A Design Methodology for System level Synthesis of Multi-core System Architectures

†Karim Yehia, ††Mona Safar
†Department of Electronics and Communications Engineering, Cairo University, ††Department of Computers and Systems Engineering, Ain-Shams University Cairo, Egypt

Hassan Youness, Mohamed AbdElSalam, Ashraf Salem
Mentor Graphics Egypt Cairo, Egypt
{hassan_youness, mohamed_abdelsalam, ashraf_salem}@mentor.com

Abstract—A multi-core system is an integrated circuit containing multiple processor cores that implements most of the functionality of a complex electronic system and some other components like FPGA/ASIC on a single chip. In this paper, we present a novel approach to synthesize multi-core system architectures from Task Precedence Graphs (TPG) models. The front end engine applies efficient algorithm for scheduling and communication contention resolving to obtain the optimal multi-core system architecture in terms of number of processor cores, number of busses, task-to-processor/channel-to-bus mapping, optimal schedule, and HW/SW partition. The back end engine generates a SystemC simulation model using a well-known commercial tool model generation library. The viability and potential of the approach is demonstrated by a case study.

I. INTRODUCTION

Multi-core designs represent today the main trend for future architectural designs. It poses number of problems [1] such as extracting parallelism from applications, partitioning application into tasks, allocating and scheduling tasks to multiprocessors and coordinating communication and synchronization between processors.

Furthermore the process of automating the generation of realistic multi-core architectural designs from system level specifications remains a challenge for nowadays EDA design tools and the true complexity of the problem is the reason why until to date no tool in this area has become a commercial success. Several system level modeling and design space exploration methodologies for streaming multiprocessors especially heterogeneous media and signal processing systems have gained wide acceptance in system level community, among those methodologies are SPADE [2], Sesame [3], CASSE [4], and CoWare’s approach [5]. These methods share the Y-chart concept that separate between applications and architectures. SPADE and Sesame start with a Kahn Process network (KPN) as design entry and use trace-driven simulation, where execution of architecture model is driven by traces from execution of application model. CASSE follows a programming model based on Task Transaction Level Interface (TTL) [6] and uses SystemC based simulation. On the other hand, CoWare uses a set of un-timed reactive SystemC tasks communicating through a Transaction Level Model interface. In work of [7] they used an extended Task Graph (eTG) as design entry, and generated a scheduled SystemC performance model that is further synthesized into interacting H/W modules with control unit implemented as FSMs. Researchers have also proposed system level modeling methodologies for integrated HW/SW embedded systems, but aiming at automated HW/SW partitioning, co-design, code generation from formal specification languages, and refinement from abstract system level models to more realistic implementations including communication refinement. The Ptolemy [8], Metropolis [9], and COSY [10] environments are pioneering work in that direction.

In this paper, we present a system level synthesis approach to automate the process of architecture exploration and generation of specialized multi-core system architectures from Task Precedence Graphs (TPG). Fig. 1 shows the outline of the proposed synthesis approach.

The design entry is a Task Precedence Graph with communication channels (TPG). The output is an Architecture Level Model (ALM) of a specialized multi-core system architecture described in eXtended Markup Language (XML) format. The ALM description is then processed by a utility we developed (XML Parser/Generator) that will interpret the structure and operation of an optimal (or semi-optimal) configuration of our specialized multi-core system architecture, instantiate the appropriate SystemC TLM 2.0 compliant models from a library of fast generic models and connect the target multi-core system architecture.

The front-end engine of our synthesis approach is an efficient algorithm we developed to solve the task allocation/scheduling problem to obtain the optimal schedule on a multiprocessor system and reduce the number of processors in the target system. The algorithm also resolves communication channels conflicts and reduces the overall execution time of the application task graph by using
HW/SW partitioning that depends on fast conversion from SW to HW tasks.

The rest of this paper is organized as follows. Section II describes the output & input of our proposed synthesis approach and the algorithm of the front-end engine. Section III shows an illustrative example “case-study” to clarify the algorithm operation and back-end generation. Section IV draws some conclusions.

Fig. 2. Specialized Multi-core system architecture

Fig. 3. TPG practical example

The geometry of the task graph divides it into paths and levels; the path is the way from the start node to the exit node considering all the nodes and the edges in this way. The levels are depths of nodes. In scheduling a task graph G onto a target system with a set of processors P, each node must be assigned to one processor. Classical assumptions [11] that are made about the target system are:

i) Tasks are non-preemptive

ii) Local communications are cost free

Let \( t_i(n_i) \) be the execution start time of node \( n_i \), and \( w(n_i) \) be the weight of node \( n_i \). The execution finish time of node \( n_i \), \( t_f(n_i) \) can be denoted as

\[
t_f(n_i) = t_i(n_i) + w(n_i)
\]

