A Software Oriented Approach to Hardware/Software Codesign

Axel Jantsch, Peeter Ellervee, Johnny Öberg, Ahmed Hemani, Hannu Tenhunen
Royal Institute of Technology
KTH-Electrum, ESDlab, Electrum 229, S-16440 Kista, Sweden
Email: axel@inmic.se

Abstract

We present a software oriented approach to hardware/software codesign by applying traditional compiler techniques to the hardware/software partitioning problem and linking a compiler to a state of the art hardware synthesis technology. The system is specified in C or C++. Time critical regions are identified by means of profiling and are automatically implemented in user programmable logic with high level and logic synthesis design tools. The underlying architecture is an add-on board with user programmable logic, connected to a Sparc based workstation via the system bus. We present a novel partitioning technique based on a hierarchical candidate preselection scheme, that utilizes profilers and estimators for performance and cost. Our approach allows (a) efficient collection of profiling data due to the usage of C and C++ as specification languages, (b) fast partitioning due to a candidate preselection scheme, and (c) high complexity of the hardware partition due to a logic emulation system.

Introduction

A critical issue in complex system design is finding effective hardware and software partitioning quickly with good area predictions and performance estimations. This early system partitioning must also result in more refined specifications, which can be directly used for automatic hardware and software generation. The advent of mature high level and logic level synthesis tools

for hardware design allows for automation of HW/SW codesign and for systematic exploration of trade-offs of HW/SW partitioning at the system level, because they bridge the gap between and algorithmic specification and its implementation at the layout level.

In this work we present an approach to HW/SW codesign, that utilizes state of the art software compiler and hardware synthesis technology, and propose a novel partitioning technique. We believe that a software oriented approach that implements everything per default in software and moves only time critical parts into hardware is superior to an approach that starts with an all-hardware solution [1] because it allows a much higher system complexity. C++ serves well as an implementation neutral system specification language and was chosen based on our experiences in telecommunication systems, where C++ or its dialects like μC++, is used directly in system specification and modeling, or it is generated by other system specification tools [15].

The architecture of the demonstrator system consists of a conventional workstation with a user configurable add-on board that is connected to the workstation via the system bus [13]. User configurable logic can be provided either as field programmable gate arrays from Xilinx or as a logic emulator system with up to 100000 gates from Quickturn.

Objectives of this research are (a) to define the class of programs that are well suited for acceleration with the proposed architecture and (b) to explore the relations between properties of the program and architectural parameters. Although we are concentrating on conventional programs we are convinced that our results are relevant for embedded system design and our techniques can be easily adopted to this area, since similar approaches have been taken there ([3] [4] [5]).

Figure 1 shows the overall design flow. The source program is partitioned by an extended version of the GNU CC compiler [2] into hardware and software parts. The hardware partition is derived from one or more source code regions and translated automatically into behavioral VHDL. VHDL is a standard hardware description language and is used to describe hardware at the structural and the behavioural level. The behavior of a hardware component is described as an algorithm which is sufficiently similar to imperative programming languages like C to make the translation process painless. The high level synthesis system SYNT [9] [14] translates the behavioural description into a structural VHDL description by allocating hardware units and scheduling operations in control steps. The logic synthesis system Synopsys and finally ASIC vendor specific place and route tools transform the design into configuration files for the user programmable logic. The software part will usually constitute the larger fragment and is compiled for the host workstation.

In this paper we focus on the partitioning problem and suggest a *hierarchical candidate selection scheme* which, together with the use of *profilers* and *estimators* allows us to solve the partitioning efficiently with a dynamic programming technique. We also present results from initial experiments with the programs egrep and gzip, and we demonstrate how we can

