The Potential of Resource Budgets in Complex Real-Time System Design

Mattias Weckstén and Jonas Vasell
Department of Computing and Communication
Halmstad University
[Mattias.Wecksten, Jonas.Vasell]@ide.hh.se

April 2 , 2002

Abstract

To make the complex system design task clearer this paper suggests a tool that will generate, evaluate, and annotate budget proposals. The generation of budgets is based on a task graph representation of the system together with an architectural description. Some optimization criteria (e.g. timing, energy, and cost) will be evaluated for each budget and compared with earlier results in order to get the "best". The best proposals will then be annotated for better understanding of how the separate parts of the system depends on each other. To do exhaustive optimization of this search space is not possible for larger problems why a number of heuristics are introduced.

1 Design of Complex Real-Time Systems

When designing complex real-time systems it is desirable to find a practical method to evaluate the proposed solution as early as possible in the design cycle, improving the possibility of meeting the non-functional demands. To make the design task iterations easier it is also desirable to make relations between different parts of the system clearer. A typical design process could look like Figure 1, where the work starts with functionality analysis and requirement specification and ends with an implementation.

1.1 Approach

The suggested approach to accomplish this is to identify functional units in the system and then assign a resource budget for each unit, in such way that the possibility of implementing the function to fulfill the non-functional demands is maximized. To make the system more stable to corrections and alterations, the dependencies between budgets should be annotated. The term *resource* include, among other things

- Execution time
- Processor element
- Memory

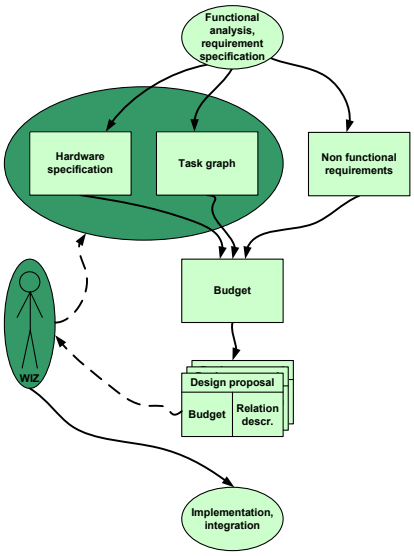

Figure 1: A typical design process where functional and non functional demands are used to generate a budget proposal. The proposal could then be feed back for more iterations or implemented.

- Energy

Note that resource budget assignments should be based on relative estimates of the actual resource need since we are at early design stages.

1.2 Work Focus

In this paper we will assume that input data is present in the form of an acyclic task graph describing the precedence constraints, where each task has an execution time estimate and an estimate of the accuracy. The architecture will be modelled as homogenous resources connected by a single bus, which gives us the maximum number of resources and the bus speed as parameters. When resource budgets are assigned we will only address the assignment of allowed execution time and static processor to reside on. To do this we will utilize each tasks estimated execution time in conjunction with the end-to-end requirements for the system, in our case deadlines and release-times. The result we look for is the task graph with the minimum tightness, where "tightness" is equal to the path through the task graph with the most work to execute in comparison to the length of the path. This will all sum up to a number of practical methods oriented towards giving a system designer support in finding and evaluating design alternatives. These methods does not necessary strive for optimality (in the case that it would be possible to define), but rather a set of good estimates that will point us in the right direction.

1.3 Related Work

One solution for part of this problem was presented in [Zha99], where the author assumes we have an allocated system in the form of a task graph with end-to-end timing constraints and the only thing left is to try different execution orders for the tasks on the different resources in the system. The measure of how good a specific "order" is inversely proportional to the tightness. Based on this measure we can allocate more, or if needed - less, time for each task in the system so the idle time is kept at a minimum, but still budgeted in a "fair" fashion. A similar method that scales well with problem size is presented in [AH98] that has the drawback that it sometime "misses" feasible assignments.

The author of [Axe97] discusses resource budgets and their importance and calls attention to that one great benefit of the resource budgets is that they localize the global constraints. He also point out that [Mey88] introduced resource budgets under the notation "programming by contract".

