A Development Method for Reconfigurable Hardware Systems With Real-Time Wavelet Image Compression Application

Jeffrey A. Shafer
Waleed W. Smari, Ph.D.
Frank A. Scarpino, Ph.D.

Department of Electrical and Computer Engineering
University of Dayton
300 College Park Avenue, Dayton, Ohio 45469-0226

Keywords:

Reconfigurable Computing, Development Model, Waterfall Model, Spiral Model, Image Compression, Wavelet Transforms, FPGA

Abstract

The primary emphasis of this paper is the exploration of a method of development that combines high-level prototyping of algorithms with low-level hardware development on FPGAs. The method of development was applied to implement a wavelet image compression algorithm as a real-world test. This development method is based on standard software-engineering development models because of the similarities in programming hardware via hardware description languages and traditional programming of general-purpose computers. This method shows promise as a technique to implement any computationally intensive algorithm in hardware, and can both improve the quality of the end product and reduce development time.

1. Introduction

System development processes have two primary functions. First, they specify the order in which distinct project stages are accomplished. Second, they specify the criteria to transition from one development stage to the next. Or, as Barry Boehm, creator of the spiral model, wrote, they answer the questions “What shall we do next?” and “How long shall we continue to do it?” Such processes, when formalized and enforced, prevent developers from jumping into coding without having a clear and documented understanding of what system is to be developed. Although ad-hoc design and coding may succeed for small, clearly defined projects, such a method is fraught with risks on real-life systems where the full complexity and client requirements of a project may not be initially evident.

The development process proposed here has significant potential by enabling high-level software languages to be used on general-purpose computers during a system prototyping phase while still allowing for the end result to be implemented in hardware for faster execution. The use of high-level rapid prototyping tools early in the development process can be of immense value in any research or development program where deciding what system to implement is as much the goal as actually producing a working implementation. If the high-level system is properly structured and designed, then the prototype can be re-implemented in hardware through a language such as VHDL.

To validate this development model, it is applied to a real-world real-time wavelet image compression system. Image and video compression is an integral part of today’s digital environment. The compression process must be done within a few tens of milliseconds or less for the process to occur in “real-time.” Today’s general-purpose computers are approaching the computational power necessary to perform real-time video compression in software. The wealth of rapid prototyping tools available for the general-purpose computer greatly accelerates algorithm development and testing, even if such tools limit overall performance. New algorithms, however, almost always require more computational power in order to increase the compression ratio while maintaining high-quality visual results, keeping the tradeoff between compression ratio and compression speed a perennial issue [1].

In contrast to the software-based approach, reconfigurable systems based on field-programmable gate arrays (FPGAs) can be employed to utilize higher-quality algorithms with greater computational needs and still deliver real-time results [4]. By taking advantage of the parallelisms available in hardware design, such systems can even compress multiple input streams simultaneously, limited primarily by the bus system providing the raw input data and saving the compressed output streams [8]. While such dedicated hardware can substantially accelerate the compression process, system development is significantly more complex. This not only increases the cost of the end product, but also makes it more likely for software-based solutions under simultaneous development to close the performance gap as the current state-of-the-art computer performance improves. Although higher-level hardware description languages like VHDL and AHDL alleviate the tedium of manual chip layout in producing dedicated hardware systems, they cannot come close to approaching the wealth of rapid-prototyping tools available in software [6]. Thus, what is needed is some way to take advantage of high-level software environments early in development while still producing a high-performance hardware implementation as the final product.

This paper will outline historical software development models in Section 2, propose a new method of development for reconfigurable hardware systems in Section 3 and 4, and apply it to a real-world image compression problem in Section 5.

2. Historical Development Models

The system development process proposed here contains some elements from the widely publicized “waterfall model” in computer-science fields, where the top-most stage is completed first and fed into the subsequent lower stage as the starting product. This standardized model is shown in Figure 1.

