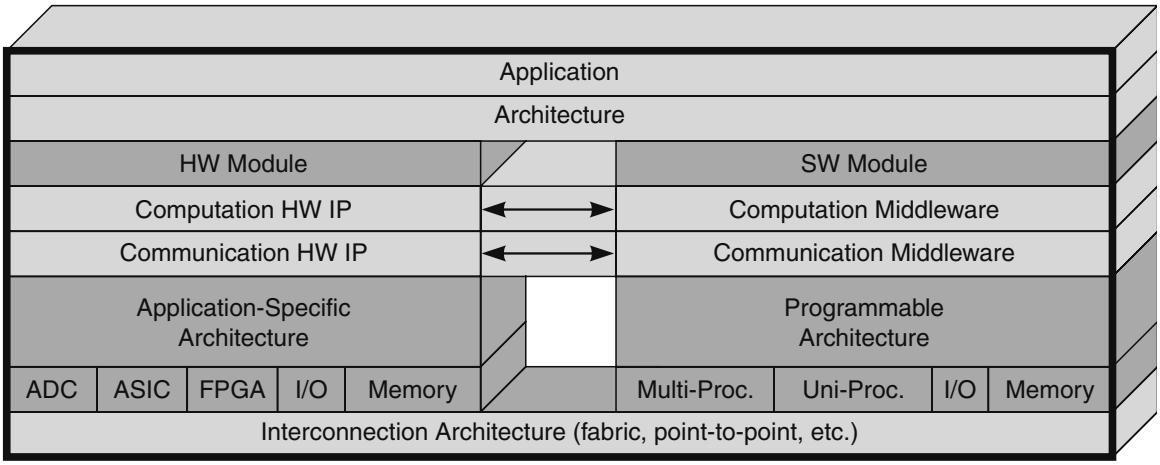
4 High Performance Embedded Computers: Development Process and Management Perspectives

Robert A. Bond, MIT Lincoln Laboratory



This chapter briefly reviews the HPEC development process and presents a detailed case study that illustrates the development and management techniques typically applied to HPEC developments. The chapter closes with a discussion of recent development/management trends and emerging challenges.

4.1 INTRODUCTION

This chapter presents a typical development process for high performance embedded computing (HPEC) systems, focusing on those aspects that distinguish HPEC development from the mainstream of embedded system development. HPEC system developments are influenced by requirement, plans, and implementation decisions of the systems in which they are embedded. In an advanced radar system, for example, it may be advantageous to field the analog radio frequency (RF) subsystems (antenna, transmitters, receivers, and control) prior to the availability of the high performance signal and data processors. This practice allows field data collections that will then lead to refinement of the algorithms targeted for the HPEC systems. However, to perform data collections, HPEC front-end subsystems are needed to perform analog-to-digital conversion, signal demodulation, and low-pass filtering. A high throughput, high-capacity recording system, which is itself an HPEC system, is also required. To accommodate the overall radar development plan, the

HPEC system is architected and implemented as three subsystems: front-end, data recorder, and back-end. These subsystems are then developed and integrated with the overall radar to support the phased, rapid deployment of the antenna. This, in fact, is the plan employed in the case study presented later in this chapter.

In addition to such external factors, the need to control technical, cost, and schedule risks also strongly influences HPEC development and management. A typical HPEC system will be a hybrid consisting of both custom hardware and programmable processors. It may consist of field programmable gate arrays (FPGAs), custom application-specific integrated circuits (ASICs) implemented in very-large-scale integration (VLSI) technology, custom boards, and a commercial off-the-shelf (COTS) programmable multicomputer [with several digital signal processors (DSPs) or microprocessor nodes]. The management of the development must be tailored to meet the particular technology choices since each technology has its own development cycle, cost, technical limitations, and risks. For example, developing a custom ASIC will tend to slow down the implementation; sometimes, especially if the chip is complex, it will be prudent to plan on two fabrication runs. It might make sense to mitigate this schedule risk by developing a lower-performing standard-cell or FPGA solution with a plan to retrofit the custom ASIC in a later iteration. With this approach, the ASIC design can proceed off the critical path of the system development schedule. Programmable processors have their own risks. In particular, the use of COTS technology, while providing ready and cost-effective access to state-of-the-art processor technology, may increase integration complexity and risk. COTS components must generally be used "as is," and, hence, integration issues must be accommodated by increased complexity in the surrounding system. Software for COTS processors also represents a major source of risk. Often, an HPEC system will employ the latest technology offerings, and as a consequence the software development tools may be primitive and the system runtime software may be immature. It is important for management to factor into the cost and schedule the added development effort that must be expended to overcome these deficiencies. Also, an HPEC system must meet throughput and latency requirements using processors that must accommodate severe size, weight, and power constraints. These requirements demand highly efficient software implementations, which, in turn, may require substantially more development effort. Usually, parallel processing codes, which are especially difficult to implement, are needed.

In this chapter, the HPEC development process is briefly reviewed, followed by a detailed case study that illustrates the development and management techniques typically applied to HPEC developments. The reader is referred to Chapter 3 for a more detailed technical discussion of the processor described in the case study. The chapter closes with a discussion on recent development and management trends, as well as emerging challenges.

4.2 DEVELOPMENT PROCESS

Conventional HPEC system development can be understood as an adaptation of the spiral development process, as defined by Boehm (1988). Figure 4-1 shows the basic process. Although the model was originally designed for software development, it can also be applied more generally to system development. The basic idea behind the spiral model is that it is most effective to manage the system development in a series of prototyping and development cycles that explicitly address risks. The iterative nature of this risk-driven methodology acknowledges that requirements are not expected to be fully known at the outset and must be explored through a series of prototypes.

As Boehm explains in his original article (1988):

> Each cycle of the spiral begins with the identification of the objective of the portion of the product being elaborated (performance, functionality, ability to accommodate change, etc.); the alternate means of implementing this portion of the product (design A, design B, reuse, buy, etc.); and the constraints imposed on the application of the alternatives (cost, schedule, interface, etc.).

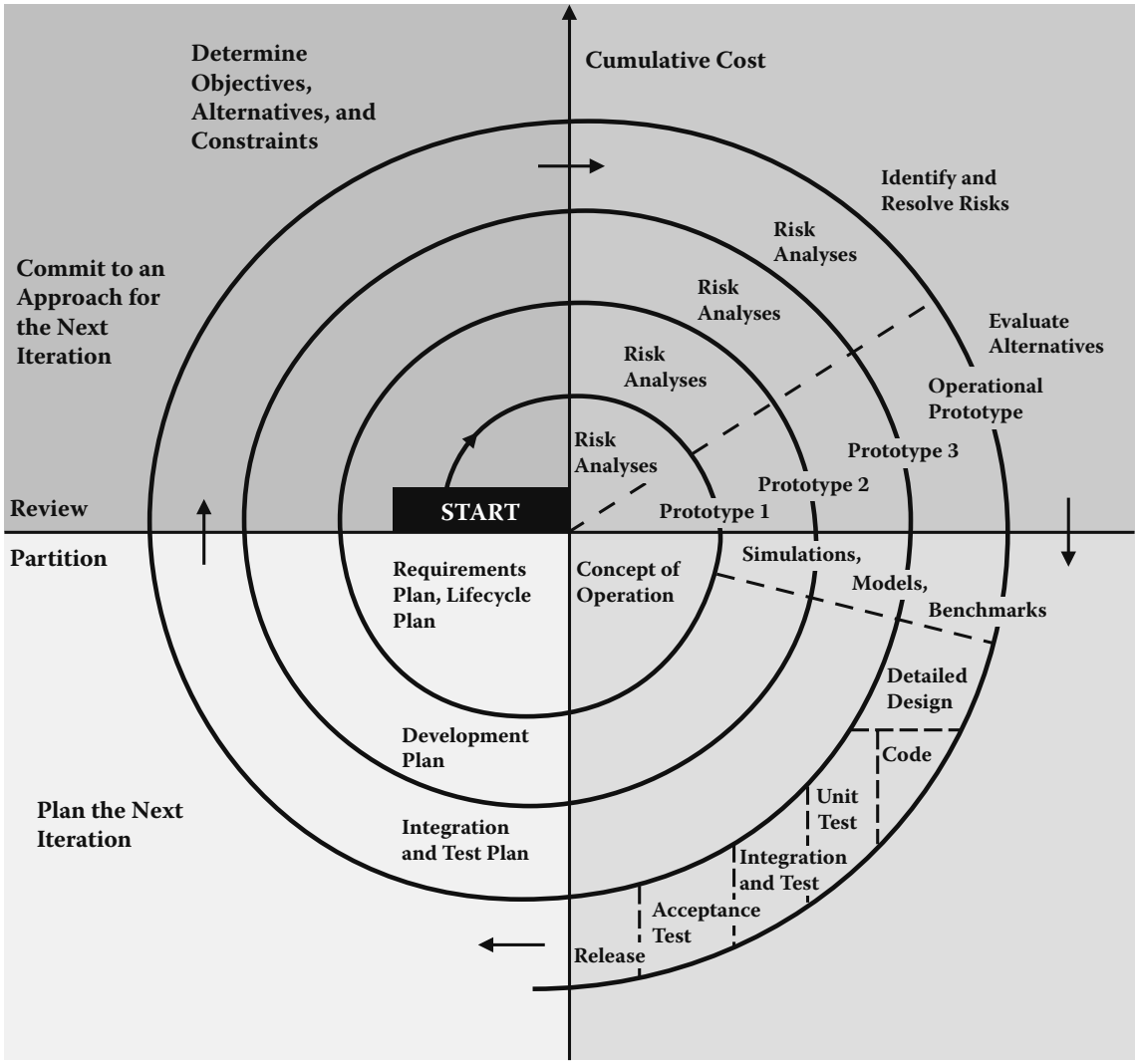


FIGURE 4-1 The spiral development process consists of a series of iterations that are viewed as building spirally to develop the overall system. Each cycle is intended to address a set of major risks in the development. A cycle may involve the development of a prototype, with each subsequent prototype addressing and removing risks and adding functionality. At some point, a transition from rapid prototyping to a production development approach (for example, a waterfall model) occurs, and subsequent spirals continue to add functionality iteratively. The complexity and duration of a spiral may be as short or long as the program pace and system evolution dictate.

In an HPEC development, the objectives are specified in terms of the overall system into which the computer is to be embedded. For example, consider a synthetic aperture radar (SAR) embedded in an unmanned aerial vehicle (UAV). The operational and performance goals of the UAV missions will determine radar performance objectives. The UAV form-factor constraints and the radar performance requirements translate into requirements for the SAR signal and data processors. The first cut at SAR performance would include cross-range and along-range resolution, sensitivity, modes of operation (e.g., scan and spot mode), false-alarm rates, etc. Candidate algorithms would be developed for each mode, which, along with operational parameters such as bandwidth and dynamic range, determines the throughput requirements of the processor. The UAV payload budget places constraints on the size, weight, power, cooling, and ruggedization for the HPEC system. Alternative HPEC architectures are developed to implement the modes and algorithms within specified form-factor constraints. Quite often, MATLAB codes of the algorithms will be developed and evaluated against real data or simulations. The MATLAB programs are then used to specify processing requirements. As illustrated more fully later in this chapter, the MATLAB programs can

be restructured to mirror the architecture of the processor and can then be used to support processor integration and verification. In this approach, datasets are input to both the MATLAB program and the real-time processor; the output of the MATLAB program is compared to the output of the processor at each processing step to verify processor functionality.

Once the requirements, alternatives, and constraints for the processor are established, the next steps are to perform a risk analysis and then to define a prototyping spiral designed to retire some portion of the risk. At this point, management will determine the cost and schedule for the iteration and will also extrapolate a cost and schedule for the overall development. To size the processor development, a basic architecture design will be needed, including, for example, allocation of functionality to hardware and software; specification of the hardware that needs to be developed or procured; specification of the basic algorithm suite; and estimation of the size of the embedded software program.* The development time constraint imposed by the overall sensor development also influences the approach and the cost. For example, trying to compress a software development schedule beyond a certain point leads to an exponential increase in schedule risk and development cost. To reduce software risk, the program may be structured so that key software functionality is developed in the earlier spirals even though it may lead to software rework as requirements change and mature later in the development.

In the early spirals, the goal is to reduce the largest technical risks and to initiate lengthy tasks that may drive schedule risk. The most common risks in an HPEC development addressed in these early cycles include the following:

Form-factor constraints: The form-factor or throughput constraints may warrant the development of either custom hardware, based either on ASICs (custom VLSI or standard-cell chips) or efficient FPGA implementations. At this point, a deeper analysis and simulation will be required in which the precision and complexity of the algorithms will need to be weighed against the implementation complexity of the hardware. The efficiency of key algorithms on FPGAs may be a deciding factor, so a rapid prototyping of algorithm kernels on a candidate FPGA may be carried out. Often, the co-design of the algorithm and the hardware architecture can result in a significant reduction in system complexity and implementation risk. If the FPGA prototyping indicates that custom VLSI is needed, then the VLSI design can be initiated at an early stage in the program.

Algorithm and functional uncertainty: Typically, the most suitable algorithm techniques are generally not well understood at the outset and will continue to evolve throughout the program. The complexity of the algorithm, however, drives the throughput requirements of the processor and, when coupled with the form-factor constraints, will dictate implementation technologies. The particular algorithmic techniques may or may not have efficient implementations in hardware. Often, a simulation of the operational environment is used to help refine and focus algorithm alternatives. Usually, data collections in representative environments are carried out. Using real data has the advantage of helping to validate assumptions that may be evident in the simulations. An effective risk-mitigation strategy at this stage is having the processor architects work with the algorithm developers to determine the fundamental algorithm kernels and the likely range of operation of these kernels. Then, an analysis of alternatives and benchmarks of prototype implementations can be performed. In any case, it is again important that algorithms and architectures are developed in a co-design approach. Often, simple changes from an algorithmic perspective, such as using single precision versus double precision or integer arithmetic versus floating-point arithmetic, can have dramatically simplifying effects on hardware without much loss in performance.

