To appear in a special issue of the IEEE Journal of Real-Time Systems

Flexible Design of Complex High-Integrity Systems Using Trade Offs

Biswa Sengupta & Iain Bate
Real-time Systems Group
Department of Computer Science
University of York, York, YO10 5DD, UK.

sengupta@ieee.org, iain.bate@cs.york.ac.uk

Abstract

Large-scale complex embedded systems pose unique problems. To reduce overall development times, there is a need to develop the system in a concurrent fashion, involving the development and verification of software at the same time as designing, building and verifying the hardware. This requires a two-phase trade-off analysis approach to the hardware software co-design problem. The first phase is platform independent: it allows system requirements to be met and also supports other important objectives, e.g. scalability, upgradeability. The results of the first phase include deriving requirements and design constraints placed on the platform dependent phase (eg. resource budgets including time). The second, platform dependent phase, chooses the actual software and hardware implementation that satisfies the requirements derived in phase 1. This paper addresses the first part of the problem through trade-off analysis. This establishes the design decisions in a traceable manner whilst capturing the rationale and assumptions made. It then searches the design space for the solution that best meets the system's objectives. The approach has been developed for the needs of critical systems and has already been applied to the logical design of systems.

1 Introduction

Large-scale embedded systems, such as those found in aerospace applications, are characterised by their functional complexity, size (in terms of required software / hardware), relatively long lifecycles and requirement for validation and verification of their *fitness for purpose* prior to deployment [1]. Often in large systems, the unit cost of hardware is not an overriding concern – the relatively few units made mean one-off development costs are the prime cost consideration.

Conventional development processes for systems contain an early hard partitioning of system functionality between hardware and software. It is performed with minimal use of trade-off techniques, but instead relying on high-level systems engineering principles [2]. Essentially, a "best guess" is made when functions are partitioned between hardware and software. Invariably an underestimate of the amount of software is made (hence the computing platform is under resourced).

One consequence of the development process is that hardware is developed in isolation from software, usually prior to software development (due to the long lead times for custom hardware or as yet unavailable hardware) and only later are they integrated. A critical problem occurs when / if additional functions are identified after system partitioning, these are usually pushed into software as the hardware is fixed. The hardware is considered fixed as it is expensive to redevelop the hardware to cope with either additional functions or to provide increased computing resource for the software components.

Codesign [6] recognises that systems implement required functions using a mixture of hardware and software components. Trade-offs can be explored between the choice of whether system functionality is implemented in hardware or software. Given a partitioning of functionality into hardware and software components, design / synthesis of hardware and software can proceed in parallel. Subsequently, the separate hardware and software are integrated to form the final system. A key element of the codesign process is that alternatives for the hardware / software partitioning are evaluated.

For this approach to be successful, it is important that requirements are established for properties and operations across the system's boundaries (e.g. between hardware and software). These requirements are referred to as *interface requirements*. The interface requirements allow change to be managed such that two distinctly different parts of the design can be developed in separation. To reduce the impact of changes (i.e. allow the designs to proceed in relative isolation), the content of the interface requirements should be chosen so that not only are the initial design objectives met but there is flexibility within the design to support change. The use of interface requirements and reducing the impact of change also allows some level of analysis, albeit with inaccuracies, when only scant or approximate design data is available earlier on in a project.

It should be noted that *managed change* is considered to be a secondary quality attribute of the system, ie. it is not considered essential to the system's operation. Primary quality attributes are those essential to the system's operation. An example of a primary quality attribute is the meeting of timing requirements in hard real-time systems.

In the timing domain an example of an interface requirement is a set of timing budgets (e.g. Worst-Case Execution Times (WCET)) and attributes (e.g. offsets and priorities) for the tasks that if met lead to the system's timing requirements being met. To support managed change the timing budgets and timing attributes should be chosen so that the scalability and flexibility of the system is improved. An example of scalability is the ability to add additional tasks into a

system without preventing the existing tasks from meeting their timing requirements.

A key issue when defining any part of the system, eg. timing budgets, is the trade-offs between different objectives of the system. For example, having larger budgets may mean the software can be developed cheaper because it doesn't have to be optimised as much. However larger budgets would make the design of the hardware more difficult in that it needs greater optimisation for the particular application or leads to the use of more powerful (and hence expensive – expensive can be in terms of cost, power etc) hardware components. The tensions between different objectives need to be traded off during the design process.

Once the interface requirements have been established, both hardware and software should be designed so that these interface requirements are upheld on both sides of the interface. This second phase of the trade-off analysis problem is not addressed in this paper. The budgets derived should be appropriately proportioned where possible. That is, there should be a significant penalty in giving a smaller budget to a larger/more complex piece of functionality unless other factors mean it is better to spend more time and money on a particular part of the system. Other work has shown how such budgets can be used to support the development of large systems where portability between hardware platforms is a key success criterion [8].

This paper contends that important benefits arise by embedding a codesign process as a sub-process within the conventional system development process. This enables early partitioning of functions whilst still allowing functions identified later to be subject to a codesign process.

To support this approach, three main requirements must be met:

1. The structured capture of design choices, definition of system objectives and design information for later use in the codesign process or as part of design certification.
2. As part of the early partitioning, resource is reserved for future functions.
3. The specification of timing properties such as offset, WCET and Best-Case Execution Time (BCET) budgets etc that allow the systems timing requirements to be met. These properties are shown to be met at integration time.

The contributions of this paper are:

1. The application of the trade-off analysis to the problem of timing in the development of systems in order to capture the design choices and the assessment criteria designs are judged against;
2. The use and derivation of interface requirements between the software and underlying platform to help manage complexity;
3. The use of scenario-based analysis to assess secondary quality attributes, such as scalability, in conjunction with traditional timing analysis to assess primary quality attributes, such as whether timing requirements are met.

