Migrating Software to Hardware on FPGAs

Russell A Klein
Mentor Graphics Corp.
[russell_klein@mentorg.com]

Rajat Moona
Indian Institute of Technology Kanpur
[moona@iitk.ac.in]

Abstract

The demands on embedded processors are growing faster than CPU developers can respond. This has lead to a number of academic and commercial processors, such as Tensilica [13, 14] and ARC [15], which allow hardware extensions in the form of new instructions to improve the overall throughput. Unfortunately, it is rarely obvious at the application level which new instruction would accelerate a given function. This paper proposes a design flow that migrates performance critical sections of software into hardware by automatically creating application specific hardware blocks that accelerate the overall software execution. The hardware implementation of the function is interfaced to the general-purpose processor, which runs the remainder of the software. Targeting FPGA fabric maintains the re-programmable nature of the algorithm that was originally in software. We will present our results on using this flow on the G.729 audio encoding algorithm.

1. Introduction

Designers of embedded systems often face a problem where the final design of the system fails to meet the expected performance. The problem is often resolved by one of the several "hacks", including migration of software functionality to the hardware. We are proposing a design methodology that allows designers to efficiently migrate functions from software as part of the body of embedded code running on a general purpose processor to an efficient implementation in FPGA gates as a means of accelerating the overall throughput of the system. Such an approach is extremely useful for prototype systems, platform FPGAs with embedded processors and finally in the product realizations. Our proposed methodology starts with a system that is implemented on an FPGA with an embedded processor (either hard or soft) containing both hardware and software - the hardware represented at the register transfer level (RTL) in a hardware description language (HDL) and the software as executable image. This system is profiled to determine the overall throughput and the time spent in individual software functions. Using this profile the designer determines the functions that are taking the most time in software. Using an estimation tool, the designer tests moving various functions from software to hardware. The estimation tool shows the designer the area that will be taken by the function in hardware and the amount of computation and communication time consumed by the function in hardware on a specific interface to the processor. This is compared with the time the various functions took in software on the processor. Once a function or a set of functions is selected, an automated tool will convert the software from a procedural language representation, in this case C, into a synthesizable RTL HDL representation. A software function is automatically created that interface the remaining software on the processor to the Hardware function that has been created. Finally, a verification testbench with stimulus and expected responses is created for the new HDL implementation.

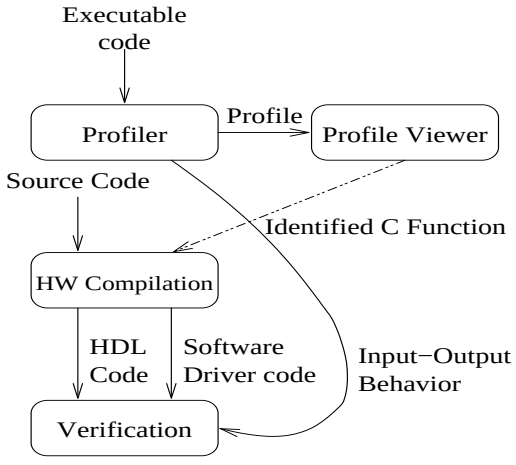

Figure 1. The design flow

2. Related Work

There are numerous academic and commercial tools that convert modified versions of software programming language source into an HDL representation suitable for

RTL synthesis. These include Handle-C from Oxford University [4] and Spec-C from the University of Irvine [5] and a host of others [1, 2, 3, 6, 7, 8, 9, 16]. Generally they have started with the C language and targeted either the VHDL or Verilog language. Since C has no notions of parallelism or the passage of time, these concepts are added, either as extensions to the language (or via "pragmas") or are built into the tool and annotated by the programmer against the original C description. In effect, these tools allow a designer to use a software programming language as a hardware description language. This can be quite useful when the original implementation of an algorithm is in a given programming language, assuming that the performance and area constraints can be met by the tool. Some other approaches [12] perform extensive analysis of the program to come up with the hardware to be implemented. Also, an interesting idea has emerged from Carnegie Mellon University (CMU) [11] to seamlessly migrate software functions into hardware while providing a facility to allow the hardware to make system calls into an RTOS.