where a processor can execute only one task at the same time, two nodes \( n_i \) and \( n_j \) assigned to the same processor should satisfy \( t_f(n_i) < t_f(n_j) \) or \( t_f(n_j) < t_f(n_i) \). For an edge \((n_i,n_j)\), if \( n_i \) and \( n_j \) are assigned to the same processor, communication cost becomes zero since there is no data transfer between processors. Hence, we define edge finish time \( t_f((n_i,n_j)) \) as the following equation where \( c(n_i,n_j) \) denotes communication cost of edge \((n_i,n_j)\):

\[
t_f((n_i,n_j)) = t_f(n_i) + c(n_i,n_j)
\]
\[ t_r((n_i, n_j)) = \begin{cases} t_f(n_i), & \text{if } n_i, n_j \text{ on same processor} \\ t_f(n_i) + c(n_i, n_j), & \text{if } n_i, n_j \text{ on different processors} \end{cases} \]

Node \( n_i \) can start to execute after all data are ready, i.e., all edges connected to node \( n_i \) are finished. Hence data ready time of node \( n_i \), \( t_R(n_i) \), is defined as:

\[ t_R(n_i) = \max_{(i, l, n_j) \in E} \left\{ t_f(n_i), t_f(n_j) \right\} \]

Scheduling length of scheduling \( S \), \( SL(S) \), is the time when all nodes are finished. Hence \( SL(S) \) is defined as follows:

\[ SL(S) = \max_{n \in V} \{ t_f(n) \} \]

Two processors \( p_i \) and \( p_j \) are isomorphic if the ready times of them are equal and the tasks \( n_i \) and \( n_j \) are equivalent when the following conditions are satisfied.

i) \( \text{pred}(n_i) = \text{pred}(n_j) \)

ii) \( w(n_i) = w(n_j) \), and

iii) \( \text{succ}(n_i) = \text{succ}(n_j) \)

According to the above assumptions, we can derive the following fact. Scheduling length of optimal task assignment to \( P+1 \) processors is always less than or equal to the one to \( P \) processors,

\[ SL(S_{opt}(P+1)) \leq SL(S_{opt}(P)) \]

where \( SL(S) \) represents scheduling length of scheduling \( S \) and \( S_{opt}(P) \) represents optimal scheduling to \( P \) processors.

A. Front-End Engine Algorithm

The main role of the front-end engine algorithm is to obtain optimal design parameters for the multi-core system architecture given a Task Precedence Graph (TPG) model. The objective is to maximize performance in terms of execution speed under some constraints as precedence, area, and communication conflicts. In order to achieve this we divided the front-end engine algorithm into two processes.

1) Scheduling, Cores Reduction, and Channel Conflicts Resolving Process

We used the A*-star as a best-first space search algorithm that maintains two lists \( OPEN \) and \( CLOSED \). It starts by putting the initial state into the \( OPEN \) list. The states in the list are sorted according to the cost function \( f(s) \). The cost function \( f(s) \) is the addition of \( g(s) \) and \( h(s) \), where \( g(s) \) represents the cost from the initial state to the state \( s \) and \( h(s) \) is the estimated cost from the state \( s \) to the goal state. In each step, the state \( s \) with the lowest \( f(s) \) value is taken from the list and put onto \( CLOSED \) list. The algorithm terminates when the state \( s \) is the goal state.

The task graph is partitioned according to the geometrical shape of paths and levels. The path length (sum of weights of nodes and edges) are calculated and sorted in descending order. From paths and levels we build a matrix called the geometrical matrix \( (G_{matrix}) \). Then, we use pruning techniques to reduce the number of search states, for example, the property of isomorphic processor and node equivalence. Lastly, the algorithm reduces the number of processors to be used for scheduling in case of idle time.

The Geometric A* Algorithm (GAA) [12] is outlined below where \( Q \) represents the set of used processors (Note. at least one task is assigned to the processor)

The Scheduling Algorithm (GAA)

1. Input the task graph and the number of Processors
2. Initialize \( OPEN, OPEN_R \) and \( CLOSED \) list
3. Build the geometrical matrix \( G_{matrix} \)
4. Extract the priority matrix \( P_{matrix} \)
5. For level = 1 to \( L \) do
6. Take the repeated tasks from the \( P_{matrix} \) into the \( OPEN_R \) list and non repeated tasks in the \( OPEN \) list
7. While (\( OPEN_R \) or \( OPEN \) list not empty) do
8. For \( i = 1 \) to \( ready\_nodes \) do
9. For \( j = 1 \) to \( P \) do
10. If (\( P \) is isomorphic) then
11. \( n_i \rightarrow \) only one of \( P \)
12. Else if \( P_i = Q \) then
13. \( n_i \rightarrow P_j \)
14. Else if \( P_i \rightarrow P_j \) then
15. \( n_i \rightarrow P_j \) only one of \( P \)
16. End if
17. End for
18. End for
19. Transfer the scheduled node to \( CLOSED \) list
20. While False do
21. End while
22. End for
23. End for

The algorithm schedules the node to only one processor in case of there are many isomorphic processors and at the same time it searches for the used processors that has a ready time less or equal to the ready time of the isomorphic one (step 14) to reduce the number of processors that can be used in the scheduling process. The algorithm is explained in section III as illustrative example.