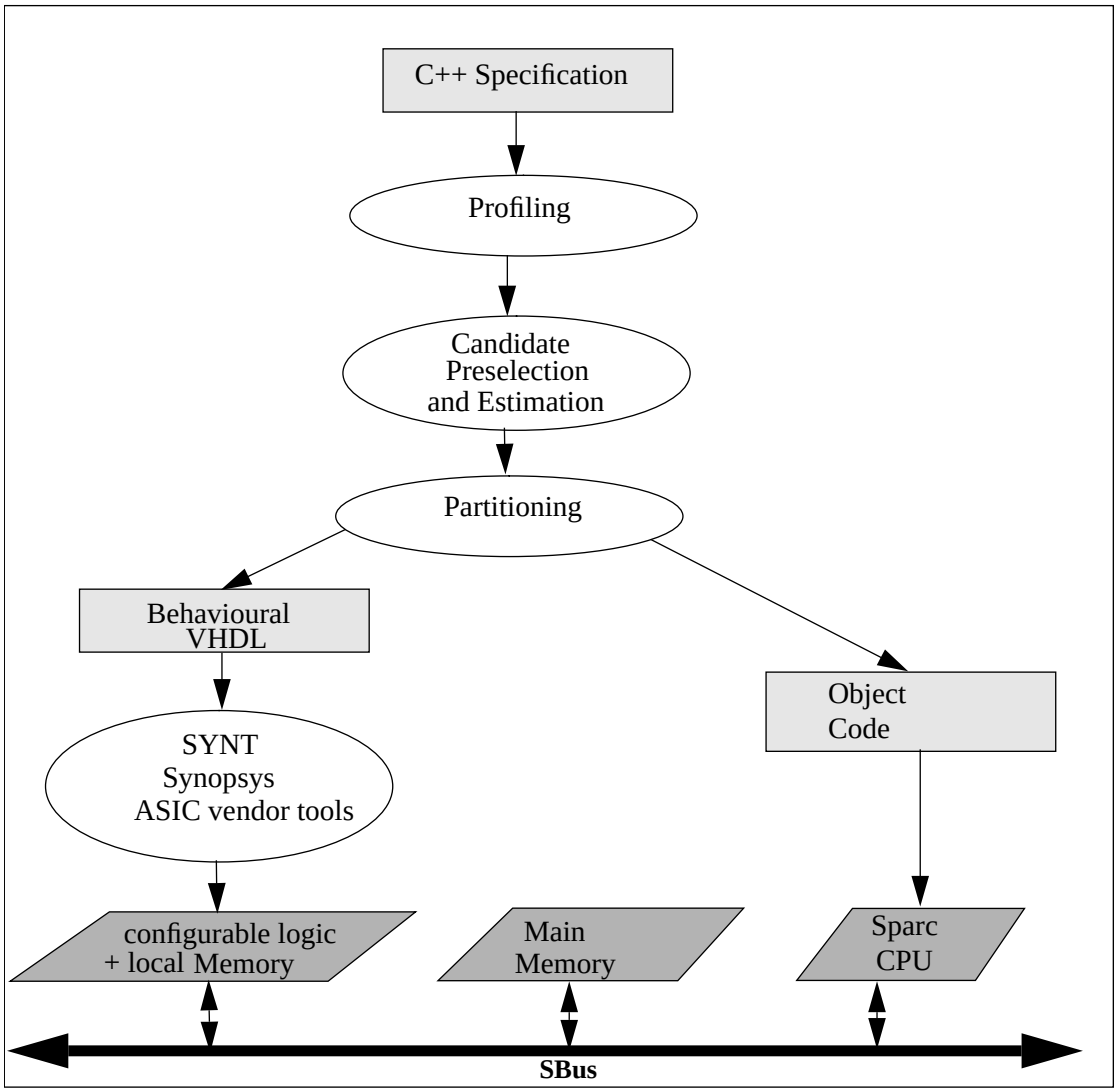

FIGURE 1. **Design flow**

rapidly explore the effects of architectural parameters on the gained speedup for a given program.

Related Work

Athanas 6 ambition to automatically partition a C program in HW and SW parts and implement the HW parts in FPGAs is similar to ours but less general. He considers only pure combinational circuits without using high level synthesis techniques. He does not allow main memory access from the FPGA board and the partitioning process is guided by a human user.

Ernst and Henkel 4 use an extension of C as specification language for embedded controller design. Their objective is to extract code segments for implementation in hardware when timing constraints are violated. Their partitioner is based on simulated annealing and considers every segment of C statements as a possible candidate, which is inefficient since

most of the segments are not good candidates when the program includes loops. The profiling data is collected by a simulator which is much slower than the compilation and execution of a C program. This makes it hard to run long programs with extensive test data.

