Codesign-Extended Applications

Brian Grattan[1], Greg Stitt[2], and Frank Vahid[2,3]

[1] Department of Electrical Engineering
[2] Department of Computer Science and Engineering
University of California, Riverside
[3] Also with the Center for Embedded Computer Systems at UC Irvine

{bgrattan | gstitt | vahid}@cs.ucr.edu, http://www.cs.ucr.edu/~vahid

ABSTRACT

We challenge the widespread assumption that an embedded system's functionality can be captured in a single specification and then partitioned among software and custom hardware processors. The specification of some functions in software is very different from the specification of the same function in hardware – too different to conceive of automatically deriving one from the other. We illustrate this concept using a digital camera example. We introduce the idea of codesign-extended applications to deal with the situation, wherein critical functions are written in multiple versions, and integrated such that simple compiler/synthesis flags instantiate a particular version along with the necessary control and communication behavior. By capturing a specification as a codesign-extended application, a designer enables smooth migration among platforms with increasing amounts of on-chip configurable logic.

Keywords

Hardware/software partitioning, hardware/software cospecification, configurable logic, system-on-a-chip, platform-based design.

1. INTRODUCTION

Hardware/software partitioning has been shown to provide excellent performance as well as power and/or energy improvements compared to software-only implementations in embedded computing systems [4][7][9][11][15][16][18][25][28][29]. Making such partitioning even more attractive is the appearance of single-chip platforms, some of which are intended for consumer products, that include both a microprocessor and configurable logic [1][3][13][23][27][30].

Most approaches to hardware/software codesign assume that a designer initially describes the behavior of an embedded system using one (or possible more than one) executable language. Languages proposed for such purposes include C [10], C++ and Java [14], as well as Statecharts [12], Esterel [8], SpecC [24][31], and SystemC [26]. Most hardware/software partitioning approaches assume in particular that the main functions of the system are each described once in the specification. Those partitioning approaches then consider the tradeoffs between compiling each function to software versus synthesizing to a custom hardware processor. The goal of such partitioning is to make best use of existing hardware to improve the performance and/or energy of the system.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CODES'02, May 6-8, 2002, Estes Park, Colorado, USA.
Copyright 2002 ACM 1-58113-542-4/02/0005...$5.00.

In our investigations of the performance and energy advantages of partitioning for single-chip platforms, we have found that the assumption that each function can be described using one algorithm in a specification, from which software or hardware implementations can be derived, does not apply to many functions. In some cases, the algorithm we would use to implement the function in software is very different from the algorithm we would use for a hardware implementation.

To cope with this situation, we propose the idea of codesign-extended applications. In short, a designer finds the most frequently executed functions, and then writes two versions of those functions, one for software, the other for hardware. An automated partitioning tool then chooses between the versions.

In this paper, we illustrate the different specifications of some functions in software and hardware by using a digital camera example. We then describe the concept of a codesign-extended application and highlight our particular implementation of the concept. We show how using codesign-extended applications can enable smooth migration from all-software implementations to hardware/software implementations using evolving single-chip platforms.

2. DIFFERENT ALGORITHMS IN SOFTWARE AND HARDWARE

A system's specification typically consists of a set of functions, where each function's granularity is that of perhaps tens or hundreds of lines of sequential program statements, corresponding roughly to an algorithm. Hardware/software partitioning seeks to map each function to either software executing on a microprocessor, or to a custom hardware processor. Most approaches seek to keep as many functions in software, due to software's low cost and flexibility, while gaining as much speedup as possible by mapping certain functions to hardware, subject to hardware size constraints.

Some functions give good speedups in hardware, due perhaps to more concurrency, more efficient bit-level processing, and/or less instruction fetch and decode overhead. A good hardware synthesis tool will maximally exploit existing hardware by transforming a function's algorithm to expose parallelism. Such transformations may include loop unrolling, subroutine inlining, subroutine cloning, and even process extraction. Thus, the resulting hardware algorithm may look very different from the original algorithm in

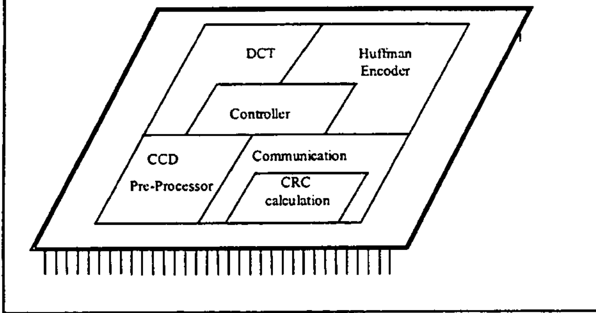

Figure 1: Digital Camera Functions on an SOC.

the specification. Nevertheless, the algorithms are fundamentally the same, achieved through a straightforward series of transformations.