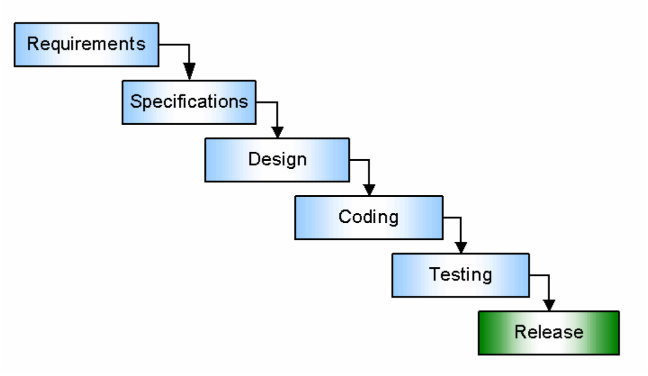

Figure 1: Waterfall Development Model [5]

In the standard waterfall model, the initial requirements and specifications stages serve to identify the product that is to be created. In the design phase, the key modules of this product and the relationships between the modules are determined. Finally, in the remaining three stages, the product is actually coded, tested, and released. Each stage is completed in series. Ideally, the design phase would be completely finished, properly documented, and then handed to the coders who use it in the coding phase.

In the waterfall model, the earlier in the process that problems are identified, the cheaper and easier the solution will be [5]. It will always be easier to rework a problem in the requirements phase than before the program architecture has been designed, just as it will also be cheaper to rework a problem in the design phase before the program has been coded. Further, although it will always be easier to rectify a problem in the testing phase before the program has been released, it would have been far cheaper to rectify that problem in the design phase. Because of this increasing-costs hierarchy, a formal method’s emphasis on complete system documentation before coding has the potential to save significant resources by encountering and fixing problems early on.

Recently, modified software development models have been proposed with the purpose of, among others, making product testing a continual part of the development process instead of an ending step, or making accommodations for program development on high-risk projects [5]. High-risk projects are defined as projects where success is uncertain upon starting, as the problem to be solved (such as an air traffic control system) may be too unwieldy or complex to program effectively, or where the development resources (i.e. money) required for a successful implementation cannot be initially estimated with any reasonable accuracy.

One new development model that allows for high-risk projects is the spiral model as developed by Barry Boehm. Rather than being document-centric like the waterfall model, it relies heavily on prototypes to segment its various stages and determine when the project should advance from one stage to another [2]. The spiral model utilizes an overall theme of evolutionary development with prototyping, but depends on the waterfall model to actually complete each prototype. Note that it is referred to as being “evolutionary” in process rather than “incremental.” In an incremental development process, the final end product is known at the beginning of development, and each development stage completes a known project module that was identified when the project was initiated. Rather, in evolutionary development, the project starts by addressing the design feature that carries the highest risk of failure. As risks are surmounted, new risks are unveiled and solved with each turn of the spiral. Thus, the exact course the eventual design solution will take is not known when the project is started.

As shown in Figure 2, the spiral model gets its name because the development process starts at the center of the spiral and proceeds radially outward in a clockwise direction. As more turns of the spiral are completed, each loop builds upon the results of the previous loops as the project evolves towards completion.

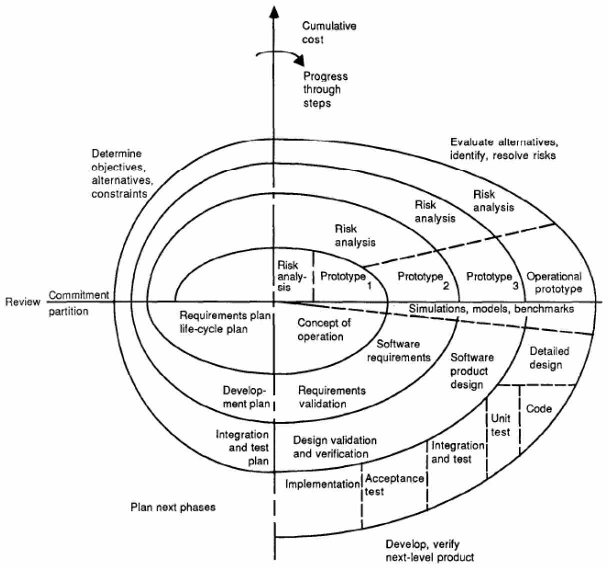

Figure 2: Spiral Model of Software Development [2]

As Boehm describes, the radial dimension of the model represents the total project cost to date, while the angular dimension of the model represents the progress made in completing the current stage of the cycle. Each turn of the spiral model is a miniature waterfall model, starting with the identification of the project objectives. Because the spiral model uses a top-down approach, the highest-priority features are implemented first. The possible methods of implementation are studied, as well as the constraints imposed by financial resources, time, or the end-user. From this study, significant risks faced in implementing these features are identified. To address these risks, prototypes are built, tested, and presented to the end-user to determine if the project risks have been surmounted. If they have, the model proceeds to the next turn of the spiral to implement the next most important feature set and address their inherent risks. If, however, the risks have not been surmounted, the current spiral must either be repeated with a new prototype and different design, or the project must be downscaled to a more solvable complexity and size. The spiral model ends when either the project is finished and successfully meets its goals, or when the project is canceled after a prototype fails a key design test.

