HARDWARE-SOFTWARE TRADEOFFS IN RELIABLE SOFTWARE DEVELOPMENT

Rein Turn
TRW Defense and Space Systems Group
Redondo Beach, CA

Abstract

Advances in computer technology are offering opportunities for innovative hardware-software tradeoffs for reducing system development costs and improving software reliability and performance. However, some project managers are not yet taking full advantage of these opportunities. They tend to proceed immediately to hardware specification and acquisition, and only later turn to software planning and design. The likely outcome is cost overruns, inefficient software structures, inadequate performance, and poor software reliability. An approach more apt to produce satisfactory results is to generate hardware and software specifications together as a synergistic whole, and then acquire the two as balanced and compatible parts of the system's architecture. This paper addresses hardware-software tradeoff opportunities, computer tools for tradeoff analyses, and a management approach.

INTRODUCTION

Many organizations have experienced serious difficulties in the acquisition of complex computer-based systems, such as those embedded in other large systems and/or operated in real time. Especially troublesome has been the development of their software components. Problems have included (1) large cost overruns and schedule slippages in the design, (2) inability of the delivered system to perform as expected in the operational environment, (3) lack of software reliability, and (4) extraordinarily high support costs in the operational phase [1-3]. Fred Brooks, in The Mythical Man-Month, likens large software development efforts to the struggle of prehistoric beast trying to escape from tar pits -- the accumulation of simultaneous and interacting factors brings progressively slower motion [4].

Among the reasons for the stickiness of the computer system development problem, especially in large military systems that include embedded computers as subsystems, has been the inability of project managers to appreciate the subtleties of integrating computers with other systems, and their failure to appreciate the need for a total-system approach in acquiring computer subsystems. Instead, they have proceeded to develop or procure the computer hardware first and then (sometimes only after hardware has been delivered) the software, hoping to integrate them successfully later with each other and with the rest of the system. Some of the systems developed under this philosophy are still stuck in the development tar pit.

Underlying such unsuccessful development efforts has been a set of management beliefs -- a set of myths -- which do not reflect the true nature of computer systems, computer software, or their development processes. For example, it is not true in general that: (1) hardware and software can be acquired separately and independently and successfully integrated later; (2) software acquisition can be treated as a bulk purchasing process similar to the off-the-shelf procurement of standard hardware; (3) hardware inadequacies can be easily rectified by simple software changes; (4) software, once accepted operationally, hardly ever again needs to be changed; and (5) it is a simple task to state valid and complete requirements for software and then to implement them in computer code.

The described management attitudes and approaches have tended to all but preclude any consideration of hardware-software tradeoffs in any phase of the computer system life cycle process, and have contributed considerably to the difficulty of developing reliable software and reducing software development costs. Under the "hardware first" philosophy, hardware has been specified and acquired without fully appreciating the software complexity, and its needs for processing speed or memory space. Figure 1 illustrates the penalty in programming cost of inadequate hardware resources [3].

![Figure 1. Software Cost Impact of Inadequate Hardware Resources](https://example.com/figure1.png)
into a total-system framework, and software development activities on an engineering-like basis [7]. Thus the user organizations have increased the support of software engineering research (e.g., [8]) and software developers have created policies, procedures, and automated design aids to increase the quality and reliability of their products and productivity of their programmers (e.g., [9-12]). In particular, deliberate efforts are now being made by managers of computer system development projects to assure adequate performance of hardware-software tradeoffs in the specification and design of both of these components of computer systems. This paper discusses a rationale, criteria, techniques, expected benefits, and management procedures for performing such tradeoffs.

