stages*, or *stages*. Any two tasks in different stages run concurrently. Any two tasks in adjacent stages may communicate through FIFO channels. Each stage is a single *program-state*. As Stage1 in Figure 1b exemplifies, a program-state can be either decomposed into sequential program-states, decomposed into parallel program-states or a leaf program-state we call a *process*. A process contains codes written in high level languages such as C/C++. Sequential decomposition implies there is a finite set of hierarchical sub program-states in the program-state; a set of state transitions is defined as well. A state transition may be affected by data. Parallel program states in a single stage run in parallel with their sibling program-states. Any two states may communicate through channels. For example, in Figure 1b, p0, p1, p4, p5 and p6 are executed concurrently in a pipelined manner. p1 is decomposed into two parallel program states: p2 and p3 running in parallel with each other. p2 is sequentially decomposed into two processes: p7 and p8. However, note that, according to the data **c** and **d**, p7 and p8 can be executed any arbitrary times even in a single cycle of the pipelined execution. On the contrary, in the MoCs that the previous work addressed in Section 2, the number of executions of p7 and p8 are statically given.

The platforms we target are combinations of general purpose processors (GPPs), DSP, FPGA and other processing elements.

4. PROBLEM DEFINITION

We merge small contiguous stages and partition large stages. This is done to minimize the execution time of the system. We assume the execution time of the system depends on the execution time of the stage which has the longest delay. To formalize the problem, we define a partition as either a subset of program-states in a single stage or the stage itself. V, S and $P(s_i)$ are the set of PEs, stages and partitions in a stage s_i, respectively.

For a partition p_j, $T(p_j)$ is execution time of p_j. $T(s_i)$ is the execution time of stage s_i and defined as follows:

$$T(s_i) = max\{T(p_j)\}\ for\ p_j\ \in\ P(s_i) \qquad (1)$$

Our goal is to find the set of stage S and mapping $M : V \rightarrow S$, where $max\{T(s_i)\}\ for\ s_i\ \in\ S$ is minimized. Note that S changes during mapping and that $T(s_i)$ may be reduced by partitioning s_i and assigning more PEs to the stage.

5. HIERARCHY-AWARE MAPPING

5.1 Algorithms

Heuristic-based approaches can be justified as follows; If even a stage can be divided when it is hierarchically described, the complexity is not polynomial.

The number of PEs $\|V\|$ can be initially less than, equal to or greater than the number of stages $\|S\|$. Figure 2 covers the case in which $\|V\| = \|S\|$. The other two are very similar.

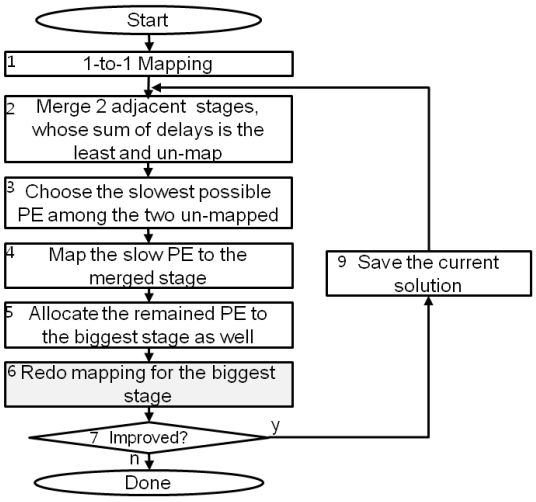


Figure 2: Hierarchy-Aware Mapping: $\|V\| = \|S\|$

In Figure 2, $\|S\|$ is or has become equal to $\|V\|$. S is sorted by the execution time given in the execution profile. V is sorted by speed. Following that, the one-to-one mapping in Box 1 Figure 2 is performed in order. One-to-one mapping has room for improvement since a set of stages are too small while another set of stages are too large. The quality of the design can improve if we save extra PEs by merging more consecutive small stages and assign the saved PEs to the stages with longer delays. In Boxes 1 through 5, Hierarchy-Aware mapping algorithms repeat merging stages, saving a PE, assigning the PE to the stage with the longest delay and testing whether there is improvement. Once the saved PEs are added to a stage, the stage should be repartitioned and remapped. Heuristics for repartitioning and remapping will be explained in the next section.

