# Use of Modern Processors in Safety-Critical Applications

IAIN BATE, PHILIPPA CONMY, TIM KELLY AND JOHN MCDERMID

*Department of Computer Science, University of York, Heslington, York YO10 5DD, UK*
*Email: iain.bate@cs.york.ac.uk*

**This paper investigates the implications of using modern superscalar processors in the safety-critical domain. Firstly, a description of current certification practice and devices is given as background. This is followed by an exposition of the certification argument for a processor when used in a safety-critical application. Throughout the presentation of the argument two types of modern processor are considered, commercial off-the-shelf (COTS) processors and purpose-designed bespoke devices. This allows the elaboration of positive and negative features of processors that can be used as part of the selection (for COTS) or design (for bespoke) process.**

*Received 6 November 2000; revised 13 April 2001*

## 1. INTRODUCTION

Safety-critical systems typically use well-established, well-understood scalar processors, such as the Motorola 680x0 range, where instructions are sequentially executed in their original order. However, this generation of processor is generally no longer in production or is incapable of meeting performance requirements, leaving companies that build safety-critical systems with limited options for new development.

One solution to this problem is to make a lifetime buy of the remaining processors and keep them in storage. However, the supply could be exhausted through use in production runs and also through storage degradation. In addition, there are difficulties in preserving the development environment over, say, a thirty-year life span (such longevity is not unusual in the aerospace domain). Although this approach is currently used in practice, it is increasingly being called into question.

A second solution is to design and manufacture a purpose-built processor, so that in the future the changing commercial market does not affect the company. This solution would necessitate the development of bespoke support tools, e.g. a compiler.

A third solution could be to use one of the available commercial off-the-shelf (COTS) processors. This solution would have the benefit of COTS tools being available. However, the drawback is that the modern processors available are generally significantly more complex than those currently used in critical systems.

The latter two solutions are considered in this paper in order to determine their relative benefits and drawbacks, and their effect on the certification argument. Although the two types of processor would have many similar features, such as a pipeline and cache, a COTS processor is likely to be built for optimum average case performance. In contrast, the bespoke processor would be designed to ease the certification process (e.g. to ensure predictability). Typical types of evidence required for certification are worst-case execution time (WCET) of instructions, hardware reliability and information on systematic design flaws in the processor.

The purpose of this paper is two-fold: first, to describe the processor-related safety arguments typically required as part of system certification. Second, to illustrate the impact of processor choice (COTS vs. bespoke) on the evidence required to support these arguments.

The structure of the remainder of the paper is as follows. Section 2 provides background on the factors influencing the choice of processor in safety-critical applications. Section 3 provides an overview of the regulatory context influencing processor use. Section 4, the main body of the paper, presents the outline structure of the required processor certification arguments. Finally, conclusions are presented in Section 5.

## 2. CHOICE OF PROCESSOR

The principal influence on COTS processor development has been the general-purpose computing market (e.g. personal computers, mobile telephones, etc.), driven by market forces and economy of scale. The entire production run of a safety-critical system may use a few thousand processors, whereas a successful mobile phone will use millions. Since a great deal of academic research is influenced by industry, then the technical basis of bespoke processors is subject to similar influences to COTS processors, however processor features can be carefully selected.

The following section describes the two principal features of processors, memory and execution of instructions, and looks at how they have changed from scalar processors to produce superscalar devices.

### 2.1. Execution of instructions

The execution of instructions on a scalar processor takes place one at a time and in the same order as the instructions appear in object code. The processor may have multiple units (e.g. load/store units and floating-point units) that combine to provide the necessary functionality, but these are never used concurrently. The execution strategy is therefore simple to understand and analyse. This is particularly useful when performing timing analysis. No matter what type of scheduling mechanism is used, all forms of timing analysis depend on knowing the maximum time a piece of software takes to execute. This is usually known as the WCET. WCET analysis can be performed by splitting software into blocks, where a block is a sequence of instructions that has only one branch at the start or at the end. Every feasible path through the blocks is then examined in order to find the longest [1]. The length is determined simply by adding up the WCET of each individual instruction.

In general, non-safety-critical systems are primarily concerned with good average-case performance, even if this is at the expense of poor but rarely encountered worst-case performance. In contrast, the development of safety-critical applications is concerned with being able to demonstrate that requirements are met, which necessitates an ability to bound the best and worst-case performance. Additionally for applications that are part of embedded control systems (of which there are many), jitter (the variation in when an event occurs, e.g. the release time of tasks) is often a concern because of its effect on the stability of the system's response to stimuli [2]. The emphasis on average-case performance has led to dramatic increases in the speed of processing compared to memory speed (over 20 years, a factor of 100,000 for processing times compared to 10 for memory). Consequently, there are increasing numbers of wait states inserted during instruction execution whilst waiting for memory. A wait state is the term used to describe the condition when processing is halted for a clock cycle whilst waiting for an event to occur, e.g. for a memory access to complete.

Figure 1 shows how pipelines, caches, registers and main memory are integrated in a typical modern processor. Figure 2 illustrates the pipeline structure for the PowerPC 603e [3], a typical example of a modern processor. The figure shows that the execution of instructions is split up into a number of stages and that some of these stages (principally the execute stage) have multiple units to support concurrent execution of instructions. This approach allows multiple instructions to be handled simultaneously and means the processor can do something productive rather than remain idle if the currently executing instruction has been delayed (e.g. by a cache miss). Processors of this type are often referred to as superscalar.

The following list contains some features of the pipeline mechanism. It should be noted that these features cannot simply be 'turned off' and the programmer has little influence over their operation. If WCET analysis of a superscalar processor were to produce accurate results, it would have to include an accurate model of the pipeline.

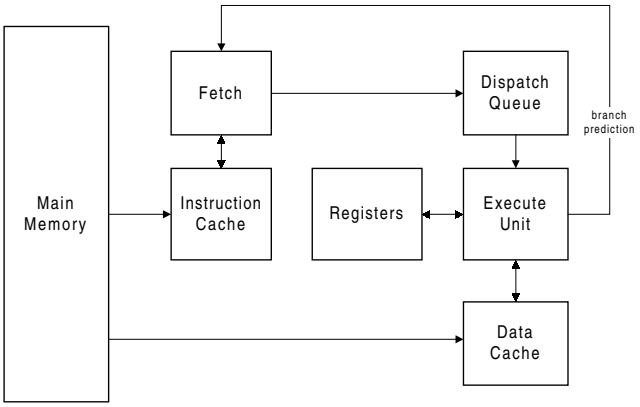


**FIGURE 1.** Typical processing architecture.

(1) *Multiple Issue*. The ability to fetch, dispatch and complete a number of instructions per clock cycle.
(2) *Parallel Execution*. From Figure 2 it can be seen that there are four types of processing units in the execute stage: floating point, system register, load/store and integer (possibly multiple).
(3) *Out of Order Execution*. This can prevent the pipeline stalling when an instruction takes a long time to execute and/or there has been a cache miss.
(4) *Speculative execution*. The pipeline mechanism does not always wait for the result of a comparison before processing the associated branch instructions (and subsequent instructions after the branch folding or falling through). Instead the pipeline mechanism guesses which branch is the likeliest to be taken using branch prediction.