DEVELOPMENT OF RELIABLE SOFTWARE

Several definitions of software reliability are in use. One of these states that software reliability is the probability that a software system will perform its intended functions for a specified number of input cases under stated input conditions [13]. The probabilistic aspect of this definition is due to the uncertainty in the selection of input cases when the systems is in operation. In the testing phase, certain representative input cases are selected and used to test the software, since exhaustive testing of all possible input cases is usually unfeasible. Thus, there may exist design or implementation errors in software that manifests themselves only when untested input cases occur in actual operation of the system. Errors in software may also be induced by transient or otherwise undetected hardware errors and techniques must be incorporated to detect, correct, or mask these errors. However, the primary concern in achieving software reliability focuses on (1) avoiding introduction of design faults in the course of the software development life cycle (see Figure 2) and (2) using various techniques to increase fault tolerance of operational software.

There is now ample evidence that the early phases of software development life cycle are most critical in determining the reliability of a software system [1,3]. For example, results obtained at TRW show that of the 224 types of errors discovered during testing, 64 percent were design errors and only 36 percent were coding errors [14]. It is also clear that the later in the development life cycle an error is found, the more costly its correction will be: errors in formulating software requirements are likely to cost ten times as much to correct when they are detected during acceptance testing as when detected in the design phase [6].

To avoid design faults and implementation errors in software development, systematic design procedures supported by automated software-tools have been developed by most of the software vendor organizations. One such set of procedures in use at TRW [15] is based on the following principles:

- Maintenance of the traceability of software requirements to software design to the final code
- Top-down incremental software development
- Stringent application of design and coding standards
- Use of automated aids for requirements and design validation, and for code testing
- Continuous application of quality control techniques

These principles are implemented at TRW in SRM -- Software Requirements Engineering Methodology (SREM). Its key components are (1) RSL -- a formal language for statement of requirements (2) R-Nets -- Requirements Networks for describing the logic of requirements, and (3) REVS -- a Requirements Validation System of automated tools for checking the requirements for completeness and consistency, maintaining traceability, and generating simulations to validate the requirements' correctness. Figure 3 illustrates the SRM structure. Validated R-Nets form a firm baseline for the preliminary design of the software system. The described techniques are one aspect for error-avoidance in the development life cycle. Another aspect is the systematic consideration of hardware-software tradeoffs.

Figure 2. Reliability-Oriented Software Development Life Cycle
(3) designing both the software and new hardware for an application.

Contemporary interest in hardware-software tradeoffs focuses on new hardware architectures to aid the programming problem: direct execution of higher-order languages, replacement of data base management software with data base machines [17], communications processors for distributed data processing systems, and implementation of parts of the operating system and subroutines in read-only memories. Also a third dimension has been added to the tradeoffs — firmware. That is, it is now both technologically feasible and economically attractive to implement certain computing functions in microprograms. For example, if a computer's instruction set is implemented as a microprogram in an alterable read-only memory, a user can use this capability to design new instructions or modify existing ones to enhance execution of his programs. Furthermore, such a computer can be made to emulate the characteristics of other computers by microprogramming into the host computer the instruction sets of the target computers. This capability is important in performing experiments with hardware-software-firmware tradeoffs prior to committing to hardware designs, or prior to procuring off-the-shelf equipment [18].

Tradeoff Goals

Given the recent advances in hardware technology which have reduced hardware costs to a relatively small fraction of the software system development cost, it is not surprising that most of the tradeoff choices are from software to firmware or hardware (there is also an increasing trend to move functions from hardware to firmware to gain flexibility). Among the principal tradeoff goals in designing new hardware as well as selecting existing hardware are to:

1. Achieve improved performance (e.g., hardware or microprogram critical functions which require too much time in software, such as a fast Fourier transform processor).
2. Reduce software complexity (e.g., provide special instructions needed for data base management).
3. Reduce programming effort (e.g., provide a large instruction repertoire and variety of data representations and assume that their execution is sufficiently fast; avoid special needs to scale variables or to overlay memory space).
4. Improve reliability and fault-tolerance (e.g., provide built-in error diagnostics, recovery features, or error detection/correction codes).
5. To improve integrity and security (e.g., implement hardware features such as processors mode indicators, security tags, or memory protection bounds registers).
6. Reduce the overall system life cycle cost (e.g., provide growth capability, improved performance measurement, and diagnostic aids).