Martin Edwards 3 describes a system in C, collects profiling information by compiling and executing the program. But he considers only functions as candidate regions and may miss more promising larger or smaller regions like inner loops. As in Athanas' work the regions are selected by a human user and no automatic partitioning is reported yet.

Pengs and Kuchcinskis 7 partitioner is based on simulated annealing and operates on a petri-net based internal representation. It uses profiling information, which is collected by a simulator, and information about static connectivity between operations. As in the case of Ernst and Henkels project the partitioner considers far too many partitions and the simulator is much slower in collecting profiling data in comparison to the execution of a C program.

The work on the Mark-I architecture by Lewis et.al. 8 has the goal to accelerate a class of programs by defining an application specific instruction set that is executed on an array of 16 Xilinx FPGAs. The definition of a special purpose instruction set for a given program is similar to the identification of candidate regions and their implementation in FPGAs. Lewis et. al. have not yet automated the partitioning, but focused on the development of the board architecture, while we are concentrating on the partitioning problem.

Gupta and De Micheli 1 propose a hardware oriented approach and use HardwareC, a limited subset of C, as system specification language. Their design system gradually moves functions into software while considering timing constraints and synchronism. The hardware oriented approach and the use of HardwareC severely limits the overall system complexity.

Our approach allows (a) efficient collection of profiling data due to usage of C and C++ as specification languages, (b) fast partitioning due to a candidate preselection scheme, and (c) high complexity of the hardware partition due to a logic emulation system.

Partitioning

The GNU CC compiler 2 consists of more than 20 passes. Each function is parsed and processed by all passes before the next function is started. System partitioning is performed by a separate pass between data flow analysis and instruction scheduling. It uses information from the data flow analysis and generates pseudo instructions for the code regions that are implemented in hardware. The pseudo instructions can be scheduled by the instruction scheduler pass as any other instruction to exploit potential parallelism between the CPU and the accelerating add-on board. In the assembler code generation phase the pseudo instructions initiate generation of code that passes parameters and invoke the design in the user configurable hardware.

The hierarchical candidate selection scheme preselects code regions for hardware implementation in a bottom up way starting from inner loops and leaf functions. Thus, we consider a region of code as candidate,

- iff it does not include floating point operations or calls to external library and operating system functions
- and
  - if it is an inner loop or leaf function
  - or it includes only loops and calls to functions that are candidate regions themselves.

This definition of candidate regions has two main advantages:

1. Real programs vary considerably in how hot regions of code look like. For instance, in egrep we found the most promising region to be an inner loop; in gzip it is a loop containing another loop.
2. By considering only these regions for hardware implementation we have to deal with a limited subset of all possible partitions. This allows us to solve the partitioning problem very efficiently and we do not rely upon computational expensive heuristics as e.g. 7 and 4.

We use profilers and estimators to guide the selection process and, as we use C++ or C as specification language we can compile, execute, and collect profiling data using standard Unix profiling methods. However, as these methods allow only profiling of functions we enhanced them to collect data for loops as well. The profiling information tells us which regions account for large amounts of run time and *should* be implemented in hardware. The estimators estimate area and performance of the hardware implementation, thus telling us which regions *can* go into hardware and what the expected speedup factor is. Because of the long synthesis path from the partitioner to the user configurable logic implementation it is necessary to use estimators which predict the combined effect of this chain of synthesis tools. These estimators are based on neural networks and currently under development in our group 10. For the experiments described below we used a simpler estimator that does not consider interconnect area and delay.

To estimate speedup factors we also need an architectural model that describes how the hardware implemented regions communicate with the main program and memory. We use a set of parameters that define the size of the local memory, the number of local memory banks, the access time to local and main memory, etc. By changing the values of these parameters we can efficiently explore the relationship between program and architectural properties.

Based on a set of preselected candidate regions we formulate the partitioning problem as to find a subset of the candidate regions that would gain the greatest speedup while still fitting in the available user configurable logic. This is similar to the well known knapsack problem which usually can be solved efficiently by applying dynamic programming techniques 11 12. The only difference in our problem is that candidate regions in the same hierarchy, such as an inner and an enclosing loop, are mutually exclusive. Any algorithm that solves the knapsack

problem can easily be adjusted to this situation by excluding some of the solutions. But the number of candidate regions that account for a reasonably large portion of the running time, e.g. more than 10%, is small enough that even an exhaustive search algorithm is feasible.