We have observed that human designers, however, often use fundamentally different algorithms when describing the same function for software versus hardware. As a simple example, consider a function that sorts an array of integers, perhaps forming part of a portable electronic phone book system. If a designer knows that the sorting function will be implemented in software, the designer may describe the function using the Quicksort algorithm [5]. On the other hand, if the designer knows the function will be implemented in hardware, the designer may describe the function using Mergesort, since it can be parallelized very nicely and forms the basis of many hardware-based sorting approaches [5][17][19]. Quicksort and Mergesort are fundamentally different algorithms that can't be derived from one another. Even if we implemented Quicksort in hardware, we would use a non-recursive version, whereas the software version is usually recursive. Thus, this initial example illustrates that a single algorithm for a function is not always sufficient as input to hardware/software partitioning—two versions (or even more) might be more appropriate.

As a second example, consider a digital camera chip, whose main functions are illustrated in Figure 1. The complete functionality could be described in a single software specification, with the following functions. The *CCD pre-processor* reads the charge coupled device and communicates data to the controller. The *DCT* component performs a discrete cosine transform. The *Huffman Encoder* performs Huffman encoding. The *Controller* is the main controller of the system. The *Communication* transmits and receives data to and from other devices. To take a picture, the controller would signal the CCD pre-processor to gather data from the CCD, signal the DCT unit to transform the data, signal the encoder to encode the data, and then store the data. At a later time, the controller may upload or download data with other devices, like a personal computer. We assume the communication method could be RS-232, USB, wireless, or some other method, but that the methods may use a CRC (Cyclic Redundancy Check [21]) for error checking.

The CRC performed during communication is a time-consuming function and thus is a good candidate for hardware implementation. If hardware is not available, the CRC can be done in software. However, the standard algorithm for a software CRC is radically different from that for a hardware CRC.

Figure 2: Software CRC algorithm.

```
unsigned short icrc1(unsigned short crc, unsigned char onech)
{
  int i;  unsigned short ans=(crc ^ onech << 8);
  for (i=0;i<8;i++)  {
    if (ans & 0x8000) {ans = (ans <<= 1) ^ 4129;}
    else  {ans <<= 1;}
  }
  return ans;
}
typedef unsigned char uchar;
#define LOBYTE(x) ((uchar)((x) & 0xFF))
#define HIBYTE(x) ((uchar)((x) >> 8))
unsigned short icrc(unsigned short crc, unsigned char *bufptr,
                    unsigned long len, short jinit, int jrev)
{
  unsigned short icrc1(unsigned short crc, unsigned char
onech);
  static unsigned short icrctb[256],init=0;
  static uchar rchr[256];
  unsigned short j,cword=crc;
  static uchar it[16]={0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15};
  if (!init) {
    init=1;
    for (j=0;j<=255;j++) {
      icrctb[j]=icrc1(j << 8,(uchar)0);
      rchr[j]=(uchar)(it[j & 0xF] << 4 | it[j >> 4]);
    }
  }
  if (jinit >= 0) cword=((uchar) jinit) | (((uchar) jinit) << 8);
  else if (jrev < 0)
    cword=rchr[HIBYTE(cword)] |rchr[LOBYTE(cword)] << 8;
  for (j=1;j<=len;j++)
  cword=icrctb[(jrev < 0 ? rchr[bufptr[j]] :
               bufptr[j]) ^ HIBYTE(cword)] ^ LOBYTE(cword) <<
8;
  return (jrev >= 0 ? cword :
          rchr[HIBYTE(cword)] | rchr[LOBYTE(cword)] << 8);
}
```

A standard software CRC, taken from [21], is shown in Figure 2. This code uses the first function (icrc1) to create a table of the CRC of 256 characters. It then uses this table to calculate the CRC of an array of characters passed to icrc. This relies heavily on looking into arrays—a task easy to do in software, but not efficient in hardware.

