An Automatic Programming Tool for Heterogeneous Multiprocessor Systems

Adriano Tavares
Department of Industrial Electronics
University of Minho
4800 Guimarães, Portugal

Carlos Couto
Department of Industrial Electronics
University of Minho
4800 Guimarães, Portugal

Abstract - Recent advances in network technology and the higher levels of circuit integration due to VLSI have led to widespread interest in the use of multiprocessor systems in solving many practical problems. As the hardware continues to diminish in size and cost, new possibilities are being created for systems that are heterogeneous by design. Parallel multiprocessor architectures are now feasible and provide a valid solution to the throughput rates demands of the increasing sophistication of control and/or instrumentation systems. Increasing the number of processors and the complexity of the problems to be solved makes programming multiprocessor systems more difficult and error-prone. This paper describes some parts already implemented (mainly the scheduler) of a software development tool for heterogeneous multiprocessor system that will perform automatically: code generation, execution time estimation, scheduling and handles the communication primitive insertion.

I. INTRODUCTION

Control and/or instrumentation application frequently requires very high peaks of processing power demands that can not be satisfied by a single processor system, and often the solution is found in a multiprocessor architectures. Developing software for such kind of architecture is very hard and usually realized at each individual processor's level with the compatibility among processors carried out by the programmer. The lack of a software development tool, independent of the multiprocessor hardware platform is the main reason against the use of integration potentialities in the development of the On-chip multiprocessor architectures.

The Commercial manufactures usually make integration of a processor and diverse kinds of peripherals but never multiprocessor integration. The few existing multiprocessor architectures emerge from discrete processors that after a completely independent development (one for each processor) are later integrated in one chip - ASIC.

This problem becomes harder since with instrumentation and/or control system, skills are necessary to deal with additional degree of complexity that emerges from different contributions of the different processors.

Parallel processing enhancement can be divided into two broad categories: On-chip versus off-chip parallelism. The Texas Instruments C80 (contains four DSPs and one RISC processor) is an example of the newest approach in on-chip parallelism that consist to put more than one CPUs onto one IC. Off-chip parallelism employs several processors, working together on a task.

We consider a heterogeneous multiprocessor system to be a system which makes use of several different types of processors, and/or connectivity topologies to optimize performance and/or cost-effectiveness of the system. For example, different processor types could include vector processing, SIMD[1] processors, MIMD[2] processors, special purpose processors, and data flow processors. Similarly, different connectivity paradigms could include bus, point-to-point, ring, or a mixture of these. Examples of heterogeneous systems can be found in:

1) Embedded systems which are application specific systems containing hardware and/or software tailored for a particular task. General purpose processors, programmable digital signal processors, and ASICs are among the many components of these system. In this kind of systems, heterogeneous processors are tightly coupled with low IPC[3] overhead but with heavy resource constraints.

2) Distributed systems, which consist of various workstation, main computers, and even super computers, with significant overhead of interprocessor communication.

3) SHHiPE[4] which is an embedded system developed by integrating COTS[5] processors and interconnect components in a 'plug-and-play' fashion. These systems offer advantages of low-cost, scalability, easy programmability, software portability, and ability to incorporate evolving hardware technology [1, 2].

Examples of application tasks where heterogeneous multiprocessor systems are desirable can be found in any domains, including instrumentation and/or control, robotics and digital signal processing.

Even if the hardware is built, the application program must be written, debugged, and maintained. When programming this kind of system new problems arise compared to programming a single processor system. Some of these are: program scheduling (good scheduling algorithms are essential to exploit the full parallelism of the hardware), load balancing, processors synchronization and communication routing. These tasks are often tedious and therefore programming facility is needed to simplify the development task while maximizing performance. Programmers must be able to specify algorithms at a sufficiently abstract level so that they are not overwhelmed with trivialities while still having access to those details

[1] Single Instruction Multiple Data
[2] Multiple Instructions Multiple Data
[3] IPC = Interprocessor Communication
[4] Scalable Heterogeneous High Performance Embedded
[5] COTS = Commercial Off The Shelf

that are needed by system software (compiler, linker, and run-time support) to efficiently execute the code. Furthermore, as applications become more sophisticated, debugging becomes more complex.

Our main purpose in this project will be the development of a visual and automatic programming environment for heterogeneous multiprocessor system based on microcontrollers and DSPs for instrumentation and/or control applications, that take advantage of available system resources and make the programming as easily as possible. After some studies and reflections, we decide that this programming environment will be composed of four tools: a graphical editor for representing the program and the target machine graph, a scheduler for partitioning and mapping the parallel program onto the processors, the code generator for each processor types, a predictor for the worst-case execution time of each task presents in the program graph, and finally a program performance visualizer.