The compiler has to be invoked two times, once to insert the profiling marks and secondly to generate the assembler and VHDL code according to the selected partitioning, which can be seen in figure 2. After the profiling information has been collected a separate program invokes the estimators to estimate the properties of the hardware implementations of possible candidate regions and calculates the speedup factors for each individual candidate region. When all these data has been collected and calculated the partitioner takes a global view of the whole program, which might consist of many functions and files, and selects those candidate regions for hardware implementation that fit on the available hardware and together provide the highest speedup.

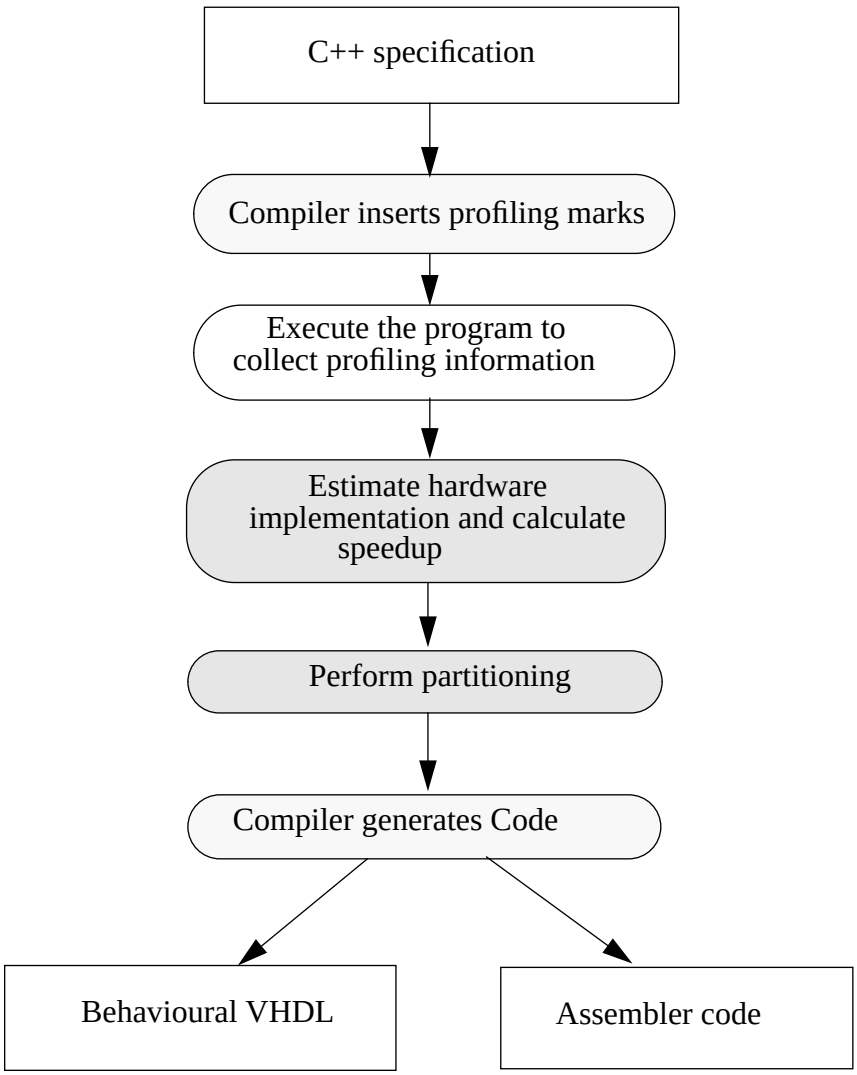

FIGURE 2. **Partitioning tool flow**

Implementation of Candidates

The preliminary mapping strategy of candidate regions onto user configurable logic, which we describe in this section, is only possible due to the progress in high level synthesis techniques and tools during the last ten years.