For a hardware CRC, bit operations can be executed in parallel. Thus, a hardware CRC consists of numerous bit-wise exclusive OR operations. Figure 3 illustrates a hardware CRC in VHDL, created automatically by the CRC generator tool in [6] (another tool can be found at [1]). Notice how different the hardware CRC algorithm is from the software CRC algorithm, even though the functions give the same result.

As another example of a function with different software and hardware algorithms, consider the DCT function, which is one of the most time consuming functions during picture taking. The DCT is thus is a good candidate for acceleration using hardware. It is also a popular hardware unit, and there are publicly available cores [20]. A major part of the DCT is the matrix multiplication of an input matrix by a constant matrix. A simple implementation of a DCT in C is shown in Figure 4. Some functions have been left out in the interest of brevity.

Figure 3: Hardware CRC algorithm.

```
-- Copyright (C) 1999 Easics NV. This source file may be used and
distributed without restriction provided that this copyright statement is
not removed from the file and that any derivative work contains the
original copyright notice and the associated disclaimer. THIS SOURCE
FILE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE
IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR
A PARTICULAR PURPOSE.
package body PCK_CRC16_D8 is
 -- polynomial: (0 5 12 16)   data width: 8
 -- convention: the first serial data bit is D(7)
 function nextCRC16_D8 ( Data: std_logic_vector(7 downto
0);
                         CRC: std_logic_vector(15 downto
0) )
   return std_logic_vector is
     variable D: std_logic_vector(7 downto 0);
     variable C: std_logic_vector(15 downto 0);
     variable NewCRC: std_logic_vector(15 downto 0);
 begin
   D := Data;  C := CRC;
   NewCRC(0) := D(4) xor D(0) xor C(8) xor C(12);
   NewCRC(1) := D(5) xor D(1) xor C(9) xor C(13);
   NewCRC(2) := D(6) xor D(2) xor C(10) xor C(14);
   NewCRC(3) := D(7) xor D(3) xor C(11) xor C(15);
   NewCRC(4) := D(4) xor C(12);
   NewCRC(5) := D(5) xor D(4) xor D(0) xor C(8) xor C(12)
                xor C(13);
   NewCRC(6) := D(6) xor D(5) xor D(1) xor C(9) xor C(13)
                xor C(14);
   NewCRC(7) := D(7) xor D(6) xor D(2) xor C(10) xor C(14)
                xor C(15);
   NewCRC(8) := D(7) xor D(3) xor C(0) xor C(11) xor C(15);
   NewCRC(9) := D(4) xor C(1) xor C(12);
   NewCRC(10) := D(5) xor C(2) xor C(13);
   NewCRC(11) := D(6) xor C(3) xor C(14);
   NewCRC(12) := D(7) xor D(4) xor D(0) xor C(4) xor C(8)
                 xor C(12) xor C(15);
   NewCRC(13) := D(5) xor D(1) xor C(5) xor C(9) xor C(13);
   NewCRC(14) := D(6) xor D(2) xor C(6) xor C(10) xor C(14);
   NewCRC(15) := D(7) xor D(3) xor C(7) xor C(11) xor C(15);
   return NewCRC;
 end nextCRC16_D8;
end PCK_CRC16_D8;
```

When implementing a DCT in hardware, a key change made to the algorithm is to utilize an algorithm based on fixed point rather than floating point numbers. Thus, some precision is typically traded off for hardware efficiency. In addition, more than one process could be used to control dataflow. We omit a hardware description of the DCT for conciseness.

Again, the two functions are quite different from each other in appearance and execution. This time, the results will be different, as the VHDL code will introduce quantization noise due to the conversion to fixed-point arithmetic.