At this moment, we have concluded the implementation of the graphical editor and of several scheduling algorithms for homogeneous and heterogeneous systems.

II. GRAPHICAL EDITOR

The graphical editor's main purpose is the representation of the program graph and target machine graph and has two very important features:

1- browsing large graphs through incremental layout.
2- visualization of dynamic graphs to be used later by the performance tool.

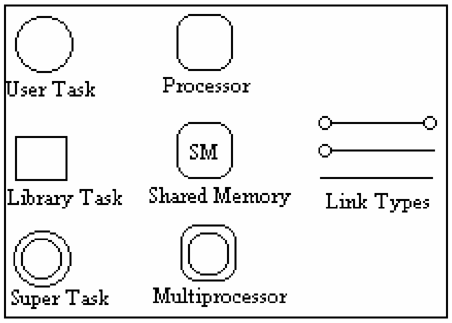

Fig.1 Some Graphical Editor Objects

A parallel program is composed by N separate cooperating and communicating tasks and its behavior is represented by a directed acyclic graph (DAG) named program graph or task graph (fig.2). A DAG is defined by a tuple G = (V, E, C, T) where V is the set of task nodes and v = |V| is the number of nodes, E is the set of communication edges and e = |E| is the number of edges, C is the set of edge communication costs and T is the set of node computation costs. The values $c_{i,j} \in C$ is the communication cost incurred along the edge $e_{i,j} = (n_i,n_j) \in E$, which is zero if both nodes are mapped in the same processor. The value $t_{ij} \in T$ is the execution time of node $N_i \in V$ onto processor p_j.

The multiprocessor system, onto which the tasks are scheduled is represented as a weighted undirected graph (fig.3) $G_p = (V_p, E_p)$, where $V_p = \{v_q : q = 1,2, \ldots,m\}$ is the set of processors with associated service rates u_q and $E_p=\{(p,q): p, q = 1, 2,\ldots,m, p\neq q\}$ is the set of links with associated link capacities c_{pq}. Fig. 1 shows some drawing objects used to represent the task and system graphs. The graph system in fig.3 represents an architecture compose by three processors communicating through a shared memory connected to a shared bus and a dedicated point-to-point link between processors P_0 and P_2 (fig.4).