Figure 3. Software Requirements Engineering Methodology (SREM) Steps and Products

Historically, hardware-software tradeoffs of the first category were concerned with the instruction set composition (e.g., whether or not to include the square root instruction, or whether or not to provide floating-point, double-precision, or decimal arithmetic). The deciding factors were the market demand for specialized instructions versus the resulting hardware complexity, physical size, power consumption, and cost. For example, even in the early 1970s some airborne or space-borne computers where physical size is critical, lacked floating-point arithmetic, thus greatly complicating their programming. The advent of small- and medium-scale integration in hardware manufacture, and then large-scale integration, greatly reduced the concerns with hardware characteristics and costs, and permitted inclusion in third-generation computers of a variety of hardware enhancements to improve performance, reliability, and programming [16]. Examples are specialized concurrently operating processing units (such as in the CDC 7600), variable-length-field and byte-oriented processing, extensive look-ahead features in arithmetic and control units, virtual memory addressing capability for automated management of peripheral units, pipeline processing architecture, and associative memory capability.
Improvement of a computer system's reliability and fault-tolerance can be obtained by using hardware techniques, software techniques, or both [19]. A tradeoff analysis is required to determine a combination that best achieves the reliability goals and satisfies the constraints. The following are among hardware techniques for improving reliability:

- Use of redundancy on circuit, subsystem or system level, such as triple modular redundancy (TMR), where it is assumed that the results/values produced by a majority of replicated units represent the correct results.
- Use of error detecting/correcting codes in data transfer paths, and automated computation of check-sums in data bases.
- Use of standby spare units to be switched into the system when any of the active units fails.
- Extra status bits and registers to aid in software diagnosis.

Included in reliability and fault-tolerance techniques usually implemented in software are the following:

- Redundant storage of critical programs and data, and redundant computation of critical functions (possibly using independently coded programs).
- Frequent recording of snapshots of the program and system states to provide a roll-back capability. When errors are diagnosed in the computation, system is restored to the stored state and computation repeated. As computation proceeds, roll-back points are advanced. Hardware-implemented error indicators and software-based diagnostic routines can be used to detect faults.
- Recovery blocks [20] in software: the use of standby software modules which are switched into action when the primary modules are found to contain faults (i.e., produce erroneous results as determined by diagnostic routines). The use of standby software modules is becoming an important technique for improving software fault-tolerance [20,21].

Tradeoffs in providing fault-tolerant systems are mostly in the direction of implementing in hardware techniques presently common in software, such as the recovery module approach described above. Any such tradeoff analysis must take into account the reliability goals of the system, performance requirements, costs of implementing various combinations of hardware or software techniques from the life-cycle point of view, and impacts on other system characteristics. In addition, likely to exist are other, nontechnical constraints that influence tradeoff decisions. For example, while system designers may find an optimal combination of hardware and software techniques that satisfies the tradeoff criteria, it is likely that the hardware system involved requires a custom-made design. This may conflict with institutional policies of the organization, such as standardization of hardware equipment. Consequently, the design would have to be modified to make it compatible with those hardware architectures/systems that are available on a "list of approved equipments". For example, standardization of computer architectures is now a joint Army/Navy project [22] and triservice-wide standardization of computer systems for tactical applications is an important item on the Department of Defense agenda for ADP system acquisition policy, similar to standardization of higher order programming languages [8].

A framework for evaluating tradeoff options in the design of reliable computer systems has been suggested by Chandy et al. [19] who have developed efficiency indices for hardware- and software-implemented reliability techniques. These indices can be used to compare proposed implementation approaches and provide one set of inputs for tradeoff decisions. In general, however, tradeoff analysis in complex systems are themselves quite complex and require automated aids.