* This is an incomplete list of the important requirements that the system designer will need to consider when assessing program cost and risks. For example, testing requirements, including integration test beds, test datasets, field testing, etc., are significant cost and schedule drivers. Mission assurance and system availability requirements will also affect cost and schedule. For systems requiring fault tolerance, such as mission-critical command-and-control modules in spacecraft, redundant equipment will be needed, driving up costs and testing requirements. Maintenance and upgrade requirements will influence up-front design costs. All of these and more must be considered in the overall HPEC development.

Synchronization and control: In HPEC applications, it is important to be able to exploit the regularity of data flow in the computation. This leads to simpler, lower risk implementations. Current trends toward multifunction and multimodal sensors and communication systems, however, are significantly increasing the need for context changes, buffering, and complex communication and synchronization requirements. In the early stages of the development, it is important to take explicit aim at simplifying the control complexity of the application through a co-exploration of system alternatives and processor architectures. System-level simulations, concept-of-operations (CONOPS) studies, and up-front processor architecture simulations will be employed to uncover the control requirements. Often, for the sake of reducing implementation complexity, the flexibility of the processor will be constrained, in turn constraining the CONOPS of the overall system.

Software complexity: Software programs in HPEC systems from two decades ago would typically have had a few thousand lines of algorithm code. Much of the HPEC architecture was implemented in custom hardware, either in custom boards using COTS hardware components or custom chips. Today, however, HPEC systems with 100,000 to 500,000 source lines of code are common. The development of large software codes for HPEC systems can be controlled through the use of early prototyping of key software modules, the reuse of software codes, the use of software frameworks previously developed for the particular class of application, and the use of middleware to raise the level of development abstraction, which has the effect of reducing overall development complexity.

Commercial off-the-shelf processor technology integration: COTS hardware is procured "off-the-shelf," thereby avoiding costly custom design and development. However, it brings its own set of risks that need to be addressed early in the development. For example, a single-board computer will need to be evaluated for application suitability. The board should be benchmarked using the key algorithm kernels at the scales that will be needed by the application. Preferably, a simplified application example, often referred to as a mini-application or compact application, should be implemented and benchmarked on the board. If the memory, input/output (I/O), or computation resources of a single board will not meet the application throughput and latency requirements, then additional boards will be needed. For most HPEC applications, multiboard, multicomputer COTS implementations are needed to meet performance requirements. Sometimes, it is not feasible or cost-effective to evaluate all candidate solutions at full scale; therefore, extrapolation to the larger system must be made. The sooner representative hardware can be benchmarked, the better. Moreover, the performance is not the only risk dimension. COTS programmable systems generally consume more power and require more space than ASIC counterparts. Thus, form-factor risks must be assessed. Furthermore, the development tools, runtime tools, scalability, and maturity of the product are all important risk factors.

Custom ASIC designs: Custom processor designs can deliver excellent performance with efficient size, weight, and power, but they are costly and time-consuming. They are inherently less flexible than are programmable systems. Therefore, for programs that require rapid development, the decision to use ASICs will require early commitment to an algorithm technique, with limited ability to change the algorithm later on. If the early algorithm variants are lower performing or, worse, inappropriate, the overall objectives of the system will be at risk. On the other hand, increasing the complexity of the hardware to accommodate potential algorithm changes increases the complexity of the custom designs, thereby leading to technical risks.

At the outset of each spiral, management assesses risk areas and decides on mitigation approaches. In the earlier cycles, "throwaway" prototypes and models may be used. The prototypes at this phase are focused on retiring specific risks. For example, a prototype FPGA implementation of a key algorithm kernel may be undertaken, or an efficient parallel code for a key kernel may be developed. In a subsequent spiral, it might be determined that the prototype is mature enough that it can serve as an operational baseline. For example, if the FPGA implementation turns out to be very efficient, it can then become the basis for a portion of the final architecture. Moreover, if the development has reached the point where requirements have stabilized for some portion of the system, then that portion can undergo a more traditional "waterfall approach" while other, less

stable, components can continue in the prototyping cycle. An essential aspect of the spiral model is the review process that occurs at the completion of each cycle. This review is the opportunity for management and technical personnel to assess the success of the cycle and to establish the plans for the next cycle. It is also the opportunity to refine cost and schedule estimates and, if necessary, to adjust the scope of the program.

4.3 CASE STUDY: AIRBORNE RADAR HPEC SYSTEM

The RAPTOR* space-time adaptive-processing (STAP) radar signal processor project provides an excellent example of the HPEC spiral development process. At the time that this system was implemented (circa 1989), it represented the state of the art in HPEC systems. RAPTOR, shown in Figure 4-2, consists of three major subsystems:

1. *Custom front-end processor (FEP)*: The front-end requirement could only be addressed through the use of high performance custom VLSI chips. The VLSI chips were the heart of the digital in-phase and quadrature sampling (DIQ) subsystem, which demodulated the received radar signals and applied low-pass filtering. This subsystem processes 48 receiver channels and operates at a throughput of 100 billion operations per second (GOPS).
2. *Tape recorder subsystem (TRS)*: The high-capacity tape recorder system that operates at 30 Mbytes/s is capable of recording 30% of the range extent after DIQ low-pass filtering.
3. *Programmable signal processor (PSP)*: To perform digital filtering, jammer nulling, clutter nulling, detection, and target parameter estimation, the PSP consists of nearly 1,000 processors, housed in four interconnected chassis, and is capable of performing 85 billion floating-point operations per second (GFLOPS). The software program consisted of over 180,000 source lines of C language code, designed to implement highly optimized radar signal processing.

This case study focuses on the programmable signal processor of RAPTOR, although aspects of the overall RAPTOR development are briefly reviewed to provide context. During the first spiral, the overall requirements for the processor were set. It was determined that RAPTOR would process 48 narrowband receiver channels from an airborne radar phased-array antenna. The received RF signals would be downconverted to an intermediate frequency and digitally sampled. The digital signals would then be demodulated to baseband and low-pass filtered. After that, airborne moving-target indication (MTI), employing STAP, would be performed. After constant false-alarm (CFAR) detection, the target reports would be passed to a tracker and displayed to the system operators. The required airborne MTI processing chain comprised a set of sophisticated and computationally demanding signal processing algorithms. The transmitted radar waveform is referred to as a coherent processing interval (CPI). A CPI consists of a series of pulses transmitted at a regular interval referred to as the pulse repetition interval (PRI). The dataset received during a CPI can be thought of as a data cube. Each return signal can be labeled in the data cube by a {range gate, channel, PRI} triplet. The major functions performed in the RAPTOR airborne MTI (AMTI) digital processor are as follows [the reader is referred to Ward (1994) for more information on STAP MTI processing]: First, the receive signals are match filtered along the range dimension with replicas of the transmitted waveform to improve the SNR and localize the target returns in range. This stage is referred to as pulse compression. Then, the signals are filtered along the PRI dimension into Doppler frequency bins, each of which corresponds to a specific range rate of the return signal relative to the radar platform velocity vector. These returns are then adaptively combined along the channel dimension in an adaptive beamformer to produce 48 beams. The beamformer adaptive weights are computed by sampling the environment for jammer energy and then solving a least-squares optimization prob-

* Reconfigurable Adaptive Processing Test bed for Onboard Radars (RAPTOR)

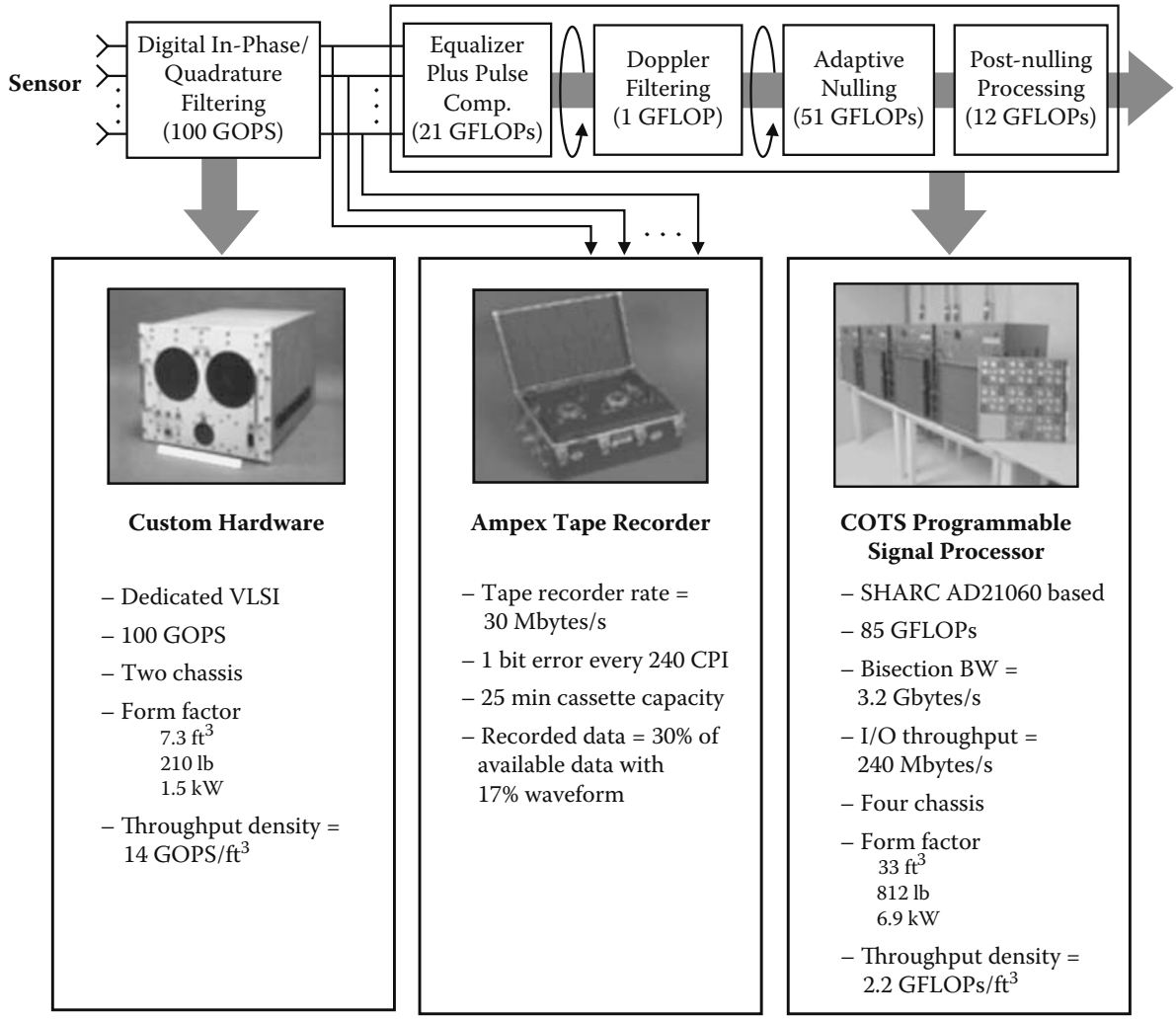


FIGURE 4-2 Airborne signal processor. (From Martinez, D.R., T.J. Moeller, and K. Teitelbaum. Application of reconfigurable computing to a high performance front-end radar signal processor. *Journal of VLSI Signal Processing Systems* 28(1–2): 67, Figure 3 © 2001 Kluwer Academic Publishing, with kind permission of Springer Science and Business Media.)

lem to determine a set of beamforming weights that remove (null) the jamming while maintaining high gain in the beam pointing directions. Subsequently, these beams are adaptively recombined to mitigate clutter interference. The clutter mitigation stage uses a space-time adaptive beamformer, producing another 48 beams that are adaptively determined to minimize clutter while preserving the signals of interest. The samples in these beams are tested for detections. State vectors consisting of range, azimuth, elevation, range rate, and radar cross section are estimated for all target detections. Finally, target reports consisting of target numbers and state vectors are sent to the back-end tracking and display subsystems. The required end-to-end algorithm turns out to be very complicated, involving over 100 signal processing steps. At the outset of the first development spiral, only a simplified version of the algorithm existed. The throughput requirement was initially estimated at between 100 GOPS and 200 GOPS, including both the front-end and PSP processing.

In the first spiral, an initial HPEC system architecture was established, and the system was factored into its three main subsystems. The following major risks were identified:

1. The throughput in the front-end of the system, where digital IQ processing is performed, would exceed 100 GOPS, although the operations were very regular and could be carried out in each of the 48 receive channels in parallel. Analysis and simulation determined that

18 bits (complex) of precision would be needed. Programmable processor technology at the time could not handle the throughput requirement, and so special-purpose hardware would be required.

2. The STAP and detection algorithms were immature and would require extensive development. Data collected from the radar antenna and receivers in a representative environment would be needed to validate the algorithm techniques. The late specification of these algorithms, the need to evolve the techniques as more insight into the operational environment was garnered, and the sophistication of the algorithms all pointed to the need for a programmable signal processor. It was likely that this PSP would need to be rated at over 50 GFLOPS. A programmable adaptive radar processor of this scale had never before been implemented.

