

Automating Hardware/Software Partitioning Using Dependency Graph

Somayeh Malekshahi, Mahshid Sedghi, Zainalabedin Navabi

Department of Electrical and Computer Engineering
University of Tehran, Tehran, Iran
{malekshahi, mahshid, navabi}@cad.ece.ut.ac.ir

Abstract

Hardware/Software (HW/SW) partitioning is a major step in embedded system design flow. This paper studies a new method of automated HW/SW partitioning. The main goal of this method is to improve performance in terms of the execution time and consumed area. Proposed algorithms are useful for partitioning using a dependency graph. This Graph models complete dependencies and flow of a design. We applied our method to a high-level description of MP3 decoder as a case study. The results show that a performance-improved automatic partitioning can be achieved using this method.

1. Introduction

Since the complexity of embedded systems increases with a fast pace, implementing all design in hardware (HW) may increase the necessary HW area [1-3]. Therefore, there is a trade-off between the utilized area and the complexity of the HW implementation. One of the most important solutions to this problem is partitioning the system into HW and SW [4]. As the complexity of embedded systems increases, HW/SW partitioning is becoming the critical problem in system-level design flow. A designer should make the final decisions to implement system components either in HW or SW.

This paper proposes a HW/SW partitioning method that improves the performance of a design to a good extent. In this method, a partitioned design is obtained from the high-level design description based on a few estimations for the execution time by *software profiling* and for the consumed area by *hardware profiling*.

The rest of this paper is organized as follows. Section 2 reviews the previous works briefly. Our method and algorithms are discussed in Section 3. In Section 4, the details of applying our method to a high-level description of a MP3 decoder as a case study are explained. Experimental results are also presented in this section. Section 5 concludes the paper and briefly discusses the future works.

2. Previous Works

There are several approaches to HW/SW partitioning. POLIS [5] is a strong method of system-level design. In

the POLIS co-design method, the target architecture is composed of a microcontroller and several hardware co-processors. The environment used in this method is based on Co-design Finite State Machine (CFSM) model. A group of CFSMs with an asynchronous communication model between them build a system. Static specification analysis is introduced in [6]. Because of the effect of data dependencies on control flow, the static analysis is not enough to solve this problem. Therefore, another analysis phase is required based on dynamic behavior of a design. This is done by design execution or simulation [7]. In [8, 9] a designer partitions the system into several components. Afterward, the designer implements the components with complex functionality in SW and leaves time-consuming components for HW.

With the increasing complexity of embedded systems and the importance of the time to market, manual partitioning has become one of the SOC design bottlenecks. In recent years, a lot of attention has been given to automatic HW/SW partitioning from higher abstraction levels [10]. Penry et al. [11] propose a method that estimates execution time and performs code analysis on system level, by the cycle count (CC) and hardware size using the gate count (GC). Next, functional HW/SW partitioning has been done using an extended Kernighan/Lin heuristic algorithm. In another work presented in reference [12], an exact integer linear programming, and a KLFM-based heuristic formulation is proposed for minimizing application execution time, schedule length, mapping each task to HW or SW, and comprehensive HW/SW partitioning for partial dynamic reconfiguration (RTR).

In [13], authors use a hybrid GA to solve the HW/SW partitioning problem. They start from a high-level description, and use a dynamically-weighted fitness function, while most parts of the design are implemented in HW to achieve execution time speedup. The large search space of the partitioning problem makes GA inappropriate for solving this problem. The main goal of the authors of the paper is to shrink the search space.

3. Our HW/SW Partitioning Method

In this paper, we mainly focus on achieving an automatic HW/SW partitioning from a high-level description of a design. High-level languages like C/C++ are not suitable for describing a design at system-level,

since they do not model hardware delays and the concurrency between hardware modules. Moreover, they do not support some essential HW data types, signals, and channels needed in the system-level design. C-based languages, like SystemC, and C-hardware are the most popular system-level description languages. As SystemC supports both HW and SW, it can be a good choice for automatic HW/SW partitioning.

We try to solve the partitioning problem in three phases: 1) exploiting complete data dependencies to identify the concurrent parts and presenting it through a graph called *dependency graph*, 2) extracting HW/SW specifications by profiling, and 3) selecting candidates for running on CPU or implementing on FPGA by evaluation of the analyzed design.