To quantitatively observe the difference between software and hardware algorithms, we examined the CRC further. The results of translating the CRC from the software description to a hardware description are found in Figure 5. Only the main process is shown to give a general idea of how it could be done. This segment of code illustrates how the body of the loop from icrc is translated into a hardware process that reads the arrays (which would have to be initialized in advance) from memory and

Figure 4: Software DCT algorithm.

```
static const short code COS_TABLE[8][8] = {
 {32768, 32138, 30273, 27245, 23170, 18204, 12539,  6392},
 {32768, 27245, 12539, -6392, -23170, -32138, -30273, -18204}
 {32768, 18204, -12539, -32138, -23170,  6392, 30273, 27245}
 {32768,  6392, -30273, -18204, 23170, 27245, -12539, -32138}
 {32768, -6392, -30273, 18204, 23170, -27245, -12539, 32138}
 {32768, -18204, -12539, 32138, -23170, -6392, 30273, -27245}
 {32768, -27245, 12539,  6392, -23170, 32138, -30273, 18204}
 {32768, -32138, 30273, -27245, 23170, -18204, 12539, -6392}
};
static const short ONE_OVER_SQRT_TWO = 23170;
static short xdata inBuffer[8][8]; static short xdata outBuffer[8][8];
static int idx;
static float Y(int a, int b) {
   return COS_TABLE[a][b] / 32768.0;
}
static float C(int h) {
   return h ? 1.0 : ONE_OVER_SQRT_TWO / 32768.0;
}
static int F(int u, int v, short img[8][8]) {
   float s[8], r = 0;   unsigned short x;
   for(x=0; x<8; x++) {
      s[x] =
         img[x][0] * Y(0, v) +      img[x][1] * Y(1, v) +
         img[x][2] * Y(2, v) +      img[x][3] * Y(3, v) +
         img[x][4] * Y(4, v) +      img[x][5] * Y(5, v) +
         img[x][6] * Y(6, v) +      img[x][7] * Y(7, v);
   }
   for(x=0; x<8; x++) {
      r += s[x] * Y(x, u);
   }
   return (short)(r * .25 * C(u) * C(v));
}
void CodecDoFdct(void) {
   unsigned short x, y;
   for(x=0; x<8; x++) {
      for(y=0; y<8; y++) {
         outBuffer[x][y] = F(x, y, inBuffer);
      }
   }
   idx = 0;
}
```

outputs the resulting CRC. This preserves the sequential execution from the software algorithm. The VHDL version of the software algorithm uses an external memory that must be loaded with the arrays that are stored in *icrctb* and *rchr* in the C version. If these were implemented directly in the hardware, they would represent 768 bytes of memory that would take even more area. We assume a best-case scenario, where memory can be read from in one cycle.

Table 1 summarizes the results of implementing the hardware version and the version translated from software. FPGA Compiler from Synopsys was used with the default synthesis flags set and targeting the Triscend E5 family. There is a 3 times speed up by using the hardware version. There is also a significant savings in area. Clearly, it would be a mistake to go straight from a C description to a VHDL description without considering how the algorithm could be changed to take advantage of the hardware.

Going the other direction, from a hardware description to software, gives poor results also. In order to mimic the hardware

Figure 5: Portion of VHDL code translated from C for the software CRC algorithm.

```
if( clk'event and clk = '1' ) then
  case exe_state is
    when GET_ENABLE =>
      if (enable = '1') then
        exe_state <= GET_RCHR;
      else
        exe_state <= GET_ENABLE;
      end if;
    when GET_RCHR =>
      addr := rchr_addr + ("000000000000000000000000" &
bufptr_j);
      START_RD_RAM(addr);
      exe_state <= GET_ICRCTB;
    when GET_ICRCTB =>
      addr := icrctb_addr + ram_in_data;
      START_RD_RAM(addr);
      exe_state <= DO_CALC;
    when DO_CALC =>
      temp := std_logic_vector(ram_in_data(15 downto 0))
              xor
              ("00000000" & cword (7 downto 0));
      temp2 := temp(7 downto 0) & "00000000";
      cword <= temp2;
      output <= temp2;
      exe_state <= GET_ENABLE;
     when others => null;
  end case;
```

description in software, we need to execute numerous bit-wise operations for each character processed. This becomes extremely inefficient because each command is executed sequentially. The results for running roughly 200 characters through each example are found in Table 2. In order to get these results, LCC was used with the default compiler flags in order to compile for a MIPS.