The trade-off analysis method is summarised in section 2. Item (1) from the list is presented in section 3. Section 4 discusses the differing requirements and relationship with other methods. Section 5 of this paper presents our framework for determining the optimum design solutions to satisfy items (2) and (3) from the list. The approach is evaluated in section 6.

2 Overview of Design Trade-Off Analysis Method

In [5] our method for architectural trade-off analysis for use within a systems engineering process was presented. It should be noted that [5] applies the same method in this paper but to the problem of the logical design systems. The trade-off analysis together with the inserted codesign process has the following properties:

- *Derivation of choices* – identifies where different design solutions are available for satisfying a goal.
- *Manage sensitivities* – identifies dependencies between components and design decisions.
- *Evaluation of options* – allows evaluation of alternative solutions against required properties / specification.
- *Influence on the design* – identifies constraints on how components should be designed to support the meeting of the system's overall objectives.
- *Collection of design rationale* – forms a repository for design decisions to aid traceability throughout the design.

The proposed approach could be used within the nine-step process of the Architecture Trade-Off Analysis Method (ATAM) [3]. The key difference between our strategy and other existing approaches, e.g. ATAM, is the way in which quality attributes are derived. (Quality attributes are assessment criteria used to evaluate solutions, e.g. does the design support predictability?) Our proposed approach was chosen due to the following reasons.

- The techniques used in our approach are already accepted and widely used.
- The techniques offer strong traceability and the ability to capture design rationale.
- Information generated from their original intended use can be reused, rather than repeating the effort.
- The method is equally intended as a design technique to assist in the evaluation of the architectural design and implementation strategy as it is for evaluating a design at particular fixed stages of the process.

Figure 1 provides a diagrammatic overview of the proposed method. Stage (1) of the trade-off analysis method is producing a model of the system to be assessed. This model should be decomposed to a uniform level of abstraction. Currently our work uses class diagrams from UML for this purpose, however it could be applied to any modelling approach that clearly identifies components and the interfaces between the components.

In stage (2), the key objectives and properties of the system are decomposed into detailed design requirements that need to be satisfied. Rationale for

these detailed requirements is encapsulated by structured *arguments*, along with the appropriate context, identifying where design choices are available. The arguments are structured using *Goal Structuring Notation (GSN)* [4] (refer to section 9 for further details of GSN).

Key properties of interest include: lifecycle cost, dependability, and maintainability. Clearly these properties can be broken down further, e.g. lifecycle cost into development, future upgrades and maintenance. Objectives of interest include; managed change, ease of integration and ease of verification.

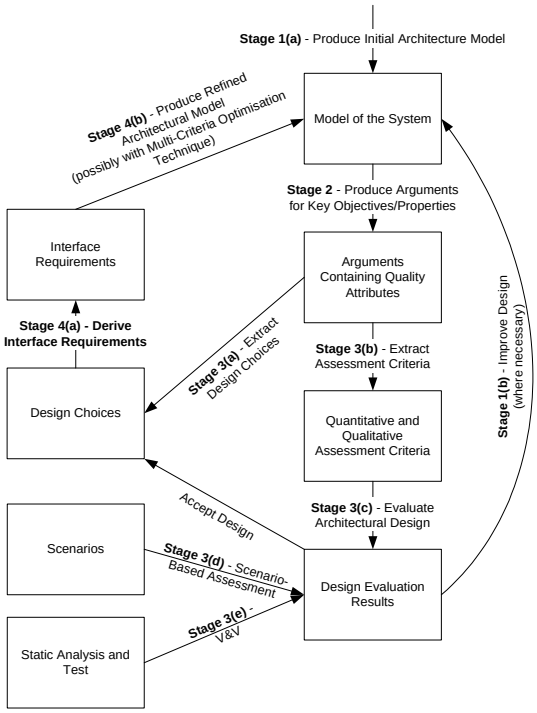

Figure 1 - Overview of the Method

Stage (3) uses the structured argument to further derive design and verification options, and to determine assessment criteria to judge how well a particular design solution meets the system objectives. Other approaches for deriving assessment criteria from systems objectives include Goal Question Metrics (GQM) [12]. Initially in the early stages of design, the evaluation may have to be qualitative in nature but as the design is refined then quantitative assessment may be used where appropriate. Part of this activity may use representative scenarios to evaluate the solutions. In the case of timing, representative scenarios will include situations where the software/system is changed which leads to modified task execution times and added/removed tasks.

Before stage (4) of the process, based on the findings of stage (3) the design is modified to fix any problems that are identified – this may require stages (1)-(3) to be repeated to show how the revised design is appropriate. When deciding on design solutions, the results from more than one assessment criteria have to be traded-off because a design modification that suits one assessment criterion may not suit another. For example, introducing an extra processor may reduce the load across the processors in the system making task schedulability easier. However it may increase the load on the communications bus making message schedulability more difficult and increasing power consumption.

When the design modification process is complete and all necessary design choices have been made, stage 4(a) of the process extracts interface requirements from the arguments. Then, as part of stage 4(b) of the process, the process returns to stage (1) where the system is decomposed to the next level of abstraction using guidance from the arguments. Components reused from another context could be incorporated as part of the decomposition. Only proceeding when design choices are complete (and any identified problems are fixed) is preferred to allowing trade-offs across components at different stages of decomposition because the abstractions and assumptions are consistent.

In this paper the refinement of the design (stage (4) of the process) is performed automatically using multi-criteria optimisation. Automatic optimisation is possible in this case because the assessment criteria can be analysed for in a quantitative fashion by tools based on static analysis and scenario-based assessment. Other than reducing the workload of engineers, another advantage of automatic optimisation techniques is their ability to trade-off the needs of different assessment criteria and balance any tensions between different system properties. There has been some previous research into the topology of optimisation algorithms including the use of optimisation for the software allocation problem, for further details refer to [11].