3.1 Dependency Graph

In this section, we just discuss the role of the *dependency graph* in details. The algorithms of extracting this graph are not explained here. This graph is used for finding a complete parallelism. The main characteristic of this graph is its independent parts located in the same level. In addition, some nodes of this graph need to be partitioned more based on their internal function calls. So, this graph can be a hierarchical graph. One way of improving the execution time of the design is to exploit concurrency in order to implement independent parts in HW. This results in a significant improvement of the execution time while adds a large HW area overhead.

In this hierarchal partitioned design, some main parts are functions located in the *main()* body. They contain several sub-parts such as loops, condition blocks, switch blocks, functions that are called in the caller functions, and basic blocks. Then this partitioned high-level code is analyzed to determine complete dependencies to exploit concurrency in HW/SW partitioning, while focusing on the global variables as the main criterion. The results are presented using a *dependency graph*. In this graph, independent parts are located at the same level, and they are candidates for parallel execution. Also, an edge between two nodes of the graph means that these two nodes should be executed sequentially.

3.2 Extracting HW/SW Specifications

In our method, information necessary for determining whether to implement each component in HW or SW is obtained by using *hardware* and *software profiling*. Therefore, CPU time and FPGA area of a candidate will be compared. *Software profiling* is a dynamic analysis used to extract run-time characteristics of a high-level design. One of the most important characteristics is the execution time of each part when it is running on the CPU. Functions with long execution time are potential candidates for being implemented in HW. This is just an initial assessment, and will be finalized by considering

the HW area and HW implementation complexity. This makes *hardware profiling* very essential.

Since the available design is described at a high level of abstraction and existing high-level synthesis tools do not support all the features of a high-level design, the evaluation is limited by the level of abstraction. Therefore, unlike *software profiling*, there is no way of automatic *hardware profiling*. In *hardware profiling*, we need to know how much area is needed for implementing each part and how long each part takes to execute. In *hardware profiling*, the necessary information are the number and the kind of each component in each line of the high-level description. In embedded system design, the partitioning process includes two steps: *allocation*, i.e., the selection of architectural components, and *mapping*, i.e., the binding of system functions to these components [14]. After generating *dependency graph*, each node of the graph is analyzed to extract hardware information. In order to perform such analysis, we use Sequential and As Soon As Possible (ASAP) strategies for scheduling and binding respectively. Then, the necessary HW area and execution time of each node of the graph are estimated. Our binding method improves the execution time without imposing a large area overhead which is a result of the sequential running. For *hardware profiling*, a library named *hw_library* is used which contains the exact implementation of all components.

3.3 Deciding on HW/SW Parts

In this approach, some decisions are made for determining hardware and software parts by considering two key parameters, time and area. The starting point is the manager of the high-level description, *main()*. All parts of *main()* should be evaluated separately. If an exact decision cannot be made for each part, the same work should be repeated hierarchically. This work contains two steps and we take advantage of the information in the *dependency graph*. These steps are explained bellow as an algorithm shown in Figure 1.

Step1: In *software profiling*, all parts are evaluated for running on the system's CPU. If the sequential execution time of a part is a small fraction of the total sequential execution time of the design, referred to as T_{CPU} , we choose this part for running as a CPU task.

Step2: After Step1, all remaining parts are candidates for being implemented in HW. Calculation of the total HW area, which we refer to as A_{total_HW} , is based on the *dependency graph* and the time and area of each node obtained by *hardware profiling*. The role of the *dependency graph* is extracting two types of concurrencies automatically, the HW parallelism and simultaneous work of HW and CPU to improve the execution time.

The best option for improving the execution time is implementing all HW candidates on FPGA and trying to utilize concurrency as much as possible.

```

Input: high level design
Output: a partitioned design
partitioning(){
  for (all parts of the design){
    if (Tcpu(part(i)) < 20% Tcpu ){
      Tdiff = Tcpu(part(i)) - Thw(part(i));
      if (Tdiff is not considerable)
        SW_candidate(this Part)= true;
      else
        HW_candidate(this Part)= true;
    } else
      HW_candidate(this Part)= true;
  }
  for each level of graph
  {
    All levels of graph  Number of parts in this level
    A(S)=Atotal_hw=∑j=1n (∑i=1n (Area HW (part(i))));
    All levels of the graph
    Time=∑j=1n (MAX(Timehw:all parts of each level));
  }
  if (A(S) = Atotal_hw < Areaavailable)
  {
    implemented in HW(HW candidates);
    Check HW complexity;
  }
  }else ...