Generally, another current difference between software and hardware descriptions has to do with the input level of the compilation and synthesis tools in common use. While compilers operate on algorithmic level code, the vast majority of synthesis tools operate on register-transfer level code. This represents another practical difference between software and hardware descriptions of functions.

3. CODESIGN-EXTENDED APPLICATIONS

We define a codesign-extended application as a software description of an application, extended with additional versions of key functions using hardware algorithms, and using macros (or some other means) to enable existing compiler and synthesis tools to automatically generate a complete working implementation of the system using any of the function versions.

In order to facilitate a codesign-extended application, we need a way to implement certain functions in software or in hardware. To do this, we have developed a standard method for different partitioning schemes to be compiled/synthesized by only choosing different compiler flags and synthesis flags. This gives the designer an efficient way to test his or her codesign-extended application with several different configurations, and then on chips that have varying amounts of programmable logic available

Table 1: Results of implementing the software CRC algorithm versus the hardware CRC algorithm in configurable logic.

Hardware Implementation	Size (Blocks)	Delay (clock cycles/character)
Hardware CRC algorithm	19	1
Software CRC algorithm	44	3

Table 2: Results of implementing the software CRC algorithm versus the hardware CRC algorithm on a MIPS processor.

Implemented in Software	Size (Assembly Lines)	Clock Cycles
Software CRC Algorithm	1061	180,000
Hardware CRC Algorithm	1298	814,000

and different microprocessor cores so that cost/performance trade-offs between chips can quickly be determined.

This method consists of putting macros around sections of code that can be implemented in hardware. After these macros are in place, a section of code is added that implements a handshaking routine to signal the hardware section to run. The complements of the macros that were placed around the original code are placed around the handshaking routine. For example, a designer could place *#ifdef*s around the original code, and place *#ifndef*s around the handshaking routine. The handshaking routine could be as simple as setting a bit and then waiting on a different bit to be set by the hardware, or it could set a bit and put the microprocessor into a sleep or idle state and wait for an interrupt to be asserted by

Figure 6: CRC Code Modified for Codesign Extended Application (updates **bolded**).

```
xdata crc_start_flag = 0;
xdata crc_done_flag;
...
...other functions...
...
unsigned short icrc(unsigned short crc, unsigned char *bufptr,
unsigned long len, short jinit, int jrev)
{
#ifdef hw_crc
crc_start_flag = '1';
while (!crc_done_flag)
#endif
#ifndef hw_crc
  unsigned short icrc1(unsigned short crc, unsigned char onech);
...
...function code for crc...
...
  return (jrev >= 0 ? cword : rchr[HIBYTE(cword)] |
rchr[LOBYTE(cword)] << 8);
#endif
   }
```

Table 3: Results for different partitions run on an example program.

Partitioning			Energy (Joules)
Multiply	Sum	Bit-Share	
SW	SW	SW	12.4
SW	SW	**HW**	8.6
SW	**HW**	SW	8.8
HW	SW	SW	8.0
SW	**HW**	**HW**	4.8
HW	SW	**HW**	Does not Route
HW	**HW**	SW	Does not Route
HW	**HW**	**HW**	Does not Route

the hardware. This can be seen in Figure 6 using the CRC as an example.

The VHDL that would be written would have to be aware of the address of the external data variables "crc_start_flag" and "crc_done_flag." It would then use the "start flag" as an enable and notify the processor of its completion using the "done flag."

Using this methodology, we have implemented a simple codesign-extended application on two different platforms. The first platform was a two-chip solution consisting of a standard 8051 and a Xilinx FPGA. The second implementation was a Triscend system on a chip—the TE520. This chip contains a "turbo" 8051 and an array of configurable system logic.

Our codesign-extended application was a signal processing example containing three functions that were described for both software and hardware. The VHDL files were written separately, but with a simple tool, we were able to merge the files and associate the different functions with different enable signals. This way, we could compile the C program with a certain set of compile-time macros set and then choose the VHDL file that complimented the functions that were left out of the software, and synthesize that VHDL file. For example, to compile the program with the matrix multiply function in the configurable logic, the command used would be: *c51 three.c -df(hw_matrix_multiply).*

Depending on the design, and the tools used, the synthesis and place and route could take anywhere from a few minutes to several hours. Some of the combinations were not able to be placed and routed onto the TE520 chip. This is to be expected, and gives motivation to experiment with higher capacity chips to determine how much performance could be increased for a given increase in price. The metric we used to rate the different implementations was energy consumption. The energy savings are shown in Table 3. Energy was determined by using a digital multimeter that communicated with the workstation and having the workstation time the execution of the given programs. Therefore, the workstation could calculate the energy, knowing the voltage, current, and time.

