

A Practical Approach to Hardware and Software SoC Tradeoffs Using High-level Synthesis for Architectural Exploration

David J. Pursley
Forte Design Systems
1501 Reedsdale Street #302
Pittsburgh, PA 15233
+1-412-321-4350 x15
pursley@ForteDS.com

Brett L. Cline
Forte Design Systems
100 Century Center Court
San Jose, CA 95112
+1-978-264-1855
brett@ForteDS.com

ABSTRACT

The complex world of SoC design is not immune to Moore's law. Many SoCs are currently pushing the five to ten million-gate count on average. In addition, many contain a huge amount of software being processed by a DSP and/or general-purpose processor.

We present a practical methodology to intelligently partition between hardware and software by using hard synthesis-based metrics. This approach is centered around architectural exploration from a single system-level model using existing modeling languages and existing synthesis tools.

We highlight the advantages of this methodology compared to previous methodologies and the prevalent ad hoc approach, with a specific bias toward real-world applicability.

1. INTRODUCTION

A central challenge in designing complex devices is creating an architecture that partitions the hardware and software functionality optimally by considering hardware constraints and costs such as area, power, and system speed. Often, a system has specific performance goals, but other limitations (such as cost or need for programmability) restrict the amount of the design that can be put into silicon. This requires a tradeoff analysis of silicon area vs. performance, with part of the system ending up in hardware and the remainder in software.

The process of hardware and software partitioning is generally somewhat one-sided. In many devices the DSP's performance and MIPS budget are the gating factors for deciding what will be hardware and what will be software. That is, if the software will not be fast enough or if it will use up too many MIPS, it must go in hardware. Unfortunately, this decision is often made without regard the hardware design considerations such as area, power, and performance. Estimates are often wrong causing significant costs later in the design process.

The idea of intelligently partitioning between hardware and software is not new, dating back 10 or more years in the literature [4][5][8]. The basic problem of partitioning is well-defined.

In short, any given system can be thought of as composed of a number of atomic units that will never be split across hardware and software. These atomic units range in size from single

instructions to basic blocks to entire processes, depending on the approach taken. Because the size of these atomic units varies widely in the literature, we will use the generic term "chunk" to refer to an atomic unit.

A set of hardware and software implementations is created or assumed present, so that each chunk has both a hardware and software implementation. Then, a cost metric is developed for both hardware and software implementations, and the problem can be solved via a hill-climbing heuristic.

It should be mentioned that hardware-software partitioning is only one part of the overall hardware-software co-design solution. Full systems have been developed for hardware-software co-design, including [2][4][5] and [8].

Nonetheless, the academic maturity of this problem has yet to be leveraged in most real-world design flows.

In this paper, we'll present a practical, realizable methodology for systematic partitioning between hardware and software. We start by discussing the prevalent approach to partitioning, and then augment the approach with some pragmatic concerns. We discuss the proposed methodology, with a bias toward real-world applicability. Using case studies of an encryption algorithm and an algorithm for an audio device, we apply the methodology to make well-informed tradeoffs between hardware and software in a hypothetical PDA (personal digital assistant) system.

2. EXISTING PARTITIONING MODEL

Although a mature, well-defined, systematic methodology is often absent in the design of today's systems, it is not to say that the issue of partitioning is not addressed. It is, in fact, addressed, but in an ad hoc manner. The existing methodology, discussed below, is true for both ad hoc and more systematic approaches.

To discuss hardware-software partitioning effectively, we must first place it in the context of an overall co-design approach. Again, this could be one of the systems referenced above, or a complete ad hoc approach. Regardless, the same decisions must be made.

A general approach to partitioning, as done today, is shown in Figure 1.