3 Application of Trade-Off Analysis

A key part of architecture trade-off analysis is deriving the top-level properties and objectives (i.e. goal) for the systems such that arguments can be produced that systematically break them down to lower-level goals. These goals are then used to form assessment criteria that can be used to judge whether a proposed solution is appropriate. During the production of these arguments, choices of how they can be supported (e.g. implement in hardware or software) will emerge and assumptions identified. (The assumptions are important when trying to reuse designs since they allow the basis for the existing components' design to be evaluated in the new context.) The following subsection proposes some properties for use in derivation of the interface requirements that later in the section are developed into arguments that can be used.

3.1 Key Properties

The following objectives are considered as being important. It should be noted that most objectives are derived from the overarching objective of maximising profit in some way.

- *Correctness* – using appropriate verification techniques sufficient evidence needs to be gathered that what is being produced meets its requirements. Sufficient is dependent on the nature of the application, for example it would be expected in critical systems development that more evidence is needed than for non-critical systems. In general,

hardware development is considered to have verification techniques that can provide stronger evidence for correctness.

- *Managed change* – the system produced should be changeable or upgradeable in an efficient manner. For most applications, typical change patterns or potential upgrades can be predicted with reasonable confidence. For some applications, there are known killer changes that are likely to occur and result in significant re-design effort being needed. In general, software is considered easier to change but as Ariane 501 demonstrated the assumptions that exist within the design and implementation of a component being reused are not always handled appropriately.
- *Efficiency* – the system produced should make the best use of the available resources. The efficiency of a technology is strongly dependent on the nature of the technology. For instance, a FPGA is an effective means (in terms of the amount of silicon used is small) of implementing logic such as found in Statecharts but may not be as effective at implementing floating point operations.
- *Sufficiency* – the technology used in the implementation must be able to represent the design. There are many factors here. Considering just timing,
  - Von Neumann architectures are often considered to have the benefit of providing raw processing power, however for applications where hard real-time guarantees are needed the difficulty in modelling modern Von Neumann processors can lead to large amounts of pessimism in the analysis that reduces/eliminates the benefits [14].
  - FPGAs are better at handling concurrency [7].
  - FPGAs have little or no difference between their best, average and worst-case performance whereas Von Neumann do [7]. Variability in timing behaviour makes many applications, e.g. control systems, harder to produce [7].

The rest of this paper considers sufficiency and its relationship to architectural design in greater detail.

3.2 Arguments for Sufficiency

Figure 3 presents the top-level argument for the property *sufficiency*. This shows how the goal, **G0001**, that the implementation strategy is sufficient is broken down into three sub-goals that concern the functional (**G0002**), non-functional (**G0003**) and operational environment requirements (**G0004**). To satisfy any of the goals in Figure 3, the assumption **A0001** has to be satisfied that the requirements are specified in an appropriate manner, i.e. the requirements are complete and consistent – note correctness is not important for the implementation strategy only for the final system being fit for purpose. Goal **G0004** that deals with operational environments is left undeveloped.

Goal **G0002** that deals with functional requirements is satisfied using a generic goal **G2001** – *Implementation Sufficient for the Requirements to be Met*. The argument (**SufficiencyArg**) for the generic goal is presented in Figure 4. In the case of functional requirements it is assumed, **A0002**, either hardware or software can equally well be used for meeting the requirements. Goal **G0003** is decomposed into an argument that deals with whether the implementation strategy can satisfy the system's non-functional requirements. The goal **G0003** is split into five parts (one for each sub-property of non-functional - memory, safety, timing, reliability and communications) which are also satisfied by the generic goal **G2001**.

In Figure 4, the argument for goal **G2001**, originating in Figure 3, is split into three parts; the implementation allows the requirements to be met (goal **G2002**), requirements are specified appropriately (goal **G2003** which is left un-developed – indicated by a diamond below the goal), and sufficient evidence can be gathered the requirements are met (goal **G2004**).

The goal **G2004** results in two choices (a choice means that at least one of *N* strategies proposed is followed) which are, evidence is either gathered by static or dynamic analysis. A feature of the technique developed is that where a choice exists then justifications and assumptions should be captured and these normally relate to pros and cons of the choices that need to be explored. In this case, the assumptions and justifications indicate that static analysis has the advantage since it can show absence of faults but with the disadvantages that getting appropriate models is difficult, hard to validate and often not practicable,. In contrast, dynamic analysis is more generally applicable but can't guarantee the absence of failures. Often, the best compromise is a combination of the options.

The goal **G2002** is split into three parts; that the infrastructure is predictable (goal **G2005**), that the mapping of the application onto the infrastructure is predictable (goal **G2006**), and the application can be shown to meet its requirements (goal **G2007**). From the goals, the key points that emerge are the choices in the hardware used (between FPGA, microprocessor and discrete circuitry) and the design notation used (between hardware-based and software-based languages). The latter of these choices shows that software-based languages have the advantage that they can represent both hardware and software but a disadvantage is raised that hardware-based languages often support interaction with devices and concurrency better.

Each of the hardware choices can be satisfied by the same argument, given in Figure 5. This argument shows that the choice should be based on whether the type of hardware has appropriate models available and whether these can be validated.

3.3 Using the Arguments

From the argument in section 3.2, a number of design choices and objectives (i.e. goals or quality attributes) have been identified. This section is to consider how these objectives should be used during the identification of interface requirements; issues include whether they should be used as part of a qualitative or quantitative assessment, whether the process is manual or automated and how they are used as part of making an overall decision. In general, qualitative assessments consist of

asking questions which are based on experience and some consideration whereas quantitative assessment consists of a checklist of activities to be completed. A key difference is qualitative assessment can be performed anytime during a project whereas quantitative assessment often relies on certain design information (e.g. task execution times in the case of timing analysis) being available which may dictate when it can be performed.