One key benefit of the spiral model is that its evolutionary "top-down" approach means that the most significant and far-reaching product implementation risks can be identified and resolved first. If the key risks are surmounted, the project can continue and implement all of the "detail work," confident that such work is not being done in vain. This is in contrast to the waterfall model where a risk is not truly surmounted until the coding stage. If this stage fails, all of the work done up to this point (requirements, specifications, and design) is at risk and must be reworked. With the spiral model, however, if a prototype fails to surmount a risk, only that turn of the spiral is at risk and must be redone, while all earlier turns (presumably solving more important problems) are still intact. Even at the worst case, where the project must be abandoned due to unanticipated complexity or expense, the spiral model will determine this fate far sooner than the waterfall model, saving time and money. It is this theme of allowance for risky projects, and its solution through the use of prototypes and evolutionary development, that is incorporated into the new hardware development method presented later in this chapter.

3. Proposed Development Model

While such models are very useful in the software development world, they have several key shortcomings when applied to hardware development, particularly in reconfigurable devices such as FPGAs. If you applied the traditional waterfall model to hardware development, you would start off by specifying the algorithm ("requirements / specifications phase") and then lay out the broad architecture of the FPGA system to be implemented ("design phase"). Finally, the system would be realized in hardware by programming in a language such as VHDL ("coding phase"). Unfortunately, while driving straight to the hardware may be possible for small development projects, it presents significant risks for larger real-world projects. What if the algorithms implemented do not produce the correct mathematical results? What if the algorithms implemented to not produce as "good" of results (via the applicable quality benchmarks) as desired? What if the algorithms implemented are too slow on hardware to satisfy the performance requirements? Finding out after hardware implementation that the algorithm is deficient in one (or more) of the above ways is hardly useful. Although reconfigurable computing (as compared to an ASIC) does provide the flexibility to rectify these problems, a significant amount of time and money has still been wasted in the low-level hardware implementation that can only be partially recovered if the existing algorithm can be "tweaked" into correctness.

With the risks inherent in hardware development, then, the spiral model would seem to be a better choice. It too has drawbacks, however. Recall that the spiral model as written attempts an evolutionary design flow by attacking the highest-risk items first. In a reconfigurable computing project, speed, device capacity, and memory architecture considerations often pose the largest risks in successfully completing the project. To this end, the spiral model would attempt a basic hardware implementation early in the project cycle, and obtain client feedback based on that prototype. Unfortunately, even a basic hardware prototype is far more time consuming than a software prototype in a high-level language, so if that spiral fails, significant time will have been lost. Similarly, if the client makes many changes to the hardware prototype they are first shown, it is very possible that significant architectural changes will need to be made to the prototype before it can be incorporated into the next evolutionary stage of the project.

The system development process described here attempts to surmount the previously mentioned risks in hardware development by attacking them early in the process through the use of software prototypes. Recall that the earlier such risks are identified and solved, the lower the total development cost. Such software prototypes also allow for evolutionary development, as the client can see a working product and make any necessary changes before the product is implemented in hardware. This process assumes that the end product should be a hardware implementation, although "escape routes" are available at the end of certain stages in case a hardware implementation proves to be either unworkable (due to complexity or expense) or unnecessary (due to rapid performance increases of general purpose computers). This new development process contains four key stages: *Concept Development*, *Feasibility Study*, *Hardware Emulation*, and *Hardware Implementation*, as shown in Figure 3.