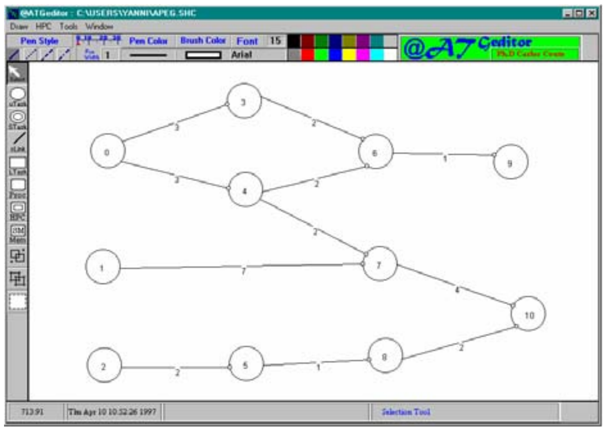

Fig.2 Task graph representation

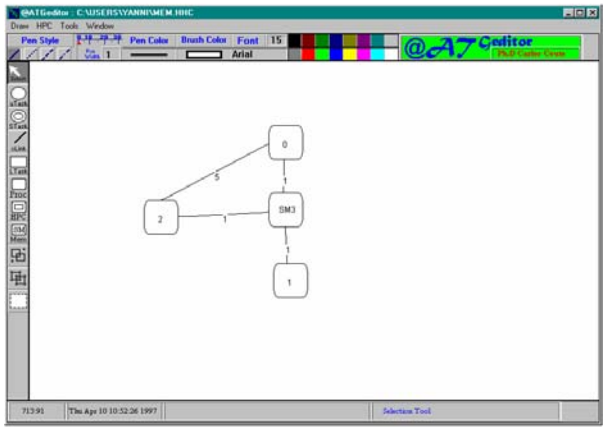

Fig.3 System graph representation

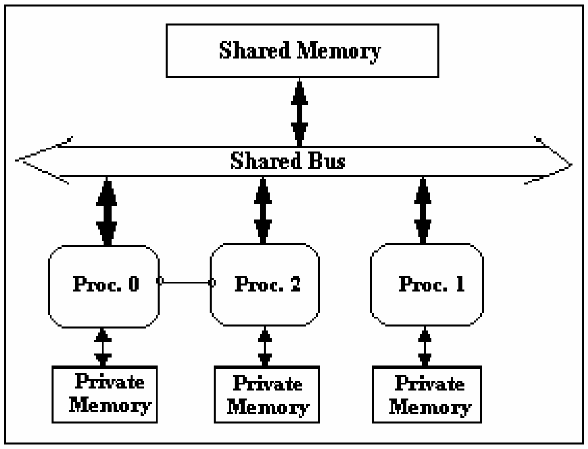

Fig. 4 Architecture represented by the system graph of fig.3

III. THE SCHEDULER

The objective of scheduling is minimization of the total execution time and its output is a Gantt chart, which contains information about the distribution of the tasks included in a given program graph onto the various target machine's processors. This problem is NP-hard, i.e., the computational requirements for an optimal solution increase exponentially with the number of tasks and number of processors, which makes the finding of the optimal schedule, even for moderately sized problems, not practicable. Therefore, nearly all practical scheduling algorithms incorporate heuristics. Since scheduling is expressed as an optimization problem, some approaches that can be taken are [3, 4]:

1) **Enumerate methods** search every point related to an objective function's domain space, one point at a time. They are very simple to implement but may require significant computation. The domain space of many applications is too large to search using these techniques.

2) **Calculus based techniques** use a set of necessary and sufficient conditions to be satisfied by the solutions of an optimization problem. It can be divided into two groups, direct and indirect methods. Direct methods is computationally feasible, but can be use only on a restricted set of "well-behaved" problems. Indirect methods look for local extrema by solving a nonlinear set of equations resulting from zeroing the gradient, and so is computationally heavy.

3) **Heuristic methods**, solve the NP-hard problems within reasonable time, but never garantee an optimal solution since it only search part of the entire solution space and suffer from the so called horizon effect[6].

4) **Guided random search techniques** including both simulated annealing and genetic algorithms are based on enumerative techniques but use additional information to guide the search.

Scheduling can be dynamic or static. The static scheduling approach can be applied to only a subset of problems whose run-time behavior is predictable, i.e., the precedence-constrained task graph must be known beforehand. Therefore, constructs such as branches and loops must be excluded to provide deterministic program behavior. The disadvantage of dynamic scheduling is its inadequacy in finding global optimums and its additional execution overhead resulting from the fact that the schedule must be determined while the program executes.
All schedulers we have been implemented are statics and make explicit consideration about interprocessor communications, because excluding IPC from the scheduling is unrealistic.

The scheduling algorithms we will present are modified versions of Lee's dynamic-level [3] and Pattipati's [4] scheduling that incorporate memory constraints and the possibility of utilizing the schedule-holes in processors and channels.

As we said above the processor types composing our target machine are DSPs and microcontroller, and so would be unrealistic if we do not incorporate memory constraints in the schedulers. Therefore some modeling technique must be used to express memory requirements and we adopt the nonlinear memory constraints model suggested by Kaneshiro [5], since the linear model is unpraticable. The memory requirements are evaluated and test against the current available memory resource every time a task is analyzed for allocation. The dynamic level of task to be schedule on a given processor is a very large negative number if the given processor cannot satisfy its memory requirement at any point of time.

Ram Murthy suggested in [6] the exploitation of schedule-holes to improve the Lee's dynamic level scheduling for homogeneous system. We made a soft change in the Lee's dynamic level calculation for heterogeneous system in order to incorporate the Ram Murthy´s suggestion and memory constraint. The new dynamic level is given by

$$\text{newDL}_1(N_i,P_j, \Sigma(t))=SL^*(N_i) - FAH(N_i,P_i, \Sigma(t)) + \Delta(N_i,P_j) \quad (1)$$

$$\text{oldDL}_1(N_i,P_j, \Sigma(t)) = SL^*(N_i) - \max[DA(N_i,P_i,\Sigma(t)), TF(N_i,P_j, \Sigma(t))] + \Delta(N_i,P_j) \quad (2)$$