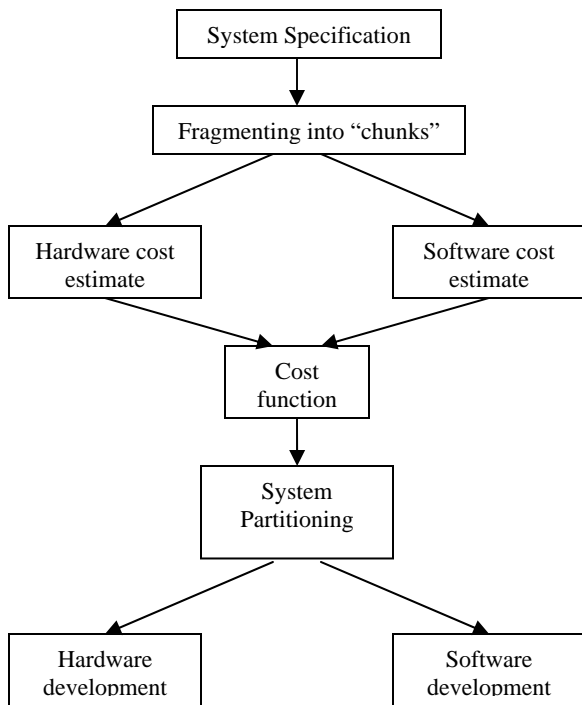


Figure 1: Current approach to hardware-software partitioning.

All systems start with a specification, which could be represented on paper, as executable code [4][5], or in another form (such as Esterel [2] or SpecCharts [8]).

The specification is then fragmented into “chunks” that can be considered for either hardware or software. In ad hoc approaches, the chunks are usually processes or functions, while the automated approaches in the literature may consider chunks as small as basic blocks [4] or even instructions [5].

Cost estimates are created for each chunk, based on the relevant cost metrics for both a software and hardware implementation of the chunk. The exact cost metrics vary from design to design (and approach to approach), but may consider performance, area (usually only for hardware), power, etc. More complex but accurate cost functions also account for the communication between hardware and software [4][5].

Finally, the partitioning is selected, and the hardware and software are created (often by separate teams). Also, the interfaces must be chosen and created at this time.

3. REFINING THE EXISTING MODEL

The model above is incomplete, as it fails to address some basic pragmatic issues. It is quite possible this is the reason more structured co-design efforts are not in place today. Below, we will refine the above model into a model that is both more practical and more powerful when designing real systems.

We should mention that this is not a presentation of an automated approach, although this methodology could be used within the automated frameworks of others. Instead, it highlights some

necessary but missing components of the above methodology, and refines it into a more practical, realizable approach to system design.

3.1 System Specification

Although a paper specification is by far the most common starting point, an executable specification, that is, a specification written in a language that can be compiled and run as an executable, is becoming commonplace for many types of complex systems. This has many advantages, the greatest of which is an unambiguous representation of the system that all implementations can be concretely measured against. A fuller discussion of executable specifications can be found in [6].

Given that an executable specification is the preferred type of specification, from a pragmatic point of view, the language chosen should be one that can easily represent both software and hardware. Furthermore, if a more automated methodology is desired, the language should support both software and hardware synthesis. Given these requirements, C and C++ are logical starting points, as they can be directly used as software implementations, and/or augmented with SystemC [7] constructs so high-level synthesis can be used to generate hardware gates.

3.2 Fragmenting the System

Much research has gone into partitioning of various sized chunks, ranging from the instruction level up through processes. To be sure, fine-grained partitioning will be a significant advantage in co-design systems of the future.

However, from a practical, non-theoretical point of view, system designers often know the granularity with which they will partition. For example, in DSP and graphics systems, the design often very logically decomposes into blocks of reasonable size.

While it is foolish to assume that no additional benefit could ever be gained by looking at smaller pieces of the design, we can safely, if not optimally, rely on the expertise of the designer (and natural composition of the system) to fragment the system for us.

3.3 Cost Estimates and Synthesis

We combine the steps of estimating costs and synthesis because they can be done hand-in-hand. Clearly, that is already the case with getting performance estimates for software, as that can be easily done by compiling and executing. The technology is now available to provide a similar approach for cost estimates of candidate hardware portions of the system.

First, however, we must correct a basic assumption about partitioning. In Section 1, we mentioned that a hardware implementation is created or assumed present for each chunk. However, in many cases, that is a grossly insufficient model. To truly optimize the system, multiple hardware implementations must be considered for each chunk. For example, there could be a high performance implementation, a low performance implementation, and any number of implementations in between those extremes. Each of those implementations should be considered when determining the hardware-software partitioning of the system. The addition of this one-to-many mapping of chunks to hardware implementations, and accurately determining

the relative cost and benefit of each, allows better system architecture exploration.

Of course, this also complicates the problem, by requiring the ability to determine cost metrics for multiple hardware implementations from a single specification. Also, since there are now multiple implementations being judged against each other, the cost metrics (whatever they are) must be fairly accurate. The best way to get accurate estimates is to make measurements on real RTL code.