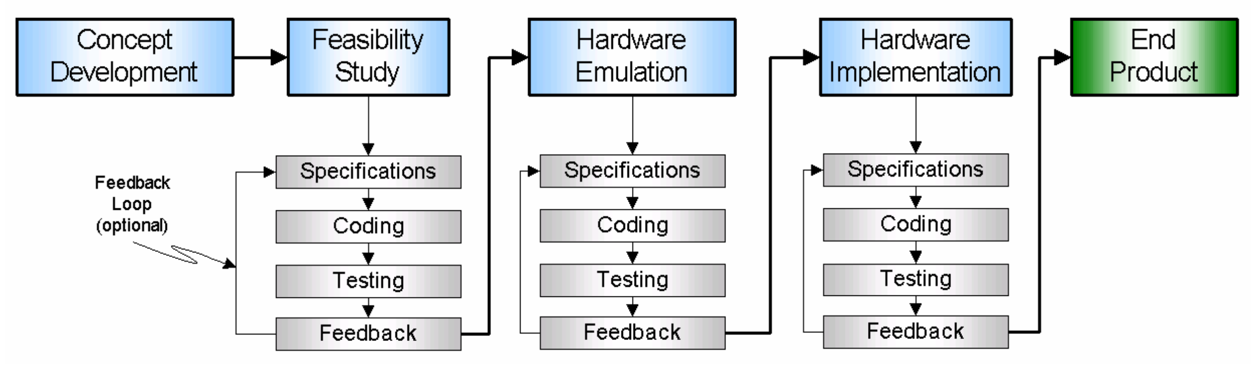

Figure 3: System Development Process for Reconfigurable Computing

Inherent in the middle three stages of the development process are elements from the standard waterfall model, as denoted by the three grayscale sections beneath the feasibility study, hardware emulation, and hardware design sections. These "sub-processes" serve to highlight how a design team might go about accomplishing, for example, the feasibility study, through a systematic process of defining specifications, coding, testing, and soliciting feedback from clients. Note that these plan-based sub-processes work best when the requirements can be defined in advance and remain relatively stable, changing only about one percent a month [3]. Because of this, the waterfall model may not be strictly applied to the feasibility study stage, where requirements may not be fully known in advance. By the time the hardware stage is reached, however, a more rigid methodology like the waterfall model should be used because the system requirements should be nearly fixed.

Evolutionary development is incorporated into the model via the use of the optional feedback loops shown below the three middle stages of the development process. Implemented with the same purpose as the spiral model, these loops allow developers in these three stages to attack the most risky element of the design first. For example, in the feasibility study stage, the riskiest element of the project would be examined first. Once that risk has been overcome, the project is reevaluated and the feasibility study continues on to the next riskiest element. The solution to this project element may have been influenced in some unanticipated way by the result from the previous solution. By dynamically building upon previous results and allowing for solutions which were not anticipated when the project was initiated, the feedback loops are able to go one step beyond the simple incremental development discussed previously and achieve true evolutionary development. If a risk proves insurmountable, the project can be aborted or redesigned with the minimum amount of lost resources. Or, once all the risks of the feasibility study are addressed, the team can progress to the next step of the development process.

4. Development Model: An In-Depth Look

4.1 – Concept Development

The first stage of the system development model is the *Concept Development* stage. In this stage, the problem to be solved is identified and various algorithms leading to a solution are proposed. It is here that the problem is analyzed to determine if a reconfigurable hardware implementation is necessary. Does the performance requirements of the problem justify the development time and expense of FPGA design? Can a general-purpose computer be used instead? If a general-purpose machine can be used, the system development model is aborted, and a purely software-based model (such as the waterfall or spiral model) is used instead. If, however, the computational and timing needs of the proposed algorithms cannot be satisfied by a general-purpose machine, the system development model continues to the feasibility stage.

4.2 – Feasibility Study

In the previous stage, possible algorithmic solutions to a problem have been identified. Here, these solutions are verified for algorithmic correctness, not performance. In essence, this stage is a high-level prototype of the system to be developed. All algorithms to be evaluated are implemented in a high level programming language such as C++, Java, or a more mathematically oriented package such as Matlab. This code is written to be easily designed and modified for rapid testing and debugging, and is not optimized for performance in any way that would obscure the algorithmic intent of the code.

Note that the "feasibility" discussed in this section is primarily focused on technical feasibility and not economic or legal feasibility. Briefly, technical feasibility is concerned with whether the project can successfully be developed given the available resources and technology. In contrast, economic feasibility is concerned with a cost-benefits analysis between the development costs and the final projected revenue from the product. Finally, legal feasibility deals with any liability or patent issues that may stem from pursuing development of the product [7]. These last two issues, while highly relevant to product development, are outside the scope of this paper.