Each region that is selected for hardware implementation is translated into a VHDL process. This translation is straight forward since all simple data types and operators of C can be expressed in behavioural VHDL when the appropriate arithmetic packages are used. Pointers in C are translated to memory accesses. Since the FPGA board has access to main memory in the same way as the CPU, pointers, arrays and compound data types can be implemented on the board. For the main program the invocation of such a region is similar to a function call. Values that reside in registers at the time of the invocation and are used or updated inside the region, are passed as parameters. The main memory access requirements of the region are analyzed and we distinguish two cases. If a particular memory location is frequently used it is transferred to the local on-board memory before the invocation and written back to main memory after the invocation; otherwise it is accessed directly in the main memory when the need arises. At compile time we do not know how large a dynamically allocated memory region will be. If such a region is considered to be moved to local memory, we insert a run time test, which conditionally moves the region to local memory, depending on the size of the allocated memory. It can also depend on a run time check, if the corresponding code region is executed in hardware or in software. In this case a software and a hardware implementation must be generated for the same code region.

The SBus interface chip implements direct virtual memory access, thus allowing reasonable short memory access time. Memory locations only used within the regions are allocated in the local memory and are not transferred to or from main memory. In C programs this case is particularly hard to identify since the memory addresses are not known at compile time. C++ on the contrary guarantees that variables local to an object are only accessed by functions of that object. This knowledge can be used to reduce data traffic between the board and main memory considerably when all the functions of an object are implemented in hardware.

The generated VHDL code is processed by SYNT, a high level synthesis system which generates a dedicated datapath and a separate controller, Synopsys, and the ASIC vendor specific place and route tools. The rest of the program is augmented with code that performs the communication between the software part and the regions implemented on the board and a startup routine that loads the configuration files for the user configurable logic down to the board.

Experiments

To test our analysis technique and to direct our research we have used the programs egrep, grep, gzip and sed as test cases. grep and egrep are pattern matching programs, gzip compresses and uncompresses files, and sed is a stream editor. Tables 1 through 3 show some properties of the most promising candidate regions. The entry “Level” indicates the hier-

TABLE 1. **egrep, target architecture 1**

region type	Level	% run time	tot-sw [µs]	tot-hw [µs]	region speedup	program speedup	gate count	function name
func	2	67.80	1.43e+07	4.48e+06	3	1.87	10304	bmg_search
loop	1	67.80	1.43e+07	4.47e+06	3	1.87	5440	bmg_search
loop	0	67.80	1.43e+07	4.47e+06	3	1.87	896	bmg_search

TABLE 2. **egrep, target architecture 2**

region type	- Level	% run time	tot-sw [µs]	tot-hw [µs]	region speedup	program speedup	gate count	function name
func	2	67.80	1.43e+07	2.69e+06	5	2.23	10304	bmg_search
loop	1	67.80	1.43e+07	2.68e+06	5	2.23	5440	bmg_search
loop	0	67.80	1.43e+07	2.68e+06	5	2.23	896	bmg_search

archy. A value of 0 denotes an inner loop or leaf function, greater values denote higher levels of hierarchy. Tables 1 and 2 show the result for egrep assuming two different architectures. Architecture 1 allows only one access to local memory per cycle whereas architecture 2 allows two memory accesses per cycle, as our proposed board architecture will allow due to two separated memory banks.

Table 3 shows the results for gzip. The results for grep and sed are similar to those of

TABLE 3. **gzip, target architecture 2**

region type	Level	% run time	tot-sw [µs]	tot-hw [µs]	region speedup	program speedup	gate count	function name
func	2	63.01	4.88e+07	1.95e+06	25	2.67	20000	longest_match
loop	1	61.70	4.76e+07	1.88e+06	25	2.58	10944	longest_match
loop	0	6.52	4.95e+06	1.1e+05	45	1.07	4000	longest_match
loop	0	9.66	7.86e+06	1.68e+06	5	1.10	672	deflate

egrep, but less promising.