In [Eke01] the author describes how constraint programming can be applied to the problem of scheduling and allocation and shows how this can be used for modelling as well as for implementation.

In [GiKHS95] the authors present a method to break down end-to-end requirements to task specific demands and, if this is not possible, to identify bottlenecks in the system. This information is then used as input in a design tool where the system could be restructured for another iteration step.

In [NS94] we find a similar method called slicing that does not address budgets but focus on how to increase online schedulability in a distributed system with off-line calculations. A discussion about the same method but under more relaxed forms (i.e. at early design stages) is found in [JS97].

2 Solution Overview

As mentioned earlier, the assumption is that in-data is available as an acyclic task graph with estimations of the execution time in each task and also end-to-end timing constraints for each possible trace. Besides the actually estimate of the workload we need to describe the actual architecture, which for now is modelled as homogenous resources connected with one single bus. The parameters for the architecture are thus maximum number of resources and bus speed. With this raw data the budgeting tool is supposed to try out all possible (or promising) solutions, or in other words generate budgets for all possible allocations and execution orders in the graph. The budgets with the lowest tightness are kept in a repository and finally returned to the user. The actual process chain consists of the following stages:

Ordering Decides in which order the tasks in the task graph should be executed if they were allocated on one single resource.

Allocation Decides on which resource a task should reside. After the actual allocation communication overhead is added.

Budgeting Distributes the "slack" in the task graph among all nodes to make implementation easier.

2.1 Budgeting

The budgeting aims to distribute as much of the idle time in the processors as possible in some "fair" fashion. The proposed scheme starts with an ordered and allocated graph and distributes the slack in proportion to the estimated execution times in the tasks. This will always render a valid solution if it is done with the path with least slack first and then in ascending order. To do this we need to find the "tightest" path through the graph. To do so, we find all possible paths through the graph, where a path is a number of nodes that can be accessed in the given order (see Section 2.3), following the edges in the graph. For each possible path we assign a "tightness" that is the total work on the path, or the sum of all estimates on the path, divided with the length of the actual path, or the deadline for the last node minus the release time for the first (see Equations 1 – 4).

$$P = [t_1, t_2, \ldots, t_N] \tag{1}$$

$$W_p = \sum_{i=1}^{N} e_i \tag{2}$$

$$L_p = D_p - R_p \tag{3}$$

$$T_p = \frac{W_p}{L_p} \tag{4}$$

The tightest path is the path with the largest T_p-value. When found, time is distributed over this tightest path proportional to the execution time estimate in each node. To make the new W_p fill all slack between R_p and D_p, each estimate is multiplied with the inverse of the tightness (see Equation 5).

$$\sum_{i=1}^{N} \frac{e_i}{T_p} = \frac{\sum_{i=1}^{N} e_i}{W_p} \cdot L_p = L_p \tag{5}$$

To make a distinction between the original estimate and the new proposal, the new value is named "Allowed Execution Time" or for short AET. The nodes on the tightest path is now removed and replaced with deadlines or release times where so is needed. The procedure is now repeated until no more nodes are left in the graph.

Unfortunately this method will render a lot of possible paths to search through for each budget and it seems to not be especially scalable for many nodes. What is even worse is that this algorithm is the kernel in the system and will be executed millions and millions of times for each run. We have to come up with something better than the naive "search-all" method.

Budgeting Based on Uncertainty

One interesting variation of the standard budgeting algorithm, where execution time estimates is base for the budget allocation, is to assign a relative uncertainty value to each of the tasks to describe how sure you are of your execution time estimate. The new budgeting algorithm would divide the available slack in proportion to the execution estimate and the uncertainty. In other words, the task which we know the least about will get the largest extra allocation.