Since practical high-level synthesis from C++ (SystemC) exists today, this is the preferred path to getting the desired cost metrics. RTL can be quickly created for a number of implementations, and then very accurate cost metrics (such as latency, area, etc.) can be extracted from the RTL.

This approach provides more accurate estimates as well as the ability to do more extensive system exploration by considering multiple hardware implementations of each chunk. These limitations may be why systematic hardware-software partitioning is noticeably absent from many system-level design flows.

3.4 Pragmatic Partitioning Model

The refined, pragmatic hardware-software partitioning model is shown in Figure 2.

The system is represented as an executable specification in C or C++, and the designer determines the relevant chunks to be considered for hardware or software implementations. For example, in a DSP design, this is likely the DSP “building blocks” (FFT’s, filters, etc.), while in graphics it would likely be functions that make up the graphics pipeline. By allowing the designer to determine the relevant chunks, this methodology is applicable across application spaces (DSP, graphics, networking, etc.).

To get the cost metrics (performance, etc.) for the software implementation, a compilation can be done. More time can be spent refining the C++ model before compilation if a more accurate software measurement is needed. Also, in many cases, especially DSP and graphics, the software costs (most likely performance) of an optimized design may be known *a priori* from previous systems.

Hardware cost metrics are obtained by using high-level synthesis on the C++ model to get a range of hardware implementations. The specifics of this process are detailed in the next section. Then, cost metrics can be directly obtained from the resulting RTL or gates.

The hardware and software cost metrics are plugged into the cost function, and the system level partitioning is determined (either by hand or via an automated process with a hill-climbing heuristic). Although the cost function may be the same as in the previous methodology, at a minimum it must be able to support an n-way decision, as compared to the binary decision of hardware vs. software, as there are now multiple candidate hardware implementations.

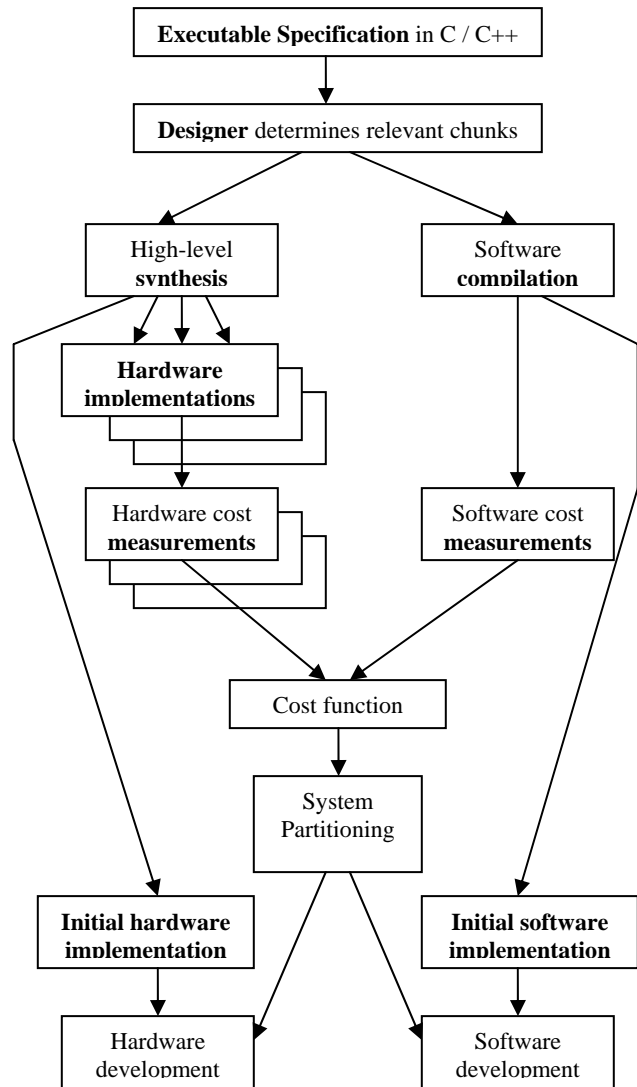


Figure 2: Pragmatic hardware-software partitioning model.

Once the partitioning is decided, hardware and software design begins. However, with this methodology an initial implementation of each chunk is already available, giving the design process a significant head start.

Finally, note that this methodology is not specific to any one application space or any one set of cost metrics. It is a generic, widely applicable methodology that can be put into use today.