### 2.2. Memory access

The principal storage area for data and instructions is main memory. This is often significantly slower than the processor. Although faster memory is available, this is expensive and it is only practical to use it where small quantities are sufficient. Thus, some processors utilize cache memory, which is relatively small and fast, as temporary storage. A proportion of the memory accesses will be made from the cache, rather than accessing the slower main memory for each instruction. A memory manager is used which attempts to optimize the contents of the cache memory for maximum throughput. However, using a cache memory increases the complexity and accuracy of the analysis, in so far as:

- there are greater variations in the execution times of the software [4];
- it is harder to deduce the actual WCET of the software as the analysis would have to include a model of the cache mechanism—it could be assumed that all memory accesses result in a cache miss but this gives results much greater than the actual WCET, leading to wasted resources [4];

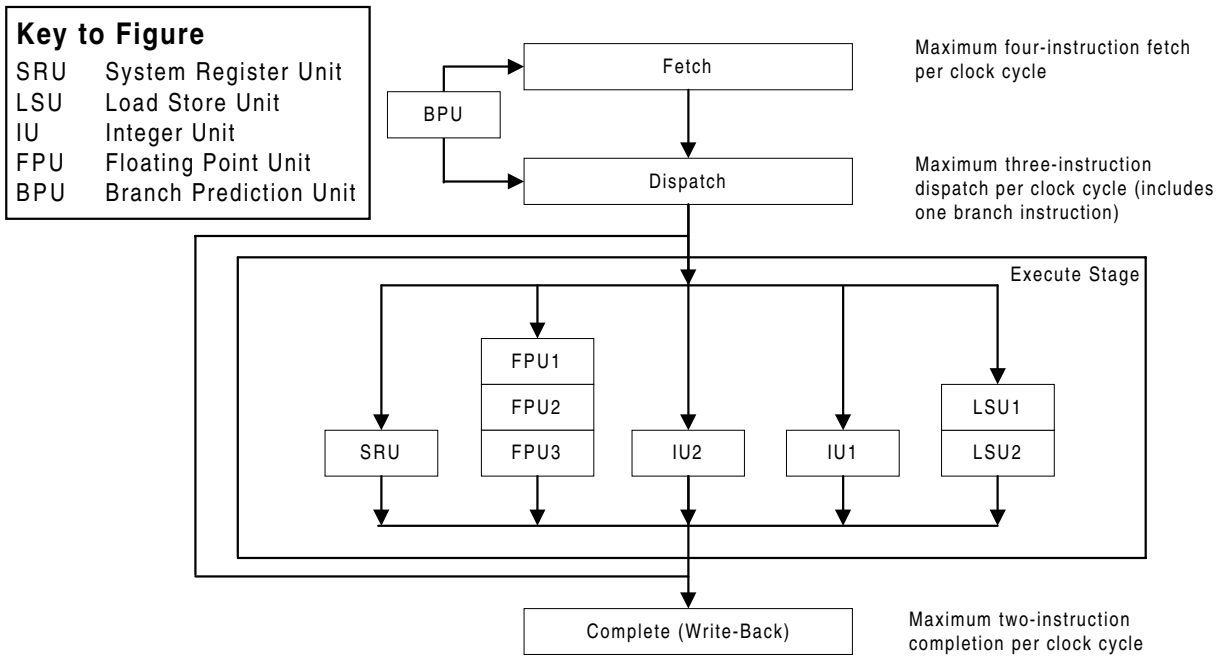


**FIGURE 2.** Pipeline structure of a typical modern processor—the PowerPC.

- the more complex the processor is, the harder it is to validate the models used and the higher the likelihood of systematic design errors (in the processor and the accompanying models) [5].

Therefore, if the processor has cache memory it is often switched off for safety-critical use. However, if it is used then it is assumed that each memory access results in a cache miss. The longest execution path is then intuitively the worst case.

Due to the increasing disparity in speed between processing and memory, modern processors have become more reliant on cache memory and some processors have two levels of cache. The first level of cache is a small but very fast 'primary' cache on the actual processor (typically with zero wait states). The other is a larger 'secondary cache' which has more wait states than the primary cache, but less than main memory. The methods for optimizing the contents of, and access to, the cache have also changed significantly. One complication of having cache is allowing for the impact of an interrupt/pre-emption on the cache contents and hence on the program flow.

### 2.3. Trade-offs in bespoke processor design

The opportunity in developing a bespoke processor for safety-critical applications is that processor features can be limited to those that may easily be analysed. This enables a trade-off between processor performance and processor predictability. For example, a bespoke safety-critical processor design may only include a primary cache where its contents are statically defined and the pipeline only uses a subset of the potential features, such as parallel execution and multiple issue.

## 3. REGULATORY CONTEXT

There are a number of safety standards and guidance documents that govern the development of hardware for use in safety-critical applications. Before discussing the 'core' certification arguments, we first provide a description of some of the specific requirements as defined by a number of the applicable safety standards.

### 3.1. UK defence standards

The UK Defence Standards 00-54 [6], 00-55 [7], and 00-56 [8] set out requirements and guidance for the development of safety-critical systems. Defence Standard 00-54 defines requirements specifically for safety-related electronic hardware (SREH) and is similar in nature to the software standard, Defence Standard 00-55.

Defence Standard 00-54, Part 2 Clause 8.2.1 advises that a hardware safety case should be constructed to present 'a readable justification that the hardware is safe in its specified context'. The context of a processor includes both the application software and its physical operating environment. Discussion of operating environment limits and conditions is outside the scope of this paper, but key issues include temperature, pressure and humidity ranges. Part 1, Clause 7.3.1 (b) of Defence Standard 00-54 supports the concept of establishing safety arguments (as an SREH safety case) for hardware elements of a system that will be used within the overall system safety case:

> The development of SREH shall include the following safety management activities and documentation ... [including] the production and maintenance of a safety case as a constituent part of System Safety Case.

Defence Standard 00-54 is a general electronic hardware standard and, as such, does not define requirements at the level of processor characteristics. Instead, it sets out general requirements concerning the specification, development, verification and validation, and safety assurance of hardware. In addition to the traditional requirement for evidence to demonstrate satisfaction of random failure rate targets, 00-54 also requires the identification and analysis of systematic errors in the hardware.

Defence Standard 00-55 requires both static code analysis and software testing as part of the system safety case. Static software analysis examines those properties of the code that can be determined prior to execution. Results of code analysis that are affected by the processor are:

- the WCET taken and the amount of memory required by the software should be bounded and statically determined;
- the software should be tolerant to and respond to random failures of the hardware;
- the system should have built-in tolerance to avoid overload, and still be capable of running in a degraded state; and
- any direct access from software to the hardware needs to be analysed.

### 3.2. Civil avionics guidance

The civil avionics arena uses guidance documents rather than standards although, in practice, it would be difficult to get approval for systems that did not follow the guidance. The relevant guidance documents are DO-178B [9] for software and DO-254 [10] for hardware. DO-254 has only very recently been published, therefore our discussion focuses upon DO-178B.

DO-178B requires the developers to produce an accomplishment summary identifying the evidence that the software meets its requirements (this is analogous to the Defence Standards idea of a safety case). It places much greater emphasis on testing and human review than Defence Standard 00-55. The summary includes evidence that the software is compatible with the hardware—which implicitly includes timing. It also requires the gathering of certification evidence through hardware/software integration testing. The testing should demonstrate that high-level requirements are met for software running on the target hardware, including timing requirements where these are significant.

DO-254 is similar in philosophy to DO-178B, although it places more emphasis on formal analysis.

### 3.3. IEC61508

IEC61508 [11] is an international generic safety standard that provides guidance on the development of electrical, electronic and programmable electronic systems (E/E/PES). Part 2 of the standard is particularly concerned with the development of electronic hardware and is therefore most relevant to the subject of this paper.