4. CODESIGN-EXTENDED APPLICATION METHOLOGY

A design methodology incorporating codesign-extended applications requires more designer effort up front, but that effort may pay off in the long run. A designer can start by writing an all software version of the application -- something typically done today anyway. The designer can then determine the most critical functions of the application, either through his/her own estimation (in many cases, the critical functions are well known), or through profiling of the application with expected input vectors. For each critical function, the designer can determine if a unique hardware algorithm is necessary, in which case the designer can write hardware-suited code for that algorithm.

Although earlier we mentioned that the hardware-suited code might be captured in a hardware description language, like VHDL or Verilog, this is not absolutely necessary. If the language used for the all software version can be compiled to either software or hardware, then the hardware-suited code can simply be written in the same language. For example, if the designer is using a hardware/software capable environment based on C, such as Proceler's environment [22] or SystemC [26], then the hardware-suited algorithm can still be written in C.

Notice that the codesign-extended application idea can be extended to support more than two versions of the same function. Likewise, if two functions were to both be implemented in hardware, the idea can be extended to allow a special combined version of those two functions that might perform better than the two hardware versions of those functions. Of course, extensions like these can quickly cause codesign-extended applications to become unwieldy, so must be used sparingly.

Notice that we could write a hardware version of a function even if the algorithm is the same as the software version -- the advantage being that no automated hardware/software partitioning tool would be necessary. Partitioning could simply be carried out by a script that sets compiler/synthesis flags and then executes the application.

The benefits of the additional up-front designer effort in creating two versions of critical functions can be illustrated by a simple example. Consider the digital camera example of this paper. A designer may initially create an all software version of the application, and then create a codesign-extended application wherein the CRC and DCT functions also have hardware-suited versions. Such a task takes extra effort (although in this case not much because those functions have fairly standard hardware versions readily available), not only because of the initial coding effort, but also in the functional verification of the different combinations. The payoff for this extra effort comes during design exploration. Without any software or hardware recoding, the designer can explore the suitability of different computing platforms, such as a processor only, or processor platforms with varying amounts of on-chip logic, choosing the platform that gives best performance/energy for a given cost. Furthermore, a designer can try new platforms months or years later (as they are evolving quickly and their costs are changing) -- without having to rewrite the application.

5. CONCLUSIONS

The basic assumption that hardware and software can be derived from the same specification is an assumption that does not apply in some cases. In some cases, the hardware algorithm is very different from the software algorithm for the same function. We propose the concept of codesign-extended applications to deal with this situation. A designer determines the most critical functions, and implements two versions of them – one for software, one for hardware. The designer uses a standard

modeling approach to enable a compiler and synthesis tool to automatically include or exclude versions, thus generating unique hardware/software partitions without requiring code rewriting or a sophisticated partitioner that can parse the software and hardware languages. Codesign-extended applications enable graceful evolution of an application onto evolving platforms that include faster processors and/or additional on-chip configurable logic. In the future, we plan to generalize the concept of codesign-extended application to include more than two versions of a function, with multiple software and hardware versions that support tradeoff performance, size and power.

6. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation (CCR-9876006) and NEC C&C Research Labs.

7. REFERENCES

[1] Altera Corporation, *ARM-Based Embedded Processor PLDs*, August, 2001

[2] Actel Corporation, *Cyclic Redundancy Code Generator Macro v4.0*, Actel Corporation, January, 2002.

[3] Atmel FPSLIC, http://www.atmel.com/atmel/products/prod39.htm.

[4] A. Balboni, W. Fornaciari and D. Sciuto. Partitioning and Exploration in the TOSCA Co-Design Flow. International Workshop on Hardware/Software Codesign, pp. 62-69, 1996.