To derive the time complexity, suppose that the application has \( n \) nodes with \( l \) levels, \( h \) paths and \( r \) repeated node and this application should be scheduled on \( p \) processors. Step 5 shows that there is \( l \) loop of repeated nodes \((r)\) and non-repeated nodes \((n-r)\) on \( p \) cores. Then \( l(lp) + ((n-r)p)) = lpn \) so, in the worst case \( l = E \) (i.e., \( O(Epm) \)) and if all the nodes are fully connected then \( E = (n^2-n)/2 \), so the time complexity is \( O(n^3-n^2) \).

Channel conflicts are resolved by mapping conflicting channels to different buses. This scheme sacrifices area (bus interconnect) to preserve the optimal schedule length.

This can be formulated as follows; Given a Channel Conflict Graph, defined as undirected graph \( G_{Conflict} = (V, E) \), where \( V \) denotes communication channel and \( E \) is conflict relation on \( V \). The algorithm maps \( c : V → B \) where \( B \) is the set of buses such that, \( c(u) ≠ c(v) \) for \( u, v \) in \( G \).

2) Partitioning Process

The partitioning starts after scheduling by searching for the critical path (i.e., the Processor core that has the longest schedule length) and converts SW tasks running on that core
to HW tasks provided that SW tasks attributes indicate feasibility to convert to HW. This conversion has an impact on the execution time of each task by a reduction of \((1/3 \sim 1/5)\) of its original computation time value.

III. CASE STUDY AND EXPERIMENTAL RESULTS

We present a case study to illustrate the operation of the front-end engine algorithm using the task graph shown in Fig. 3; Initially we assumed the target system contains three processor cores on a single bus:

Step (1): The task graph is partitioned into paths and levels, these paths are sorted in descending order by their values. There are 5 paths and 3 levels from start node to the goal node.

Path1 is T2, T3, T6 value (path1) = 121642 cycles
Path2 is T1, T3, T6 value (path2) = 17783 cycles
Path3 is T1, T5, T7 value (path3) = 10119 cycles
Path4 is T1, T4, T6 value (path4) = 8993 cycles
Path5 is T1, T4, T7 value (path5) = 8958 cycles

Step (2): Levels
Level 1 is T1, T2
Level 2 is T3, T4, T5
Level 3 is T6, T7

Step (3): The \(G\) matrix is built from paths and levels; it gives a matrix with 5 rows and 3 columns

\[
G_{\text{matrix}} = \begin{bmatrix}
T2 & T3 & T6 \\
T1 & T3 & T6 \\
T1 & T5 & T7 \\
T1 & T4 & T6 \\
T1 & T4 & T7
\end{bmatrix}
\]

Step (4): According to the target system \((P=3)\), a priority matrix called \(P\) matrix, has rows and columns equal to the number of processors in the system and the \(P\) matrix is extracted from \(G\) matrix by taking the first three rows and three columns.

\[
P_{\text{matrix}} = \begin{bmatrix}
T2 & T3 & T6 \\
T1 & T3 & T6 \\
T1 & T5 & T7
\end{bmatrix}
\]

Step (5): As noticed from the \(P\) matrix, there are some nodes that are repeated in the same level according to their relation to the child nodes as T1 here has 3 children. So, these nodes are called the repeated nodes and should be put in the \(OPEN_R\) list and the other nodes are called non-repeated nodes and should be put in the \(OPEN\) list.

\(OPEN_R = T1, T3, T6\)
\(OPEN = T2, T4, T5, T7\)

Step (6): The scheduling process will begin as in the algorithm using the \(P\) matrix and the A-star fitness function \(f(s)\), Tables (1, 2) show the function values of scheduling for tasks using 3 and 2 cores respectively.

\[
\begin{array}{|c|c|c|}
\hline
\text{Level} & \text{Task} & \text{Core 1} & \text{Core 2} & \text{Core 3} \\
\hline
1 & T1 & 6762+11021* & & \\
2 & T2 & 108821+12821* & & \\
3 & T3 & 120536+1106* & 118536+1106* & 120536+1106* \\
4 & T4 & 7687+1106* & 7887+1106 & \\
5 & T5 & 8848+1071 & 9048+1071* & \\
6 & T6 & 119642+0 & 119442+0* & 119642+0* \\
7 & T7 & 11904+0 & 10990+0* & \\
\hline
\end{array}
\]

Fig. 4 shows the Gantt chart (schedule) for each case. By observation, it turned out that scheduling on two cores has the same schedule length as three (i.e. cores are reduced). We need two buses when three cores are used (c1 conflicts with c3 and c5 conflicts with c6 “when drawing the GANTT chart to scale”). Using graph coloring (c1 and c5) are mapped to bus 1 and (c3 and c6) to bus 2.
The partitioning process is done after scheduling/processor cores reduction, and channel conflicts resolving stage as a final step to compact the SL along its critical path. The second choice of two cores on one bus is selected for the partitioning process as outlined next.