However, describing hardware and writing software are two quite different tasks and they are not easily interchangeable. Using code that was originally written as software, and simply compiling it through a converter to synthesizable HDL rarely (and only coincidentally) produces satisfactory results when compared with what a skilled hardware designer could achieve. Writing Spec-C or Handle-C for an efficient hardware implementation requires a different skill set than writing a C program to be run on a general purpose processor.

3. Software to Hardware Migration

The input to our approach is software, as it would be written for an embedded system - not a modified or extended language. For the purposes of this paper the specific language used will be ANSI C, but the same concepts should apply to any programming language. We will note the exceptions to the ANSI C standard, which cannot be supported by our approach. Our design flow will build on the existing C to VHDL tools, specifically Bach-C from Oxford University [18].

3.1 Bach-C

Bach-C system has several features which make it attractive to use in our design flow. Firstly, it performs a control flow analysis to schedule as many operations in parallel as possible subject to the constraint on the hardware use. There are several controls that can be described using the code annotations (using pragmas) to control the C to RTL conversion for speed or for space. The system can identify the parallelism within the loops and can pipeline the loop execution making the generated hardware efficient.

3.2 Limitations

Bach-C had a number of limitations on the ANSI C that it would support, we extended the Bach-C compiler to address many of these, but there are still several C constructs that we do not support. Because the algorithm is converted to a complete hardware implementation (as opposed to a programmable architecture and a program) there is no stack, thus all programming structures that rely on the stack cannot be supported, these include variable argument lists and recursion. The "goto" keyword is not supported – support for this could cause inefficient hardware to be created. The union operator is not supported, this was simply a matter of time. Finally, real floating point is not supported – our approach automatically converts floating points to fixed-point numbers. In many cases this yields a more efficient implementation, but if the application needs actual floating point values then it cannot be supported by this approach.

3.3 Implementation Efficiency

Even though the implementation created by the Bach-C compiler does not reach the performance that skilled hardware designers create, we believe that it will be adequate to significantly improve the throughput of the overall system.

In our experience, a hardware implementation automatically created by a high-level synthesis program, such as Bach-C, runs the algorithm in hardware at least 20 times faster then the same algorithm running on a general purpose processor such as ARM926ejs or PowerPC 405d5. A skilled hardware designer may be able to improve on this implementation, in some cases, by as much as a factor of 10. However such a handcrafted solution would improve system throughput only marginally as shown in Figure 2. In general, any further performance improvements than what is obtained through automated tools will provide only minimal impact to the overall performance of the system with diminishing returns for the effort.

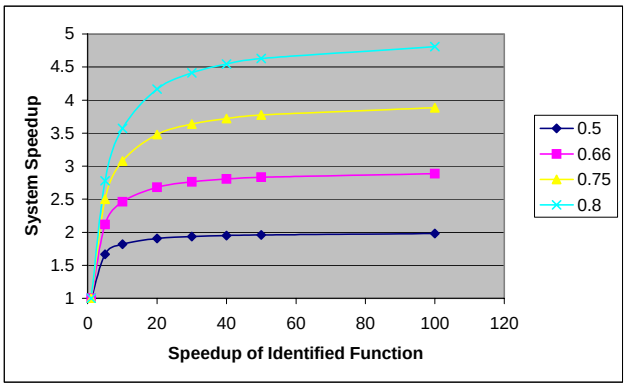

Figure 2. System Speedup vs. Speedup due to the migration of the functionality to hardware. Each curve corresponds to different percentage of total time consumed by the software functionality being considered for migration

4. Profiling the System

Our design flow begins with getting a detailed profile of the system being developed. This profile includes artifacts of code execution, specifically the entry and exit from all software functions and all branch instructions that are taken. From this set of data the complete flow of the program can later be reconstructed. On function entry the input parameters of the function are logged and on exit the return value is logged. The profile also includes all memory transactions from the processor core to any local caches as well as all memory transactions from the caches to main memory. Finally, the profile includes the bus traffic on buses specifically identified by the designer.

This profile is initially used to determine the amount of time spent on different software functions over time as the system executes. We built a viewer system that allows the design to view this profile data in a number of ways to help identify candidate functions for conversion to hardware. One viewer shows the code profile. It shows the percentage of time spent in each function, much like code profilers found on general purpose desktop computers like gprof [19]. The major difference being that the designer can specify an arbitrary starting time and end time for the profile. Quite often, at least in embedded systems, it is important to see the profile during specific operations and not from the start of code execution to the end of code execution. Another view shows a Gantt chart of the software execution by function over time. This provides the designer with a view of how the software is executing over time.