Table 7 provides a partial trade-off analysis of the objectives given in the arguments presented in section 3.2. The analysis has lead to qualitative and quantitative assessments being derived as well as an indication of the relative importance of each. The rest of the section explains how this can be used as part of deriving the necessary interface requirements. The qualitative assessment criteria can be used elsewhere as part of review checklists, and the quantitative assessment criteria are verification and validation requirements (e.g. the need to perform timing analysis).

Table 7 shows many of the quality attributes that have to be made when performing trade-offs. When exploring the trade-offs it would be necessary to produce a balanced design where the important constraints are met and others are achieved as well as possible. Table 7 provides examples of how using the method can help capture rationale and assumptions. For instance, any hardware should support a predictable mapping of the software onto it – justifications **J2003** and **J2004**.

Putting this into context, consider the design of a system whose principal function is to support a number of control loops. Key requirements for such a system are the ability to meet requirements and the ability to detect when the system is not performing as expected. Taking the first of these and considering timing, the main requirements are that precedence constraints are maintained, data is sufficiently fresh when used in calculations and later output, and jitter controlled because of its impact on stability.

One trade-off decision is whether a conventional microprocessor is used or a FPGA. For simplicity when making this decision, other variables could be considered as fixed, for example Ada as the programming language. Consider the assessment related to goal **G3001**, for a FPGA the models of the circuitry (i.e. individual cells and overall circuitry) are well-defined and comprehensive models available. However for anything other than a simple microprocessor models are rarely available, and even if they were available validation would be difficult if not impossible [14]. Additionally from a timing perspective related to **G2014**, it would be hard to meet the control systems timing requirements because modern microprocessors tend to have high variability in their executions; whereas FPGAs tend to have constant execution times, which makes meeting jitter requirements easier [7].

Despite the problems of using microprocessors, obviously as common practice suggests they do provide a means to implement control systems especially where much of the processing demands does not need strict timing behaviour (e.g. health monitoring functionality). One key advantage of the trade-off analysis approach proposed is that the output helps guide the design of the system, for example the design of the scheduler should minimise the jitter of certain tasks related to the operation of the control loops.

Another key benefit of the arguments used in the trade-off analysis is the way they support the objective of the systems engineering process being partitioned into clear and distinct parts that can then be performed as independent entities. For instance in Figure 4, there are goals for whether the implementation meets its requirements. From this, sub-goals are derived that separate the meeting of the requirements for the application, infrastructure and mapping from one another. These activities can then be performed in isolation of one another. However the goals capture key assumptions that must be considered and later shown to hold such that when the activities merge back together that the overall requirements are met. As the design is further decomposed, more assumptions between these activities would be derived.

4 Relationship with Existing Methods

There are a wide variety of existing methods for deriving interface requirements between hardware and software and then exploring the search space. These methods have differing capabilities. For an overview of these refer to [6]. The approach being developed within this paper is different for a number of reasons that originate from the need to support large-scale complex systems that take many years to develop. The main differences are:

- The need for flexibility to account for incomplete specifications and changing designs. Hence the design derived should be able to handle change and not just meet the “current” requirements.
- The need to be able to tolerate failures.
- The need to be able to partition up parts of the design, e.g. individual sub-systems or processors, so that individual suppliers can work in isolation. This also means that the partitioning and allocation needs to be flexible otherwise a constant re-negotation would be needed between suppliers and customer which would be expensive.
- The need to defer the choice of implementation solution with respect to hardware and software. For this reason a two stage process is proposed that produces an allocation and assigns properties to the system but the final implementation details is not decided. For this reason interface requirements are established between the two phases. In the case of timing, these interface requirements are in the form of timing budgets.

The following section describes the framework we have been developing for the timing aspects of phase 1 of the process. Current plans is to use the constraints and interface requirements derived from phase 1 to drive phase 2 of the process which could be based on an existing technique.

5 Co-simulation and Optimisation

5.1 Cost Function

Stage 4 of the design assessment is exploring the available design solutions for the combination that best meet the system's objectives. From the contents of Table 7 this means that from a timing perspective analyses need to be performed to demonstrate the following:

1. Normal (showing timing requirements are met).
2. Determining how the task set copes with extra tasks being added and changing execution times (WCET and BCET).
3. Tasks have low jitter.
4. Scheduling is fault tolerant, i.e. some tasks' execution can be repeated, in case of failure, without affecting the ability to meet timing requirements.

For the purposes of this work, it is assumed that no re-allocation of tasks to processors or change to the budgets is made when the nature of the task set changes or when a fault occurs. Instead it is expected that the assignment derived for the task attributes and budgets can cope with the changes or failures. It should be noted that this does not preclude the use of replication to also provide fault tolerance. In cases where changes to the task set leads to the requirements being met, then it would be expected that a re-allocation and re-assignment of budgets would then be performed.

Since the assessment is to be performed in a platform dependent manner, actual WCETs and BCETs are not known. Therefore budgets are to be derived for WCETs and BCETs. However rather than generate completely abstract and infeasible budgets, some control is provided by one of the assessment criteria being whether the budget is broadly in line with an estimate. This estimate is found by a combination of; whether the relative budgets between tasks is comparative to the tasks' execution times on another platform, and whether the budgets are comparative to the tasks' size and complexity. Later in the development of the system, the estimates could be obtained by analyzing or measuring each task's actual execution time. Using these estimates though do not prohibit a set of budgets being derived that mean particular tasks need more effort and optimisation to meet their budgets if it provides enough benefits in other areas.

The optimisation was performed using a simulated annealing algorithm and a cost function whose parameters (e.g. parameters) are described in Table 1.