The requirements of Part 2 in IEC61508 are divided predominantly into two categories. Firstly, process-oriented requirements are presented concerning the avoidance of systematic error introduction in the hardware development process (e.g. concerning hardware specification techniques). Secondly, product-oriented requirements are defined concerning the tolerance to, and control of, both random and systematic hardware faults.

In the following section, rather than presenting certification arguments targeted to a specific safety standard, we present the certification arguments that can be regarded as 'core' to all standards, e.g. the identification, avoidance and control of systematic errors in a processor design.

## 4. PROCESSOR ARGUMENTS WITHIN THE SYSTEM SAFETY CASE

### 4.1. Introduction to goal structuring notation

Within this and the following sections, we use the goal structuring notation (GSN) [12] to outline the safety arguments that need to be made to support the use of a processor within a safety-critical system. Any safety case can be considered as consisting of requirements, argument, evidence and definition of bounding context. GSN—a graphical notation—explicitly represents these elements and (perhaps more significantly) the relationships that exist between these elements (i.e. how individual requirements are supported by specific arguments, how argument claims are supported by evidence and the assumed context that is defined for the argument).

The principal symbols in the notation are shown in Figure 3 (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 [12].

### 4.2. Top-level argument that the system is safe

Figure 4 presents the top level of the 'generic' safety argument required to establish the certification case for a modern processor. It is important to note that system- and hazard-specific arguments concerning the safety of the software (application) running on the processor should be presented alongside such an argument and are outside the scope of this paper. Separate software safety arguments often contain implicit assumptions about correct and error-free operation at the processor level. These assumptions will be discharged through the argument we present.

The argument presented in Figure 4 starts conventionally (goal: 'ProcAcceptSafe') by arguing that the processor

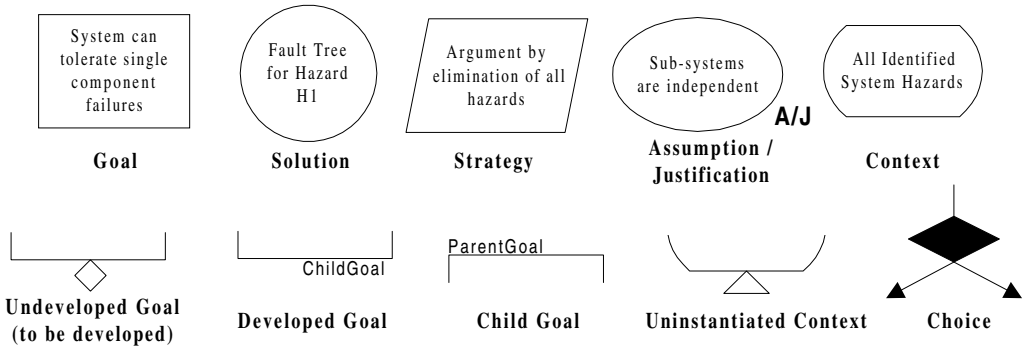


**FIGURE 3.** Principal elements of the goal structuring notation.

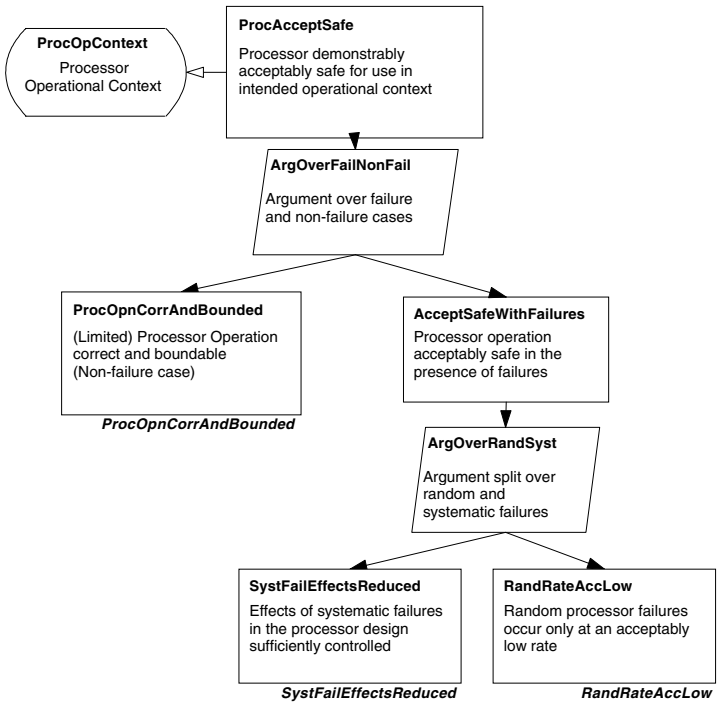


**FIGURE 4.** Top-level processor safety argument.

is demonstrably acceptably safe within the intended operational context (context: 'ProcOpContext'). It is important that the operating context is explicitly identified as it defines both the explicit (functional) and implicit (non-functional) interface between the processor and its environment, which could influence the safe operation of the processor; e.g. the electromagnetic, thermal and mechanical contexts. The argument strategy adopted is to split the argument into two sub-arguments. The first of these arguments puts forward the positive claim that in normal operation the processor will 'do what it's told' (i.e. will operate correctly) and that its operation is predictable. This argument is developed further in Section 4.3. The second of these arguments acknowledges that both random and systematic failures are inevitable and puts forwards the claim that the processor operation can be argued to be acceptably safe in the presence of these failures. The arguments of safe operation in the presence of systematic and random failures are developed in Sections 4.5 and 4.7 respectively. The concept of acceptability introduced at the beginning of the argument can only truly be defined in the target context of the processor. Included in this definition of acceptability may be a random failure rate requirement used as the basis for defining the 'RandRateAccLow' claim found at the bottom of Figure 4.

### 4.3. Correct processor operation

The argument supporting the assertion that the operation of the processor is both correct and boundable is presented in Figure 5. Within the definition of the claim 'ProcOpnCorrAndBounded' is the notion of defining *limitations* on the processor operation. The strategy in developing and supporting the argument of determinable processor operation may involve disabling or restricting the use of processor features (e.g. caching) so that operation