We collect our profile data using a commercial simulation environment called Seamless from Mentor Graphics [10]. This is a tool that includes an HDL simulator for the external logic of the design and an instruction set simulator (ISS) for the software running on the processor. This tool has a number of optimizations that allow for execution of the design faster than traditional logic simulations [20]. However, even while taking full advantage of these "optimizations" this environment ran quite slowly, sometimes requiring runs greater than 48 hours. This tool did give us the ability to non-intrusively collect data on the system operation needed for later analysis. The same data could have been collected off an FPGA evaluation board like the ML-300 from Xilinx [21] and we are looking into ways to perform this data collection as part of our future work.

5. Performance Estimation

Once the candidate functions have been identified, the next step is to determine the impact of migrating the function from software to hardware. There are three areas that need to be considered, the area in FPGA resources that will be consumed by the hardware implementation, the time in clocks (at a given frequency) that the function will take to complete in hardware, and finally, the communication overhead associated with having the function in hardware. When considering the communication overhead, the existing load on the communication paths should be included in assessing the overall performance. Estimation of these items is performed in the following manner. First the embedded C function (or set of functions) is wrapped in a Bach-C function. Bach-C implements a set of extensions to the ANSI C standard. These extensions introduce "channels" as a way of describing hardware type interfaces between functions. We developed a utility that reads the program signature from the C file and creates a Bach-C wrapper that implements registers of an appropriate size for the input parameters, the return value (if not a "void"), any output parameters (output parameters are those passed by reference). There is also a 1 bit register added as a "busy" semaphore, a 1 bit register as a ready bit – signifying that the inputs are in place and processing can commence, and a done bit, which is written by the function after the return value and outputs have been registered. This Bach-C wrapper is then put around the ANSI C function and the result is run through the Bach-C compiler and it is directed to generate a cycle accurate model in C, but at the RT level of detail. Thus all of the state machines and control logic that will end up in the final implementation are created, but the output language is C and not VHDL. This C code describes the hardware in a way that is quite similar to System-C [22]. While this implementation is constructed, it uses a characterized version of the FPGA synthesis library to determine the size of the implementation. As Bach-C goes through the compilation process, it records the number of functional units used, (LUTs, multipliers, etc.) and reports this usage at the end of the compilation. This provides an estimate of the area that will be consumed by the function in hardware. Most RTL synthesis tools can provide further optimizations

than Bach-C, so this estimate generally ends up being 5 to 10 percent higher than the final implementation.

To estimate the performance, the cycle accurate C model created by Bach-C is run in a simple cycle simulator. The inputs to the function, which were collected as part of the performance profiling step are passed as inputs to the input registers and the function is executed and the results are received and compared against the results that were received when the code was run as software. This verifies that the behavioral synthesis transformation resulted in a correct implementation. Also, since the simulation is clock cycle accurate, this provides an accurate measure of the time in clock cycles that the function will take when implemented in hardware. This takes into account all of the data dependent computation time, since the same input values are presented to the hardware block as we processed when the function was implemented in software. We must stress here that to draw a valid conclusion on which function to migrate requires that a representative set of data be run through the simulation of the system when the profile is obtained.

To determine the communication overhead, the interface to the processor needs to be known, as well as some of the characteristics of the software interface. The software interface is created at the same time that the Bach-C wrapper is created, so the number of transactions and their attributes are known at this time. Using this data, along with the bus loading from the system profile, the communication overhead can be computed.

To present the performance data to the designer, a new performance profile is created which can be viewed by the viewing tool referenced above. This is done by removing all data from the original performance profile associated with the function in software and then replacing it with the new performance data collected during the cycle simulation of the C model. In this way, the designer can quickly see the impact of the change to the overall system. This method requires simulation of only those parts of the design that are to be changed, thus it is much faster to observe the results. There are some second and third order effects which are not taken into account by this method; in most cases these effects are inconsequential. One area where this estimation method would fail is if hardware related events were moved from one time-slice given to an RTOS task, where the RTOS is operating in a pre-emptive mode. In this case a small change in the timing of software events in a give task can have large effects on the operation of the system.