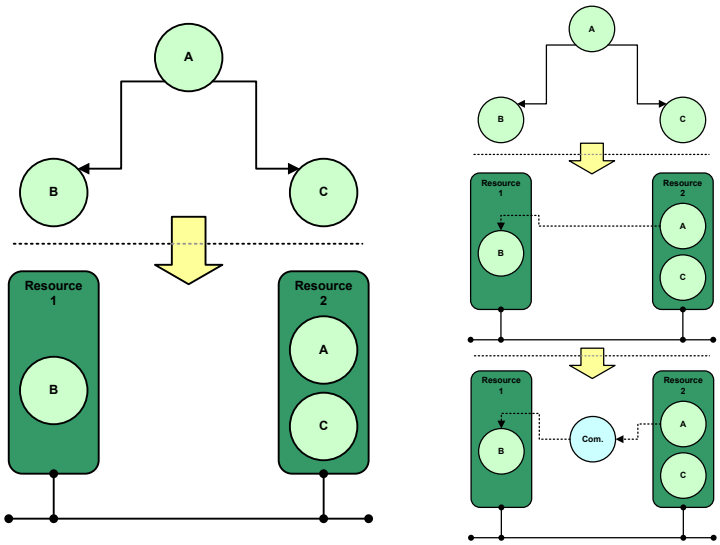

Figure 2: Allocation of three ordered tasks on two resources. After allocation we add communication penalties for tasks on different resources and have to use the bus.

2.2 Resource Allocation

The aim of the allocator is to test all possible allocations for a given order of the tasks in our system. The only limit we have for this is a maximum number of processors allowed in the system. Since it is a rather big (and unnecessary) task to test *all* possible allocations it is useful to introduce some expert knowledge. One simple rule we could apply would be that if there is a precedence constraint between two nodes (e.g. due to data flow) these two nodes should be placed in same resource. The reason for this is that it is impossible for these tasks to run in parallel, and to distribute them will only result in increased communication (i.e. not considering load balancing). If we consider a simple graph of three nodes where one node is the parent for the other two, this could be allocated on two resources in a 1-2 or a 2-1 configuration. If we name the parent A and the children B and C, we will see that the interesting allocations will be $\{\{A, B\}, \{C\}\}$ or $\{\{A, C\}, \{B\}\}$ (see Figure 2). The case with three processors is not interesting at this stage since this only will generate overhead communication as well as for the allocation $\{\{A\}, \{B, C\}\}$.

Communication

After the system is allocated we need to add communication penalties for tasks communicating between resources. For each data flow between two tasks, the number of bytes is specified. This translates into a communication delay and is finally represented as task locked on the bus resource (see Figure 2). To determine a schedule (or an order) for the bus traffic, the communication tasks inherits the order of their "parent". Since all tasks in the system first of all is ordered in a global order this will result in an unambiguous bus schedule (see Section 2.3).

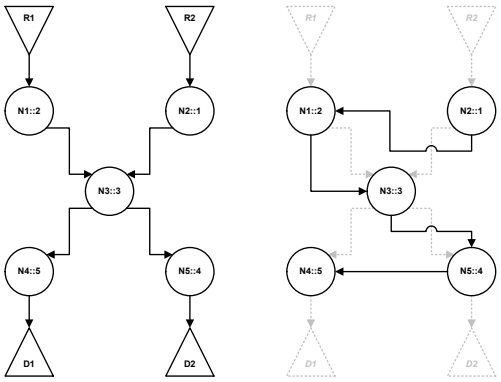

Figure 3: Based on the precedence constraints (e.g. data dependencies) an execution "path" is decided through the graph. This path shows the execution order of the tasks if they were to be executed on a single processor.

2.3 Ordering

To decide in which order to execute two parallel tasks if they end up in (or sharing) same resource due to allocation (see Section 3) we have to order all tasks. Our suggestion is to order all tasks (see Figure 3) in the same way as they would have been ordered if merely one processing element had been used (and therefore no communication is added at this stage). The example in the Figure 3 shows how the graph is put in the order N_2, N_1, N_3, N_5, N_4. To find all possible orders for a large system is also a huge task so we need some narrowing criteria. One suggested criteria is that the tasks with "latest possible finishing time" earlier than other tasks "earliest possible start time" should be put in the mentioned order. The reason for this is that if we try to do it in the other way and these two tasks end up on the same resource we will not be able to find a solution (see Section 3.1).