Tools for Tradeoff Analysis

An hardware-software tradeoff analysis for selecting an overall architecture that satisfies the system performance or reliability requirements is usually an iterative process which proceeds as follows:

1. Functional requirements of the application to be implemented are formulated and validated, and detailed specifications of the subfunction modules and their interactions are produced (e.g., using requirements engineering techniques such as SREM).
2. Classes of computer equipment are identified which can satisfy timing, throughput, loading, and storage capacity requirements. Alternatively, when new computer hardware designs are considered, technically feasible ranges of these characteristic are postulated as a baseline, along with instruction representations, memory hierarchy structures, capacities and access times, and so forth.
3. Functional simulation models of the relevant external environments and systems, and their dynamics, and users demands, are produced.
4. Trial assignments are made of subfunction modules to be executed in software or implemented hardware. If multiple-processor or distributed hardware architectures are considered, the choice of different processors for executing software modules or incorporating hardware implemented modules are considered.
5. Trial assignments are evaluated using performance evaluation models which may be built into automated tools or applied manually.

Tradeoff questions impacting software reliability, in addition to special techniques for this purpose, include the speed and memory characteristics of the selected hardware. It is well established, especially in real-time systems, that as the available response time and memory capacity are being exhausted in the software design process, programmers have to resort to clever techniques that may save time and reduce storage space needed, but at the expense of increasing software complexity and greater expenditures of design and testing time. In aerospace systems with limited hardware resources this has been one of the major causes of poor software reliability, cost overruns, and schedule slippages [14].

A diagnostic emulation facility in use at TRW [18] provides an example of a set of automated tools for analyzing hardware-software tradeoffs in system design, and for improving software reliability. Figure 4 illustrates its structure. In such a facility a microprogrammable host computer is used to emulate the hardware architecture under study (e.g., its instruction set as well as any subfunctions that are candidates for hardware implementation). The software part of the application is coded in terms of the target architecture instructions.

![Figure 4. Structure of Diagnostic Emulation](image)

Automated capabilities are provided in this facility to observe and collect data, and check for errors while the target software is executing on the emulated hardware in a simulated system environment. A particular requirement here is that emulations be relatively easy to produce in microcode of the host machine, and that such microcode be essentially error-free itself. The TRW facility has a nonprocedural, higher-order language, SMITE, for expressing the target computer characteristics and a compiler to produce the corresponding microcode for the emulator-computer. SMITE and its compiler are one of the significant advantages of this facility over other emulation facilities where microcode must be written directly by the analysts.

Another advantage of emulation is in analysis speed and cost. By emulating a hardware architecture on a microprogrammable minicomputer, such as Varian 73 or Nanodata QM-1, large improvements can be obtained over performing similar evaluations entirely as interpreter computer simulations (ICs) even if the latter are performed on much larger computers. For example, using Varian 73 as the emulator versus an IC simulation on CDC 6600, in evaluating a portion of UNIVAC's Weapon System Controller (WSC) software, resulted in the following advantages in favor of emulation [18]. A portion of the software which requires 1.78 microseconds on WSC required 16.2 microseconds on Varian 73, but 66.1 microseconds on CDC 6600 with ICs, thus the slow-down factor was in favor of emulation, 1 to 4. Furthermore, given the 1.1 to 1 computer time cost advantage of Varian 73, an overall advantage of 45 to 1 was obtained overrunning the ICs on the CDC 6600. Use of a QM-1 computer as an emulator offers similar advantages.

In the future, the same hardware technology advances (e.g., low cost logic and memory) that make attractive the migration of software to firmware or hardware can also improve automated tools for tradeoff analysis. For example, specialized test beds at microprogrammable minicomputers which have built-in performance evaluation and bookkeeping capabilities can be constructed for emulating system architectures.