[5] T. Cormen, C. Leiserson, R. Rivest. *Introduction to Algorithms*. McGraw-Hill Book Company, 1997.

[6] Easics Corporation, http://www.easics.com/webtools/crctool.

[7] P. Eles, Z. Peng, K. Kuchchinski and A. Doboli. System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search. Kluwer's Design Automation for Embedded Systems, vol2, no 1, pp. 5-32, Jan 1997.

[8] Esterel Synchronous Language Web Main page, http://www.esterel.org.

[9] D.D. Gajski and F. Vahid and S. Narayan and J. Gong. SpecSyn: An Environment Supporting the Specify-Explore-Refine Paradigm for Hardware/Software System Design. IEEE Transactions on VLSI Systems, Vol. 6, No. 1, pp. 84-100, 1998.

[10] M. Gokhale, J. Stone. NAPA C: Compiling for hybrid RISC/FPGA architectures. IEEE Symposium on FPGAs for Custom Computing Machines, FCCM '98.

[11] J. Grode, P. Knudsen, J. Madsen. "Hardware Resource Allocation for Hardware/Software Partitioning in the LYCOS System." Proc. of the 1998 Design Automation and Test in Europe.

[12] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231-274, 1987.

[13] J. Hauser, J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. IEEE Symposium on FPGAs for Custom Computing Machines, pages 12-21, Napa Valley, CA, April 1997.

[14] R. Helaihel and K. Olukotun, "Java as a Specification Language for Hardware-Software Systems," Proc. ICCAD '97, pp. 690-697, 1997.

[15] J. Henkel, Y. Li. Energy-conscious HW/SW-partitioning of embedded systems: A Case Study on an MPEG-2 Encoder. Proceedings of Sixth International Workshop on Hardware/Software Codesign, March 1998, pp. 23-27.

[16] J. Henkel. A low power hardware/software partitioning approach for core-based embedded systems. Proceedings of the 36th ACM/IEEE conference on Design automation conference, pp. 122 – 127,1999.

[17] C.Y. Huang, G.J. Yu and B.D. Liu. A Hardware Design Approach for Merge-Sorting Network. IEEE International Symposium on Circuits and Systems, 2001, pp.534-537.

[18] K. Kucukcakar. An ASIP Design Methodology for Embedded Systems. International Symposium on Hardware/Software Codesign, May 1999.

[19] S. Olarlu, M.C. Pinotti and S.Q. Zheng. An Optimal Hardware-Algorithm for Sorting using a Fixed-Size Parallel Sorting Device. IEEE Transactions on Computers, vol.49, (no.12), Dec. 2000, pp. 1310-1324.

[20] Opencores Web-Site, http://www.opencores.org/

[21] Press, William H. et al., *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 1992.

[22] Proceler. http://www.proceler.com.

[23] C. Snyder. FPGA Processors Ready for Takeoff. Microprocessor Report, Novemeber 2000, pp. 25-29.

[24] SpecC Technology Open Consortium Web Page, http://www.specc.gr.jp/eng/index.htm.

[25] G. Stitt, B. Grattan, J. Villarreal, F. Vahid. Using On-Chip Configurable Logic to Reduce Embedded System Software Energy. IEEE Symposium on Field-Programmable Custom Computing Machines. April 2002.

[26] SystemC Homepage, http://www.systemc.org/

[27] Triscend Corporation, http://www.triscend.com, 2002.

[28] G. Vanmeerbeeck, P. Schaumont, S. Vernalde, M. Engels and I. Bolsens. Hardware/Software Partitioning of Embedded System in OCAPI-xl. International Symposium on Hardware/Software Codesign, pp. 30-35, 2001.

[29] M. Wan, Y. Ichikawa, D. Lidsky, J. Rabaey. An energy conscious methodology for early design exploration of heterogeneous DSPs. Proceedings of the IEEE 1998 Custom Integrated Circuits Conference, p.111-117, Santa Clara, May 1998.

[30] Xilinx Corporation, *Virtex-II Pro Platform FPGA Handbook*, January 31, 2002

[31] J. Zhu, R. Domer, and D. D. Gajski. Syntax and Semantics of the SpecC Language. In Proceedings of the SASIMI Workshop, pages 75 -- 82, 1997.