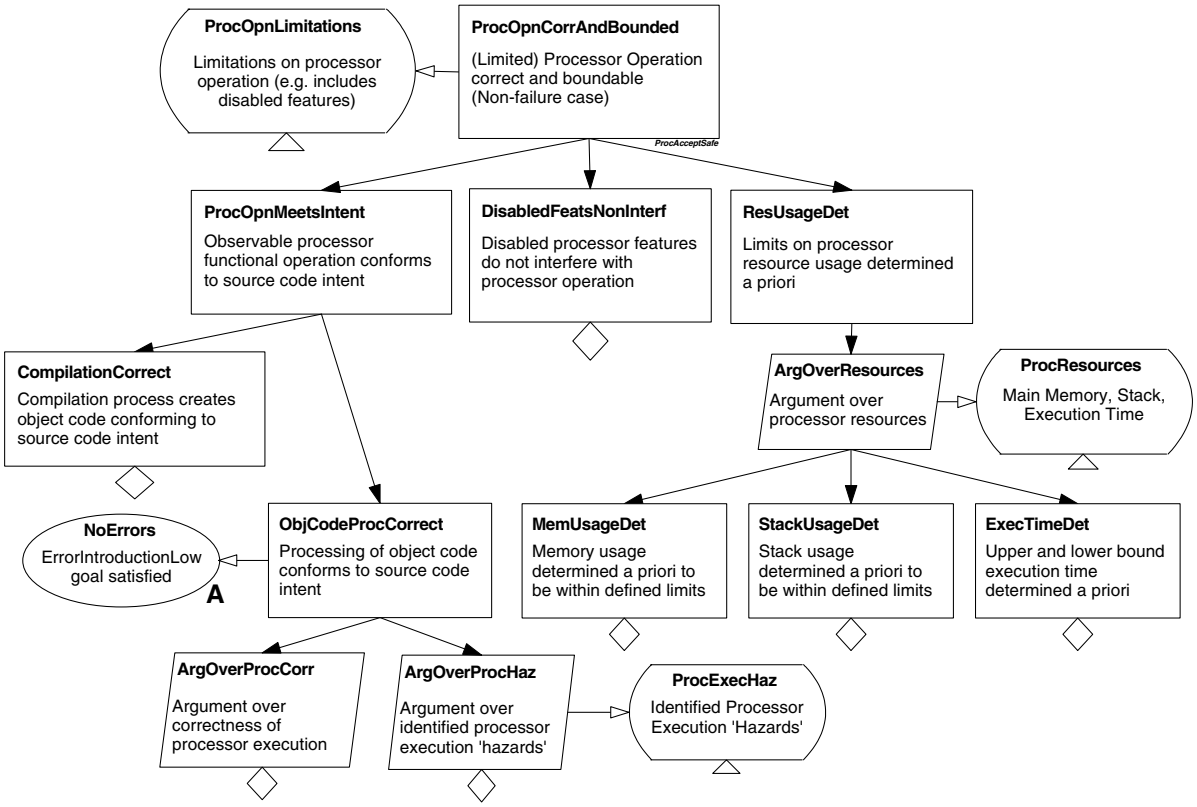


**FIGURE 5.** Correct processor operation argument.

is easier to predict and hence analysis effort is eased. Such disabling of features will obviously be at the expense of average-case processor performance and potentially the worst case. However, this may be an acceptable trade-off.

The first leg of the argument in Figure 5 puts forward the claim that the operation of the processor conforms to the intent as defined in the source code of the application software. For example, at the level of a single operation, if the source code describes an add operation, the argument is that the processor will perform this operation correctly. For the purposes of the processor argument we have been careful to define correctness with respect to the intent implicitly defined in the source code. Systematic (logical) errors in the source code that could lead to unsafe behaviour are outside the scope of this argument. The claim 'ProcOpnMeetsIntent' is first supported by an argument of the correctness of the compilation from source to object code, and secondly concerning the correctness of the processing of object code.

The compilation argument remains undeveloped (as indicated by the diamond underneath the claim 'CompilationCorrect') in Figure 5. Typical means of supporting such a claim include appealing to object-code verification evidence. This involves performing checks that the characteristics of the source code level (e.g. control flow) are reasonably represented at the object-code level. In this context, an advantage of the bespoke processor approach is that a bespoke compiler would be produced and hence produce output which eases the task of object-code verification.

Two strategies are put forward for supporting the object-code processing argument (under claim 'ObjCodeProcCorrect'). The first of these is concerned with arguing the correctness of instruction processing. (This goal is explicitly connected with the claim to be found in Figure 8—discussed in Section 4.5—that the processor contains an acceptably low number of design flaws.) The second strategy is concerned with processor execution 'hazards'. We use this term to refer to the undesirable impact of processor features on the correct operation of the processor (e.g. pipeline hazards). As is shown in Figure 5, this strategy requires some *a priori* analysis (referred to by the context: 'ProcExecHaz') of the hazards possible given the architecture and features of the processor.

Processor hazards could be introduced by out-of-order execution. For example, control-flow analysis is used to ensure that software is executed in the correct order, whilst data-flow analysis should ensure variables are not used prior to being set a value. This analysis is normally performed at the source-code level based on the assumption that the software semantics are preserved when executing on the hardware platform. However, a limited amount of flow analysis is performed at the object-code level to show the results are consistent with the software executing on the actual processor. Verification would have to be performed to demonstrate that the characteristics of the source code are upheld by the processor. This is complicated by the fact that the processor's pipeline is rarely observable. It would be further complicated if a COTS processor is selected, or a

bespoke processor designed, involving dynamic scheduling of instructions such as out-of-order execution.

When moving to a modern processor, further object-level verification is likely to be needed to deal with the advanced pipeline features of the processor. The extra verification would have to justify that the instructions' functional semantics (at the object-code level) were correctly represented by the processor's operation. This all adds to the cost associated with verification. For a COTS processor this is extremely difficult because there is no specification that can be relied on as the basis for verification. Therefore, the verification must be based on black-box testing but there is a limit to the evidence that can be derived by this approach. For a bespoke processor this involves showing that the processor's specification is appropriate and that the implementation meets this for all instructions, but a limited set of data.

Hardware validation needs to be included and has to show that the results are consistent within the processor itself and at its outputs. Part of this work is to demonstrate the accuracy of calculation units, such as the floating point unit. Hardware validation could be used to demonstrate the processor correctness, e.g. conformity to floating-point standards IEEE 754 [13] and IEEE 854 [14].

In order to take full advantage of fast execution times and expanded memory it is desirable to share processing resources between more than one application. Sharing memory requires a partitioning mechanism that can guarantee that a number of applications (possibly with multiple criticality levels) can share the same memory resource without risk of data corruption [15]. This involves the detection of attempted memory violations by an application. The Memory Management Unit (MMU) in many processors provides functionality to create interrupts when an illegal memory access occurs. To use this feature it would need to be demonstrated that the MMU would reliably provide interrupts when detecting an attempted violation and that the system software could deal with these interrupts. Since the integrity of the software relies heavily on this feature, a bespoke processor could be designed to support the necessary safety argument. In some cases where the MMU of a COTS processor has been used to support partitioning, it has been necessary to provide bespoke hardware to externally monitor for correct behaviour [15].

The second leg of the argument in Figure 5 (shown in claim 'DisabledFeatsNonInterf') goes hand in hand with the notion of limiting use of advanced processor features (e.g. disabling multi-processor cache coherency). The claim that must be supported here is that disabled features truly *are* disabled and that they can no longer interfere with the operation of the processor. The details are left undeveloped as this could involve tying down processor pins, e.g. cache enable, or policing the use of features in the programming language, e.g. object creation, at source level.

The third leg of the argument in Figure 5 (shown under claim 'ResUsageDet') focuses upon the *a priori* determination of processor resource usage. This argument is decomposed over the various resources of concern (identified through context reference ProcResources). Key resources include memory usage, stack usage and execution time. It must be possible to support the claims 'MemUsageDet' and 'StackUsageDet' that memory and stack usage are determinable *a priori*. Equally, it is essential to be able to argue that upper and lower bound execution time can be determined *a priori* (as claimed in 'ExecTimeDet').

### 4.4. Worst-case execution time analysis

Considering the claim 'ArgOverResources', and more specifically its sub-claim 'ExecTimeDet', raises one of the most significant practical problems, i.e. the subject of WCET analysis. (The most significant problem is verifying the correctness of COTS hardware but there is probably no practical 'complete' solution, only ways of raising confidence.)