3 Handling Complexity

Elementary analysis shows that the suggested algorithm scales very badly why we need to isolate the reasons for this and try to find ways to reduce complexity.

Ordering

The complexity in the system originates from different sources. When we address the ordering of the task graph the number of nodes is not the main issue; but how they are connected. The more independent tasks, the higher complexity (see Figure 4). Complexity will probably increase with the number of tasks anyway, since the general case is loosely coupled. This is unfortunately the case for many regulator systems. The "fix" for this was suggested to make graphs more connected by inserting edges (or pre ordering so to say) where time limits show that other solutions would be pointless.

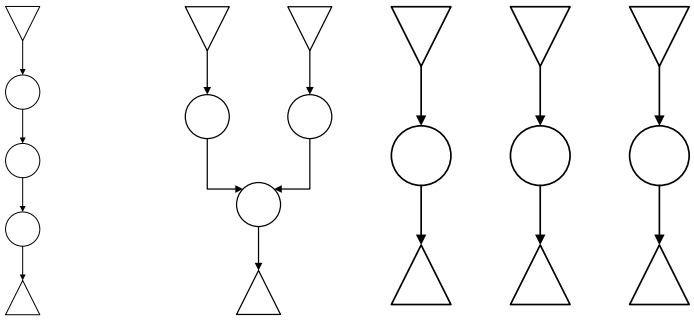

Figure 4: All three graphs have three tasks, but the first one has just one order, the second has two orders, and finally the third has six possible orders.

Allocation

When it comes to the complexity of the allocation algorithm the main factor here is the maximum number of resources allowed. For the moment, only homogenous systems are allowed, but heterogeneous systems will be supported, and this will increase the complexity of the search.

Budgeting

For the moment, budgeting is done by finding all possible paths through the system. The complexity for this search is dependent on the number of possible paths through the graph. The number of paths through the ordered allocated graph is based on the number of edges, which will increase in the step between allocation and budgeting. This is therefore important to keep the number of edges at a minimum.

3.1 Reducing Complexity

During the construction of the development tool it showed clearly that the runtime would scale very badly with the problem size (the problem is just a special case of multiprocessor scheduling). Due to this fact a number of expert methods were developed to shrink the search space and get reasonable computation loads for searches.

Virtual Edges

To describe a precedence constraint, between two tasks, not present because of a data dependence, we insert a virtual edge. In other words, virtual edges is to be inserted in the graph for tasks that clearly have a natural order due to their timing constraints (e.g. $LFT_a < EST_b \Rightarrow N_a \curvearrowright N_b$). This means that two nodes, where one node has to finish before the second one is allowed to start, are pre-ordered in that way. The pre-ordering will result in less possible total orders, without imposing any restrictions on the feasible part of the search space.

In the example setup "SimpleBaseStation" a run without pre-ordering results in 840 generated orders that are budgeted in approximately 230 seconds. If pre-ordering is turned on 504 orders were generated and budgeted in less than 160 seconds.

Clustering

One way to reduce calculation complexity is to make the graph data less complex. This could be done by clustering nodes that probably would end up in the same resource anyway. A typical case for this is a straight sequence of tasks where we do not gain anything by executing them on separate resources. As an example we clustered the "BaseStation" example and got the "SimpleBaseStation" example. In the clustering the number of tasks reduces from 17 to 8 nodes and this in turn reduces the number of orders to search through from 12 612 600 down to 540. The drawbacks of this process is that we do not get as good schedules as possible with the original graph; the optimal tightness in the "BaseStation" example is 0.4514 meanwhile the "SimpleBaseStation" has 0.5013, a deviation of approximately 10 percent. Despite this, clustering is especially suitable since it follows the typical project development cycle, where you start with large functions that are derived into finer grained modules during the iterations.

Expert Knowledge of Resource Allocation

Some of the tasks in the system have a natural place to reside, for example inputs or outputs have physical layout demands, and some tasks may have demand for large memory space or specific demands on instruction setup. To be able to make these restrictions the following operators are added:

SET Describes the set of allowed processors for this node to reside on.