Table 1 and 2 the most inner loops, denoted with “Level 0” would be the best choice, since the program does not spend a significant part of the run time in the enclosing loop or function. However, the implementation area would by several times larger than for the inner loop only. In this case, a technique, which only considers hardware implementation for complete functions would pay a high price in the hardware implementation. Table 3 shows a dif-

ferent situation for program gzip. The best choice depends on the avaliable area. The biggest speed up is achieved when moving from the inner loop to the enclosing loop in function longest_match. An additional speed up of 3% by including the function longest_match is payed by doubling the number of necessary gates. Note, that the region speed up in function longest_match is remarkable high with factors of 25 snd 45, but the overall program speed up is limited by the percentage of the running time the program spends in the respective candidates.

The remarkable improvement of speedup in table 2 over table 1 indicates that memory access time is a main bottleneck, since the only difference between architecture 1 and 2 is the memory access time.

Conclusions

The main obstacles to obtain higher speedup factors are (a) the relative small size of the candidate regions and (b) the frequency of the main memory access. We have to point out, however, that all example programs have not been written with an accelerating hardware in mind. The speedup factors would probably increase considerably if the programmer knows about the potentials of the target hardware architecture. Even the pure automatic approach is not fully exploited and we plan to focus on the following points.

- Implementation of local memory allocation functions that allocate and release local memory dynamicly. Care must be taken, however, that the mechanism that handles local memory overflow is not too expensive.
- Development of data transfer profilers that provide information about
  1. memory regions that are frequently accessed inside candidate regions and therefore should be moved to local memory with a higher priority;
  2. memory regions that are not used outside a candidate region and can therefore be allocated in local memory only;
  3. average and maximum size of dynamicly allocated memory regions.
- Development of analysis and synthesis routines for C++ objects which exploit their inherent data locality.
- Investigation of examples from signal processing and telecommunication applications.

Bibliography

[1] Rajesh K. Gupta and Giovanni De Micheli, “System-level Synthesis using Re-programmable Components”, *Proceedings of the European Design Automation Conference*, 1992.

[2] Richard M. Stallman, *Using and Porting GNU CC,* Free Software Foundation, December1992, version 2.3.

[3] Martin Edwards, “A Development System for Hardware/Software Co-Synthesis using FPGAs”, *Second IFIP International Workshop on Hardware-Software Co-Design*, May 1993.

[4] R. Ernst and J. Henkel, “Hardware-Software Codesign of Embedded Controllers Based on Hardware extraction”, *Proceedings of the International Workshop on Hardware-Software Co-Design*, September 1992.

[5] Th. Benner, J. Henkel, and R. Ernst, “Internal representation of Embedded Hardware-/ Software Systems”, *Second IFIP International Workshop on Hardware-Software Co-Design*, May 1993.

[6] Peter M. Athanas, *An Adaptive Hardware Machine Architecture and Compiler for Dynamic Processor Reconfiguration*, PhD thesis, Division of Engineering, Brown University, Providence, Rhode Island, USA, September 1991.

[7] Zebo Peng and Krzysztof Kuchcinski, “An Algorithm for Partitioning of Application Specific Systems”, *Proceedings of the European Conference of Design Automation*, February 1993.

[8] David M. Lewis, Marcus H. van Ierssel, and Daniel H. Wong, “A Field Programmable Accelerator for Compiled Code Applications”, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1993.

[9] Ahmed Hemani, *High Level Synthesis of Synchronous Digital Systems using Self-Organization Algorithms for Scheduling and Allocation*, PhD thesis, Department of Applied Electronics, Royal Institute of Technology, Stockholm, Sweden, 1992.

[10] Peeter Ellervee, Axel Jantsch, Johnny Öberg and Ahmed Hemani, “Nestimator - A Neural Network Based Estimator”, submitted to the *Seventh International Workshop on High Level Synthesis*, May 1994.

[11] Thomas H. Corman, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 1992.

[12] Udi Manber, *Introduction to Algorithms*, Addison-Wesley Publishing Company, 1989.

[13] Shousheng He and Mats Torkelson, “FPGA Application in a SBbus Based Rapid Prototyping System for ASDP”, *ICCD* 1991.

[14] Ahmed Hemani and Adam Postula, “A Neural Net Based Self Organizing Scheduling Algorithm”, *European Design Automation Conference*, pp 136 - 140, 1990.

[15] SDT Reference Manual, TeleLOGIC Malmö AB, 1993.