The timing analysis of the system, and WCET analysis, becomes a much more complex task for a pipelined processor. This is a crucial activity, as most safety-critical systems are dependent on real-time deadlines being met. Ignoring features of the processor when calculating WCET leads to pessimistic results. As an illustration, on a 500 MHz (i.e. 2 ns clock cycle) PowerPC processor that uses 50 ns memory, there are at least 25 wait states per memory access. Ignoring the four-stage pipeline would mean the analysis would produce a result up to four times greater than the actual execution time. This is due to the analysis having to assume that each instruction propagates through every stage before the next instruction can begin. Therefore if both cache and pipeline features are ignored, then pessimism could easily reach two orders of magnitude.

An alternative to WCET analysis is to extensively test the system to obtain a measured value for timing. For lower integrity systems this may be an acceptable solution. However, for the highest integrity level it is necessary to perform analysis in order to guarantee timeliness. Simply testing the system cannot guarantee the absence of timing overruns [4].

Developing WCET analysis for modern processors is complicated as it relies on accurate models for the types of features discussed in Section 2. However, producing a completely accurate model may not be possible due to the lack of verifiable design information. Verification is difficult because most of the processor's mechanisms are not externally observable. In our experience, COTS processor manuals do contain errors that would easily lead to incorrect analysis. Accurate models also have limited portability. In fact, producing a software model for WCET analysis that completely emulates a processor would simply take too great a time to obtain results for any reasonable size of code [4]. However, a simple model may be hard to validate because it is sufficiently different to the actual processor.

To further complicate the issue, not accurately modelling all the processor's features may introduce anomalies [16] and unmanageable pessimism into the results. An example of an anomaly is illustrated in Figure 6 where a cache

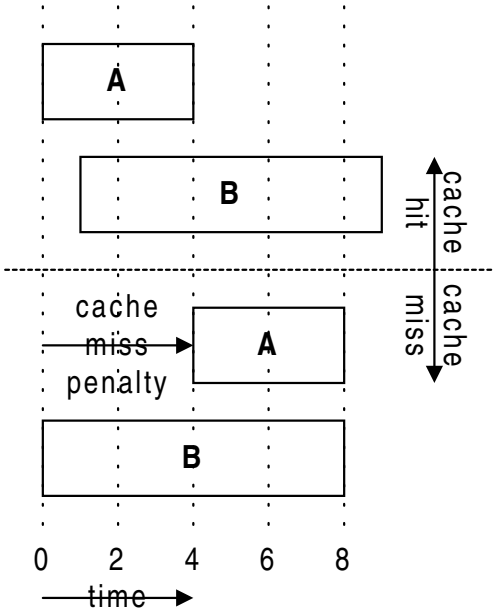


**FIGURE 6.** Example of a timing anomaly.

hit results in a longer overall execution time than a cache miss, i.e. the opposite of what is expected. Figure 6 shows two instructions that are dispatched simultaneously, where instruction (A), is due to execute first, has a shorter execution time than the other instruction (B); both instructions use different execute units (e.g. FPU and IU) that can be used concurrently. To simplify the example, the execution time of each instruction is not affected by a cache miss, only by the time at which it is executed. If A suffers a cache miss, then the pipeline mechanism identifies the stall and re-schedules the instructions, which results in B being executed before it, i.e. A and B are executed in an order other than originally intended. In this case, both instructions complete earlier than if instruction A had not suffered a cache miss and they had been executed in their original order. Therefore the WCET analysis model needs to take into account potential anomalies. Otherwise, all possible paths would have to be analysed rather than those that could be expected to be the worst case. Again for realistic software sizes, this would be intractable. Hence a balance must be struck between the pessimism of the results, the computational complexity of obtaining results and the difficulty in validating the model.

There is already a great deal of analysis available including [4, 16], and Engblom provides a useful survey of these analyses in [5]. These approaches can be tailored for our use by taking advantage of the fact that software is written in a particularly disciplined way for the safety-critical domain. For example, the software is written to make control flow analysis easier to perform, and/or conforms to a restricted subset (e.g. SPARK Ada [17]). Such assumptions allow some simplification in the way that the analysis is performed.

Some of the fundamental issues for WCET analysis of a superscalar processor are summarized here.

- *Re-targetable analysis*. Producing analysis in a generic fashion means that it can be instantiated for a particular platform with a minimal amount of rework. To achieve this aim a WCET analysis language has been defined at York which allows the WCET analysis software to parse platform-specific information and hence to customize the analysis. For example, the number of cache lines can be defined, as can the number of sets the cache is organized into, and so on. The framework is presented in Figure 7. In this figure it can be seen that for most parts of the analysis, a platform description is obtained from a file written using the defined WCET language. This information is used to instantiate a generic form of analysis. Amongst other features the WCET language allows a generic parser to understand the stream of instructions being read and identify the individual ones. The parts of the analysis not produced in this form even though they still use the platform description are the automatic data-cache analysis and program-path analysis; these are the subject of future work.
- *Validity of the analysis*. Producing a valid WCET is reliant on correct information from the manufacturer. It can be assumed that a COTS processor has not been produced or documented to the standard required for the certification of the system. Assuming that the processor cannot be re-engineered, a validation strategy is needed. Some work has been performed on this subject [18].
- *Anomalies within the analysis*. The analysis has to allow for the effect of cache misses and branch predictions without leading to anomalies which understate the worst case.
- *Data cache analysis*. Data and instruction-cache analysis mechanisms are similar. The key difference being that instructions always have a static location in memory whereas data does not. Examples of data accesses that do not have a static location include pointers and stack variables. One solution is to exclude difficult to predict accesses from the cache [19], this is possible on most processors. Data cache analysis is the subject of future work.

A suitable strategy for WCET analysis is to produce a simplified model of a modern COTS processor that can easily be tailored to a specific processor. This accepts some pessimism whilst guaranteeing that the results are safe, e.g. the predicted WCET is greater than the actual WCET. The difference between a COTS and bespoke processor is that the bespoke processor can be designed to aid predictability, hence tighter estimates can be achieved, i.e. lower pessimism, and the model would be easier to validate. As already discussed a pipeline without dynamic scheduling of instructions simplifies the gathering of evidence. Additional features that help are instructions having constant processing times (e.g. not like the ARM 7 processor [20] where floating-point multiply instructions have a processing time dependent on the size of the result)