If $\|V\|$ is less than $\|S\|$, an intuitive way to map starts with merging several small consecutive stages so that $\|S\|$ becomes $\|V\|$. The total number of ways to reduce S is ${}_{\|S\|}C_{\|V\|}$. In addition, for each reduced S, there are $\|V\|!$ ways of mapping. Nonetheless, $\|S\|$, and thus $\|V\|$ in this specific case, are not large in practice, so an exhaustive search could be reasonable. In case where $\|S\|$ is less than $\|V\|$, S and V are sorted and injective mapping is conducted in order. The same procedures in Figure 2 follow except that Box 2 is skipped whenever there is any remaining PE.

5.2 Heuristics for Repartitioning and Remapping

When a saved PE is added to a stage, the stage should be repartitioned. Repartitioning is followed by remapping. Even finding the local optimal that minimizes the execution time of the stage takes exponential time. Therefore, heuristics giving suboptimal solutions are required.

In the heuristics, a process and a PE to which the process is moved is selected. The process is moved to the PE until there is no improvement. We formalize the problem. The algorithm takes three inputs: 1) a graph G, which is the subgraph of the given MoC representing the stage under concern 2) the set of the PEs given to this stage s V_s and

```
1: double ExecTime(G) {
2: if G is a leaf node then
3:   return the execution time of G on the mapped PE
4: end if
5: if G is a set of sequential sub program-states SubG_seq then
6:   return Σ ExecTime( sub_g ∈ SubG_seq )
7: end if
8: if G is a set of parallel sub program-states SubG_par then
9:   max := −∞
10:  for all v ∈ V_G where V_G is the set of PEs running part of G do
11:    delay := 0
12:    for all sub_g ∈ SubG_par do
13:      if sub_g or its part is mapped to v then
14:        delay+ = ExecTime( sub_g ) × α, 0 < α ≤ 1
15:      end if
16:    end for
17:    max = MAX( max, delay )
18:  end for
19:  return max
20: end if
21: }
```

Figure 3: ExecTime(G) Function

3) the execution profile. The output is P_s which is a set of non-overlapping subgraphs of G and one-to-one mapping $f : P_s \rightarrow V_s$ where the $ExecTime(G)$ is minimized.

Figure 3 is the algorithm to compute ExecTime. If the state is a leaf process, the execution time of the leaf process on the currently mapped PE is given from the execution profile. If the decomposition is sequential, delay of each sub program-state should be accumulated. It is complicated to compute ExecTime function with a program-state decomposed into parallel program-states. If each parallel sub program-state is mapped to different PEs respectively, ExecTime returns the maximum execution time. However, several sub program-states can be mapped to the same PE. Moreover, each sub program-state is hierarchical. For example, sub program-state0 and sub program-state1 can be partially mapped to PE v0. The execution time of program-state0 is affected by PE configurations, bus arbitration, RTOS scheduling, RTOS overhead and many others so is not predictable at this design stage. Therefore, approximation is required. As seen from lines 11 through 19, to compute the execution time of the sub graph of G mapped to v, we sum up ExecTime of all sub graphs, each is entirely or partially mapped to v. Before summation, we multiply α. The best value of α in Figure 3 could vary. We used 1, which encourages mapping parallel program-states separately.

6. CASE STUDY

The case study is performed with two streaming applications: Canny Edge Detector [10] and JPEG encoder. The platforms are randomly generated with different cost constraints. In the platform library, there are MicroBlaze processors, three different types of custom hardware and DSP models. We capture Canny Edge Detector in PSM following the modeling style of Han et al [11]. We modeled the

Table 1: Canny Edge Detector: Average Execution Time per Frame

Randomly Generated Platforms	Exhaustive Hierarchy-Unaware	Hierarchy-Aware	Execution Time Reduction
1	50.62 ms	38.68 ms	23.59%
2	50.32 ms	38.27 ms	23.95%
3	189.65 ms	110.45 ms	41.76%
4	39.5 ms	32.89 ms	16.73%
5	39.5 ms	23.87 ms	39.57%