Assessment Criteria	Weighting	Bonus Factor	Penalty Factor
Individual task schedulability	500 per task	5	5
Multiple task schedulability	500 per dependency	5	5
Number of processors > 1	10000	N/A	N/A
Task fault tolerance	100	N/A	N/A
Task execution variability	20	N/A	N/A
Relative size	2	N/A	N/A
Task scalability	10	N/A	N/A
Execution scalability	10	N/A	N/A

Table 1 – Weightings for Each Assessment Criteria

The simulated annealing algorithm is chosen rather than standard static search techniques due to its ability to scale to large systems [13]. It is chosen over other heuristic search techniques due to its ability in finding good solutions assuming it does not get stuck in a local minimum [13]. To prevent this, if a best solution is not found after a defined number of moves, then the algorithm is re-seeded with a completely new solution. Section 5.2 contains further details of the algorithms.

The table has four columns; the first being the assessment criteria, the second the standard weighting (found through evaluation) used for the scoring mechanism and evaluation method presented in later in this section, the third a bonus factor used in cases such as when all the assessed timing requirements are met, and the fourth a penalty for when all the assessed timing requirements are not met.

The results of the analysis are converted to a score that can be used in the cost function by the following means:

1. **Individual task schedulability**: for each task that is schedulable (i.e. meets its requirements) a score of +1 is given and for each unschedulable task a score of "**–** *PENALTY"* is given (e.g. *PENALTY* is equal to +5). If all tasks are schedulable, then the final results is multiplied by a bonus factor (e.g. +5) to bias the results in favour of a completely schedulable solution.
2. **Multiple task schedulability**: for each requirement met a score of +1 is given and for each requirement not met a score of "**–** *PENALTY"* is given. Again if all requirements are met, then a bonus factor is applied to the result.
3. **Task fault tolerance**: +1 for each task that is re-runnable without affecting the ability to meet timing requirements.
4. **Task execution variability:** -1 for every clock tick that each task's WCET is greater than its BCET, i.e. sum for all tasks of (WCET-BCET).
5. **Number of processors:** -1 for every processor in the system greater than one. The reason for every processor greater than one being used is that we can't avoid having one processor.
6. **Relative size**: relationship between two tasks' estimated WCET (EWCET), which is approximated via metrics or transformation of WCETs from other processors, and their budgeted WCET. That is,

$$= \sum_{\forall i} \sum_{\forall j} \frac{\frac{EWCET_i}{EWCET_j}}{\frac{WCET_i}{WCET_j}}$$

   where i, j are individual tasks in the task set
   The aim of "relative size" is to indicate that the tasks' WCET budgets are in line with their estimated WCET.
7. **Execution scalability:** +1 for every clock tick that each task's WCET is greater than its EWCET, i.e. sum for all tasks of (WCET-EWCET). To penalise WCET budgets being assigned that are smaller than the estimated WCET, a score of **–**10 for every clock

tick that each task's WCET is less than its EWCET, i.e. sum for all tasks of (WCET-EWCET).

8. **Task scalability**: +1 for every extra randomly generated task that can be added to the task set without affecting the ability to meet the system's requirements.

5.2 Searching the Design Space

The simulated annealing algorithm used in our work can be described by the following pseudo-code. The pseudo-code features re-seeding which is used to prevent the solution getting trapped in part of the search space.

```
randomly generate an initial model
loop for each temperature(T)
    if improved solution not found after N moves
        re-seed solution with completely new solution
    loop for number of times inv prop to T
        select new model
        move to new model
        calculate cost function = Σ_{∀ cost function elements} weightings × score
        if new model has higher cost value
            adopt it
        else
            draw random number
            decide whether to adopt it
    end random moves loop
end temperature loop
```

A "new" model is found by modifying the current model of the system in a randomly selected way from a number of ways. Modifying the current model is equivalent to using a new "design tactic" as discussed in [10]. Examples of design tactics are the use of a technique such as fixed priority scheduling. The following is a list of ways in which the current solution is modified in the simulated annealing algorithm. It should be noted that these represent the sub-set of design choices originally listed in section 3.3 that are relevant to the first of the two-stage allocation process – the second stage being choosing the actual hardware (e.g. microprocessor) that allows the timing budgets to be met.

Processor level – for randomly chosen task or message

1. **Execution times (worst and best-case) –** increase, decrease or random.
2. **Ordering –** increase, decrease or random.
3. **Offset –** increase, decrease or random.
4. **Release jitter –** increase, decrease or random.

System-Level

1. **Task allocation –** move a randomly chosen task between processors. This could lead to the addition of a new processor.
2. **Processor –** remove a processor from the system.

6 Evaluation

The evaluation presented in this section is intended to show how the framework uses a set of requirements and an estimated WCET to generate what it considers the best solution. To demonstrate the way the framework operates in the available space, a small example with few tasks is chosen. However other work we have performed has shown that the approach is equally applicable to large-scale systems. In addition, a great deal of other work, including [11], has shown heuristic search algorithms can handle the scalability to allocating tasks for large systems.

For the purposes of the example considered the Fixed Priority Scheduling approach is used [9]. However the theory developed can be applied to other scheduling approaches or even to decide between scheduling approaches for a particular problem. The priorities are initially derived according to the deadline monotonic priority ordering [9] where the tasks with the shortest deadline have the highest priority. In this case where tasks have an equal deadline an arbitrary decision is taken on which has the highest priority.

The example consists of 10 tasks with an initial resource utilization (equal to the sum for all tasks of EWCET/Period) of 1.70 – initial resource estimate is based on the estimated WCETs). The tasks have 2 dependency requirements of which one is a separation requirement and one is a transaction requirement. The requirements and initial attributes are depicted in the following tables. Table 2 gives the individual task requirements and attributes. Table 3 gives the requirements for multiple tasks.