It is essential to determine the performance of the system with the new hardware. While the function being converted is almost certain to run significantly faster than a software implementation, when the overhead of communication, data transfers and bus loading are considered not all functions will end up going faster in hardware than in software.

6. Implementation

Once the candidate functions have been identified and the performance gain has been estimated, the next step is to go to implementation. This involves creating the synthesizable HDL for the function, a connection to the appropriate interface on the processor, and the replacing the software function with the created software interface.

The first step in going to implementation is creating the synthesizable HDL. For this the Bach-C compiler is used on the created Bach-C file. This time, instead of C, synthesizable VHDL or Verilog is created. There are a number of options for the creation of the HDL to allow it to conform to a number of coding styles and standards. Next, an HDL wrapper is created that combines the parameter and control registers of the created hardware to a synchronous SRAM type interface. Each of the parameter registers is assigned an address. A local decoder directs data to the appropriate register off the data input bus. A multiplexer is used to drive the outputs to the data output bus. This synchronous SRAM (SSRAM) interface is then wrapped in a bus wrapper to a standard bus. We have created bus wrappers for the ARM AMBA interface, ARM TCM and ARM co-processor interface in addition to the PLB and OCM interfaces that exist on the Xilinx VirtexIIPro and to the FIL interface for the MicroBlaze. An extensible interface was defined so that new bus interface modules can be easily added by the designer for additional bus support. For ease of integration with the rest of the circuit, for Xilinx devices, the component is packaged as a Xilinx Platform Studio component, where it can be “dragged and dropped” into the circuit [23]. From that point, on the hardware side, the remainder of the implementation uses the standard FPGA vendor tools.

A software function is created which has the same name and signature as the function being migrated. We call this function, the hardware driver. Instead of performing the computation of the original function, it handshakes with the hardware implementation, passes the input values to the hardware and retrieves the results. The original software function needs to be removed from the application and the new hardware driver function needs to be put in its place. In this way there are no other changes needed to the overall body of software.

The hardware driver function, when invoked, acquires a lock on the hardware function. This is implemented in the Bach-C wrapper as a hardware semaphore, which is set when it is read. This is done to prevent multiple threads of an RTOS or multiple CPUs from accessing the same non-reentrant hardware simultaneously. If the hardware is unavailable, the software can either wait for

the hardware to be released, or if it is being run on an RTOS it can relinquish its thread. Once the lock is acquired, the input parameters are written to the appropriate registers. Then the "go" bit is set. The "go" bit signifies to the hardware that all input parameters are set and processing can begin. The hardware block will complete the processing and write output parameter and the return value to the output registers and then set the "done" bit. The software either polls the done bit, or if it is part of an RTOS it relinquishes its thread until the processing is done. Once the done bit is set, the results are picked up by reading the output registers. Finally, the lock is released by clearing the semaphore.

The polling type interface that is created, while very inefficient, was easily created and guarantees correct operation. It is easily modified by the software designer to a more efficient interface. In some cases the polling interface is preferred, this will be the case if the function completes in a very few clocks. For functions that take an intermediate number of clocks it is most desirable to have the CPU perform software functions in parallel with the hardware function. For very long functions, an interrupt driven function would be ideal. Even though we generate the interrupt in the hardware, we do not use it in the generated hardware driver function primarily due to the complexity and lack of standardization surrounding interrupts driven systems. Our future work will include analysis of the software running on the general purpose processor to automatically find opportunities for parallelism. For now, running software in parallel with the hardware function needs to be done manually by the designer.

Pointers and arrays in the software function can be handled in a couple of different ways. In one case, the pointer may be passed to the hardware block, and the hardware block will then use the pointer as an address and dereference it through a bus master interface that is added to the hardware function. Alternatively, the software interface can copy N bytes from memory pointed to by the pointer at the start of the function and copy back the same N bytes at the conclusion of the function. If the data pointed to by the processor is referenced frequently it is better to copy the data to the hardware block, as the current implementation does not have facilities for the hardware block to maintain local caches. If the data is sparsely or infrequently referenced, then it is better to pass the pointer to the hardware block and allow the hardware to perform the references directly. If the data is referenced by the hardware block it is important to make sure the data is not in the processor's cache – as the hardware block will not have access to the current version of the data.