To address the front-end processing challenge, a set of three spirals, shown in Figure 4-3, was carried out. In the first spiral, a survey of architectural and technology alternatives was conducted. The INMOS A100 chip and others were evaluated, and it soon became evident that there were no commercial chips that could simultaneously meet the throughput and precision requirements of the digital IQ processing. At this point, a second spiral was initiated. The spiral consisted of an analysis of alternatives for VLSI designs for a digital IQ chip, a matched-filter chip, a Doppler-filter chip, and a beamforming chip. The analysis determined that the digital IQ requirements could be met with a standard-cell custom design using 18-bit integer arithmetic. Full-custom VLSI was also investigated, but the schedule risk was assessed as too high, although the design would be able to deliver higher performance. During this spiral, the preliminary front-end processing architecture was refined. Standard-cell custom top-level designs for the matched filtering, Doppler filtering, and beamforming were developed that would meet the requirements of these functions. Due to the overall complexity of the timing, control, synchronization, and data movement in the front-end processing architecture, coupled with the large number of unique custom chip designs, the front-end processor implementation was assessed as being very high risk. As a result, an alternative architecture, based on programmable signal processing, was developed. In this architecture, the digital IQ function was allocated to a standard-cell custom design, but the remaining signal processing stages downstream of this function were allocated to programmable components. This approach was also

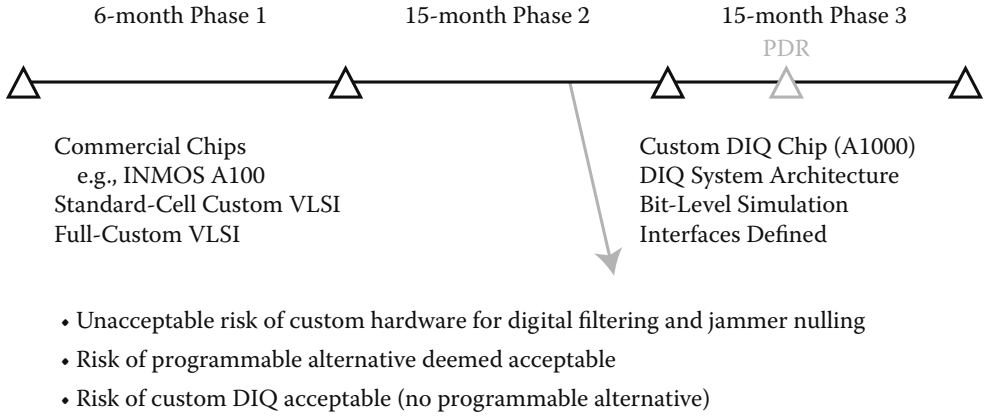


FIGURE 4-3 Custom-hardware risk-mitigation timeline. The front-end signal processing requirements represented a significant technical risk. The risk mitigation approach involved three spirals (phases). The first spiral entailed a requirements analysis and a survey of commercial chips and technology options. The second spiral took the findings of the first spiral, which recommended a VLSI technology approach, and developed a set of alternative VLSI-based front-end architectures. This spiral resulted in an architecture that relegated the digital-filtering and jammer-nulling functions to programmable hardware and allocated the digital in-phase and quadrature sampling function to custom VLSI. The final spiral developed a detailed design that followed the architecture recommended in the second spiral.

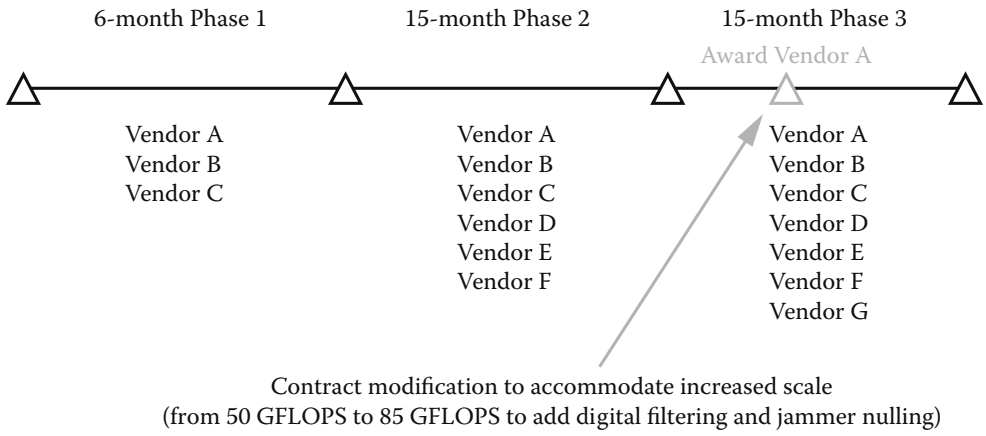


FIGURE 4-4 PSP spiral risk-mitigation timeline. The programmable signal processor hardware was a significant risk factor in the program. To mitigate this risk, a series of risk-mitigation spirals (phases) were carried out. The first spiral was a preliminary feasibility study. The second spiral was a detailed benchmarking phase. The final spiral was the processor procurement. During the procurement phase, the size and functional scope of the PSP were increased to take on some of the signal processing initially assigned to the front-end custom hardware.

recognized as a risky proposition, but pending the results of the benchmarking of the programmable processors, the risk was considered more manageable.

The risks associated with the programmable implementation of the STAP and detection algorithms were considered to be very high. Programmable signal processor technology had never been applied to a STAP problem of this size, and the need for a ruggedized, airborne form-factored computer further increased the challenge. Hence, an early investigation was conducted to assess this approach. The first step was to survey the state of the art in PSP technology. It became evident that while programmable supercomputing systems would be capable of handling the throughput, they were neither ruggedized nor of an appropriate form factor (size, weight, and power). On the other hand, embedded multicomputer technologies based on i860s, SHARC DSPs, and PowerPCs were emerging as potential candidates. Their performance on STAP algorithms, however, was unproven. At this point, management decided to address the risk through a comprehensive, nationwide study and benchmarking initiative. Three risk-mitigation cycles, which culminated in the procurement of a PSP hardware system, were carried out. The timeline and goals of the spirals are shown in Figure 4-4.

In the first spiral, a set of benchmarks and processor form-factor constraints were specified. The benchmarks consisted of several variants of STAP algorithms, specified for radars of various operational configurations. A more general-purpose linear algebra benchmark was also included. A wide range of radar and algorithm options was needed since it was still unclear at this early stage what the final configuration for the target radar would be and what algorithm variant would be chosen. A Request for Proposal (RFP) was issued to major radar prime contractors and computer vendors. There were three respondents, each of which was awarded a study contract. The results of the studies were favorable, indicating that modern multicomputer technology would be able to meet the throughput, latency, and form-factor requirements.

A second spiral was initiated with another RFP, this time requesting the implementation of the benchmarks on a scaled-down version of a proposed system and an architecture design for the target PSP system. Six vendors participated, including the original three. The results were once again quite promising, although the challenge of developing efficient parallel versions of the benchmarks was also evident. This spiral demonstrated that STAP algorithms could achieve real-time throughput on PSPs. The architectures were scaled to extrapolate to the full-scale STAP algorithm. The proposed architectures required in the range of 100 to over 400 processors. Computational efficiencies from 25% to 30% were predicted.

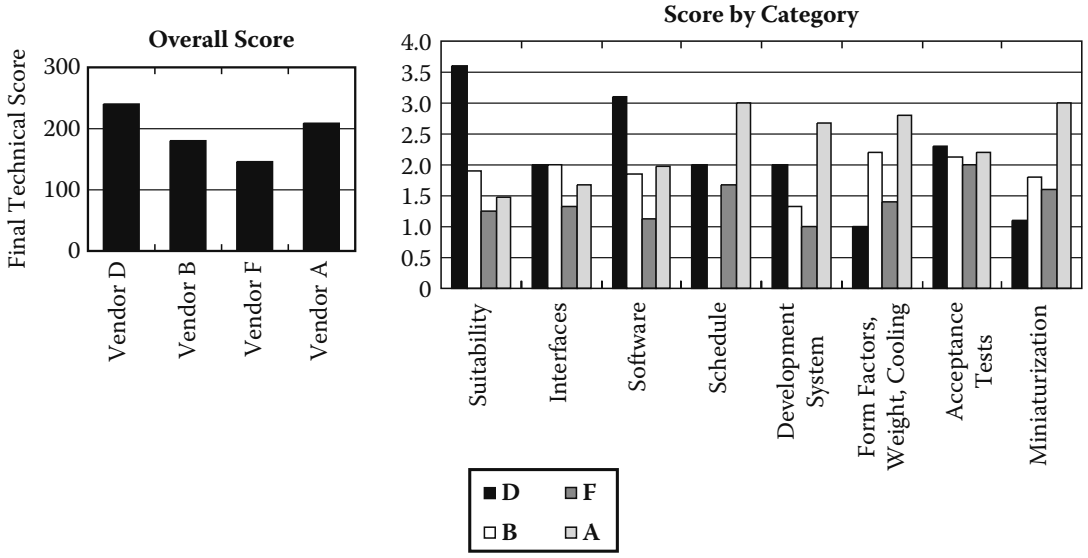


FIGURE 4-5 PSP procurement: proposal evaluation. The PSP proposals were evaluated in several categories. A cumulative threshold of 200 was established. Vendors with scores below this value (Vendors B and F) were, therefore, eliminated. Vendor D had a higher cumulative score than did Vendor A, but it was determined that the form-factor risks associated with Vendor D were too risky. Thus, Vendor A's proposal was chosen as the winning proposal.

The third spiral was initiated by a third RFP for the procurement of a full-scale, ruggedized, airborne multicomputer to perform STAP. The RFP was issued to the six participants and also to the vendor community in general. The responses were evaluated against a matrix of required capabilities (refer to Figure 4-5):

1. Suitability—an assessment of the architecture design, including predicted throughput and latency on the benchmarks
2. External interfaces—an assessment of the I/O capability
3. Software—an assessment of the runtime software suite
4. Schedule—an assessment of the processor fabrication and delivery time
5. Development system—an assessment of the availability and quality of a development system that would be delivered prior to the final system to allow early software development
6. Form factor—an assessment of the size, weight, power, and cooling metrics
7. Acceptance tests—an assessment of the quality of the proposed acceptance test procedures
8. Miniaturization—an assessment of the viability for the miniaturization in a future variant

The six vendors from the benchmarking phase and three other vendors were solicited for proposals. Of the original six, one decided not to bid. One of the newly selected vendors also did not bid. Of the remaining vendors, one was determined to be nonresponsive since many of the selection categories were not addressed. The remaining vendors (denoted vendors A, B, D, and F in the figure) were scored against the criteria listed above. Two of the vendors fell below the acceptance threshold. The comparison between the two remaining vendors was quite interesting. Vendor D had the highest overall score, but had a particularly low score in form-factor assessment. Vendor A had moderate scores across the board, but the overall suitability of the architecture was only marginal. After careful consideration, the evaluation committee decided that the form-factor risks associated with Vendor D were more likely to occur and be of potentially greater impact than the architectural risks associated with the Vendor A processor. The costs for the two processors were comparable. Thus, Vendor A was awarded the contract.

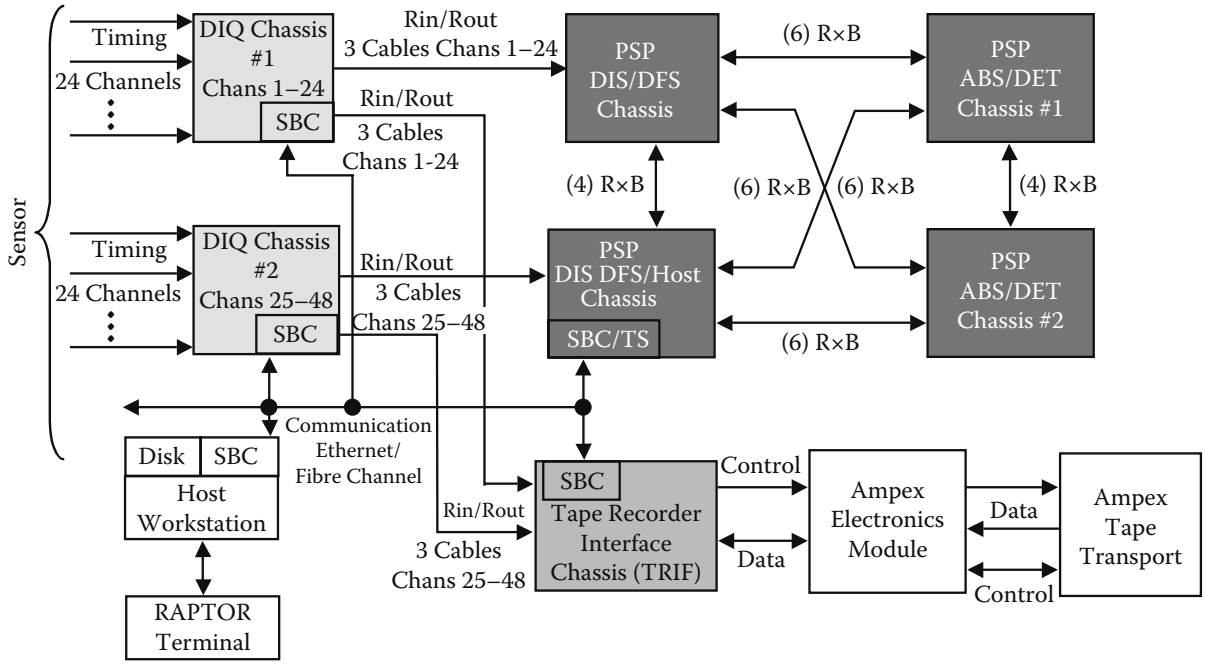


FIGURE 4-6 RAPTOR chassis-level block diagram.

At the same time that this evaluation was taking place, management was struggling with the realization that a full-custom front-end of the scope that was originally envisioned was too risky. The chief architect, who was a participant in both the front-end and the back-end spirals, developed a new architectural proposal that allocated the digital IQ functionality to custom hardware and subsequent stages to a scaled version of the Vendor A proposed processor. Program management, the chief architect, and the team leads for the front- and back-end systems conferred to weigh the relative risks and options. Out of this meeting, the critical decision was reached that the PSP would be scaled upward to encompass all of the processing after the DIQ task. Vendor A was notified and asked for a cost, schedule, and technical proposal (contract modification) to scale the processor. They sent a favorable response quickly. The proposed processor would be scaled from two chassis to four chassis, its throughput would be increased by about 40% to handle the additional processing, and the front-end I/O system would be scaled up by a factor of two to handle the greater input data rate. Although commercial hardware and operating-system software components were to be used, a system of the required size and complexity was not yet commercially available. The vendor proposed (as per contract requirements) to deliver the processor in nine months. A fully working STAP acceptance test benchmark and acceptance test would be carried out first at the factory and then at the radar system integration laboratory (SIL).