Although the eventual goal of the system development process is to design a working hardware implementation of an algorithm, high-level software languages, not low-level hardware description languages, are used for this stage of the process. This is because low-level languages such as VHDL are simply too unwieldy and cumbersome to use when the goal is as much to determine what algorithm should be implemented as performing the actual implementation work. Although the algorithm implementations at this stage should not be optimized (a waste of time since the final goal is a hardware implementation), easily-obtained performance improvements could be taken within various software environments with the goal of making the prototype more usable and enabling demonstrations to the end-user.

Upon completion of the feasibility study phase, the prototype is examined and compared to the specifications. Is the right system being built? Can it still be completed on time and on budget? Have similarly complex systems been implemented on FPGAs before? In addition, the prototype is tested with real-world test data. Do the algorithms satisfy all non-speed related "quality" metrics? Quality metrics envelop both quantifiable (e.g. Are the mathematical results correct? Is the signal-to-noise ratio high?) and non-quantifiable results (eg. Does the audio sound good?). If not, this phase is repeated with new or modified algorithms until the quality of the algorithms is acceptable. In addition, this high-level simulation on a general-purpose computer is examined to determine if it satisfies (or come close to satisfying) the necessary performance benchmarks. If so, an "escape route" is possible at this point, as the high-level prototype may only require small performance optimizations to meet the client's needs, rather than a full hardware implementation.

To complete the feasibility study, the prototype is demonstrated to the client who can test it and require corrections and additions to be made as necessary. This allows the design to evolve towards the final solution while making modifications to the final product is still relatively cheap and easy. This final prototype examination is the most important of the stage because the end-users are often incapable of knowing exactly what system they want, or are unable to express it with any degree of precision. However, when the end-user is shown an actual working prototype, it is much easier for them to determine if the right system is being built. (i.e. the system that fulfills all of their unstated objectives). This is in contrast to what the developers can determine, which is whether the system is being built correctly. (i.e. the system fulfills its initial specifications). Note that this feedback process is only possible if the prototype constructed is a complete (sub)system. If it is purely a mathematical engine or some other dedicated tool, extra software may be needed to "wrap" the algorithm and interpret its results so that it can be easily tested and demonstrated. If this is too time consuming or impossible, then the project engineers must interpret the results of the feasibility study and decide when to move to the next stage of the development process. Regardless, the feasibility study stage should continue until all significant portions of the final system have been prototyped and tested.

4.3 – Hardware Emulator

Once the prototype has been fully examined and tested, a high-level to low-level mapping is begun in the *Hardware Emulator* phase. Here, key algorithms in the existing high-level software implementation are reworked to more accurately simulate real-world hardware constraints. In essence, this stage is a low level prototype of the system to be developed. This prototype is still written in the same language as the feasibility model, however, as there are still significant risks to be addressed before authorizing the development time and expense of the hardware implementation.

In this development stage, multidimensional arrays in the high-level implementation are replaced with single-dimensional fixed-length linearly addressable models of the actual memory available in the destination hardware. This is because many reconfigurable computing systems have fixed memory capacities in multiple banks, and it must be determined if the algorithm previously modeled in a high-level fashion can fit into a real-world system. If the destination hardware and memory architecture has been previous selected (or mandated), this model should reflect its specifications. Otherwise, the designers make reasonable decisions as to what the specifications and architecture of the destination hardware will be. After completing this stage of the development process, a final list of hardware requirements will be generated which can greatly accelerate the selection of appropriate devices.

Element-to-element linear operators replace vectorized operations, which were utilized in the high-level implementation for speed of execution and algorithmic simplicity. If these operations can be executed concurrently, special comment blocks are placed in the code to relay that information to the hardware designers. High-level mathematical operations such as multiplication or division are replaced (when possible) with simple bit shift operations. If such a replacement is not easily possible (due to fractional results that must be maintained to preserve algorithmic integrity), notes are taken to document the need for floating-point hardware in the final design.

If significant design changes were made between the high-level prototype in the feasibility study stage and the low-level prototype in the hardware emulator, the current prototype can be demonstrated to the client again for further feedback. If, however, the only changes made were done to closely resemble the actual device hardware, it would be redundant to show the client what is, on the surface, the same product.