The syntax that used to direct the tool to handle the pointer in these two ways is to put N in the function signature. For example if a function dct has a buffer of 64 short integers that we want to copy down to the hardware block because the data is referenced frequently we would define the function as

```
void dct(short buf[64])
```

This is referred to the "copy" syntax. The observant reader will immediately note that a simple DMA device can be constructed by converting "memcpy()" to a hardware block. Alternatively, if we wanted to add a bus master interface to the hardware block and have the hardware reference the data directly we would use the unbounded array or pointer syntax such as one of the following.

```
void dct(short *buf)

void dct(short buf[])
```

For values that are returned from a function as parameters, rather than the return value, the "copy" syntax could be used. For example a function that returns an integer value by reference would be defined as follows:

```
int foo(int output_param[1])
```

All of these are legal C syntax, and do not interfere with the development or testing of the code. In cases where the connection to the processor is through a local memory bus or a co-processor type interface a bus master cannot be added to the hardware block. In such cases, the hardware function will not be able to dereference the data. For such interfaces only the copy syntax is supported. It may be noted that even when the hardware block uses the bus master interface to dereference, the software block running on the CPU can work concurrently on some other part of the code. As with any multiple master design, the CPU and the hardware block may collide for the bus and will arbitrate before assuming the mastership. As a recommended design practice, the hardware block must be given precedence over the CPU to get the bus access.

7. Verification

Once the new HDL is created through an automated process, it needs to be verified. In typical design cycles today verification takes significantly more effort than design [24]. Through the data that was collected as part of the performance profiling we have the input values to each invocation of the converted function and the return value. If the function performs a pointer dereference these accesses to memory have been recorded as well. For memory accesses we verify only that the last memory write to any memory location from the hardware device is

the same as the last memory value written by the software implementation. This ensures that the memory image of the system is the same after either the hardware or software implementation is run. Certain memory locations may be I/O devices, where the number and order of memory cycles is significant. For this we allow the designer to designate memory regions as "volatile". For all volatile memory accesses the verification program ensures that the number and order of memory accesses is maintained by the hardware implementation.

8. Results

In our approach, the designer is quickly able to convert the software to the hardware and assess the impact on the performance. Thus this permits him to do a "what-if" analysis for fine tuning the design. We have applied this system to a number of algorithms, which are accessible in the public domain. In this paper we present the results for the G.729 audio encoder [17].

8.1 G.729 Audio Encoder

The G.729 audio encoder [17] is a compute intensive algorithm which encodes audio data at rate of 8 KHz. It uses conjugate structure algebraic-code-excited linear prediction (CS-ACELP). The source code for a reference algorithm can be found at http://www.itu.int.
Running this code as provided by the International Telecommunications Union (ITU) on a MicroBlaze processor requires 10.3 millions clocks to process one buffer of 80 8-bit data samples. The profile of function usage is shown in figure 3. Almost 40% of the load on the CPU is being consumed by the function L_Mac(). This is a relatively small function which takes two parameters as input and returns a single value. On average, this function takes 91 clocks to complete (using the test datasets provided by the ITU). When converted to hardware this function can be completed in 4 clocks cycles. Added to this time is the time required to pass the data to the function and the time to pick up the result. From the processor to the FPGA fabric using the main processor bus through a memory mapped interface adds 8 clocks per access (in our circuit, other configurations may vary). There is a minimum of 6 accesses that need to be made to complete this function if the semaphore is used. This reduces the function to 48 clocks, since the 4 clocks of computation in hardware are run in parallel with the accesses. This yields a 19% improvement in throughput. However, in an implementation if it is known that more than one thread cannot access the hardware block at the same time, the hardware semaphore implementation can be avoided. This then results in a 25% increase in performance of the audio decoder.