4. EXPLORATION METHODOLOGY

A major change between Figure 1 and Figure 2 is the use of high-level synthesis, sometimes called behavioral synthesis, to create many implementations of a given chunk or set of chunks. We will call this additional ability hardware design exploration. Hardware design exploration is the enabling technology allowing the approach in Figure 1 to be replaced by the pragmatic approach in Figure 2.

We used Forte Design Systems’ Cynthesizer product to do design exploration and synthesize register transfer-level (RTL) hardware

from a C++ executable specification [3]. Below, we will briefly discuss the steps in hardware design exploration via Cynthesizer.

The main coding change between most C or C++ algorithms and the SystemC accepted by Cynthesizer is that I/O protocols must be defined when using Cynthesizer. In other words, the C++ algorithm may remain untouched, but it must be wrapped in an SC_MODULE and I/O methods should be added.

However, even this may not be a significant change. Since SystemC is C++, inheritance can be used to make the addition of I/O protocols extremely straightforward. The new design can simply inherit the I/O functions from a parent class, assuming a parent class exists with the desired I/O protocol. For example, the design could use an undefined read() function call to read data, and the actual I/O protocol for the read would be defined in the parent class. Depending on which parent class is used, the read() could use a ready-valid handshake, a request-acknowledge interface, etc. Custom protocols, of course, will always require explicitly writing SystemC I/O methods.

Once the I/O protocols are selected, the design is ready for synthesis with Cynthesizer. However, for effective design exploration, one additional step is suggested.

Cynthesizer allows the user to define multiple “directive sets,” which are groups of synthesis directives for the design. These directives allow the user to specify latencies for blocks of code (usually loops), whether or not to pipeline loops, whether or not to unroll loops, if arrays should be implemented as memories or registers and wires, etc. It is important to note that specifying these directives (and combinations of directives) requires no changes to the C++ code.

Each directive set (give this loop a latency of x, pipeline that loop to start every y cycles, etc.) corresponds to a set of directives that will be given to Cynthesizer during a single synthesis run. Iterating over the directive sets results in hardware design exploration.

Finally, since the timing of the I/O protocols is maintained in the synthesized output, the same testbench can be used to verify the input and output of Cynthesizer. This makes verification of synthesized RTL trivial.

5. CASE STUDIES

Below are two examples of hardware design exploration. The first example is a small example of how design exploration can sometimes result in discovering unexpected trade-offs. The second example is from a hypothetical design of a PDA or other device containing an MP3 player.

5.1 Advanced Encryption Standard

The Advanced Encryption Standard is a symmetric encryption algorithm recommended by the U.S. Government for organizations to protect sensitive information [1]. This algorithm, downloaded off the web, was used as a test case for Cynthesizer, and the results were compared against a handwritten RTL implementation.

Area was the top design goal, with performance a secondary concern (“make it go as fast as possible in minimum area”). By applying different directive sets with Cynthesizer, the designer was able to create many different RTL implementations of the same design. Each design could be verified with the original

testbench, guaranteeing that the synthesis results were not only gates, but “correct gates.”

An automated flow for design exploration was used, allowing trade-offs between competing design goals (area vs. performance in this case) to be made using actual synthesis numbers. To do this, all loops were unrolled completely, and then the latency constraint for the main loop was iteratively decremented via directive sets, as explained above.

It turns out that with each iteration, the area change was fairly minimal, so at the smallest latency that could be met (highest performance) overall area had only increased by approximately 22%. This is summarized in Table 1.

Table 1. Comparison of AES implementations

Methodology	Normalized Area	Normalized Performance	Time to create
Hand coded	1.00	1.00	1 man-month
Cynthesizer	1.22	4.85	2 man-days

Unrolling loops completely allowed much constant propagation, which immediately removed a lot of computation. However, unrolling loops also increased the number of states in the FSM considerably. As the latency constraint tightened, the relatively small functional units began to be placed in parallel (increasing area), but that was partially offset by the decrease in FSM size (fewer cycles = fewer states). The net result was a significant speed-up at only moderate area cost.

This was an interesting, unexpected tradeoff, increasing performance by almost 5x while increasing area by only 22% over a reference design that had been done by hand. Had the designer known about this tradeoff, he would have chosen this design point over the slightly smaller but much slower reference design. However, there is little chance the designer would have stumbled on this design point with a standard design process.