Id	Period (T)	Deadline (D)	EWCET	Jitter Req	Priority (P)
0	90	40	19	N/A	3
1	80	70	15	N/A	6
2	100	100	20	N/A	10
3	90	80	18	N/A	7
4	70	60	11	N/A	5
5	70	20	10	N/A	1
6	70	50	14	N/A	4
7	70	30	13	N/A	2
8	100	80	11	N/A	8
9	100	80	11	N/A	9
Resource Utilisation					1.70

Table 2 - Task Requirements and Initial Attributes

Id	Type	Precedence		Min. Separation Requirement (S)	End-to-End Deadline (TD)
		1st Task	2nd Task		
0	Separation	6	7	2	N/A
1	Transaction	8	9	N/A	90

Table 3 - Task Dependency Requirements

For the given example, Table 1 shows the weightings that were used in the context of this specific system. Table 1 shows that the greatest emphasis is given to ensuring task requirements are met (both individual tasks and dependencies between tasks) and to reducing

the number of processors within the system. (It should be noted that some of the assessment produces outputs of markedly different quantities.) For example, the degree of variability in execution time (measured in clock ticks) between a task's WCET and BCET added up for all tasks is likely to be significantly greater than the number of processors (in addition to the minimum of one) used.

Id	T	D	EWCET	BCET	WCET	A	O	RJ	P	R
0	90	40	19	14	20	0	3	1	3	34
1	80	70	15	20	20	2	4	0	6	50
2	100	100	20	20	20	0	4	3	10	77
3	90	80	18	17	20	0	5	4	7	59
4	70	60	11	11	11	2	11	1	5	38
5	70	20	10	9	10	0	4	0	1	14
6	70	50	14	8	15	2	2	2	4	19
7	70	30	13	19	19	1	4	3	2	26
8	100	80	11	12	13	1	8	3	8	43
9	100	80	11	19	20	1	9	2	9	63
Resource Utilisation									2.01	
Resource Utilisation for Processor 0									78.7	
Resource Utilisation for Processor 1									60.1	
Resource Utilisation for Processor 2									62.1	

Table 4 - Task Schedulability Results

The results of the analysis are presented in Table 4 and Table 5. In Table 4, *R* represents the Worst-Case Response Time (WCRT), *A* the processor to which a task is allocated, *O* the offset for a task, and *RJ* the release jitter for a task.

Id	Precedence		S	TD	Actual Separation	WCRT
	1st Task	2nd Task				
0	6	7	2	N/A	7	N/A
1	8	9	N/A	90	N/A	55

Table 5 - Task Dependency Results

The following is a discussion of the solution found with respect to each assessment criteria.

- **Individual task schedulability –** All the individual tasks are schedulable so a maximum score is achieved here.
- **Multiple task schedulability** - All the task dependency requirements are met so a maximum score is achieved here. It should be noted that the solution derived is such that the majority of the dependent tasks are situated on a single processor – i.e. A=1. This makes schedulability easier as there are no time critical messages since separation requirements do not require messages
- **Number of processors (greater than 1)** - The results show that the solution found features three processors. Since the resource usage of the revised task set is greater than two and less than three, then three processors is the minimum number that can schedule the system. Therefore with respect to this criterion an optimum solution has been found.
- **Task fault tolerance** - The resource utilisation on each processor is well balanced which helps increase the likelihood that tasks can be executed for a second time in case of a failure being detected. For instance, Table 6 presents the schedulability analysis results for the situation where task *1* is re-executed due to an error. In this case all tasks are still schedulable. The results from the co-simulation did however show that not all cases of task failure and subsequent re-execution mean the entire task set remained schedulable. However in the majority of these cases, it was the lowest priority task on a particular processor that became unschedulable.
- **Task execution variability –** The results indicate that the difference between the BCET and WCET is small (i.e. less than 25% difference) in most cases. The exception to this rule is the task with identifier *6* whose BCET is 8 and WCET is 15.
- **Relative size –** The results indicate that the WCET budgets chosen are broadly inline with the EWCETs and in all cases the WCET budget is greater than the value of EWCET.
- **Task scalability and execution scalability –** The resource utilisation on each processor is well balanced which helps increase the degree of scalability that is possible.

Id	T	D	EWCET	BCET	WCET	A	O	RJ	P	R
0	90	40	19	14	20	0	3	1	3	34
1	80	70	15	20	20	2	4	0	6	70
2	100	100	20	20	20	0	4	3	10	77
3	90	80	18	17	20	0	5	4	7	59
4	70	60	11	11	11	2	11	1	5	38
5	70	20	10	9	10	0	4	0	1	14
6	70	50	14	8	15	2	2	2	4	19
7	70	30	13	19	19	1	4	3	2	26
8	100	80	11	12	13	1	8	3	8	43
9	100	80	11	19	20	1	9	2	9	63

Table 6 – Fault Tolerance Schedulability Results

7 Conclusions

This work has shown how trade-offs in the timing aspects of how software can be mapped onto hardware can be handled. The approach made use of interface requirements between the hardware and software such that each of these design processes can be performed independently. Firstly, a number of design choices and assessment criteria were derived from the top-level objectives of the system using a systematic method that captures the rationale behind the design decisions in a traceable manner. Secondly, an experimental method for evaluating a particular design was produced that combined static analysis of the baseline system with scenario-based assessment of how the system may behave in the presence of change and failures.

Using the design choices available and the experimental method, optimisation tactics were employed to determine the best solution to a particular problem. Given this best solution, the hardware and software can be development in relative independence. At integration time, it would have to be shown that the low-level platform design of the hardware is sufficient to meet the interface requirements for the software that has been developed. In cases where the interface requirements are not met, then a new solution would be to be found using the framework.

Future work could include developing phase 2 of the

process and incorporating other objectives such as power.

8 References

[1] Y. Yeh, *Dependability of the 777 Primary Flight Control System*, Proceedings of the 5th IFIP Conference on Dependable Computing for Critical Applications, 1995.

[2] D.M. Buede, *The Engineering Design of Systems*, Wiley, 2000.