The sequence of risk analyses, studies, and architectural designs just described illustrates the complex and interdependent nature of HPEC system design. By using a risk-driven spiral development model, program management was able to navigate the early stages of the project through a series of important analyses of alternatives, informed by early benchmarking, simulation, and modeling. The early trade-offs between the custom and COTS hardware, between programmable and hardwired functionality, and between full-custom and standard-cell VLSI were crucial to the success of the development. In the absence of these trade-offs, management might have committed to a design whose critical shortcomings would only have become evident during a later cycle in the development, when the impact would have been much more severe.

Even with these early trade-offs, the project still had several risk areas, in particular those associated with the scale and complexity of the PSP. The architecture that emerged from these spirals is shown in Figure 4-6. The entire system was controlled by a host workstation connected over an Ethernet. Two 9U-VME chassis were allocated to the digital IQ (DIQ) subsystem. Each chassis

handled 24 receiver channels. The outputs from the DIQ subsystem were duplicated so that data could be simultaneously sent to the PSP and the tape recorder. The tape recorder had its own chassis and special mount for the tape-recorder assembly. The PSP consisted of four 9U-VME chassis housing nearly 1,000 processors having special high-speed interchassis connections. The development of the PSP is now presented to illustrate the steps taken by management to address these risks and to ensure the successful development of the challenging PSP component of this HPEC system.

4.3.1 Programmable Signal Processor Development

Once the top-level architecture of the HPEC system was determined, the PSP development progressed as another dedicated series of spirals. The PSP hardware was not scheduled to be delivered for nine months, and it would be one of the most complex embedded multicomputers of its kind in the nation. It consisted of nearly 1,000 DSPs, requiring four chassis with a specially designed interchassis communication subsystem and a new operating system (that spanned the chassis, essentially treating the four chassis as a single system). The PSP had the maximum amount of memory that could be fit on each board. It had six high-speed input channels to accommodate the 48 channels of digital IQ from the front-end system. The long lead time in the PSP delivery meant that integration with the front-end system would have to begin almost immediately after acceptance of the PSP. There would be little time for further software development, as much of the time would be needed to integrate the radar system end-to-end and then integrate it into a test-bed aircraft.

Risk in the development was mitigated by taking the following measures:

1. To get a head start on software development, an early, off-the-shelf, development system was procured. The development system, while similar to the final system, had significant differences. It occupied only one chassis, it used i860 processors at each node instead of the triple-SHARC node (still being developed) of the final system, and it used an earlier version of the operating system. While not ideal, the system allowed many software components to be developed and tested early on in the development.
2. As a bridge between the full-scale system and the development system, a 1/4-scale, single-chassis PSP was procured that would arrive six months after contract award. This processor provided early access to a parallel (but reduced-scale) signal processor. It had the same type of processor boards, I/O hardware, and development software as the full-scale PSP and, hence, served as an excellent development and test surrogate.
3. A series of spirals was designed to incrementally add algorithm capability and to scale to the full, four-chassis PSP system.

As shown in Figure 4-7, the overall PSP software system factors into seven subsystems. The allocation of the subsystems within the overall RAPTOR chassis configuration is shown in Figure 4-6. All of the subsystems map onto multiple signal processing nodes within these chassis, with the exception of the task scheduler (TS) subsystem. The TS receives radar mode commands and configuration information from the radar, describing the waveform and other operational parameters. Each command arrives a few processing intervals in advance, giving the PSP time to set up for the new mode before the data are input into the PSP. The scheduler calculates internal parameters and then transmits a mode description to the master node in each parallel PSP subsystem. Each master node then disseminates the description to all of the nodes that compose its subsystem. The nodes in a subsystem then extract the parameters they need and set up in anticipation of the arrival of new mode data.

The parallel processing subsystems perform signal processing and data reorganization tasks. The data input subsystem (DIS) receives data from the front-end digital IQ hardware each pulse repetition interval. The DIS uses six input ports, each of which handles eight receiver channels and is controlled by a PowerPC. The DIS buffers the data and demultiplexes them into a format that

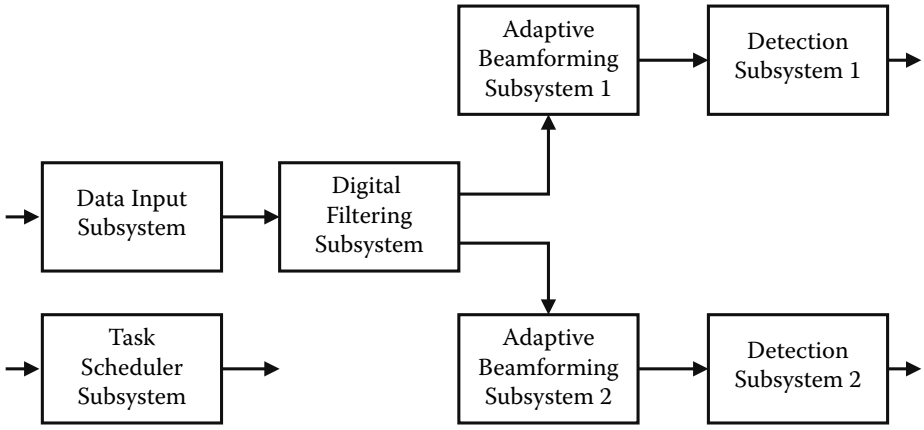


FIGURE 4-7 PSP subsystems. The programmable processors comprised seven subsystems. Five subsystems carried out complex signal processing operations: the digital filtering subsystem carried out pulse compression and Doppler filtering; the two adaptive beamforming subsystems (ABF1 and ABF2) performed jammer nulling, interference rejection, and beamforming; the two detection subsystems (DET1 and DET2) performed detection and target parameter-estimations. The task scheduler controlled the mode of operation of the other subsystems, and the data input subsystem performed high-speed data input and demultiplexing.

can be efficiently processed by the digital filtering subsystem (DFS). The DFS performs the pulse compression and Doppler filtering tasks. It exploits the parallelism in the receiver channels so that each triple SHARC node receives and processes all of the range gates for one of the 48 receiver channels. After filtering, the DFS must also perform data corner-turning, during which it reorganizes the data and transmits them to the downstream adaptive beamforming subsystem (ABS). The corner turn reorganizes the data so that the range gates for each Doppler bin are stored on a separate triple-SHARC processing node in the ABS. There are a total of 96 Doppler bins, and these are mapped one to each of 96 triple-SHARC nodes. The ABS performs jammer nulling and space-time adaptive processing, both of which are beamforming operations. The ABS is the most demanding subsystem, requiring over 30 GFLOPS. Fortunately, due to the early risk-mitigation studies and benchmarks, the throughput expectations of the SHARC nodes on the ABS computations had been assessed beforehand. It was (correctly) predicted that the 96 triple-SHARC nodes would not be able to meet the throughput requirements. To solve this, a ping-pong configuration was used, in which every odd-numbered CPI is forwarded to one set of 96 nodes and every even interval is forwarded to another set of nodes. Although this doubles the latency in the ABS, it allows the hardware to meet the throughput requirement. After the ABS, the beamformed data for three Dopplers were grouped and forwarded to a detection subsystem (DET). Odd processing intervals were forwarded to one DET subsystem and even intervals were forwarded to another subsystem. Each DET used 32 triple-SHARC nodes, with each SHARC in a node handling detections for a single Doppler bin. (However, to perform clustering properly, data needed to be shared between neighboring bins, thus complicating the synchronization and communication logic in this subsystem considerably.) Once detections were grouped and their state vectors (azimuth, elevation, range, range rate, and radar cross section) were estimated, the DET created a target report message that it forwarded to the back-end radar processor for tracking and display.

Each software development spiral followed the basic waterfall sequence shown in Figure 4-8. The sequence starts with a refinement of the requirements and a statement of the specific objectives for the spiral. MATLAB code is used as an executable specification of the algorithm requirements. The code is augmented by documentation that describes the theory and the algorithm design trades. MATLAB code is double-precision floating-point, and the PSP by default uses single-precision floating. Hence, a precision analysis needs to be conducted for any portion of the algorithm that may be expected to have precision issues. This analysis requires coordination between the algorithm designer and the real-time program software engineer. For example, during the early develop-

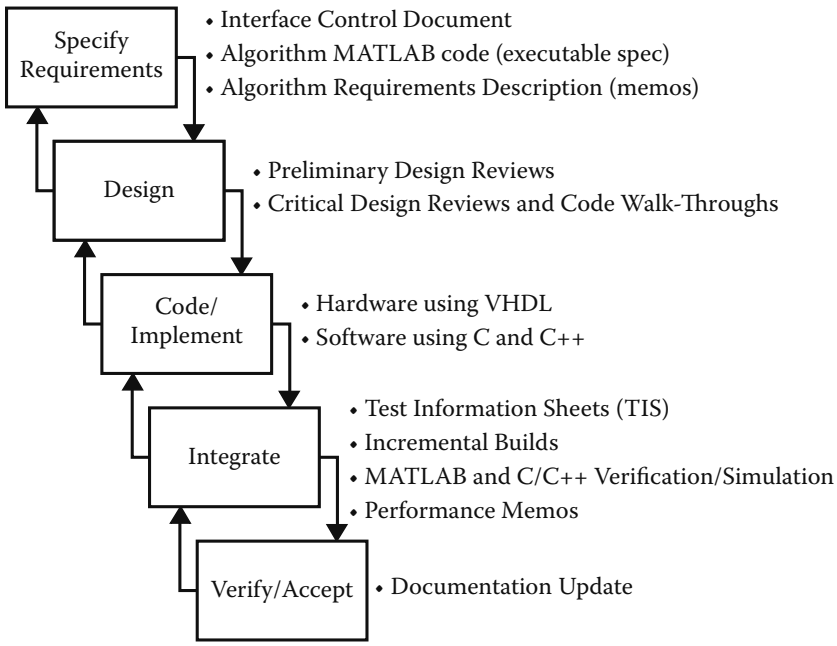


FIGURE 4-8 Waterfall development cycle. For each spiral increment that involved software or hardware development, a waterfall development model was followed. The figure also shows the major products of each phase in the waterfall.

ment of the QR decomposition for the jamming nulling stage, it was determined through numerical simulation that although the single-precision variants of the algorithm produced slightly different results from the MATLAB double-precision code, the end-to-end real-time system still operated within performance tolerances. Double-precision arithmetic requires roughly six times the number of operations as single-precision arithmetic; therefore, being able to stay with single precision was a significant computational benefit. To handle double precision, a QR co-processor or dedicated QR processing nodes would have been required, thereby increasing the size and complexity of the subsystem. Such algorithm architecture trade-offs were generally mediated between the verification and architecture teams, who had the joint responsibility to certify the completion of each spiral. When the spiral involved interfacing requirements to other subsystems in the embedded system, the interface control documents were updated and used as design references.

Figure 4-9 shows the nominal allocation of implementation effort and schedule to the development phases. It is important to realize that the actual effort for a cycle depends on the size of the overall system, not just the size of the increment, since adding new software functionality to an existing body of software requires more work than just creating the new functions; there is an increase in the number of interfaces (leading to additional design and code), the amount of regression testing, and the integration effort.

Figure 4-10 shows the planned sequence of development spirals. Management and technical leadership recognized early on that the task of scaling the PSP to its full size would be very challenging. Because of the late availability of the full-scale PSP, the software needed to be developed initially on a smaller system and ultimately scaled to the full-scale system. The 1/4-scale PSP system was a steppingstone on the way to the full-scale PSP and also served as the permanent development system once the PSP was deployed to the platform. To be able to run the software on three different systems of different parallel scales, with different node processors and runtime systems, a layered software architecture was developed, as shown in Figure 4-11. The architecture was based on a novel parallel middleware library called the STAP library (STAPL), developed using the C programming language.

STAPL provided an object-based library of scalable matrix algebra and signal processing components. Each object could be scaled from one to hundreds of nodes by changing configuration

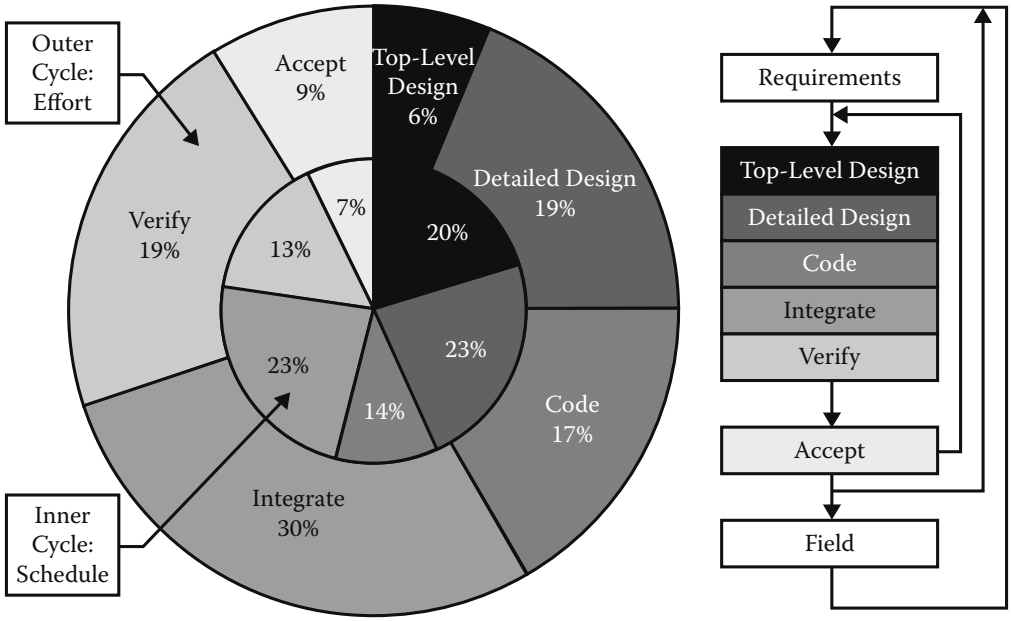


FIGURE 4-9 Software development cycle.