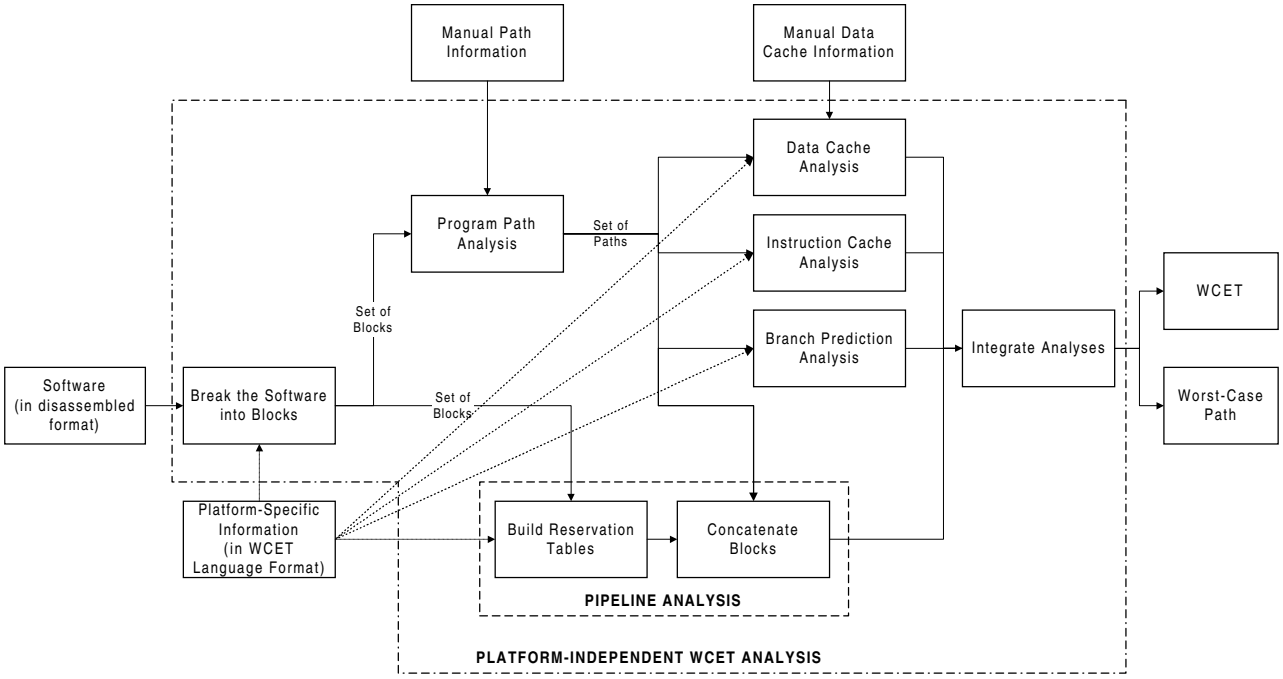


**FIGURE 7.** Re-targetable framework for WCET analysis.

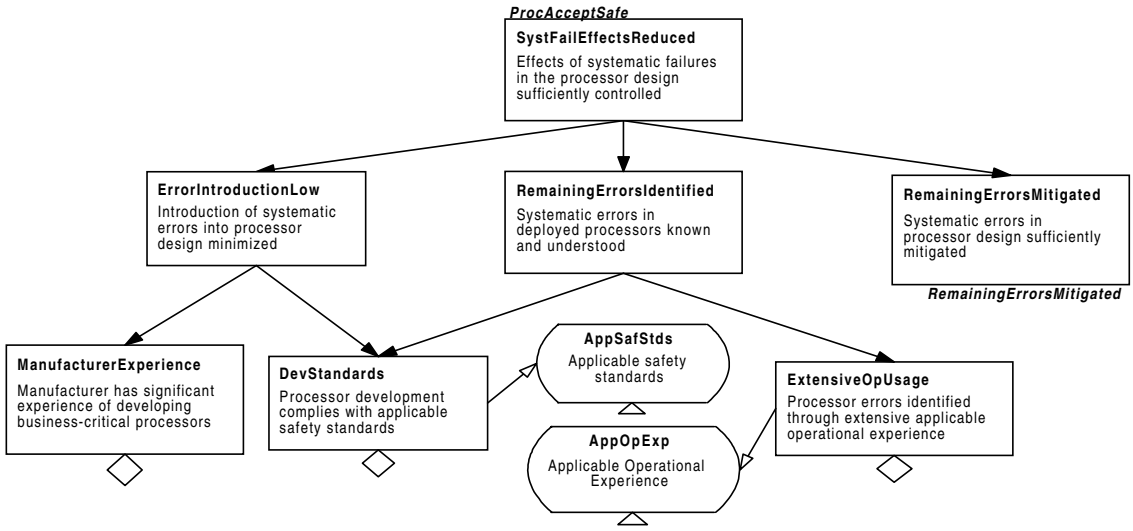


**FIGURE 8.** Processor systematic failure argument.

and a few exceptions to the normal processing times (e.g. again not like the ARM 7 where instructions with the register that holds the stack pointer as a parameter take longer than normal).

### 4.5. Processor design errors

Systematic errors, sometimes referred to as 'design errors' are those errors introduced into an artefact through mistakes in the requirements, specification or implementation of a product. As the complexity of an artefact increases, the likelihood of introducing systematic errors also increases. We should therefore expect that systematic errors may be present in modern processors, and reason about them accordingly. This applies to both COTS and bespoke processors. Although it could assume the incidence of design errors in a bespoke processor to be less than for a COTS processor, e.g. if it has a simpler design, errors may still exist as the bespoke manufacturer is unlikely to have the wealth of design experience of a COTS manufacturer. This is discussed below. Figure 8 presents the argument of processor safety with respect to systematic errors.

It should be noted that the top claim presented in Figure 8 ('SystFailEffectsReduced') is *not* that there are no systematic errors present (perhaps as some processor

manufacturers would wish us to believe). Rather, the argument is that any errors that *are* present are sufficiently managed such that they have an insignificant effect on safe operation.

For the older generation of processors it was reasonable to assume that the majority of design flaws had been found and documented, e.g. for widely used devices such as Motorola's 68020. Even if these errors had not been fixed, design engineers could tailor their system to avoid them. However, a modern processor has a significantly larger and more complex design. The Pentium III in 1999 had 28 million transistors versus 68 thousand transistors on the 68020 in 1979—note that the number of transistors is not a direct measure of design complexity because a significant number of transistors are taken by regular structures such as memory. This increase makes it less likely that all the design errors have been found and documented, although certain problems have been well publicized (e.g. the problems with the Pentium's arithmetic unit [21]—it should be noted that the processor was widely deployed before the problems were discovered). It could be hoped that all, or at least most, design errors were known with processors of the complexity of the 68020 due to their widespread use. However with significantly more complex processors, it is much less likely that all design errors are known.

To further complicate the matter, Defence Standard 00-54 notes that processors may have several different variants in a year, each of which may contain subtly different design features and flaws. It can be confidently assumed that a COTS product would not comply with the Defence Standard's requirements. However, the argument for using a COTS processor can be given weight by using a supplier with a reputation for well-designed products. In this case the chosen processor would have to adhere to the supplier's usual standards of quality. In any case the certification argument has to assume the existence of known and unknown errors.

It should be noted that a hardware manufacturer, such as Intel, has a great deal invested in the correctness of their hardware, including the value of their reputation not to mention the cost of mistakes, particularly when they lead to product recalls. In this context, the correctness of the processor can be classified as critical (in business terms) due to the potential implications of failure for the company's profitability. Since the Pentium floating-point arithmetic error was uncovered, the importance of hardware correctness has received a higher profile. Significant effort has been applied to formally proving parts of the design [22].

To gain confidence in a processor it is desirable to have some reassurances that the introduction of systematic errors within processor development was reduced as low as reasonably practicable (as claimed in 'ErrorIntroductionLow'). There are two key arguments that can be made in support of such a claim. The first of these appeals to the domain experience of the organization developing the processor. Unpublished studies have shown that previous experience is one of the most significant factors in systematic error introduction. As stated earlier, the reputation of the manufacturer can give weight to the certification argument. The second argument appeals to a 'good' process (e.g. as defined by safety-critical standards). For example, claims of low residual errors can be made by appeal to an extensive, systematic and thorough design review process.

The arguments of manufacturer experience and adherence to development standards represent areas where there may be divergence between bespoke and commercial processors. The experience argument is probably strongest for a COTS processor from a major semiconductor designer. However, although rigorous standards may have been applied during COTS processor development it is unlikely that safety-critical standards (such as UK Defence Standard 00-54) will have been used. Therefore, the development standards argument may be weaker.

The development standards argument has the potential to be much stronger with a bespoke safety-critical processor development. However, unless the bespoke processor is being developed by an experienced semiconductor manufacturer, this will be at the expense of a weaker manufacturer argument.