5.2 MP3 Filter

The second case study is the design of a portion of the MP3 polyphase filtering algorithm. MP3 encoders and decoders are found in a wide array of devices, so the system-level implications of the tradeoffs presented by hardware design exploration are more obvious than in the previous example.

Here, we will consider the case where some hardware exploration is needed to help meet system performance goals, but because this is going into a mass-produced device, cost is a limiting factor.

In this example, five directive sets were used, varying only the amount of loop unrolling and latency constraints for loops. The results of design exploration are shown in Figure 3. In this example, two tradeoffs are of special interest.

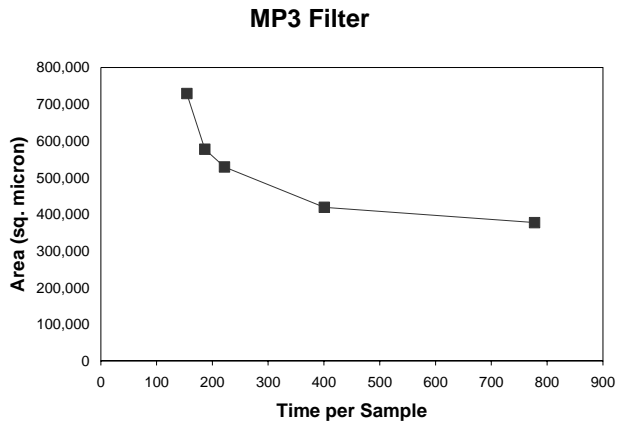


Figure 3: Design exploration for MP3 design.

First, the difference in performance between the slowest and fastest designs is 5.04x, while the area differential is only 1.93x. Perhaps, when examining the overall system, speeding up one block very significantly would be all that is needed to meet the overall system performance constraints. In such a case, maximizing the performance of this block could be a very useful tradeoff.

Second, notice that the rightmost two points show a speedup of 94% at a cost of only 11% area. This is likely to be a very good tradeoff, assuming the system needs any additional speedup at all. This additional 11% area may give the system high enough performance that it would not require additional blocks to be moved to hardware. On the other hand, if other blocks must still be moved to hardware for further speedup, the 94% speedup of this block at a modest area penalty may allow other blocks to be executed more slowly (and thus in less hardware).

Of course, increased performance may also have other system implications, such as allowing the clock to be periodically turned off, thereby saving power. It is assumed that any such possibilities would be included in the cost function.

Regardless, these tradeoffs can only be seriously considered after creating RTL to get very accurate measurements.

6. CONCLUSION

If systematic hardware-software co-design is to become commonplace when designing today's increasingly complex systems, then a more pragmatic approach to the problem is required. We have presented an approach to the partitioning problem that is both powerful and implementable today. It leverages the knowledge and experience of the designer to determine the granularity at which to look at the problem, and leverages languages and high-level synthesis technology that exists today to perform exploration and evaluate potential decisions.

The proposed methodology allows system designers to make better partitioning decisions by basing those decisions on hard numbers, instead of "back of the envelope" estimates. Adoption of this methodology may result in better systems as well as the ability to create them faster, since initial hardware models are created during the partitioning analysis.

7. REFERENCES

- [1] AES Home Page. <http://csrc.nist.gov/CryptoToolkit/aes/>
- [2] Balarin, F., et al, Hardware-Software Co-design of Embedded Systems—The POLIS Approach. Kluwer Academic Publishers, 1997.
- [3] Synthesizer User's Guide. <http://www.ForteDS.com/> Forte Design Systems, 2003.
- [4] Ernst, R., Henkel, J., and Benner, T. Hardware-Software Cosynthesis for Microcontrollers. IEEE Design & Test of Computers, vol. 10, no. 4, pp. 64-75, 1993.
- [5] Gupta, R.K., and De Micheli, G. System-level Synthesis using Reprogrammable Components. Proceedings of EDAC '92, IEEE Computer Society Press, pp. 2-7, 1992.
- [6] Sanguinetti, J., and Pursley, D. High-Level Modeling and Hardware Implementation with General-Purpose Languages and High-level Synthesis. Ninth IEEE/DATC Electronic Design Processes Workshop, April 2002.
- [7] SystemC Community. <http://www.systemc.org>
- [8] Vahid, F., and Gajski, D. Specification Partitioning for System Design. Proceedings of DAC '92, 1992.