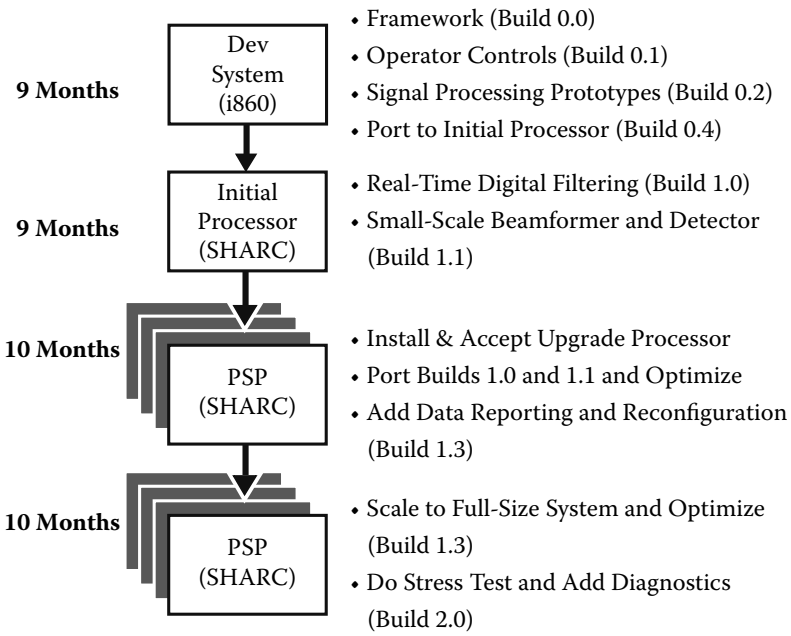


FIGURE 4-10 PSP development spirals.

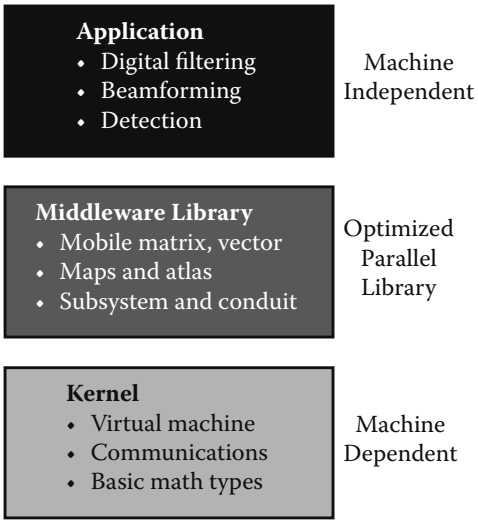


FIGURE 4-11 Layered software architecture.

parameters that were read into the program at start-up and communicated to the objects at initialization. STAPL objects used the underlying lower-level communication and computation libraries supported by the vendor system. In this manner, application software that used STAPL was both portable to the three vendor platforms (the development system, the 1/4-scale PSP, and the full-scale PSP), and it could be scaled from a few nodes to the full processor configuration.

As shown in Figure 4-12, the PSP was scaled in a series of three major spirals. It was important to show end-to-end functionality as part of each spiral, so the scaling steps were chosen to demonstrate a processing thread that exercised the major functionality of all the software subsystems. The first instantiation processed eight channels through the DFS, followed by 24 Doppler beams through the adaptive beamforming and detection subsystems. The odd CPIs were sent to one set of adaptive beamforming (ABF) and DET subsystems, and the even CPIs were sent to the other set. This first scaling step verified multichassis functionality and also allowed every major processing step to be exercised. It verified basic synchronization logic and showed that the task scheduler communication logic was correct and could meet real-time latencies. The next major scaling step increased the DFS to the full-scale, 48-channel configuration and increased the ABF and DET subsystems to half scale—48 Doppler configurations. Most significantly, this configuration was integrated into the full radar system to support system-level demonstrations. Several real-time performance issues that required further optimization were uncovered. Integration issues were also identified at all levels: operating system, middleware, and application. Although the middleware was designed to allow the application to scale, there were still several challenges in moving

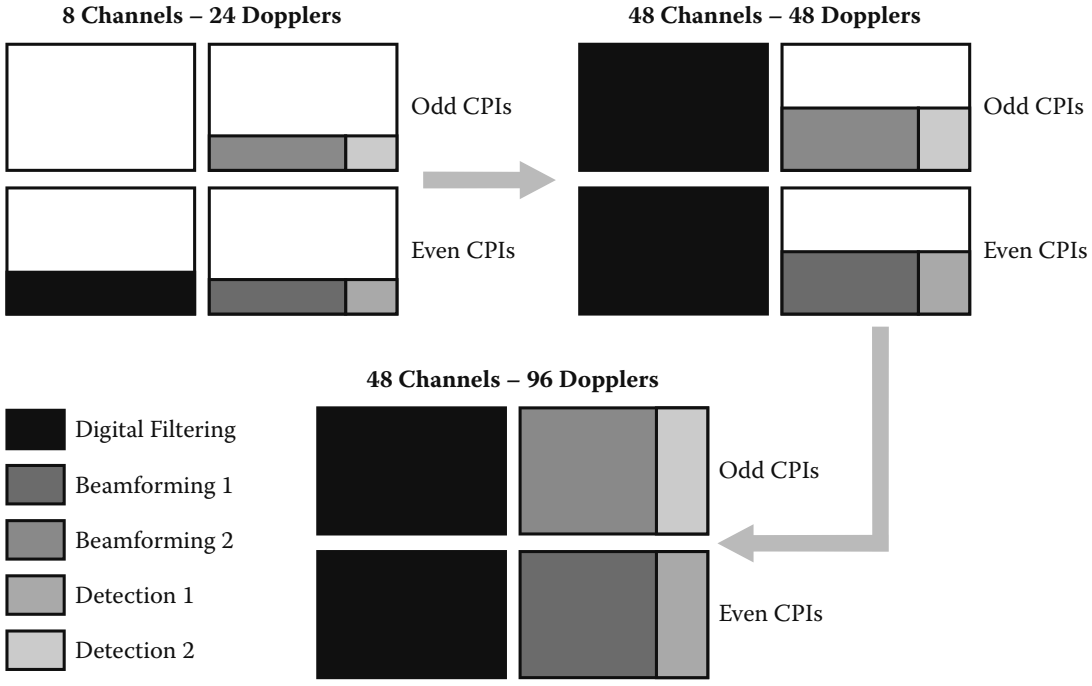


FIGURE 4-12 PSP scaling. The PSP was scaled in three major steps. The allocation of subsystems to chassis is shown for each step. In each step, the end-to-end algorithm was verified at the required scale.

from a 32-processor development system to a 1,000-processor, full-scale system. During system acceptance, the performance of the full-scale processor had been demonstrated using benchmark codes. The full-scale application, however, was significantly more complex, both in total lines of code and total throughput. As the application was scaled, the operating system resource utilization proved to be a major stress area. The full-scale application required the allocation of 1,000s of communication objects, causing the start-up time of the system to become unacceptably long. Loading and initializing the application took over two hours initially. This slowed the debug and test cycle to a crawl. The initial middleware design naively assumed that the operating system would be able to accommodate this load without deleterious consequences. Fortunately, although the solution involved numerous optimizations, the changes were mainly isolated to the middleware initialization; changes to the application were minimal. The final scaling step increased the ABF and DET subsystems to the full 96 Dopplers. The overall scaling development cycle took ten months, during which time the start-up time was further reduced in a series of optimizations, bringing the final, full-scale start-up time down to about 17 minutes—long, but acceptable. The PSP was integrated with the rest of the radar, and numerous performance optimizations were carried out on the ABS and DET subsystems to bring them to within the real-time allocation.

4.3.2 Software Estimation, Monitoring, and Configuration Control

One of the most important management tasks for an HPEC project is to develop accurate estimates of the project development effort, cost, and schedule. For programmable HPEC systems, this involves an estimate of the complexity of the delivered software product. One way to get an idea of the complexity of a software program is to estimate the size of the code in terms of non-blank, non-comment source lines of code (SLOC). Table 4-1 shows the initial estimate of the SLOC for the PSP application code. Similar estimates were developed for the communication, middleware, and mathematical kernels developed for the project. The estimates were derived after the first set of risk-reduction spirals (during the builds prior to Build 1.0, as shown in Figure 4-10). The figure also shows the measured lines of code at the end of the first full delivery (Build 2.0 shown in Figure 4-10) for each subsystem. The estimate turned out to be reasonably good (anything within 20% of the initial estimate is considered good). It was generated by an experienced team working in a familiar application area and was developed after the software algorithm requirements were relatively stable. However, the middleware library (not shown) ended up being about 52,000 SLOC, twice the original estimate. The increased complexity of the middleware was due to the extra coding required to accommodate parallel processing routines and scalability from 10s to 100s of nodes.

TABLE 4-1
PSP Code Estimates

Component	Measured Lines of Code	Estimated Lines of Code	Notes
Task Scheduler (TS)	2.6K	1.5K	
Data Input Subsystem (DIS)	3.2K	1.8K	
Digital Filtering Subsystem (DFS)	8.3K	8.8K	
Adaptive Beamforming (ABF)	9.9K	10.1K	Scope reduced
Detection (DET)	9.0K	10.1K	
System Control & System Loading (SC/SL)	4.3K	4.3K	Already coded
Host (CC/CD)	33.5K	27.9K	Display and control
Total Code	**70.8K***	**64.5K**	

* Does not include OS, Communications, Middleware Library, and Kernel code

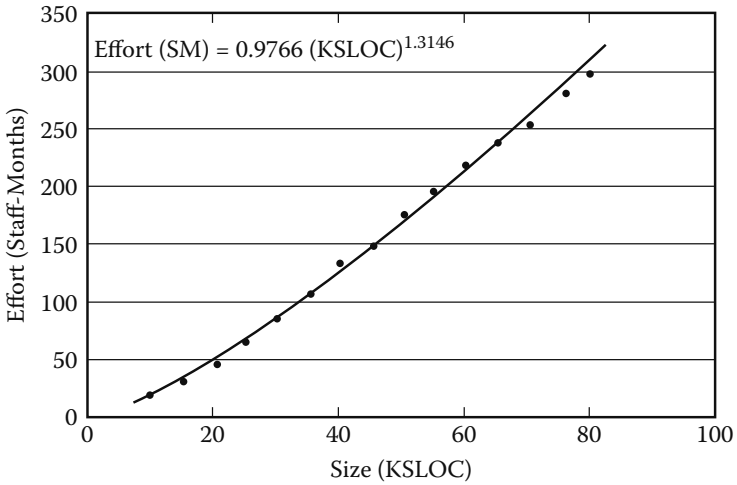


FIGURE 4-13 Software productivity. As the size of the program code grows, the effort (shown above) and development time (not shown) grow nonlinearly. Often, a power-law curve can be used to model the relationship.

There are many pitfalls to using SLOC as a means to estimate cost and schedule. Different software domains, different projects, and different organizations all influence the relationships between SLOC, software cost, and development schedule. Figure 4-13 shows the relationship used in this case study. It was based on metrics from previous projects; historical data predict that as the size of the software product grows, the effort grows with a power-law coefficient. This is intuitively explained by noting that the interactions between software components can grow combinatorially. Adding a line to an existing body of code requires testing the interaction of the new code with preexisting code (referred to as regression testing), as well as testing of the new functionality. The larger the body of previous code, the more likely that the new code will have interactions, and the more extensive the interactions are likely to be. Many other factors besides code size affect the overall cost and schedule. For example, the cost and schedule of the processor in the case study were strongly affected by the delayed availability of the processor, the system deployment schedule, the scale of the processor, the immaturity in the back-end detection and estimation algorithms, and the need to deploy the front-end processor prior to the programmable back-end processor. The series of risk-reduction spirals was also a strong determinant of development plan, cost, and schedule. It turned out that for the case study, the curve shown predicted the cost for unit-tested functionality to within about 10%, but it underestimated the integration cost by nearly 30% and project duration by several months. There was a 50% underestimation in the integration portion of the schedule, due principally to the protracted time spent scaling the processor, an unprecedented task that turned out to be much more difficult than anticipated. Thus, while software complexity estimation is a useful management tool, it must be emphasized that the predictions thus generated serve only as guidelines. Moreover, progress monitoring, configuration control, risk assessment, and project replanning are required throughout the entire development. The whole point of the spiral process is to address the technical, cost, and schedule risks in a way that allows management to iteratively rebalance priorities (cost, schedule, and technical) while meeting evolving and emerging requirements.

Figure 4-14 gives an example of software development monitoring. The progress represents a roll-up of the code integration status of the software planned for a particular spiral build. By monitoring at this level of detail, it is possible to predict the completion date of the integration phase of the spiral. Insight into the impact of external events, such as the introduction of a new operating system, can also be gleaned. Such indicators can aid in risk mitigation by pointing out where additional effort or attention is required.

Tables 4-2 and 4-3 show typical software issue-tracking reports. During each development cycle, it is important to track issues, develop solutions, and then schedule the incorporation of the

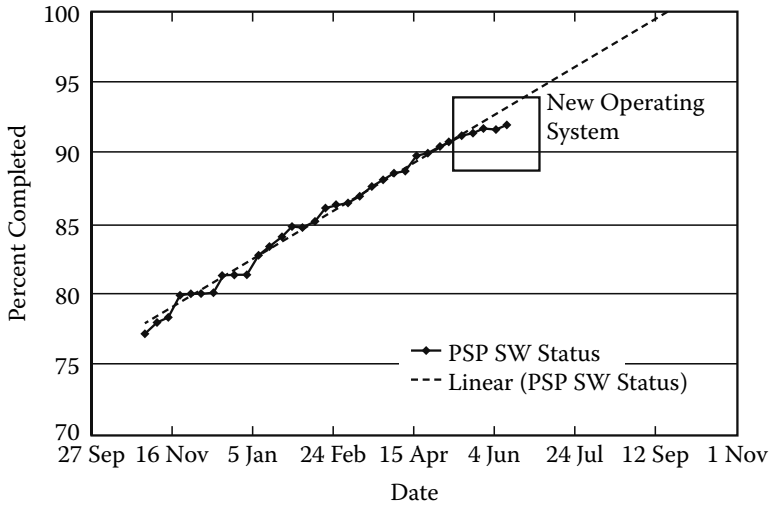


FIGURE 4-14 PSP integration progress. The figure shows the progress of code integration and verification over an eight-month period in the PSP development. The boxed region in the June time frame shows the slowdown in integration when a new operating system was installed in the PSP.