$$\Delta(N_i,P_j) = E^*(N_i) - E(N_i,P_j), \quad (3)$$

where $\Sigma(t)$, $DA(N_i,P_i,\Sigma)$, $FAH(N_i,P_j,\Sigma)$, $SL^*(N_i)$, $TF(N_i,P_j,\Sigma)$, $E(N_i,P_j)$, $E^*(N_i)$, stand for: the state of the processing, memory and communication resource at instant t, the earliest time all data required by node N_i is available at Processor P_i with memory requirement satisfied, the first available schedule hole after $DA(N_i,P_j,\Sigma)$, static level of node N_i, the time the last node assigned to the jth processor finishes execution, execution time of node N_i over processor P_j, and the adjusted median execution time of node N_i over all processors, respectively. The static level of a node N_i, $SL^*(N_i)$, represents the largest sum of execution times along any directed path from N_i to an endnode of the task graph, over all endnodes of the task graph. All others Lee's scheduler parameters, such as descendent consideration and Resource scarcity are kept unchanged.

TABLE I
EXECUTION TIMES OF THE TASKS IN FIG.2

Task / Proc	0	1	2	3	4	5	6	7	8	9	10
0	5	5	2	4	5	2	4	5	4	3	4
1	∞	3	∞	2	∞	∞	3	∞	5	3	∞
2	3	6	4	4	3	3	∞	6	∞	3	6

We also modify Pattipati's algorithm in the same way in order to carried out schedule hole exploitation. Pattipati use the same argument as Lee: a node N_i cannot begin execution on processor P_j until the data from all predecessors of N_i are available at P_j, and also until processor P_j completes its execution of the last task in the ordered set TP_j. TP_j contains the task assigned to P_j sequencing in the execution order. In this algorithm, we also use $E^*(N_i)$ in the calculation of the order of task allocation instead of the ratio service demand/service rate.

[6] This property asserts that no matter how far one looks ahead before making a local decision, there may exist something "just over the horizont" which can render the decision detrimental to the objective, i.e., heuristic methods tend to take optimal local decisions that may have a non-optimal global effect.

TABLE II
SOME PROCESSOR CAPABILITIES USED BY THE SCHEDULER

Proc	Startup	Memory	Processor I / O
0	1	20	yes
1	0	20	No
2	2	20	Yes
SM	3	20	

Fig.5 presents two Gantt charts, one for the scheduling of communication channels and another one for processors that ilustrates the results of running our first scheduler incorporated with decendent consideration and resource scarcity and using as inputs the task graph of fig.1, the system graph of fig.2, and the information presents in table I and II. All data in these tables are available when a particular task graph or system file are opened in editor mode. By double-clicking a particular drawing object a dialog box is showed with the specific information that depends on the mode used. The point-to-point link in fig.3 is full duplex.

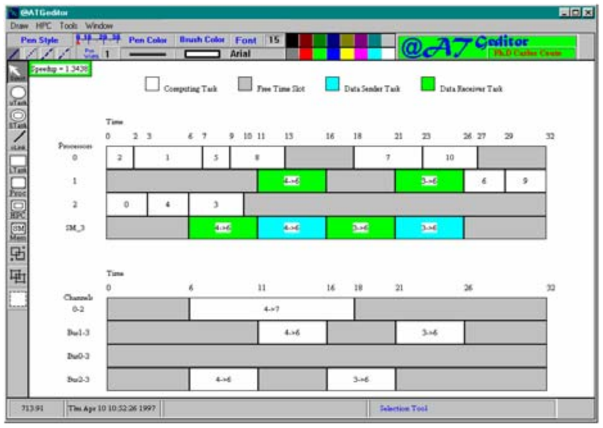

Fig.5 Gantt chart for our Lee modified Scheduler

A. Memory Constraints

As said before, an edge between two tasks allocated to different processors represents an interprocessor communication, and so, demands memory storage at the destination processor. Using the memory usage information associated with the schedule state $\Sigma(t)$ at instant t, every solutions with memory requirement that violate the memory availability can be discarded. For schedule a given task N_i on processor P_j at instant t, the memory requirement evaluation is carried out, following the next steps:

1) Deallocation Stage - the schedule state $\Sigma(t)$ is analyzed and input memory requirement is deallocated for all tasks that end execution before or at the instant t.

2) Transfer Stage - for all predecessors of N_i scheduled before on processors P_k, for all $k \neq j$, the associated output memory requirement to N_i must be transferred to P_j.

3) Output Allocation Stage - for all N_i's successors tasks memory storage must be allocated on processor P_j.

4) The schedule of N_i onto P_j at instant t is accept only if P_j has enough memory to satisfy the demands at the transfer and output allocation stages.