```

Figure1. Algorithm 1 Used for Partitioning

If the total area of HW candidates is more than the area of the selected FPGA, some parts should be removed to achieve allowable area, $Area_{available}$. We propose some strategies to do this:

Strategy 1: Consider those HW candidates that are some nodes of the graph not located in the same level of the *dependency graph*. The node with the shortest execution time is a better choice for removal from HW and execution on the CPU. See Figure 2.

```

...else{
while(Areahw>Areaavailable || HW implementation is
complex){
switch (strategy){
case 1:
for(HW candidate nodes that are not located
at the same level of the graph){
find(node with the least CPU time);
Areahw =A(s+i)= A(s) - Areahw(selected);
if (Areahw > Areaavailable)
strategy=2;
case 2:
if(Areahw(selected) == great) {
partition this node hierarchically;
strategy=1;
}
if(Areahw>Areaavailable || HW implementation
== complex)
strategy=3;
case3 :
choose( parts among parallel nodes with
the least
execution time on CPU);
if (Areahw < Areaavailable)
strategy=2;
}
}

```

Figure2. Algorithm 1 Used for Partitioning, Cont.

Strategy 2: If the node chosen by Strategy 1 has a large area, a large amount of area will be released by omitting this node from HW. It is better to evaluate this node hierarchically and try to choose only some parts of this node to remove from HW.

Strategy 3: In the worst case, if by removing nodes of Strategy 1 the consumed area still exceeds the area of the available FPGA, we should choose more parts from the parallel nodes that are in the same level of the graph. In this situation, again we can work hierarchically to remove some parts of the node with the shortest sequential execution time and in the worst case, the complete node. See Figure 2.

4. Experimental Results

We applied our method to a high-level description of a MP3 decoder [15]. At first, we generated the *dependency graph* of the description. There are three primitive parts, *Part1*, *Part2*, and *Part3*. These parts are determined based on functions of the *main()*. *Part1* has a recursive function, so we never consider it as a hardware part. In *Part2*, the fundamental portion is *MP3_decode_frame* which is located inside a *while loop* that is repeated for many iterations, so this is the bottleneck of this design that we focus on. See Table 1.

Table1. Results of HW/SW profiling

# Part	Time _{HW} in one iteration	Time _{CPU}	Area _{HW}
Part1	-	65.448, (0.3 %)	> 9236
Part2	185.197	19665.331, (81.3 %)	9381
Part3	0.017	0.127, (0.0 %)	2394

Based on Step1 of Algorithm1, *Part2* is a candidate for hardware implementation, but it does not fit in the selected FPGA area, $Area_{available}$. Therefore, it needs to be partitioned further. *Part2* contains a *while loop* with two functions *MP3_decode_frame* (with only one part, *mp_decode_frame*) and *MP3_GetFrameSize*. With the current partitioning, the area is still great, based on Step2 of Algorithm1. Therefore, another partitioning phase should be applied to this part. The *mp_decode_frame* block has two main parts *mp_decode_layer3* and *for loop* contains a function, *synth_filter* located in *for loop*.

Table2. Results of HW/SW profiling

# Part	Area _{HW}	Time _{HW}
mp_decode_frame	8997	92.567
mp_decode_layer3	8997	8.243
for (synth_filter)	6866	84.373
P2_1	8561	1.064
P2_2	5925	0.623
P2_3	6470	4.203

The part located in *for loop*, (*synth_filter*) based on case1 of Algorithm1 in Figure2 is a good choice for HW implementation. However, $Time_{CPU}$ of *mp_decode_layer3* is 10941.908 (45.2% of $Time_{CPU}$). Therefore,

mp_decode_layer3 is neither suitable for HW nor SW, so it should be partitioned further. The *mp_decode_layer3* contains two parts *P1*, *P2*. The *P2* part has three parts, *P2_1*, *P2_2* and *P2_3*. These parts are independent. But for implementing all of them in HW to run concurrent, a large area is needed. Because of their concurrency, it is more suitable to let both HW and CPU execute them. *P2_1* is more time-consuming in CPU, *P2_2* can run easily in CPU, and *P2_3* does not need a long time in CPU but the necessary HW for implementing it is considerable. See Table 2 and case3 of Algorithm1 in Figure2.

After determining the HW and SW parts by considering the *dependency graph*, the designer should be able to finalize the design, see Figure 3. The total time compared with complete SW implementation and total areas compared with pure HW implementation are decreased. Manual partitioning takes almost 48 hours. In this work, we use APEX20KE for *hw_library* components [16], and Celerone, Intel processor [17].