TABLE 4-2
Build 3.0 Issues List

Item	Subsystem	Description	Completed
128	ABS	Optimize ABS tag-files	**X**
85	ABS	110 msec glitch in ABS	**X**
123	ABS	alpha back is not being applied correctly	**X**
122	ABS	Threshold done "prior" to summing but this may be OK	**X**
121	ABS	Jammer nulling start range in CPI info not being used	**X**
110	ABS	Clutter training beam (Nc vs. Nctr)	**X**
120	ABS	Om Jns_selectClutterSet, initialize IC based on Doppler index	**X**
124	DET	Detecting targets from beam 19 twice	**X**
87	DET	Clutter editor agreement	**X**
108	DET	Increase # DET reports (again)	
89	DET	Grouper agreement	**X**
90	DET	SendToCache optimizations	
116	DFS	Corner Turn Conduit output defer until CPI interval ends	**X**
115	DFS	Avoid Corner Turn Conduit data copy for 96 Dopplers	**X**
94	DFS	Fix CPI timer	**X**
95	DFS	Tune SURV Tag files	
125	DFS	All data products fail to provide actual CPI number	X
127	DFS	There seems to be an extra word (at end of?) each of the messages	X
69	DFS	Integrate DFS data collection	X
114	TS	Debug Cal Data message timing between RCP, TS, DFS	X
64	ABS/DET	Send subset Cholesky data from ABS-> DET (36 × 36 not 36 × 180)	

Key: No shading = algorithm; light shading = performance; medium shading = integration; X = completed.

TABLE 4-3
Build 3.1 Issues List

Item	Subsystem	Description	Completed
18	ABS	Move tgate_ndx and bgate_ndx to the AbsData structure	
19	ABS	Cns_computeCorrelations() performs a validity check	
13	ABS	Change objects to Basic types B, idxTset, baz, chtaper, spatialTaper	
15	ABS	destMats() method for MobMatrixCplxFltConduit	
106	ABS	Channel masking	**X**
20	ABS	Cns_apply SHoleWeights() calls SHARC assembly functions	
34	ALL	Error Handling (ability to handle bad CPIs)	**X**
91	DET	Improve logic for limiting target reports	
86	DET	Send "Tracker Aux Info" as part of DetCommRspMsg	
8	DET	Fix MobTargetReportList::copy Shutdown freeze up – use a non-blocking send	
126	DFS	EquPc raw A-scope should contain 2 adjacent channels but only contains one	
83	DFS	Time reverse and conjugate PC coefs at start-up. Change .I files	
117	DFS	CalSigValid: Real time for the per-channel part	
65	DFS	EQU mode needs to operate in real time (tag files required)	
118	TS/DFS/ABS	Implement Sniff mode	X
97	DIS	Send timing data to Fran's tool	X
100	SYS	Merge hosts	**X**
102	TS	Send timing data to Fran's tool	X
63	TS/DFS	Transition to SysTest on any CPI	**X**
130	DET	Implement MLE-based estimation algorithms	

Key: Heavy shading = algorithm; medium shading = performance; light shading = integration; no shading = robustness/maintainability; X = completed.

solutions into the configuration-controlled code. These tables show excerpts of the issues for PSP Builds 3.0 and 3.1, respectively. Both builds occurred after acceptance of the full-scale system, during the first few months of system operation. A configuration control board (CCB) consisting of the major program stakeholders was responsible for assessing, prioritizing, and scheduling work to be performed to resolve issues. The program manager was the CCB chairman; other participants included the integration lead, the verification lead, the PSP development manager, and the application lead. The issue descriptions are actual excerpts from the development and, as such, they are rather cryptic. They are shown here to give the reader an idea of the detailed types of software issues that arise in a complicated system development. The table entries are shaded to indicate issue categories. The main focus for Build 3.0 was algorithm enhancements. During Build 3.1, the focus was on robustness. Not shown in the tables is the supporting material, which included testing requirements, estimated implementation effort and time, and an impact assessment. The issues were signed off by the issue originator, the implementer, and a verification team member. The last column in each table codes (with an X) the issues completed when this snapshot was taken. The full tracking includes a list of the affected software modules (configuration controlled), as well as sign-off dates.

4.3.3 PSP Software Integration, Optimization, and Verification

The integration of the PSP was carried out first in a smaller system integration laboratory using the 1/4-scale PSP and a copy of the DIQ front-end subsystem. Integration continued in a full-scale SIL

TABLE 4-4
Performance on Key Kernels

Kernel	Per SHARC FLOP Count (millions)	Measured Execution Time (ms)	Computational Efficiency (%)
Equalization and Pulse Compression	7.73	110.8	87.2
QR Factorization of Jammer-Nulling Training Matrix	0.71	26.6	33.5
Orthogonalization of Jammer-Nulling Adaptive Weights	0.11	3.4	41.3
Application of Jammer-Nulling Adaptive Weights	4.02	62.7	80.3
QR Factorization of Clutter-Nulling Training Matrix	1.74	25.4	85.9
Application of Clutter-Nulling Adaptive Weights	2.49	42.5	73.2
Whitening of Clutter-Nulled Data	0.80	28.0	35.8
80 MFLOP/s peak SHARC 21060 DSP			

using the full-scale PSP and another copy of the DIQ subsystem. Finally, the PSP was moved to the platform and integrated end-to-end with the radar. During platform integration, small-scale SIL was kept operational to handle initial testing of new software. The final codes, once supplied with scaled values for the radar configuration, were transferred to the full-scale SIL. To handle this code scaling, a full set of verification tests with both reduced-scale and full-scale parameters was developed. When a code base passed verification in the small-scale SIL, it was moved to the full-scale lab or (during radar demonstrations) directly to the platform, where the code was tested at full scale using tailored verification tests. Any changes made to the code base during the full-scale testing were sent back to the small-scale SIL, where the code was "patched" and re-verified. In this way, the two code bases were kept synchronized.

Code optimization was an important part of the program. At the outset, computationally intensive kernels were identified. These kernels were coded and hand-optimized during the early spirals. Table 4-4 shows the key kernels and the ultimate efficiencies achieved on the full-scale PSP. These kernels were first coded and tested on the development system. They were ported to the 1/4-scale system, where they were hand-optimized to take advantage of the SHARC instruction set and the SHARC on-chip memory. Parallel versions of the QR factorization routine were developed and the optimum number of processor nodes (i.e., the number of nodes giving the maximum speedup) was determined. During the subsequent builds, performance optimizations continued, with the most intense optimization efforts being carried out during the scaling spirals.

Earlier in this discussion, for the sake of clarity, the scaling spirals were depicted as occurring sequentially. In fact, to help expedite the integration schedule, these spirals were overlapped in time and integrated code was then transitioned to the radar for further integration and verification. The overlap is depicted in Figure 4-15. The first phase of each scaling spiral consisted of the initial scaling step, in which the system was booted, the application was downloaded, and the basic set of start-up tests was run; in the second phase, the scaled system was verified for correct algorithm functionality; during the final phase real-time performance was the major focus. It was during this last phase that extensive code modifications were carried out at several levels. For example, the movement of data and code into and out of internal SHARC memory was reviewed and carefully orchestrated to minimize external memory accesses; source-level codes were optimized by techniques such as code unrolling; data were rearranged for efficient inner loop access; and handcrafted assembly-level vector-vector multiply routines were created. After optimization in the full-scale SIL, the code base was delivered to the platform, where it underwent platform-level integration and verification. The SIL had a comprehensive set of external hardware that allowed the processor to be interfaced with other platform components and then tested.

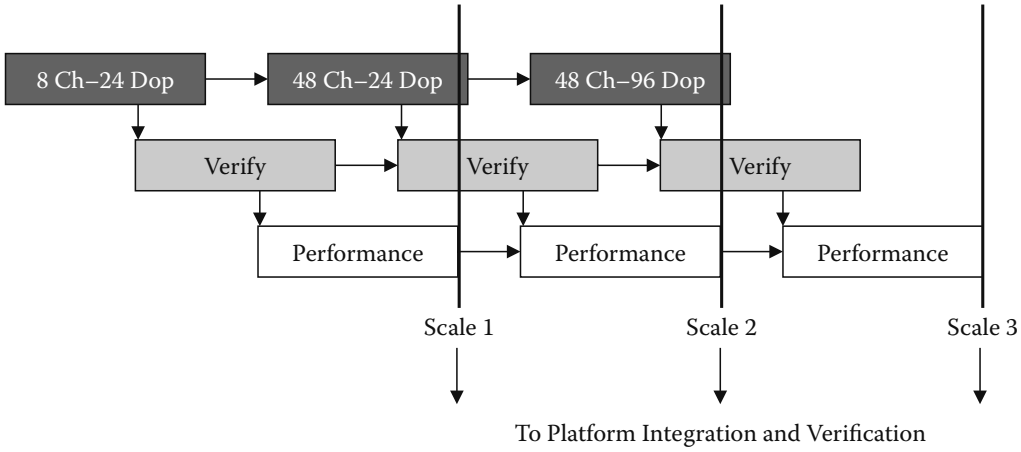


FIGURE 4-15 System integration laboratory scaling phases.

With such a hectic and complex development, tight configuration management was a necessity. The configuration management system tracked code updates and allowed the integration team to develop and manage multiple system configurations. Each subsystem program was tracked through incremental milestones. The major milestones were designated as

(a) *developmental*, at which point the code compiled successfully and was in the process of being tested at the unit level;
(b) *unit-tested*, at which point the code had passed unit-test criteria and was in the process of being integrated with other subsystems;
(c) *integrated*, at which point the code had completed verification; and
(d) *accepted*, at which point the overall subsystem code base had been accepted by an independent application verification team and had been signed off by the program manager and the integration lead.

A version-control system was used to keep track of the different code bases, and regression testing was carried out each day so that problems were detected early and were readily correlated with the particular phase and subphases in progress.

During the optimization phases, the performance of each subsystem was measured on canonical datasets, performance issues were identified, and the code was scrutinized for optimization opportunities. Some of the techniques applied included code unrolling (in which an inner loop is replicated a number of times to reduce the overhead associated with checking loop variables), program cache management (in which, for example, a set of routines was frozen in program cache when it was observed the code would be re-executed in order), and data cache management (in which, for example, data that were used sequentially by a set of routines were kept in cache until the last routine completed, only then allowing the system to copy the results back to main memory). Figure 4-16 shows a snapshot of the optimization performance figures for the 48 Doppler and the 96 Doppler (full-scale) systems, compared to the real-time requirement. At this point in the development, the adaptive beamforming and detection subsystems were still slower than real time, so these two subsystems became the focus of subsequent optimizations.

For example, once the ABS was scaled to the full 96 Doppler configuration (in the final scaling spiral), a disturbing communication bottleneck was identified, as shown in Figure 4-17. The full-scale data cube processed by the ABS consisted of 792 range gates for each of 48 channels for 96 Doppler bins. When the data were transported from the DFS to the ABS, the cube had to be corner-turned so that each ABS node received all of the range gates for all of the channels for a single Doppler bin. This communication step severely loaded the interchassis communication system. At first, the CPI data cubes were streamed through the system at slower than the full real-time rate. Full real-time perfor-

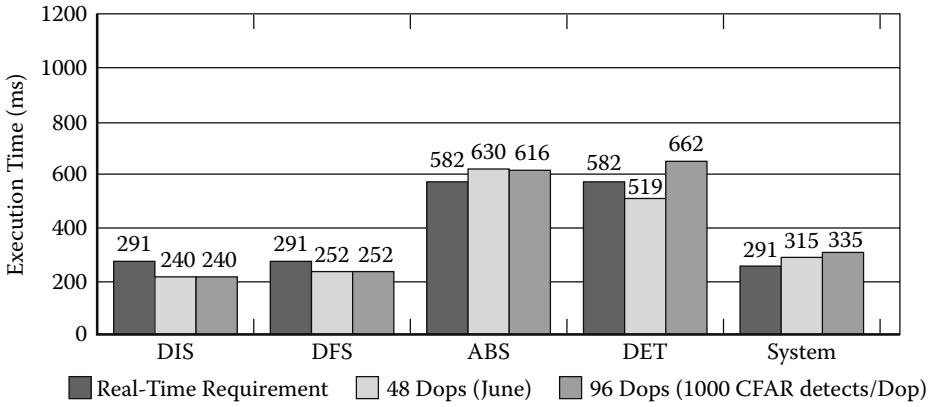


FIGURE 4-16 Performance measurements. Achieving real-time performance in the PSP was a significant challenge. The chart shows one of the many real-time performance assessments that were used during the development to focus optimization efforts. In this example, the DIS and DFS subsystems were measured at better than real-time performance for the reduced-scale (48 Doppler) system and the full-scale (96 Doppler) system. The ABS and the DET were still not at the real-time threshold at this point in the development.