[3] R. Kazman, M. Klein, and P. Clements, *Evaluating Software Architectures - Methods and Case Studies*. Addison-Wesley, 2001.

[4] T. Kelly, *Arguing Safety – A Systematic Approach to Safety Case Management*, DPhil Thesis, YCST-99-05, Department of Computer Science, Univ. of York, 1998.

[5] I. Bate, T. Kelly, *Architectural Considerations in the Certification of Modular Systems*, Proceedings of 22nd International Conference on Computer Safety, Reliability and Security (SAFECOMP 2002), 2002.

[6] G. de Micheli, R. Ernst, W. Wolf, *Readings in Hardware / Software Codesign*, pub. Morgan Kaufmann, 2002.

[7] M. Ward, N. Audsley, *Hardware Compilation of Sequential Ada*, Proc. of CASES'01, pp. 99-107, 2001.

[8] K. Shin, Y. Chang, *A Reservation-based Analysis for Scheduling Both Periodic and Aperiodic Tasks*, IEEE Trans. on Computers, 44(12), pp. 1405-1419, 1995.

[9] I. Bate, *Scheduling and Timing Analysis of Safety Critical Hard Real Time Systems*, Phd Thesis, Department of Computer Science, Univ. of York, YCST-99-04, 1999.

[10] F. Bachmann, L. Bass, M. Klein, *Illuminating the Fundamental Contributors to Software Architecture Quality*, Software Engineering Institute, CMU/SEI-2002-TR-025, 2002.

[11] M. Nicholson, *Selecting a Topology for Safety-Critical Real-Time Control Systems*, DPhil Thesis YCST-98-08, Department of Computer Science, University of York, UK, 1998.

[12] V. Basili, H. Rombach, *The TAME Project: Towards Improvement-Oriented Software Environments*, IEEE Trans. on SE, 14(6), pp. 758-773, 1988.

[13] V. Rayward-Smith, I. Osman, C. Reeves, G. Smith (Editors), *Modern Heuristic Search Methods*, Wiley, 1996.

[14] J. Engblom, A. Ermedahl, F. Stappert, *Validating a Worst-Case Execution Time Analysis Method for an Embedded Processor*, Dept. of Information Technology Tech. Report 2001-030, Uppsala University, 2001.

9 Appendix

9.1 Overview of GSN

The Goal Structuring Notation (GSN) [4] - a graphical argumentation notation - explicitly represents the individual elements of any safety argument (requirements, claims, evidence and context) and (perhaps more significantly) the relationships that exist between these elements (i.e. how individual requirements are supported by specific claims, how claims are supported by evidence and the assumed context that is defined for the argument). The principal symbols of the notation are shown in Figure 2 (with example instances of each concept).

The principal purpose of a goal structure is to show how goals (claims about the system) are successively broken down into sub-goals until a point is reached where claims can be supported by direct reference to available evidence (solutions). As part of this decomposition, using the GSN it is also possible to make clear the argument strategies adopted (e.g. adopting a quantitative or qualitative approach), the rationale for the approach (assumptions, justifications) and the context in which goals are stated (e.g. the system scope or the assumed operational role). For further details on GSN see [4].

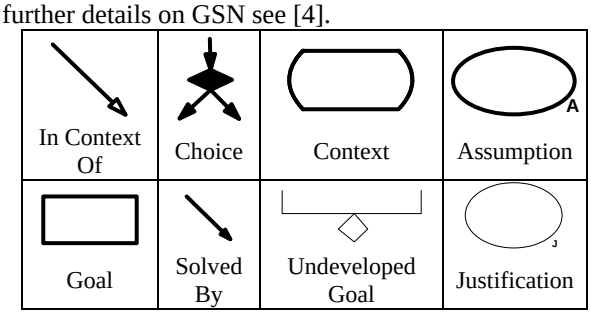

Figure 2 – Principal Elements of GSN

9.2 Arguments to Support the Trade-Off Analysis

Goal	Qualitative	Quantitative	Importance
G3001 – Hardware used is predictable	For proposed hardware, raise questions such as: a. Does documentation exist? b. Does it seem comprehensive?	Choice (originating from goal **G2004**) of either: a. derivation of models for the different non-functional properties validated against the actual hardware. b. development of an appropriate testing strategy.	Depends on the integrity of the system.
G2017 and **G2018** – Use a software or hardware based language	For a proposed language, raise questions such as: a. Have previous systems been successfully developed with it? b. Does the language provide a set of features sufficient for the envisaged application? c. Does the language allow static analysis to be performed? d. Are support tools available?	Assessment activities could include: a. Related to **G2013**, obtaining/defining semantics. b. Related to **G2008**, obtaining/producing complementary static analysis tools. c. Related to **A2006** and **A2007,** determine wether the necessary language features are supported. d. Related to **J2003** and **J2004**, is an appropriate mapping available from software to hardware?	High since changing the language used part way through development can lead to large amounts of (if not total) rework.
G2014 – Infrastructure provided is sufficient	For a proposed infrastructure, raise questions such as: a. Based on previous experience, does the resources available seem sufficient? E.g. Are more MIPS available than for other similar projects b. Are the hardware's available features sufficient for the application? c. Are support tools available?	a. Related to **G2004,** early in the project, use data from previous similar systems and other metrics (e.g. number of requirements). Later, use actual data obtained via static or dynamic analysis. At all stages, possibly perform sensitivity analysis to de-risk further development. b. Related to **G2014**, determine the hardware features needed and whether they are supported.	Depends on how easily the infrastructure or application can be changed.