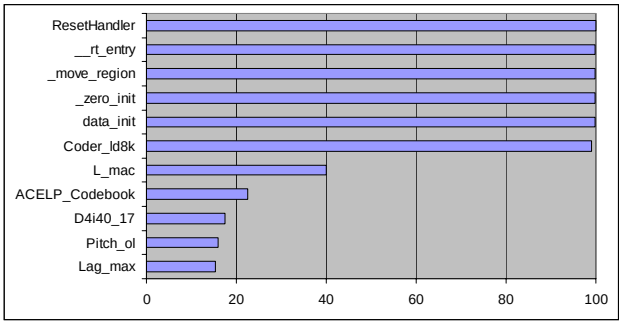

Figure 3. Code Profile, cumulative by function

A higher performance interface is available on the MicroBlaze processor, the FIL interface. This provides a co-processor like interface where a single instruction can move data to and from a non arbitrated interface. Since this function does not dereference a pointer, it can be put on this interface. Accesses on this interface take 2 clocks (again, on our design). The total time to run this function using this interface is 12 clocks. This is an 87% improvement in the throughput of this function, and gives a 35% improvement in throughput for the overall application.

Using the different interfaces on the hardware side is a simple means of selecting a different HDL "wrapper" which goes between the processor and the hardware block. On the software side, the interface function is the same for both interfaces; the code that accesses the registers of the hardware is a set of C macros. A different header file is included for each interface, which provides different macros that will access the hardware in the appropriate manner. For memory mapped registers, the macros provide a dereferencing of an offset from the base address of the hardware block. In the case of the FIL interface, the macros provide the required in-line assembly language statements that read and write to the FIL interface.

With the function L_mac() moved to hardware, using the FIL interface the encoder runs 35% faster. Profiling the new configuration of hardware and software shows that the function ACELP_Codebook() takes approximately 25% of the remaining time on the processor. The ACELP_codebook() function is comprised of 3 main functions, D4i40_17(), Cor_h(), and Cor_h_X(). D4i40_17() takes about 17% of the remaining time, with Cor_h() and Cor_h_X() taking about 5%. Each of the three functions was converted individually.

The function D4i40_17() had two changes to the source code that were required before the code could be synthesized correctly. First, in the main body of the function, there is "goto" which is used to exit the function after a maximum number of iterations. This goto was eliminated by converting the code to use if-then-else constructs. The second change was to eliminate potential

recursion in the code. The shift left function, L_shl(), calls the shift right function, L_shr(), when passed a negative parameter as the shift, and the shift right function calls the shift left function when passed a negative parameter. While this never actually causes recursion in the code, the static analysis performed by the Bach-C compiler find this potentially recursive path. Copying parts of the shift left function into the shift right function (and vice versa) eliminated this problem. The remainder of the code was converted at it was received from the ITU.

The software implementation of the D4i40_17 function takes approximately 1.5 million clocks to complete. In hardware it takes about 129,000 clocks to complete. As stated earlier, a skilled hardware designer could implement this function to take less time, but after conversion it amounts to less than 1% of the total time processing time for the algorithm.

Among the parameters to D4i40_17 are several arrays. Because these variables are reference frequently in the algorithm, the copy interface is used. Since the array parameters are only input or output parameters only it is not necessary to copy both to and from the hardware block. Through "pragmas" the arrays are defined as input or output arrays, and they are only copied when needed. Likewise on the Cor_h() and Cor_h_X() functions the arrays are passed by copy.

Cor_h and Cor_h_X are not dependent on each other, that is they can be run in parallel. This is not done automatically by the tool, but the software interfaces that communicate with the block can easily be broken down into "start" and "end" functions. In this way, the hardware for Cor_h() can be started – that is to write the input parameters to the input registers and write to the start bit. Then the hardware for Cor_h_X can be started. Then the results for each be picked up. In this way the greatest performance increase can be achieved. Although it requires restructuring the code, a similar approach can be used to run the CPU and the hardware blocks in parallel. This will effectively reduce the time for a function to be executed in hardware to the time to initiate the operation.

With D4i40_17, Cor_h, and Cor_h_X converted to hardware the system runs 22% faster. The combined effects of moving the 4 functions, the previous three plus the L_mac function, provides a 50% speed up for the algorithm. Only minimal code changes were required to one of the functions to achieve these results. The time required to perform this repartitioning of the system is a matter of a few days.