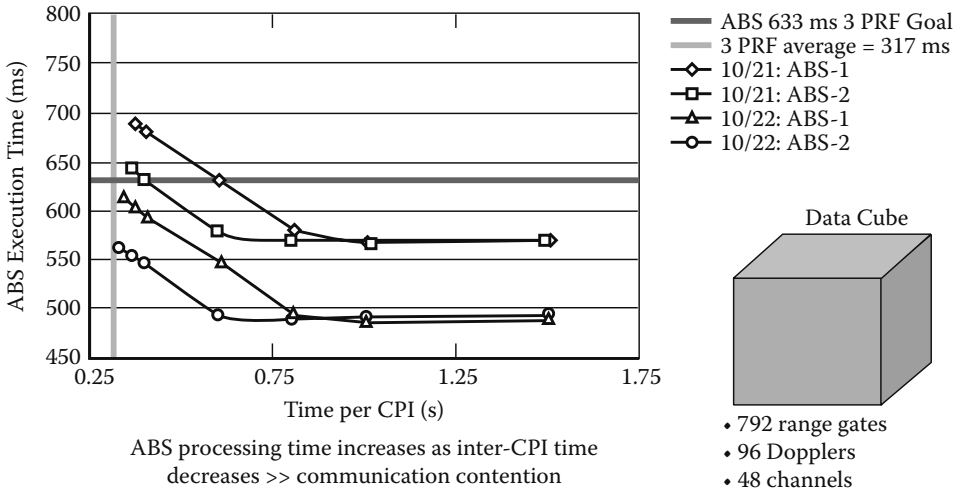


FIGURE 4-17 ABS real-time performance.

mance required the transmission of a data cube every 317 milliseconds (on average). As the inter-CPI time was reduced, a point was reached where the communication system became overloaded and the end-to-end computation time in the ABS began to rise linearly with decreasing inter-CPI time. Thus, although the ABS version that was tested on 10/21 met the real-time goal (shown for the nominal schedule of three CPIs in a repeating sequence) when operated in isolation, when it was operated in the full-scale system with CPIs arriving at the full rate, the execution time climbed to an unacceptable level. Fortunately, the optimization team had been working on a new version of the adaptive weight computation that was significantly faster. The optimized version made more efficient use of internal SHARC memory, thereby saving significant time in off-chip memory accesses. Within one day, the new version was installed and the full-rate CPI case met the end-to-end performance goal with a few milliseconds to spare. The variance in these measurements was determined to be acceptably small by running several 24-hour stress tests. (The stress test also uncovered some memory leaks*, which took time to track down but were fixed prior to the completion of the build.)

* A memory leak is a coding bug that causes the software program to overwrite the end of an array or structure, so that, in a sense, the data "leaks" into memory locations that it should not occupy. The leak can be largely innocuous until a critical data or control item is overwritten, and then the result can be catastrophic. Sometimes, hours or even days of execution are required to uncover these sorts of bugs.

TABLE 4-5
PS Integration Test Matrix (16 and 48 Channels; 48 Dopplers)

Test Item	16-Channel Data Cube	48-Channel Data Cube	Stress Test (full size)	Real Time (3 of 6 CPIs)
DIS (Internal test driver)				
DIS (External interface)				
TS				
DFS (Surveillance)				
DFS (Equalization)				
DFS (Diagnostics)				
Jammer Nulling (part of ABF)				
Clutter Nulling (part of ABF)				
DET				
Overall System				

Key: Heavy shading = partial; light shading = done. 96% completed: 9/17/99.

The verification of the PSP for each build spiral was tracked for each subsystem and for the end-to-end system. Table 4-5 shows a typical snapshot of the top-level verification test matrix for the build that scaled the processor to 48 channels and 48 Dopplers. At this point in the build, the test items had been completed on the canonical set of input data cubes, but the 24-hour stress tests had not been completed for the DFS diagnostic node, and the DFS equalization and diagnostic modes had not been verified for full real-time operation. The DFS diagnostic mode was used to test the performance of the analog components in the radar front-end. The DFS equalization mode computed coefficients for a linear filter that was used to match the end-to-end transfer function of each receiver channel prior to adaptive beamforming. Equalization was needed to achieve the maximum benefit from the jammer-nulling processing that was carried out in the ABF subsystem. The tests verifying these modes were completed in the next month.

The PSP functional verification procedure involved a thorough set of tests that compared the output of the PSP to the output of a MATLAB executable specification. Figure 4-18 shows the general approach. Datasets were read by the MATLAB code and processed through a series of steps. The same datasets were input into the DIS and processed through the PSP subsystems. The results at each processing step for both the MATLAB code and the real-time embedded code were written out to results files. The results were compared and the relative errors between the MATLAB and real-time codes were computed. Note that the relative error at a particular step reflected the accumulated effect of all previous steps. The verification teams analyzed the relative errors to determine if the real-time computation was correct and within tolerances. The MATLAB code executed in double precision, so, in general, the differences were expected to be on the order of the full precision of a single-precision floating-point word. However, since the errors accumulated, it was important to evaluate the signal processing in the context of the overall processing chain to verify the correct functionality and acceptable precision of the end-to-end real-time algorithm. Figure 4-19 shows the plotted contents of an example PSP results file. In this example, the output is the lower triangular matrix (L-matrix) in the LQ decomposition of the training matrix used in the jammer-nulling adaptive-weight computation. The relative difference between this computation and the equivalent computation performed by the MATLAB specification code is shown in Figure 4-20. The errors are on the order of 10^{-6}, which is commensurate with the precision of the single-precision computation. The end-to-end real-time algorithm verification consisted of over 100 such verification tests for several datasets for each radar mode.

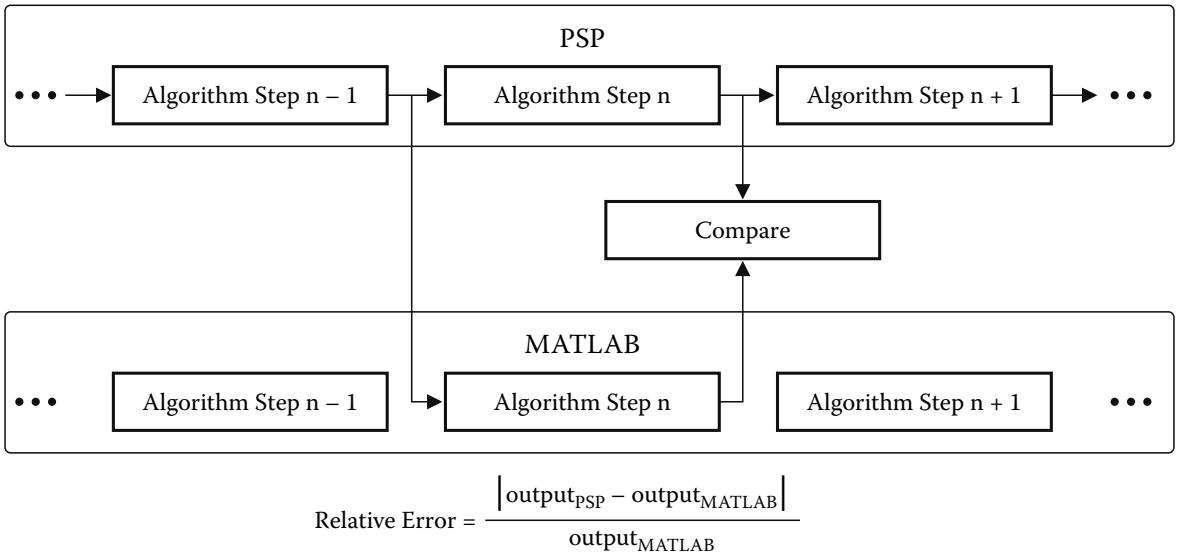
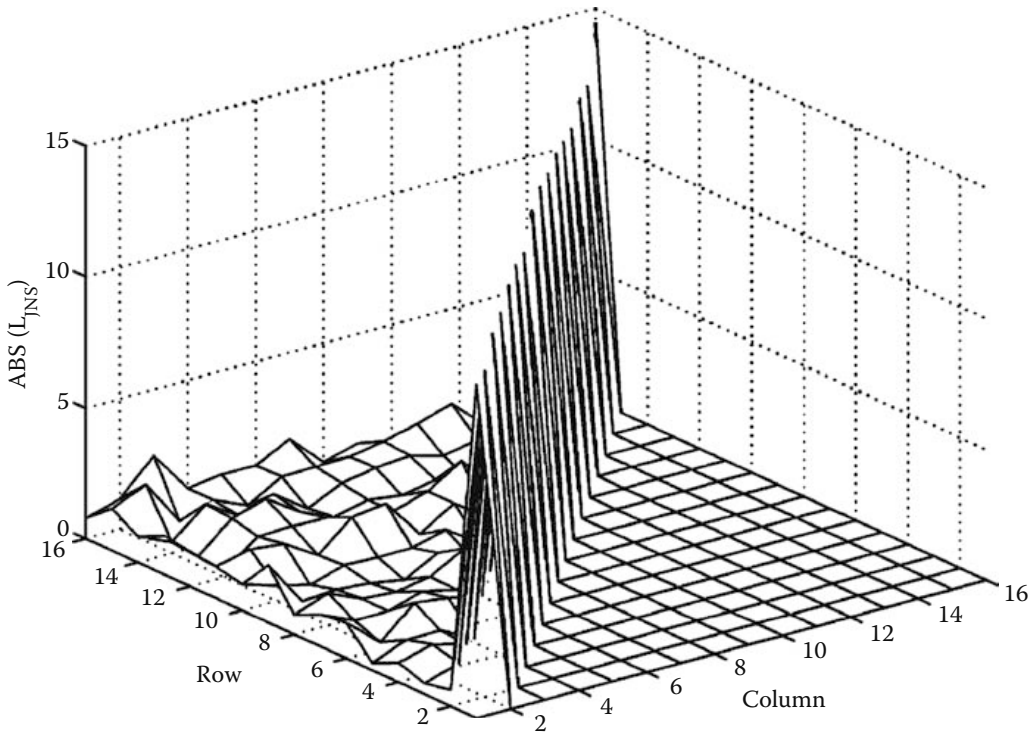


FIGURE 4-18 Verification test procedure. The same dataset is input into the PSP and the MATLAB code. The output of the PSP at each algorithmic step is compared to the equivalent output of the MATLAB. The relative error between the two computations is calculated and evaluated by the verification engineer.



ABS Step 2 Parameters

Parameter	Value
Number of Channels	16
Number of Jammer-Nulling Training Samples	171
Jammer-Nulling Diagonal Loading Level	13 dB

FIGURE 4-19 Example verification computation. The ABS step 2 computation is the calculation of an LQ matrix factorization of a training dataset. Shown here is the magnitude of the entries in the lower diagonal matrix (L-matrix) computed by the PSP code on a controlled test dataset.

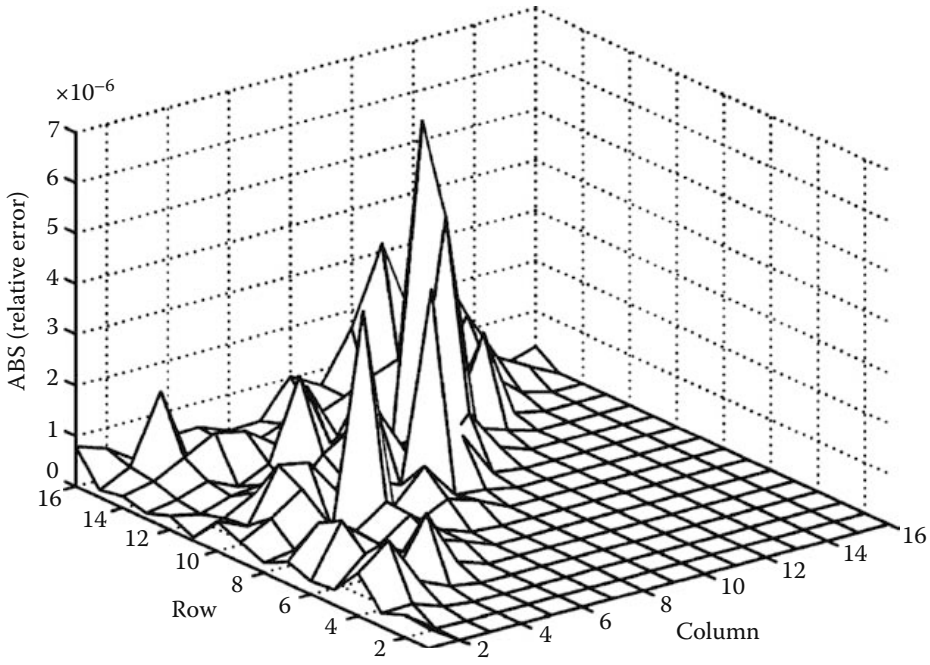


FIGURE 4-20 Example verification result. The L-matrix computed in the PSP step 2 computations is compared point-wise to the equivalent computation in MATLAB code (using the same input dataset). The relative error is plotted here. Note that the errors are on the order of the single-precision arithmetic performed in the PSP.

4.4 TRENDS

HPEC developers today have an ever-increasing repertoire of technology options. Full-custom VLSI ASICs, standard-cell ASICs, FPGAs, DSPs, microcontrollers, and multicore microprocessor units (MPUs) all have roles to play in HPEC systems. Form-factor constraints are becoming increasingly stressful as powerful HPEC systems are being embedded in small platforms such as satellites, unmanned aerial vehicles, and portable communications systems. At the same time, sensing and communication hardware continue to increase in bandwidth, dynamic range, and number of elements. Radars, for example, are beginning to employ active electronic scanning arrays (AESAs) with 1000s of elements. Radar waveforms with instantaneous bandwidths in the 100s of MHz regime are finding use in higher-resolution applications. Designs for analog-to-digital converters that will support over 5 GHz of bandwidth with over 8 bits of dynamic range are on the drawing table. Electro-optical sensing is growing at least as fast, as large CCD video arrays and increasingly capable laser detection and ranging (LADAR) applications emerge. Similar trends are evident in the communication industry. Digital circuitry is replacing analog circuitry in the receiver chain, so that HPEC systems are continuing to encroach on domains traditionally reserved for the analog system designers. Algorithms are becoming more sophisticated, with knowledge-based processing being integrated with more traditional signal, image, and communication processing. Sensors integrated with decision-support and data-fusion applications, using both wireline and wireless communications, are becoming a major research and development area.