Table 7 – Assessment Using the Contents of the Arguments

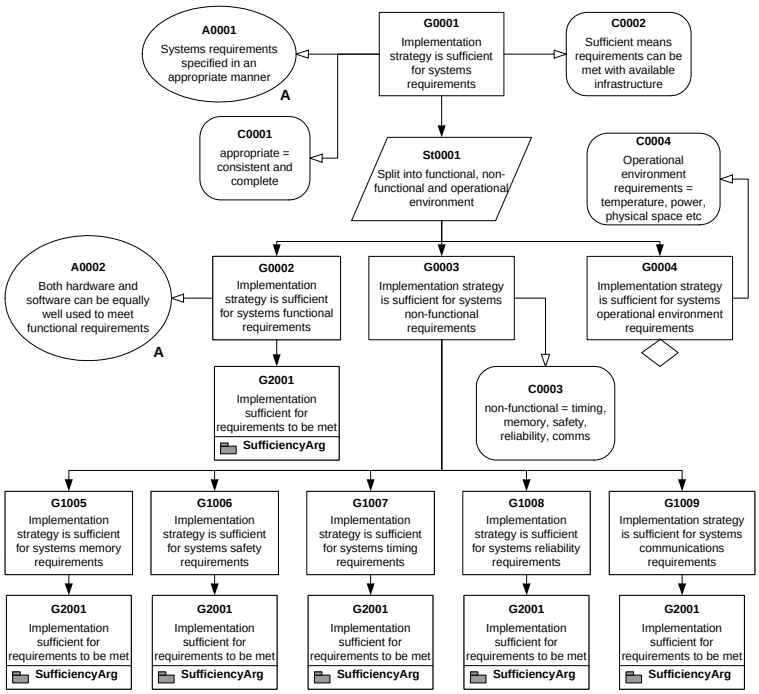

Figure 3 - Top Level Argument

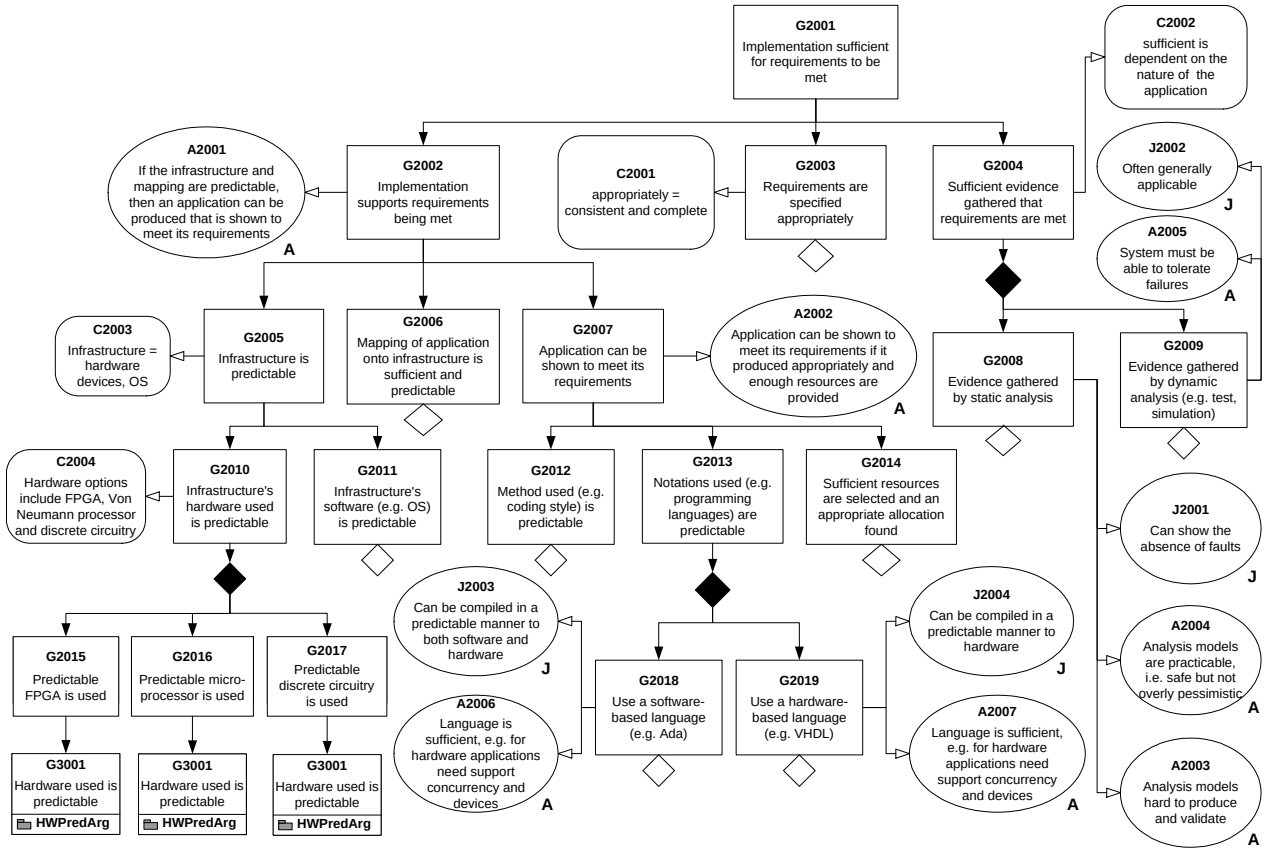

Figure 4 - Implementation Meets the Requirements – *"SufficiencyArg"*

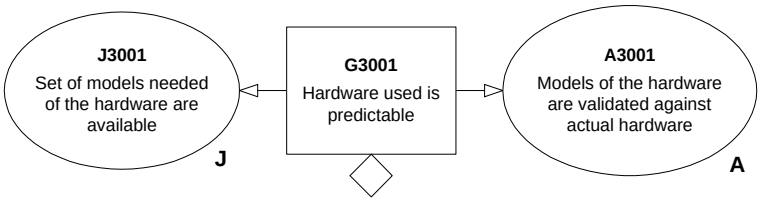

Figure 5 - Hardware is Predictable – *"HWPredArg"*