However, special care must be taken, since our schedulers perform schedule-hole exploitation. Depend on the instant t and $TF(N_i,P_j,\Sigma)$ one of the two cases have to be handled:

1) if $t \geq TF(N_i,P_j,\Sigma)$, then the memory usage is evaluated only for N_i.
2) otherwise, the evaluation must be repeated for all tasks schedule so far in a way of increase starting time order

To simplify the implementation of the memory requirement test, the transfer stage can be removed if we consider not only the testing of execution tasks but also the testing of receiver tasks with the deallocation stage carried out when the related sender onto the same processor is executed.

Note that not only the data communication must be take part of the memory usage evaluation, but also the memory for global and static variables, and the program memory. The memory program of each processor is update accumulatively after each task allocation and so, before the schedule of a given Task N_i on processor P_j at instant t, the schedule state $\Sigma(t)$ must be used to verify that P_j has enough memory program for the storage of N_i's code. Table III partially shows the memory usage information associated to the final schedule state for the schedule in fig.5.

B. Modeling Interprocessor Communication

Modeling the IPC requires an estimator for the time it takes to communicate a number of data units between to processors, and a model of the interconnection topology. If the communication time is not the same for all interconnections, then the estimator has to be included for each communication channel. The communication model reflects the way the processors communicate, which again depends on the interconnection topology. Different communication times and ways the processor can communicate arise from different topologies and topologies combinations. Topologies such as shared memory (single and multiple bus), point-to-point connections, mesh, and a mixed of shared memory and point-to-point connection are handled by our schedulers.

TABLE III
MEMORY USAGE INFORMATION OF SCHEDULE IN FIG.5

Task / Proc	0	1	2	3	4	5	6	7	8	9	10	
												Task 2
0	0	0	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	0	0	0	0	0	
SM	0	0	0	0	0	0	0	0	0	0	0	
												Task 0
0	0	0	0	0	0	2	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	0	0	0	0	0	
SM	0	0	0	0	0	0	0	0	0	0	0	
												Task 1
0	0	0	0	0	0	2	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	0	
2	0	0	0	3	3	0	0	0	0	0	0	
SM	0	0	0	0	0	0	0	0	0	0	0	
												Task 4
0	0	0	0	0	0	2	0	7	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	0	
2	0	0	0	3	3	0	0	0	0	0	0	
SM	0	0	0	0	0	0	0	0	0	0	0	
												Receiver 4-7 [6]
0	0	0	0	0	0	2	0	7	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	0	
2	0	0	0	3	0	0	2	2	0	0	0	
SM	0	0	0	0	0	0	0	0	0	0	0	
												Task 3
0	0	0	0	0	0	2	0	9	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	0	
2	0	0	0	3	0	0	2	2	0	0	0	
SM	0	0	0	0	0	0	0	0	0	0	0	

TABLE III (cont.)

Task / Proc	0	1	2	3	4	5	6	7	8	9	10	
												Receiver 4-6 [6]
0	0	0	0	0	0	2	0	9	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	0	
2	0	0	0	3	0	0	4	2	0	0	0	
SM	0	0	0	0	0	0	0	0	0	0	0	
												Task 5
0	0	0	0	0	0	2	0	9	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	0	
2	0	0	0	3	0	0	4	2	0	0	0	
SM	0	0	0	0	0	0	2	0	0	0	0	
												Task 8
0	0	0	0	0	0	0	0	9	1	0	0	
1	0	0	0	0	0	0	0	0	0	0	0	
2	0	0	0	3	0	0	4	2	0	0	0	
SM	0	0	0	0	0	0	2	0	0	0	0	
												Receiver 4-6 [11]
0	0	0	0	0	0	0	0	9	1	0	2	
1	0	0	0	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	2	2	0	0	0	
SM	0	0	0	0	0	0	2	0	0	0	0	
												Receiver 3-6 [16]
0	0	0	0	0	0	0	0	9	0	0	2	
1	0	0	0	0	0	0	2	0	0	0	0	
2	0	0	0	0	0	0	0	2	0	0	0	
SM	0	0	0	0	0	0	0	0	0	0	0	
												Task 7
0	0	0	0	0	0	0	0	9	0	0	2	
1	0	0	0	0	0	0	2	0	0	0	0	
2	0	0	0	0	0	0	0	0	0	0	0	
SM	0	0	0	0	0	0	0	0	0	0	0	
⋮		⋮		⋮			⋮		⋮		⋮	**Receiver 3-6 [21]**

The availability of some kind of processor I/O is very important in a way that it makes a processor able to calculate and communicate simultaneously.