GROUP Describes the set of tasks that we are allowed to share resource with.

UNGROUP Describe the set of tasks that we are not allowed sharing resources with.

Early Stopping

An interesting notation made during the tests is that we will find a solution as good as the best one among the first 10-20 thousand budgets for examples of size close to "BaseStation" (i.e. around 17 nodes, medium connected). If we assume that the tightness suddenly will make a "jump" close to the optimum (see Figure 5) we can introduce an early stopping criterion to get quicker results or indications. Examples of stopping criteria could be a predefined level of the tightness that is sufficient for our designers or a maximum numbers of iterations until we need the tightness estimate. Since iterations are quite correlated to time the second criteria could be expressed in a number of seconds we are willing to wait for the answer.

4 Preliminary Conclusion

The article presents an implemented method that fulfills the goal to evaluate proposed solutions in a number of aspects. Complexity is still an important problem to tackle; optimality is impossible to assign but how close will we be? Indications show that despite the complexity of finding optimal solutions it is still possible to achieve a practically and useful design support using the various heuristics.

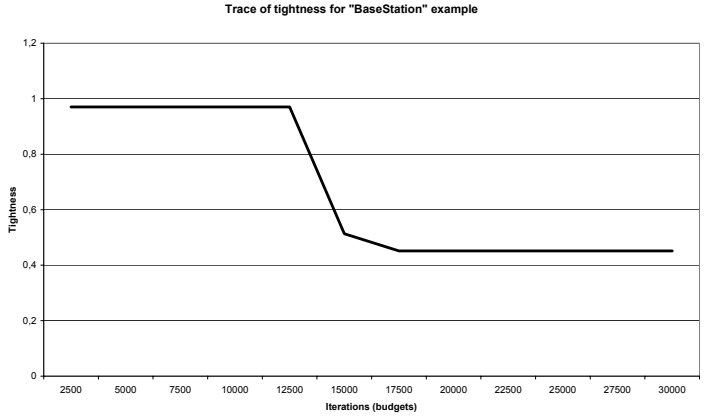

Figure 5: It would be suitable to choose an early stopping level by indicating a criterion for the tightness itself or the run time in seconds or iterations.

5 Future Work

For future evaluation we see more heuristics coming up to reduce the work load of searching. Related to this is to define new optimization criteria such as utility and cost. One very important part of the problem, barely touched, is the analysis of budget dependencies which is of importance for the design iterations (i.e. when we actually want to alter things). Since only basic homogenous architectures are supported we need to add support for heterogeneous architectures. When it comes to the restricted task graph representations we need to allow periodic tasks in the task graph, allow periods smaller than the deadline and find a way to do general task graph transformations.

References

[AH98] Peter Altenbernd and Hans Hansson. The slack method: A new method for static allocation of hard real-time tasks. *Real-Time Systems*, 15(2):103–130, 1998.

[Axe97] Jacob Axelsson. *Analysis and synthesis of hetrogenous real-time systems*. PhD thesis, Linköping University, 1997.

[Eke01] Cecilia Ekelin. Scheduling of embedded real-time systems: A constraint programming approach. Licentiate thesis, Computer Science and Engineering, december 2001.

[GiKHS95] Richard Gerber, Dong in Kang, Seongsoo Hong, and Manas Saksena. End-to-end design of real-time systems. Technical Report CS-TR-3476, 1995.

[JS97] Jan Jonsson* and Kang G. Shin**. Deadline assignment in distributed hard real-time systems with relaxed locality constraints. Technical report, 1997.

[Mey88] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[NS94] Marco Di Natale and John A. Stankovic. Dynamic end-to-end guarantees in distributed real time systems. In *IEEE Real-Time Systems Symposium*, pages 216–227. IEEE, 1994.

[Zha99] Yufeng Zhao. Derivation of local timing constraints in early design stages. Master's thesis, Chalmers University of Technology, April 1999.