Step 1: Find the longest schedule length on cores, processor P2 has SL = 119642 cycles, including the time for data write (Channel-W) and read time (Channel-R) from and to tasks that are implemented on other core respectively.

Step 2: The SW tasks on the longest SL will be converted to HW tasks to reduce the overall execution time. We assume the execution time of a HW task on FPGA equals one third to one fifth of the execution time of a SW task on CPU core. In this case study, all tasks have feasible attribute for conversion to H/W tasks.

Step 3: The target architecture is reduced to one bus, one processor core with HW tasks (i.e. FPGA) and one processor core with SW tasks. Task T1 send data to task T3 via channel (C1) and task T4 send data to T6 via channel (C5), the channels are mapped to one bus (no conflicts) and the SL is reduced to 29910 cycles, see Fig. 5.

As for backend generation, the optimal (or any semi-optimal configuration) can be selected for SystemC generation, both the structural and scheduling info is written to an XML file. The information is parsed and processed by a utility we developed to integrate the front-end engine to Mentor Graphics® Vista Architect™ Model Builder tool [14]. The utility would instantiate the appropriate SystemC TLM 2.0 compliant models from a library of fast generic models (e.g. Processor core with SW/HW Thread, Bus “AHB/AXI”, MEM/Cache, etc.) and connect the target multi-core system architecture based on the design parameters and configuration in the XML file. As for the interconnect (See Fig. 6) and dynamics (see Fig. 7) of the final generated multi-core system, we used the built-in generic memory model for modeling the shared and the local memories, the bus model which is currently an AHB bus for modeling the system interconnect, and the CPU model for modeling the Real-Time Resource Scheduler (RTRS). We build a number of processor core models for the main processing element with a SW or HW Task thread. For the interface protocols, we used an AHB master/slave TLM protocol for the bus interface between RTRS, Processor cores, as well as shared memory and the system AHB bus. We also used a Tightly Coupled Memory (TCM) protocol for the interface between the processor cores and their associated local memories. The tasks running on each core is statically bounded according to the scheduling algorithm results. Each task is modeled as an SC_THREAD that is suspended for the specified execution time based on the computation cost of the task (SW or HW). Access to channels is modeled as bus read/write transactions from specific memory addresses.

The memory address space that can be accessed through each bus is specified in a parameter file to correctly map the channels to their dedicated buses. The time spent in a channel read/write transaction (i.e. communication cost) is specified using the parameterized memory read/write associated delay timing policy. The SW thread associated with the RTRS is modified to initiate tasks at different cores to run according to the scheduling algorithm results. An SC_METHOD “control callback” in each processor core is sensitive to the control interfacing port connected to the RTRS. This method notifies the “Go” event associated with each of the statically allocated tasks which are implemented as SC_THREADS. The Scheduler thread in the RTRS is responsible for generating correct control information at appropriate time to realize the pre-determined scheduling.

Upon simulating the generated SystemC code equivalent to the three processor configuration in Fig. 4, the following statistical charts are directly obtained using the Vista Architect™ analyzing utility. Fig 8. shows three cores (red = P1, green = P2, and blue = P3) and tasks id’s on Y-axis versus time in cycles on X-axis which is equivalent the Gantt chart of Fig. 4. (c) when drawn to scale. Other analysis widgets are also available for calculating throughput and latency on busses for example Fig. 9 shows the read/write transactions on buses 1 and 2 for the given configuration.
IV. CONCLUSION

In this paper we presented a system level synthesis approach for multi-core system architectures. The front-end uses an efficient algorithm for optimal architecture selection. The algorithm solves task scheduling problem and obtain the optimal schedule on a specialized multi-core system architecture with a capability of (i) reducing the number of cores in the target system, (ii) resolving communication contention over buses and (iii) minimizing execution time by searching for the cores that have the longest schedule length and converts their tasks to be executed in hardware instead of software if feasible. The output of the algorithm which is typically an optimal ALM structure/schedule is then fed to a utility (back-end generator) that uses Vista Architect Model builder™ to architecturally build the system with appropriate TLM components and dynamics for SystemC simulation.

REFERENCES