**Payoffs in Performing Tradeoffs**

Hardware-software tradeoffs have found a firm place in the development of reliable software. They generate important choices for reducing software complexity and its development cost and, thus, in addition to yielding other benefits, improve expectations for improving software reliability. For example:

1. Any need for more capable hardware becomes apparent early in the design process, avoiding problems later on. It remains a myth that it is easy to compensate for hardware deficiencies leads to reduced reliability and higher development cost.

2. Implementation of certain application or operating system software functions in hardware reduces software complexity and improves its performance (e.g., automated memory management, control of input-output operations, and access control and other security functions).

3. Implementation of software techniques for fault tolerance wholly or in part in hardware
A study of the queue management in a general purpose computer system, with a particular emphasis on hardware size, weight, and power consumption penalties which otherwise may make the use of these techniques economically or operationally unattractive (e.g., fault detection and diagnosis).

4. Tradeoffs in designing automated analysis tools themselves can make their use more economical, speed the analysis process, and produce more useful results (e.g., simulation vs. interpretative simulation).

Most of the payoffs obtained can be evaluated in quantitative basis through the use of performance evaluation tools and techniques, others can be justified on qualitative arguments. Below are some examples:

1. The need in the 1960s for executing large programs in computers with relatively small main memories led to the development of program overlay techniques which grew increasingly more complex and contributed as much as 25-40 percent of the programming costs [17]. Development of hardware-implemented memory management capability -- the virtual memory concept -- eliminated these programming costs and provided other benefits, at the expense of small increases in hardware complexity and peripheral memory access time.

2. In computer security, implementation in software of encryption/decryption techniques to protect files can increase the file access time by a factor of four in a CDC 6400 computer using assembly language routines and employing exclusive-or addition of a pseudorandom keystream to the data [23]. Implementation of this function in hardware has a much smaller effect on the access time.

3. Database machines with specialized "intelligent" disks can improve search time over software-implemented data base management systems as much 40 times [24].

Tradeoffs can also move in the other direction -- from hardware to software. For example, the hardware size, weight, and power consumption penalties when providing triple modular redundancy may be unacceptable in some aerospace applications. Implementing similar techniques in software can provide an adequate level of fault-tolerance at lesser hardware complexity. Another example illustrates such "reverse migration" of functions: A study of the queue management in a general purpose buffer memory to be implemented in a single LSI chip with variable word width and variable queue length capabilities [25], demonstrated that the optimal allocation of functions would be to leave the provisions for handling these two capabilities in software. While the queue management software was increased 30% over the case when these functions were implemented in hardware, the hardware requirements were reduced by over 50%. The latter was considered a greater payoff in this particular case.

CONCLUDING REMARKS

Managers of large computer-based systems are only now beginning to realize the special nature of software as compared to hardware, and the need develop the two in a closely harmonized manner. However, it will take more time until it is wholly understood by managers that economizing with hardware may not be a proper policy at a time when software development costs are soaring relative to hardware acquisition costs. Thus, there is and should be every incentive for hardware-software tradeoffs.

Opportunities for tradeoffs are many and will be much more plentiful in the future -- with the possible exception of pocket calculators or table-top computers, the real impact of the LSI and VLSI hardware manufacturing capability is only now beginning to be perceived [26, 27]. In the future it will not be considered extravagant to be "lavish with hardware" in attempting to enhance software reliability and its development processes. For example, it would not be farfetched to eliminate the programs and associated overhead in a time-sharing system for swapping programs in and out of the main memory by simply replicating the main memory and switch from one unit to another to service programs resident in these units.

This paper has discussed some of the goals, means, and automated tools for performing hardware-software tradeoff analyses, and has illustrated functional areas in computer systems where tradeoff benefits are readily attainable. In general, however, each tradeoff situation requires its own analysis and must be considered in its own special context. Thus, one should not be surprised that the same tradeoff question in different systems under different systems design goals and/or external constraints can produce conflicting decisions.

REFERENCES


24. Hsiao, D., Private Communication, Ohio State University, Columbus, Ohio, October 1977.