Table 2: JPEG: Average Execution Time per Frame

Randomly Generated Platforms	Exhaustive Hierarchy-Unaware	Hierarchy-Aware	Execution Time Reduction
1	20.09 ms	14.14 ms	29.62%
2	19.03 ms	13.07 ms	31.32%
3	15.15 ms	14.14 ms	6.67%
4	19.03 ms	13.57 ms	28.69%
5	19.03 ms	17.03 ms	10.51%
6	15.15 ms	14.26 ms	5.87%
7	47.98 ms	40.99 ms	14.57%
8	41.93 ms	32.79 ms	21.80%
9	16.39 ms	16.18 ms	1.28%
10	15.15 ms	14.14 ms	6.67%

JPEG encoder on our own. The application is decomposed into four stages. The first two stages are decomposed into several hierarchical sequential/parallel sub program-states. The rest are leaf program-states. The platforms are randomly generated. The number of PEs varies from 2 to 4. The simulation framework in [12] is used to measure the execution times. Note that the framework reflects the impact of resource sharing and communication overheads.

Tables 1 and 2 compare, respectively, Hierarchy-Aware mapping to an exhaustive Hierarchy-Unaware Mapping in the average execution time of Canny Edge Detector and JPEG encoder. The latter is the optimal as long as hierarchy is not taken into account. Hierarchy-Aware mapping decreases the execution time by the average of 23.3%.

7. CONCLUSION

This paper describes Hierarchy-Aware mapping for pipelined applications executed in a pipelined manner. Balancing the execution time of each stage is crucial especially to minimizing the system's execution time. To balance stages, previous works have assumed hierarchy is flattened, merged consecutive stages or mapped parallel tasks separately. However, it is not always feasible to flatten the hierarchy of a general MoC. Therefore, our Hierarchy-Aware mapping also divides large stages/tasks into pieces while being aware of complex hierarchy in the stages/tasks. The case study is performed with JPEG encoder and Canny Edge Detector to compare Hierarchy-Aware mapping to the optimal pipeline-aware mapping that is unaware of hierarchy: the exhaustive hierarchy-unaware mapping. The execution time is decreased on average by 23.3%.

Acknowledgment

This work was supported in part by the National Science Foundation under NSF grant number 1136146.

8. REFERENCES

[1] International technology roadmap for semiconductor, 2011.

[2] Seng Lin Shee, A. Erdos, and S. Parameswaran. Heterogeneous multiprocessor implementations for jpeg:: a case study. In *Hardware/Software Codesign and System Synthesis, 2006. CODES+ISSS '06. Proceedings of the 4th International Conference*, pages 217–222, 2006.

[3] Jing Lin, A. Srivatsa, A. Gerstlauer, and B.L. Evans. Heterogeneous multiprocessor mapping for real-time streaming systems. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 1605–1608, 2011.

[4] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *SIGARCH Comput. Archit. News*, 34(5):151–162, October 2006.

[5] H. Javaid and S. Parameswaran. A design flow for application specific heterogeneous pipelined multiprocessor systems. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 250–253, 2009.

[6] A. Benoit, P. Renaud-Goud, and Y. Robert. Performance and energy optimization of concurrent pipelined applications. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, 2010.

[7] A. Benoit, L. Marchal, Y. Robert, and O. Sinnen. Mapping pipelined applications with replication to increase throughput and reliability. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*, pages 55–62, 2010.

[8] Daniel D. Gajski, Samar Abdi, Andreas Gerstlauer, and Gunar Schirner. *Embedded System Design: Modeling, Synthesis and Verification*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[9] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm. Technical report, 2001.

[10] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-8(6):679 –698, nov. 1986.

[11] Xu Han, Yasman Samei, and Rainer Doemer. System-level modeling and refinement of a canny edge detector. Technical Report # 12-14, Center for Embedded Computer Systems, November 2012.

[12] Yonghyun Hwang, Samar Abdi, and Daniel Gajski. Cycle-approximate retargetable performance estimation at the transaction level. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, pages 3–8, New York, NY, USA, 2008. ACM.