The second leg of the argument presented in Figure 8 puts forward the claim ('RemainingErrorsIdentified') that those errors remaining in the finished processor design (e.g. those discovered in operational usage) are known and understood. This argument rests upon two sub-goals. Firstly, it can be argued (as in goal 'DevStandards') that remaining systematic errors are uncovered and understood as a result of a systematic development process (e.g. one which involved detailed hazards and operability studies [23] over the processor design). A second, and perhaps more compelling argument, is that errors are uncovered and understood through extensive usage.

As with the error-introduction argument, this is again an area of divergence between bespoke and COTS processors. COTS processors have the significant advantage of large amounts of operational experience from field usage; the only caveat here being that the operational experience must be argued to be applicable and appropriate for the application context of any new system development, as highlighted by the context reference 'AppOpExp'. For example, it is obviously inappropriate to directly read across data regarding faults uncovered at room temperature to a situation where the processor may be placed in an extreme thermal environment.

The disadvantage of COTS processors can be the unavailability of evidence (from the processor vendor) to support the development-standards claim. The availability of such information is an obvious advantage in a bespoke development. However, bespoke developments lack the wealth of field data accompanying a COTS solution.

The final leg of the argument presented in Figure 8 claims (in goal 'RemainingErrorsMitigated') that the presence of residual design errors in the processor is sufficiently mitigated. This argument is developed in the following section.

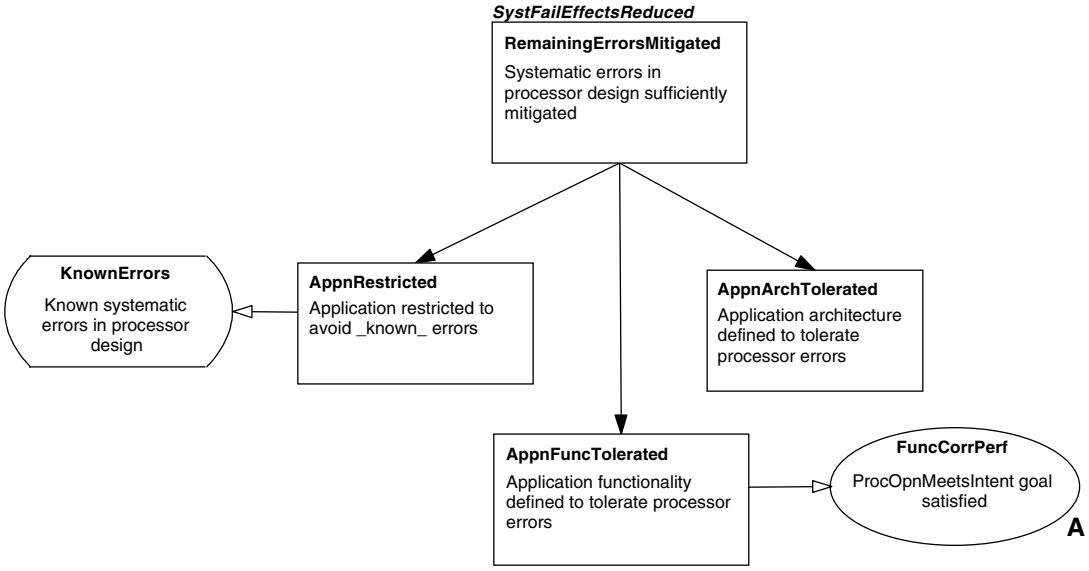


**FIGURE 9.** Systematic error mitigation argument.

### 4.6. Mitigation of processor design errors

Figure 9 presents the argument that design errors remaining in the processor are sufficiently mitigated. This argument mainly applies to the use of COTS processors. The first leg of this argument (shown by goal 'AppnRestricted') claims that the application is restricted to avoid activation of known design flaws (e.g. avoiding the trigger provided by a specific sequence of machine instructions). This is essentially a 'work-around' argument, depending heavily on the principle that errors are known about in advance of application development.

However, this application restriction needs to be verified by object-code inspection. As verification of object code against source code is often currently undertaken by visual inspection, this becomes a more complex process, requiring specific knowledge of the processor hardware. In order to ease inspection it might be desirable to use a programming language sub-set, designed to prevent the use of a particular feature if it is deemed to be unsafe, and/or difficult to analyse. Other alternatives would be to adapt the compiler to avoid the feature or modify the object code after compilation.

Both the second and third legs of the argument in Figure 9 are concerned with *toleration* of faults. Firstly, goal 'AppnFuncTolerated' is concerned with handling known faults in software at the application level, e.g. through exception handling. Secondly, goal 'AppnArchTolerated' is concerned with handling known faults through the system architecture, e.g. employing redundant and diverse channels. Standards such as DO-178B [9] consider the use of multiple dissimilar processors with dissimilar software. It notes that some of the required hardware evidence may be replaced with evidence that equivalent output and performance is achieved by both systems. The requirement for evidence of hardware failures can also be reduced. For example, if the processors are dissimilar in design it can be argued that the likelihood of simultaneous failure is lowered, reducing the amount of failure rate evidence needed. For example the Boeing 777 Primary Flight Control System [24] uses different types of processors in each of three computing channels, with cross-lane monitoring between each channel. Using a bespoke processor in a dissimilar architecture would be one way of gaining operational experience whilst reducing risk.

The advantage of this diverse processor approach to fault mitigation is that it provides a general mechanism which, although perhaps targeted at specific known errors, can provide identification and mitigation of a wide range of control, data and timing errors.

However, it should be noted that it is impossible to support claims 'AppnFuncTolerated' and 'ApplnArchTolerated' in a technology independent manner—i.e. they both require detailed knowledge of the specific implementation and operation of the processor.

### 4.7. Processor reliability

In addition to arguments concerning systematic errors in the processor design, it is also necessary to put forward and support claims of an acceptably low occurrence of random failures, i.e. failures attributable to environmental factors (such as electromagnetic interference) and material defects. This argument is presented in Figure 10.

The role of the expected operational environment of the processor (as referred to by context 'ExpOpEnv') needs to be highlighted specifically in connection with the argument. The arguments of processor reliability need to be presented in the context of the anticipated operating conditions.

Demonstration that the processor has an acceptably low failure rate for the processor can be achieved via