In order to give a unified vision of the interconnection topology to the scheduler, the following considerations are introduced:

1) a shared memory is modeled as a "dead processor", i.e., a processor without execution capability, and processor I/O, only with storage capacity.
2) a connection between two processors through a shared memory can be represented as two point-to-point connection.
3) the number of simultaneous accesses to the shared memory depend on the number of ports and the addresses to be accessed.

Using such considerations, the scheduler has an unified vision of the interconnection topologies as based on point-to-point connection. Therefore, the communication cost between two processors P_i and P_j (including the 'dead processor') can be modeled as

$$\text{CommCost}_{i,j} = \text{SetupTime} + \text{NUnits}*\text{WCTime},$$

where NUnits and WCTime, specify the number of data units to be sent and the communication time for one data unit, respectively. In cases where one of the communicating processor is a "dead processor", the SetupTime is the overhead incurred in setting the connection from a processor to the shared memory, i.e., the overhead associated with taking the shared memory from a processor and handing it to another one, and WCTime is equal to the number of wait states plus one.

IV. WCET[7] PREDICTOR

Since we are concerned with static scheduling, we need to know the execution time of each task onto each processor beforehand, and consequently, for each task code must be generated for each processor. The simpler approach to estimate the execution time of a task graph node is: for each arithmetic instruction counting the number of times it appears on the code, express the contribution of this instruction in terms of clock cycles, and update de total clock cycles with this contribution. Nevertheless, this approach is unrealistic since it ignores, cache and pipeline effects (these are features of some DSPs that can be used in our hardware architecture) and mainly the system interferences. Other two basic approaches are:

1- Isolate the operation to be measured and make time measurements before and after performing it, which is valid only when the resolution of an individual measurement will be considerably less than the time of the operation to be analyzed.

2- Execution of the operation a large number of time and at the end of the loop operation execution the desired time will be found by averaging. Even with this approach, if you want an accurate measurement, a number of complications such as, compiler optimizations, OS distortions, must be solved.

We plan to study the timing schema proposed by Alan Shaw et al [7, 8] and extend it in order to take account of the cache and pipeline effects [9- 11]. Others approach for time measurement can be found in [12].

[7] Worst-Case execution Time

V. REFERENCES

[1] Wenheng Liu et al., "Communication Issues in Heterogeneous Embedded Systems" http://www.usc.edu/dept/ceng/prasanna/projects/embed.html

[2] Prashanth Bhat, "Issues in using Heterogeneous HPC Systems for Embedded Real Time Signal Processing Applications," http://www.usc.edu/dept/ceng/prasanna/projects/embed.html

[3] Gilbert Sih, A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," IEEE-PDS, 4, 2 February 1993.

[4] K. Pattipati et al., "On Mapping a Tracking Algorithm Onto Parallel Processors," IEEE-AES, 26, 5, September,1990.

[5] Ronald Kaneshiro et al., "Task Allocation and Scheduling Models for Multiprocessor Digital Signal Processing," IEEE - ASSP, vol.38, 12, December 1990.

[6] S. Selvakumar and C. Siva Ram Murthy, "Scheduling Precedence Constrained Task graphs with Non-Negligible Intertask communication onto Multiprocessors," IEEE-PDS 5,3, March 1994

[7] Alan Shaw, "Reasoning About Time In Higher-level Languages Software," IEEE Trans.-SE 15, 7 July 1989.

[8] Chang Yun Park and Alan Shaw, "Experiments with a Program Timing Tool Based on Source-Level Timing Schema," IEEE Computer Magazine, May 1991, pp. 48-57.

[9] Sung-Soo Lim et al., "An Accurate Worst Case Timing Analysis for RISC Processors," IEEE Trans.-SE, 21,7, July 1995.

[10] Frank Mueller et al., "Predicting Instruction Cache Behavior," in ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems, June, 1994.

[11] Kelvin Nilsen and Bernt Rygg, "Worst-Case Execution Time Analysis on Modern Processors," ACM SIGPLAN Notices, Vol. 30, No.11 November 1995.

[12] Russel Clapp et al., "Toward Real-Time Performance Benchmarks for ADA", Communication of ACM, vol. 29, No. 8, August 1986.

[13] Jing-Jang Hwang, et al., "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times," Siam J. Computing, vol.18, No. 2, pp. 244-257, April 1989.

[14] Hyunok Oh and Soonhoi Ha., "A Static Scheduling Heuristic for Heterogeneous Processor," http://www.ce2.snu.ac.kr.