In the hardware emulator, benchmarks can be gathered which will determine which hardware device to utilize, or whether the specified device will be sufficiently large or fast to accommodate the design. Because the matrices in the high-level simulation have been converted to one-dimensional fixed arrays, it is a simple task in the emulator environment to track the number of memory accesses required to run the algorithm on real-world test data. Similarly, it is straightforward to wrap the algorithm with code in order to count the number of calculations required to complete the test suite, as well as determine the relative frequency of each arithmetic or logical operation. If floating-point calculations must be used, research should be undertaken to determine what pre-built or custom-built libraries could be used in the hardware implementation stage, and what the performance and cost penalties of those options will be.

Based on the memory access and computation statistics, performance predictions about the algorithm's actual performance on real hardware are made. Based on these results, the developers can determine which algorithms, if any, should be implemented in hardware. If no algorithms can satisfy the necessary performance requirements on the specified devices, the development process returns to the concept development stage to research new algorithms in search of an economical solution. But, if an algorithm has been identified as being a likely success on a real-world device package, it is programmed into hardware in the *Hardware Implementation* stage.

4.4 – Hardware Implementation

In the final stage of the system development process, the algorithms are implemented on FPGAs for maximum computational performance and flexibility. At this point, the algorithms have been verified in the high-level simulator for mathematical correctness and any applicable "quality" metrics. In addition, the algorithms have been tested in the low-level simulator to insure that they will fit in the hardware system chosen with regards to memory size, architecture, and computational performance limitations. Thus, the majority of risks inherent in hardware development should have already been addressed in the two system prototypes already created.

For the hardware implementation phase a hardware description language such as VHDL is used. Vectorized operations that were previously converted to linear element-by-element operations in the hardware emulator are now either converted to concurrent operations here (to exploit the parallelisms possible in hardware), or are included as part of finite state machines if sequential operation must be maintained. If necessary, floating-point libraries are utilized to achieve the same mathematical results as produced by the prototypes.

The hardware implementation phase incorporates the same feedback loop structure found in the feasibility study and hardware emulator stages. Thus, as mentioned previously, the highest-risk elements should be addressed first. For example, if the entire design depends upon the implementation of a high performance ALU, that element should be implemented in hardware first, and then examined and tested. If it meets its requirements, the next riskiest element of the implementation stage should begin, since that element may depend or interact with previously unknown attributes of the ALU. Thus, the design evolves towards a final solution based upon knowledge that may not have been available when the project was initiated.

4.5 – Summary

One of the key advantages of this development process is that all algorithm research and development is done in the feasibility study stage through the use of high-level software prototypes. This is because it is significantly faster and cheaper to test out new ideas and build prototypes in software than in a low-level hardware description language. In addition, the majority of client feedback is solicited and applied during this stage, as opposed to later on in the development process where changes become significantly more expensive and time consuming. This is in contrast to the traditional waterfall or spiral models where the client would not see a prototype (or make changes to one) until after at least a partial hardware implementation had been completed. If the project objectives change significantly as a result of knowledge learned when making this prototype, none of the time involved in a hardware implementation will be lost, because that implementation is done later in the development process.

Like the spiral model, most of the far-reaching product implementation risks are examined first in the feasibility study stage of this model. After this stage is complete, all of the algorithms and processes necessary for a successful solution have been verified. Many of the remaining risks, mostly involving whether the algorithms will properly fit on hardware devices, are examined next in the hardware emulation phase. Finally, confident that the key risks have been surmounted, the project can continue and implement all of the "detail work" of hardware development confident that such work is not being done in vain. In addition, evolutionary development like that found in the spiral model is also incorporated into this development process through the use of feedback loops. These loops allow developers to attack the highest-risk project elements at each stage of the process first. Then, subsequent project elements can incorporated knowledge gained from the riskiest project element; information that was likely not known or even considered when the project was initiated.

Because one of the key strengths of reconfigurable computing is the ability of the developers to modify and enhance (or correct) the product once it is in the field, this system development model allows for ongoing maintenance.

As new product objectives are identified, it is a straightforward process to simply repeat the development process shown in Figure 3 on a smaller scale. Because the original high-level and low-level prototypes are still in existence, they can simply be altered at each stage of the process to gain a full understanding of the modifications that need to be made to the hardware implementation.

5. Application of Model to Wavelet Image Compression

The system development process here was applied to develop and implement an optimized integer-based Haar wavelet transform, the Super-Efficient Haar Transform (SEHT) [9]. This optimized transform eliminates the separate row and column transformations inherent in ordinary wavelet image compression algorithms.