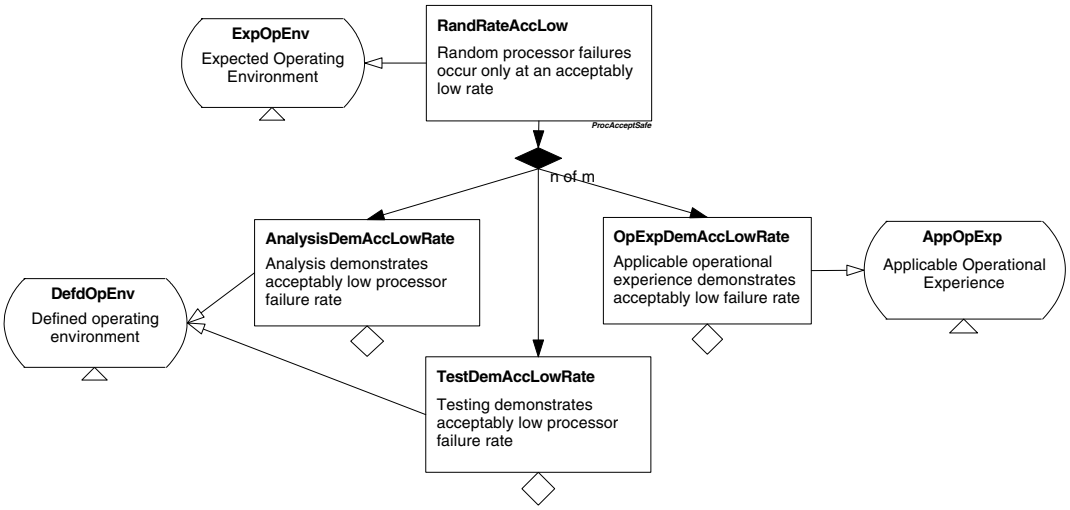


**FIGURE 10.** Processor reliability argument.

a number of different routes (as indicated in Figure 10 depicted by the choice of '*n* of *m*' sub-goals to support the parent 'RandRateAccLow' claim). The first route is based upon analysis of the processor (e.g. fault tree analysis performed at the level of the processor hardware). Such an analysis would need to make assumptions about the operating environment (as depicted by the connected Context 'DefdOpEnv'). It is obviously desirable that the operating environment as defined for the purposes of the analysis is as close as possible to the actual operating environment (as referred to by the context reference 'ExpOpEnv'). Analysis is a viable option where there is free access to detailed information regarding the processor design and manufacture. It is therefore more likely to be viable for a bespoke processor than for a commercially acquired processor.

The second route is based upon reliability testing of the processor, which is especially an issue for COTS processors where manufacturers rarely produce military specification [25] components these days, and leads to other forms of assurance being sought (e.g. including so-called 'shake-and-bake' tests where the resilience of the processor to heat and vibration is tested). Such testing will largely be 'black box' in nature, and is therefore a viable option for both bespoke and commercially-acquired processors. The relevance of the testing environment with respect to the actual operational environment remains a key factor in the confidence that can be invested in this form of argument.

The third route to supporting the reliability claims is based upon the operational experience gained from field data. Commercially available processors may have a possible advantage over bespoke processors in the availability of such data. However, as previously discussed in Section 4.5, the admissibility of this data within the certification case rests upon whether the data can be argued to be relevant and applicable to the intended target context of the processor.

## 5. CONCLUSIONS

Complete certification arguments for processors have to allow for systematic and random failures. In particular we have to acknowledge the presence of residual design faults in the processor and provide appropriate mitigation. Within the structure of the argument it is possible to see that there are advantages and disadvantages on both sides of the choice of COTS versus bespoke processors. The selection of COTS processors with particular features can significantly simplify the task of gathering evidence. Alternatively a bespoke processor can be designed in conjunction with building the safety argument so that the overall cost is reduced.

Principal advantages of COTS processor are that the widespread use stems from the extensive operational experience and thoroughly exercised support tools such as compiler. However, the admissibility of this operational experience when not directly related to safety-critical applications is a point of debate. Disadvantages of COTS processors lie in the black-box nature of the design and development process, and the fact the processors are not necessarily designed for predictability and their design goals may not be appropriate (e.g. good average-case performance).

Principal advantages of the bespoke processors lie in the ability to define a processor only using features that can easily be analysed allowing the designer to trade-off processor worst-case performance versus predictability, and that the process can be treated as 'white-box'. The principal disadvantages are that its use is not widespread resulting in a lack of operational experience, lack of freely available tools and that other users may not necessarily accept the design.

## ACKNOWLEDGEMENT

Iain Bate and Philippa Conmy are part of the BAE SYSTEMS funded DCSC group at the University of York.

## REFERENCES


[1] Chapman, R. (1995) *Static Timing Analysis and Program Proof*. DPhil Thesis, Department of Computer Science, University of York, YCST-95-05.

[2] Bate, I. (1999) *Scheduling and Timing Analysis for Safety-Critical Systems*. DPhil Thesis, Department of Computer Science, University of York, YCST-99-04.

[3] Motorola (1997) *MPC603e & EC603e—RISC Microprocessor User's Manual*. MPC603EUM/AD.

[4] Mueller, F. (1994) *Static Cache Simulations and its Applications*. PhD Thesis, Department of Computer Science, Florida State.

[5] Engblom, J. (1997) *Worst Case Execution Time Analysis for Optimized Code*. MSc Thesis, Department of Computer Science, Uppsala University, DoCS 97/94.

[6] United Kingdom Ministry of Defence (1999) *Requirements for Safety Related Electronic Hardware in Defence Equipment*. Defence Standard 00-54.

[7] United Kingdom Ministry of Defence (1997) *Requirements for Safety Related Software in Defence Equipment*. Defence Standard 00-55.

[8] United Kingdom Ministry of Defence (1996) *Safety Management Requirements for Defence Systems*. Defence Standard 00-56.

[9] RTCA/EUROCAE (1992) *Software Considerations in Airborne Systems and Equipment Certification*. DO-178B/ED-12B.

[10] RTCA/EUROCAE (2000) *Design Assurance Guideline for Airborne Electronic Hardware*. DO-254.

[11] International Electrotechnical Commission (2000) *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*. IEC 61508.

[12] Kelly, T. P. (1999) *Arguing Safety—A Systematic Approach to Safety Case Management*. DPhil Thesis, Department of Computer Science, University of York, YCST 99-05.

[13] IEEE (1985) *IEEE Standard for Binary Floating-Point Arithmetic*. IEEE 754.

[14] IEEE (1987) *IEEE Standard for Radix-Independent Floating-Point Arithmetic*. IEEE 854.

[15] Pumfrey, D. J. and McDermid, J. A. (2000) Assessing the safety of integrity level partitioning in software. In *Proc. 8th Safety-Critical Systems Symp.*, Southampton, UK, pp. 134–152.

[16] Lundqvist, T. and Stenstrom, P. (1999) Timing anomalies in dynamically scheduled microprocessors. In *Proc. 20th IEEE Real-Time Systems Symp. (RTSS'99)*, pp. 12–21.

[17] Currie, I. E. (1995) *TDF Specification*, Version 4, DRA/CIS(SE2)/CR/94/36/4.0. Defence Evaluation Research Agency, UK.

[18] Utamaphehai, N., Blanton, R. D. and Shen, J. P. (2000) A buffer-oriented methodology for microarchitecture validation. *J. Elect. Testing (Jetta)*, **16**, 49–65.

[19] Lundqvist, T. and Stenstrom, P. (1999) A method to improve the estimated worst-case performance of data caching. In *Proc. 6th Int. Conf. on Real-Time Computing Systems and Applications (RTCSA'99)*, Hong Kong, pp. 255–262.

[20] ARM (1995) *ARM 7 TDMI Data Sheet*. ARM/DDI/0029E.

[21] Intel (1994) *Statistical Analysis of Floating Point Flaw*. Intel White Paper.

[22] O'Leary, J., Zhao, X., Gerth, R. and Seger, C.-J. H. (1999) Formally verifying IEEE compliance of floating point hardware. *Intel Technol. J.*, 1–14.

[23] United Kingdom Ministry Of Defence (2000) *HAZOP Studies on Systems Containing Programmable Electronics*. Defence Standard 00-58, Issue 2.

[24] Yeh, Y. C. (1996) Triple-triple redundant 777 primary flight computer. In *Proc. 1996 IEEE Aerospace Applications Conference*, Aspen, CO, pp. 293–307.

[25] US Department of Defense (1986) *Reliability Prediction of Electronic Equipment*. MIL-HDBK 217E.