9. Conclusion

The approach presented allows embedded system designers to postpone some of the hardware/software partitioning decisions when implementing systems that include FPGAs with embedded processors. Certain functions can be implemented in software and quickly migrated to software if the performance is found to be inadequate for the application. We believe that through out quick turn-around methodology, a designer is empowered to explore different alternatives to reach to an acceptable design. With programmable hardware such as FPGAs this will also enable him to quickly prototype the design and evaluate different alternatives to improve the system performance. Our system will estimate the area and performance, including the communication overhead. A synthesizable implementation will be created, as well as the desired hardware and software interfaces. Finally, a verification testbench with stimulus and expected results is created from an earlier simulated execution of the system. Our results show that the throughput of a system could potentially doubled by utilizing the FPGA fabric to implement functions originally in software

References

[1] Synopsys Inc., "CoCentric SystemC Compiler", http://www.synopsys.com

[2] Jon Connell, Bruce Johnson, "Early Hardware Software Integration Using SystemC 2.0" Embedded Systems Conference 2002, San Francisco, 2002

[3] Celoxica Ltd., "C to RTL, Software Compiled System Design", http://www.celoxica.com, Oxford, UK

[4] S.M. Loo, B Earl Wells, N Freije, J.Kulick, "Handel-C for rapid prototyping of VLSI Coprocessors for Real Time Systems", Proceedings of the Southeastern Symposium on System Theory (SSST-2002) pp.6-10, Huntsville, AL, March 18-19, 2002

[5] D. Gajski, J. Zhu, R Domer, A Gerstlauer, S Zhao, "Spec C: Specification Language and Design Methodology", University of California at Irvine, 1999

[6] Giovani De Micheli, "Hardware Synthesis from C/C++ Models", Proceedings of Design Automation and Test Europe (DATE) pp.382-383, 2003 Munich, Germany

[7] Guido Arnout "C for System Level Design", Proceedings of Design Automation and Test Europe (DATE) pp.384-386, 2003, Munich, Germany

[8] CoWare Inc., "CoWare N2C Products", http://www.coware.com

[9] Critical Blue, "Cascade Design Tools", Edinburgh, UK http://www.criticalblue.com

[10] Mentor Graphics Corp., "Seamless Hardware Software Co-verification", Wilsonville, OR, http://www.mentor.com/seamless

[11] Mihai Budiu, Mahim Mishra, Ashwin R. Brarambe, Seth Copen Goldstein, "Peer-to-peer Hardware-Software Interfaces for Reconfigurable Fabrics", Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines 2002 (FCCM) Napa, CA 2002

[12] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist, M.Sivaraman, "PICO-NPA: High Level Synthesis of Non-programmable hardware accelerators", Technical Report HPL-2001-249, HP Labs, October 2001

[13] S. Liebson, "SoC Logic Development Using Configurable, Application Specific Processors", International Symposium on SoC 2003, Tampere, Finland, November 19, 2003

[14] Tensilica Inc., "The Xtensa Processor", http://www.tensilica.com Santa Clara, CA

[15]ARC Ltd. "ARC-700 Processor", http://www.arc.com/products/soc/microprocessors/arcprocessors/arc7_processor.html

[16] A. Singh, A. Chabra, A. Gangwar, B. Diwedi, "SoC Synthesis with Automatic Hardware Software Interface Generation", VLSI Design 2003, Delphi, pp. 585-590, January 2003

[17] International Telecommunications Union (ITU) "Coding of Speech at 8kbits/s using Conjugate-Structure Algebraic-Code-Excited Linear-Prediction (CS-ACELP)", ITU-T Recommendations G.729, March 1996

[18] T. Kambe, A. Yamada, K. Nishida, K. Okada, M. Onishi, A. Kay, P. Boca, "A C-Based Synthesis System, Bach-C, and its Application", Proceedings of the Asian South Pacific Design Automation Conference, pp. 151-155, 2001

[19] S. Graham, P. Kessler, M. McKusick, "Gprof: A Call Graph Execution Profiler" Proceedings of SIGPLAN Symposium on Compiler Construction pp.120-126, Boston MA, 1982

[20] R. Klein "Miami: A Hardware Software Co-verification System" Proceedings 7th IEEE Workshop on Rapid Systems Prototyping, 1996, p173-177.

[21] http://www.xilinx.com/products/boards/ml300

[22] W. Müller , W. Rosenstiel , J. Ruf, "SystemC: methodologies and applications", Kluwer Academic Publishers, Norwell, MA, 2003

[23]http://www.xilinx.com/publications/products/v2pro/xc_edk45.htm

[24] J. Bergeron, "Writing Testbenches, Functional Verification of HDL Models", Kluwer Academic Press, 2000