The development and implementation of the SEHT algorithm followed the system development process outlined previously. The specific project flow is detailed in Figure 4.

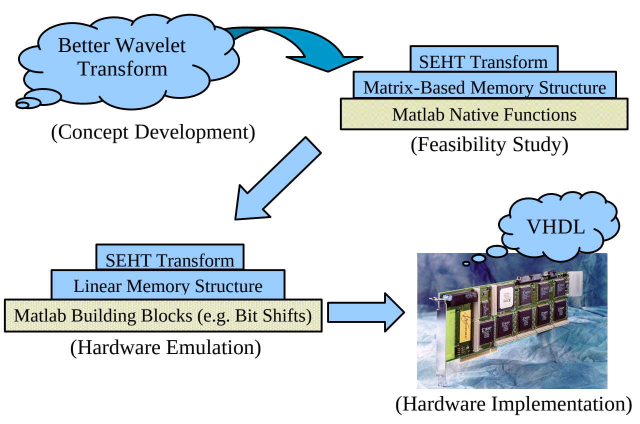

Figure 4: Image Compression Application

First, the algorithm was conceived in the *Concept Development* stage as a method to cut the number of memory accesses in half by combining the row and column transformations. The concept was programmed in Matlab in the *Feasibility Study* phase. The Matlab environment is a vector and matrix-based calculation engine upon which a wide variety of specialized scripts can be run. It was specifically chosen for this project because of its C-like high-level programming language that can natively manipulate the large matrices inherent in image processing. This helps insure transparent code, which aids in rapid development and design testing.

In the *Feasibility Study* stage, the SEHT algorithm was tested to see if the relative performance improvements over the standard Haar wavelet transform were worth pursuing. The results, shown in Figure 5, indicated that further algorithm development was justified.

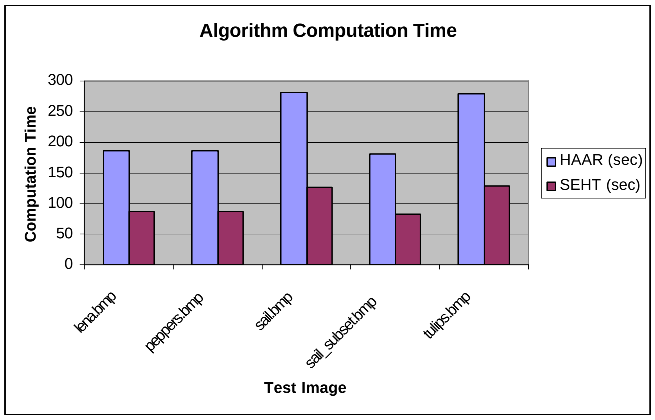

Figure 5: Performance Improvement from Haar to SEHT Wavelet Transform

Based on the promising performance improvements in a high-level environment, the *Hardware Emulator* phase was initiated. Here, a high-level to low-level mapping was begun with the goal of reworking existing algorithms in the high-level software implementation to more accurately simulate real-world hardware constraints. Vectorized operations in the Matlab code (utilized for speed of execution and algorithmic simplicity) were replaced with element-to-element linear operators. At locations where these operations could be executed concurrently, special comment blocks were placed in the code to preserve that information for the hardware implementation phase. High-level mathematical operations such as multiplication or division were replaced with simple bit shift operations. Due to the integer nature of the SEHT algorithm, no floating-point hardware was necessary for the emulator or final hardware implementation.

Also in the emulator stage, multidimensional arrays in the high-level implementation were replaced with single-dimensional fixed-length linearly addressable models of the actual memory available in the destination hardware. This is because many reconfigurable computing systems have fixed memory capacities in multiple banks, and one of the goals of this stage is to determine if the SEHT algorithm previously modeled in a high-level fashion can fit into a real-world system. In this example, the SLAAC development system (a rapid prototyping PCI board with three FPGAs and on-board memory) had been previously purchased for use in real-time applications. Thus, this hardware emulator model reflected its specifications.

As this point in this research effort, alternate methods for structuring the data in memory and managing sequential and concurrent operations were studied. This allowed the hardware implementation stage to proceed smoothly based upon a fully-realized design. Of all the algorithms that made up the image compression process, the SEHT algorithm and its supporting memory architecture was selected to be the first algorithm implanted in hardware due to its significant computational complexity and possibility for greater performance improvements.