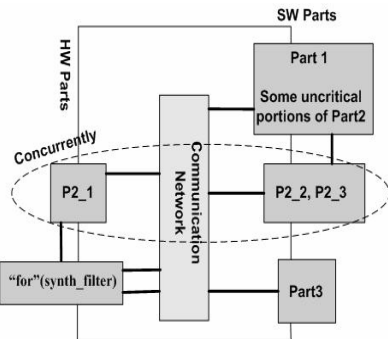


Figure3. Final Decisions for MP3 Co-design

Table4. Final comparison

Partitioning Results	Time	Area of HW
Before partitioning	24195.275	> 9381
After partitioning	16041.4673	8561
Improvement	33.7	8.75

5. Conclusions and Future Work

Today's design methodologies cannot support the complex embedded system designs. Therefore, new methods are needed to handle today's complicated designs. One of these new methods is co-design achieved by collaborating both hardware and software designers. In this paper, we have presented an approach to HW/SW partitioning at the system-level design. The proposed graph helps finding the exact dependency to exploit parallelism in order to improve execution time automatically. Some defined strategies help us finalize the partitioning decisions by considering time and area limitations. As a future work, it is useful to find an appropriate estimation for communication overhead and increase its effect on the presented design methodology.

References

- [1] J. D. Davis, L. Hammond, and K. Olukotun, "A flexible architecture for simulation and testing (FAST) multiprocessor systems," In Proc. of 1st Workshop on Architecture Research Using FPGA Platforms, 2005.
- [2] E. Nurvitadhi, and J. C. Hoe, "Full-system architectural exploration sandbox," In Proc. of 1st Workshop on Architecture Research Using FPGA Platforms, 2005.
- [3] S. Wee, J. Casper, N. Njoroge, Y. Tesylar, D. Ge, C. Kozyrakis, and K. Olukotun, "A practical FPGA-based framework for novel CMP research," In Proc. of 15th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 2007, pp. 116-125.
- [4] B. Knerr, M. Holzer, and M. Rupp, "HW/SW Partitioning Using High Level Metrics," In Proc. of International Conference on Computing, Communications and Control Technologies, 2004.
- [5] F. Balarin, E. Stentovich, M. Chiodo, H. Hsieh, B. Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli, "Hardware-Software Co-Design of Embedded Systems - The POLIS Approach", Kluwer Academic Publishers, 1997.
- [6] W. Hardt, and R. Camposano, "Specification Analysis for HW/SW-partitioning," In Proc. of GI/ITG Workshop Application of Formal Methods during the Design of Hardware Systems, 1995, pp. 1-10.
- [7] W. Ye, R. Ernst, T. Benner, and J. Henkel, "Fast timing analysis for hardware-software cosynthesis," Computer Design: VLSI in Computers and Processors, 1993, ICCD '93, pp. 452-457.
- [8] D. Chiou, H. Sunjeliwala, D. Sunwoo, J. Xu, and N. Patil, "FPGA-based fast, cycle-accurate, full-system simulators," In Proc. of 2nd Workshop on Architecture Research using FPGA Platforms, 2006.
- [9] T. Risset, Compsys, Lip, ENS-Lyon, "Some Trends in High-level Synthesis Research Tools," <http://www.ens-lyon.fr/COMPSYS>.
- [10] S. Banerjee, and N. Dutt, "Efficient Search Space Exploration for HW-SW Partitioning," Hardware/Software Co-design and System Synthesis (CODES+ ISSS), 2004, pp. 122-127.
- [11] D. A. Penry, Z. Ruan, and K. Rehme, "An Infrastructure for HW/SW Partitioning and Synthesis of Architectural Simulators," In Proc. of. Workshop on Architectural Research Prototyping, 2006.
- [12] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "Physically-aware HW-SW partitioning for reconfigurable architectures with partial dynamic reconfiguration," In Proc. of DAC 2005, pp.. 335-340.
- [13] P. Mudry, G. Zufferey, and G. Tempesti, "A dynamically constrained genetic algorithm for hardware-software partitioning" In Proc. of Genetic and Evolutionary Computation Conference, 2006, pp. 769-776.
- [14] B. Knerr, M. Holzer, and M. Rupp, "Improvements of the GCLP algorithm for HW/SW partitioning of task graphs," In Proc of 4th IAESTED International Conference on Circuits, Signals, and Systems (CSS '06), 2006, pp. 107-113.
- [15] <http://sourceforge.net>
- [16] <http://www.altera.com.cn/literature/hb/nios2>
- [17] <http://www.intel.com/products/processor>