As the capability and complexity of HPEC systems continue to grow, development methods and management methods are evolving to keep pace. The complex interplay of form-factor, algorithm, and processor technology choices is motivating the use of more tightly integrated co-design methods. Algorithm-hardware co-design techniques emphasize the rapid navigation of the trade space between algorithm designs and custom (or FPGA) hardware implementations. For example, tools that allow rapid explorations of suitable arithmetic systems (floating point, fixed point, block floating point, etc.) and arithmetic precision and that also predict implementation costs (chip real-estate, power consumption, throughput, etc.) are being developed. Techniques that map from high-level

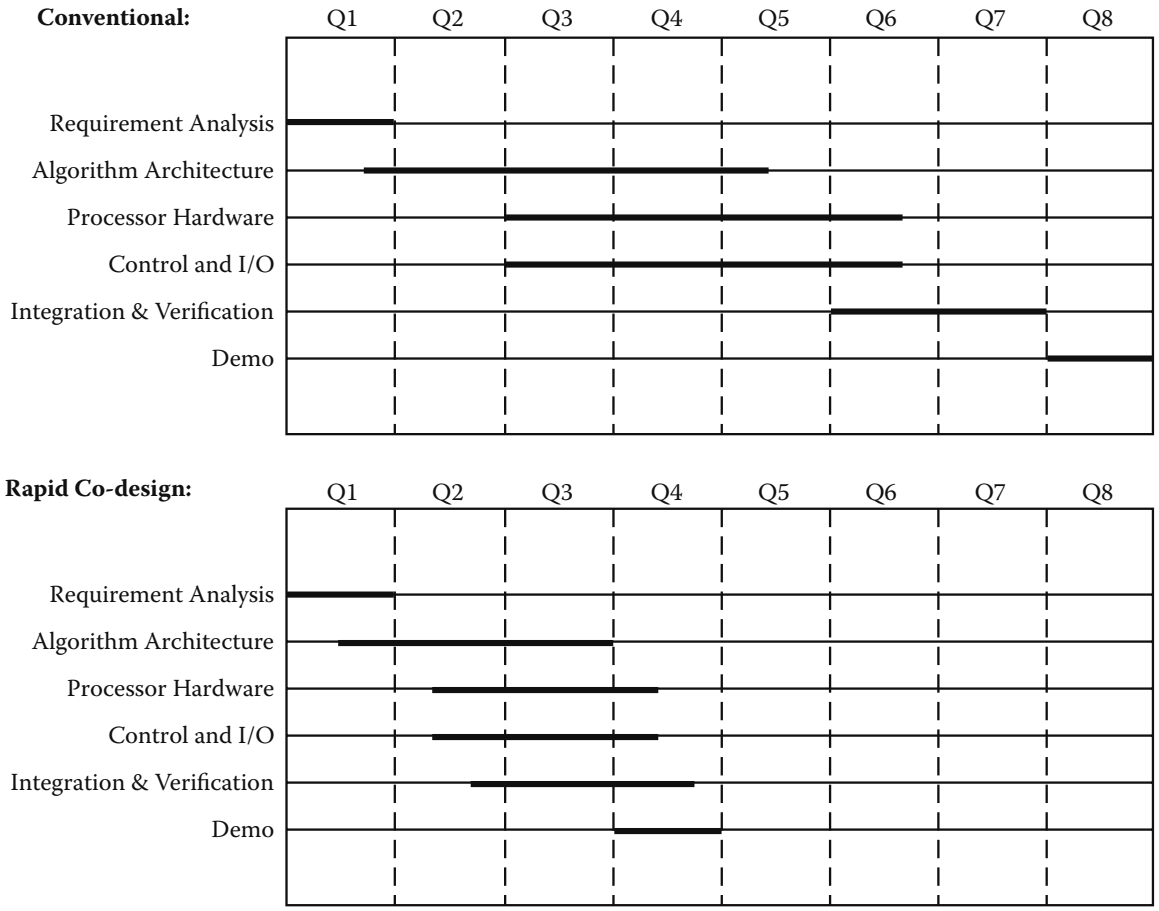


FIGURE 4-21 Rapid co-design of HPEC hardware.

prototyping languages (such as Simulink) to FPGA implementations are emerging. Recent trends are the development of fully integrated hardware design and algorithm exploration development environments. The benefits expected as these tools mature are shown in Figure 4-21 for a hypothetical HPEC system development. As the figure shows, co-design can allow the overlap of the algorithm and architecture design phases. Integration can begin sooner and will take less time since many of the integration issues are addressed earlier in the process.

For hybrid HPEC systems, hardware-software co-design techniques are emerging. These co-design approaches start by developing an executable model of the HPEC system. The mapping or binding of functionality to hardware and software is carried out using the model, permitting the design trade-offs to be explored early and the commitment to particular hardware to be made later in the development cycle. This approach leads to more effective architectures, but just as importantly, the integration of the hardware with the software proceeds more smoothly since the interfaces have already been verified via simulation. If hardware models are available, different technology choices can be explored via simulation to find an optimal assignment. Since much of the design space can be explored more rapidly (when proper models exist), this approach has the potential to dramatically shorten the overall development time through each development spiral.

Another, related, trend is the use of model-driven architectures (MDAs) and graphical modeling languages such as the Universal Modeling Language (UML) to specify software designs. The UML model can be executed to verify correct function; the overall architecture and interaction of the components can be explored; and then the actual code (for example, C++) can be generated and integrated into the HPEC system. Although this approach has not been widely adopted in HPEC designs, partly due to their complexity and the need to produce highly efficient codes, model-driven architecture design is expected to find application as it continues to mature.

Software middleware technologies, pioneered in the late 1990s (for example, STAPL discussed in this chapter), are becoming indispensable infrastructure components for modern HPEC systems. Most processor vendors provide their own middleware, and the standard Vector Signal Image Processing Library (VSIPL) is now widely available. In 2005, a parallel, C++ variant of the VSIPL standard (VSIPL++) was developed and made available to the HPEC community [http://www.vsipl.org]. Middleware libraries that support parallelism are particularly important since programmable HPEC systems are invariably parallel computers. The parallel VSIPL++ library gives HPEC developers a parallel, scalable, and efficient set of signal and image processing objects and functions that support the rapid development of embedded sensors and communication systems. Using libraries of this sort can reduce the size of an application code significantly. For example, a prototype variant of VSIPL++ called the Parallel Vector Library has been used to reduce application code by as much as a factor of three for radar and sonar systems.

Open system architectures (OSAs) are also being developed for HPEC product lines. The advantages of OSAs include ease of technology refresh; interoperable, plug-and-play components; modular reuse; easier insertion of intellectual property (e.g., advanced algorithms and components); and the ability to foster competition. For example, the Radar Open System Architecture (ROSA) was developed circa 2000 at MIT Lincoln Laboratory to provide a reference architecture, modular systems, common hardware, and reusable and configurable real-time software for ground-based radars. ROSA has had a revolutionary impact on the development and maintenance of ground-based radars. For example, the entire radar suite at the Kwajalein Missile Range was upgraded with ROSA technology. Five radars that previously were implemented with custom technology were refurbished using common processing and control hardware. Common hardware, nearly 90% of which was commercial off-the-shelf componentry, and a configurable common software base were used across the five systems. Currently, ROSA is undergoing an upgrade to apply it to phased-array radars, both ground-based and airborne.

Integrated development environments (IDEs) are becoming increasingly powerful aids in developing software. IDEs provide programmers with a suite of interoperable tools, typically including source code editors, compilers, build-automation tools, debuggers, and test automation tools. Newer IDEs integrate with version-control systems and also provide tools to track test coverage. For object-oriented languages, such as C++ and Java, a modern IDE will also include class browser, an object inspector, and a class-hierarchy diagram. At the same time, the environment in which programmers develop their software has evolved from command-line tools, to powerful graphical editors, to IDEs. Modern IDEs include Eclipse, Netbeans, IntelliJ, and Visual Studio. Although these IDEs have been developed for network software development, extensions and plug-ins for embedded software development are becoming available.

In conclusion, Figure 4-22 depicts the future envisioned for HPEC development, shown in the context of a high performance radar signal processor such as the one covered in the case study in this chapter. Future HPEC development is anticipated as a refinement of the classical spiral model to include a much more tightly integrated hardware-software co-design methodology and toolset. A high-level design specification that covers both the hardware and software components will be used. The allocation to hardware or software will be carried out using analysis and simulation tools. Once this allocation has been achieved, a more detailed design will be carried out and code will be generated, either from a modeling language such as UML or a combination of UML and traditional coding practices. The code will use a high-level library such as VSIPL++ and domain-specific reusable components. VSIPL++ components will rely on kernels that have been optimized for the target programmable hardware.

On the hardware side, the high-level design will be used in conjunction with parameter-based hardware module-generators to carry out the detailed design of the hardware. The hardware modules will be chosen from existing intellectual property (IP) cores where possible, and these IP cores will be automatically generated for the parameter range required by the application. Other components will still require the traditional design and generation, but once they have been developed,

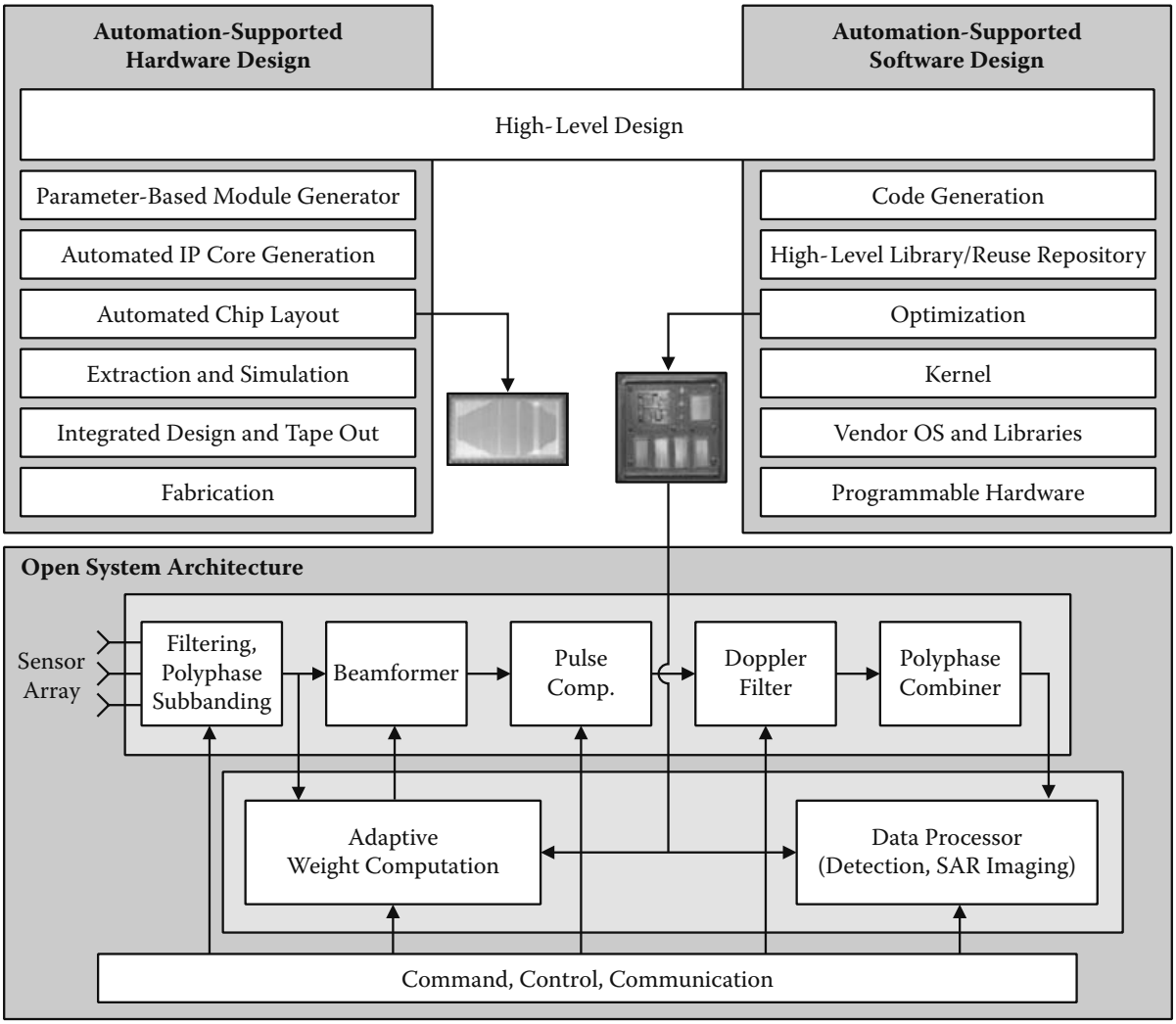


FIGURE 4-22 Vision of future development of embedded architectures. A high-level co-design environment is envisioned that will allow the system design to be allocated between hardware and software in a seamless and reconfigurable manner. Below the high-level design specification are toolsets that help to automate the detailed design, fabrication (or compilation), and testing of the HPEC system. The architecture interfaces are specified in the high-level design to enforce an open-architecture design approach that allows components to be upgraded in a modular, isolated manner that reduces system-level impact.

they can also be included in the overall IP library for future reuse. The next steps in the process will involve greater support for automatic layout of the chips and boards, extraction and detailed simulation, and finally tape out, fabrication, and integration. With a more integrated and automated process and tools as depicted, the spiral development and management process can be applied as before. However, the process will encourage reuse of both hardware and software components, as well as the creation of new reusable components. Domain-specific reuse repositories for both hardware (IP cores) and software (domain-specific libraries) will thereby be developed and optimized, permitting much more rapid system development, and will significantly mitigate cost, schedule, and technical risks in future, challenging HPEC developments.

REFERENCES

Boehm, B.W. 1988. A spiral model of software development and enhancement. *IEEE Computer* 21(5): 61–72.

Ward, J. 1994. *Space-Time Adaptive Processing for Airborne Radar Submitter.* MIT Lincoln Laboratory Technical Report 1015, Revision 1. Lexington, Mass.: MIT Lincoln Laboratory.