In the final stage of the system development process, the *Hardware Implementation*, the key image compression algorithms were implemented in VHDL on FPGAs for maximum computational performance and flexibility. At this point, the algorithms have been verified in the high-level simulator for mathematical correctness and any applicable “quality” metrics. In addition, the algorithms have been tested in the low-level simulator to insure that they will fit the SLAAC board with regards to memory size and architecture as well as in regards to the computational performance limitations of the device. Thus, the majority of risks inherent in hardware development have already been addressed in the two system prototypes already created.

For the hardware implementation phase VHDL is used as the appropriate hardware description language because of the availability of the necessary compiler and supporting place and route software. Vectorized operations that were previously converted to linear element-by-element operations in the hardware emulator are now either converted to concurrent operations here (to exploit the parallelisms possible in hardware), or are included as part of finite state machines if sequential operation must be maintained.

The hardware implementation phase incorporates the same feedback loop structure found in the feasibility study and hardware emulator stages. Thus, as mentioned previously, the highest-risk elements should be addressed first. In this case, the overall memory structure and routing was implemented first. Once it was evaluated for correctly storing and routing data, the next riskiest element of the implementation stage, the SEHT transformation algorithm was started, since that element may depend or interact with previously unknown attributes of the memory architecture. Thus, the design evolved towards a final solution based upon knowledge that may not have been available when the project was initiated.

6. Concluding Remarks

When applied to the real-world image compression problem, the system development process proposed here demonstrated several advantages. First and foremost was the ease of algorithm development with the high-level Matlab language. The transparency of the high-level code and the integrated debugger made it easy to optimize the SEHT algorithm and the supporting data systems.

Such tools also proved very useful when, during testing at the emulator stage, the algorithm implementation was found to have subtle problems in the memory structure and re-use of certain memory elements; problems that were not present in the pure high-level simulation. Because the emulator was still in the Matlab environment, however, these errors were much easier to locate through the use of the integrated debugger. Thus, they were solved before the time-consuming hardware implementation stage, at which point such logic problems would have been effectively obscured by implementation details and would have been much more costly to locate and fix.

The use of evolutionary development to build upon recently gained experience was of particular value in the hardware implementation stage. Because all the particulars of the implementation were not fixed early in the project, the developers were able to take advantage of the algorithmic fixes and optimizations learned in the first two stages of the process. Thus, when the hardware implementation was reached, little was left to chance, and little knowledge was wasted.

Overall, the use of high-level rapid prototyping tools early in the development process can be valuable in any research or development program where deciding what system to implement is as much the goal as actually producing a working implementation. If the high-level system is properly structured and designed, then this real-world problem showed that the prototype can be efficiently re-implemented in hardware through a language such as VHDL.

7. References

[1] Balster, E.J., Scarpino, F.A., and W.W. Smari, “Wavelet Transform for Real-Time Image Compression Using FPGAs,” *12th IASTED International Conference on Parallel and Distributed Computing and Systems*, Las Vegas, Nevada, Nov. 6 – 9, 2000, pp. 232-238.

[2] Boehm, B., "A Spiral Model of Software Development and Enhancement", *IEEE Computer*, Vol.21, #5, May 1988, pp. 61-72

[3] Boehm, B., "Get Ready for Agile Methods, with Care", *IEEE Computer*, Vol.35, #1, January 2002, pp. 64-69

[4] DeHon, A., “The Density Advantage of Configurable Computing,” *IEEE Computer*, Vol 33, No. 4, April 2000, pp. 41-49

[5] Hamlet, D. and J. Maybee, *The Engineering of Software*, Addison-Wesley, 2001

[6] Hutchings, B.L. and B.E. Nelson, “Using General-Purpose Programming Languages for FPGA Design,” *37th Design Automation Conference*, Los Angeles, CA, June 5-9, 2000, pp. 561-566

[7] Pressman, R.S., *Software Engineering, A Practitioner’s Approach*, 4th Edition, McGraw Hill, 1997

[8] Swan, R. et al., “Re-configurable Computing,” *ACM Crossroads*, Issue 5.3, Spring 1999

[9] Turri, W, “Design And Hardware Implementation Of A Wavelet-Based Color Image Compression System,” University of Dayton Masters Thesis, May 2002