Hardware and Interface Synthesis of FPGA Blocks using Parallelizing Code Transformations

Sumit Gupta[†] Manev Luthra[†] Nikil Dutt[†] Rajesh Gupta[§] Alex Nicolau[†]
[†]Center for Embedded Computer Systems, University of California at Irvine, U.S.A.
{sumitg, mluthra, dutt, nicolau}@cecs.uci.edu
[§]Dept. of Computer Science and Engineering, University of California at San Diego, U.S.A.
gupta@cs.ucsd.edu

Abstract

Reconfigurable logic such as FPGAs is increasingly being used on system-on-chip (SoC) platforms to provide a flexible, programmable co-processor that augments the core processor. In this paper, we present a tightly coupled hardware synthesis and interface synthesis approach that forms part of our hardware-software co-design methodology for such FPGA-based platforms. For hardware synthesis, we use a parallelizing *high-level synthesis approach that employs aggressive coarse-grain and fine-grain code parallelizing and code motion techniques to discover circuit optimization opportunities beyond what is possible with traditional high-level synthesis. We have implemented this approach in a framework called* Spark *that takes a behavioral description in ANSI-C as input and produces synthesizable register-transfer level VHDL. Our interface synthesis approach is based on a novel memory mapping algorithm that uses scheduling information from the high-level synthesis tool to map data used by both the hardware and the software to shared memories on the reconfigurable fabric. We present experimental results for the synthesis of computationally intensive portions of multimedia applications that demonstrate that the code transformations in* Spark *lead to up to 50-70 % improvements in circuit delay with fairly constant circuit area. We also present a case study of the hardware-software co-design of a portion of multimedia application onto a FGPA platform using our methodology.*

KEY WORDS
System synthesis, high-level synthesis, interface synthesis, parallelizing transformations, FPGA, platform design

1 Introduction

The platform based design methodology is pegged as a flexible and scalable solution for handling the growing complexity of chip designs [1]. In this work, we target a range of emerging platforms that contain a processor core assisted by dedicated hardware for computationally intensive tasks. The hardware assists are implemented on FPGA blocks integrated on the platform. These FPGAs provide a flexible, reconfigurable co-processor that is useful for speeding-up target applications.

Mapping the application functionality to software and hardware requires automated methods to specify, generate and optimize the hardware, software, and the interface between them. The interface should be fast, transparent and require minimal hardware and software resources. This paper presents a methodology to generate the hardware component and the hardware-software interface.

We focus on multimedia and image processing applications that are data-intensive and also have complex control flow (conditionals and loops). The presence of control flow has a particularly adverse effect on the quality of hardware synthesis results. To alleviate this problem, we have developed a *parallelizing* high-level synthesis approach that employs a range of coarse-grain and fine-grain compiler, parallelizing compiler, and synthesis transformations. These transformations are guided by heuristics that optimize the circuit quality in terms of latency, cycle time, circuit size, and interconnect costs.

Since media applications typically operate on a large data set, we find that when these applications are partitioned into hardware and software, a large amount of data has to be communicated and shared between the hardware and software components. Configurable logic blocks in FPGAs are typically inefficient for use as memories; if we store each data element in a register and provide an independent access mechanism for each one, then the resulting memory implementation occupies a large portion of the FPGA fabric. Instead, we have developed interface synthesis approach that efficiently clusters the data elements onto embedded RAMs on the FPGAs.

In this paper, we present our parallelizing high-level synthesis framework called *Spark*, along with the interface synthesis approach that uses scheduling information from *Spark* to map data to memories. We present an overview of the parallelizing code transformations in *Spark* and results for experiments performed on computationally expensive portions of multimedia and image processing applications. We then present a case study of the hardware-software co-design of the motion estimation portion of the MPEG-1 multimedia application onto a FPGA-based platform.

The rest of this paper is organized as follows: in the next section, we discuss related work, followed by an overview of our co-design methodology. In Section 4, we present the *Spark* synthesis framework and then, we present our interface synthesis approach. We then present results for experiments performed using *Spark*, followed by a co-design case study for a FPGA platform.

2 Related Work

Hardware-software partitioning [2, 3] and high level synthesis [4, 5, 6, 7] have received significant attention over the past decade. Interface synthesis techniques have focused on various issues like optimizing the use of external IO pins of micro-controllers and minimizing glue logic [8, 9].

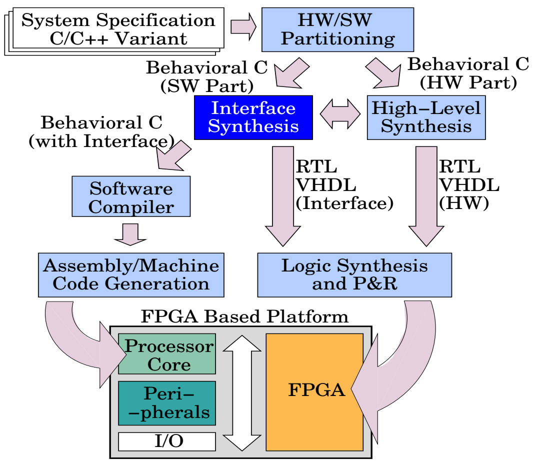

Figure 1. **Role of high-level synthesis and interface synthesis in our co-design methodology.**

Gharsalli et al. [10] generate wrappers to interface processors and memory components in a multiprocessor SOC.

High-level synthesis has been a subject for research for almost two decades now [4, 5]. Early work focused on scheduling heuristics and algebraic and re-timing optimizations for data flow designs. Recent work has presented scheduling heuristics for mixed control-data flow designs, many of which employ speculative code motions [7, 11, 12, 13, 14]. A range of code transformations similar to those we use for high-level synthesis have been presented for software (parallelizing) compilers [15]. However, these need to be re-instrumented for synthesis to use hardware cost models for operations and resources.

Previous work on memory mapping and allocation of multi-port memories has focused primarily on data flow designs [4, 5, 16]. Memory mapping and register binding algorithms in the data path synthesis domain are based on variable lifetime analysis and register allocation heuristics [4, 17]. Previous work for FPGAs has focused on packing uneven sized data structures into fixed aspect memories available on FPGA [18] and on taking the parameters of the memory banks into account during memory mapping [17].

3 Overview of our Co-Design Methodology

An overview of our hardware-software co-design methodology is shown in Figure 1. High-level synthesis and interface synthesis are important aspects of this methodology. After hardware-software partitioning, the hardware part is synthesized using the *Spark* high-level synthesis tool [6] and the scheduling information is passed to the interface synthesizer. The interface synthesizer generates the hardware interface and re-instruments the software component of the application to make appropriate calls to the hardware component via this interface. It also passes the addresses of all registers that have been mapped to memories in the hardware interface to the high-level synthesis tool. The RTL code generated by the high-level synthesis tool and the interface synthesizer is then downloaded to the FPGA. We utilize the compiler suite for the processor on the platform to compile the software component and download the code into the instruction memory of the processor.

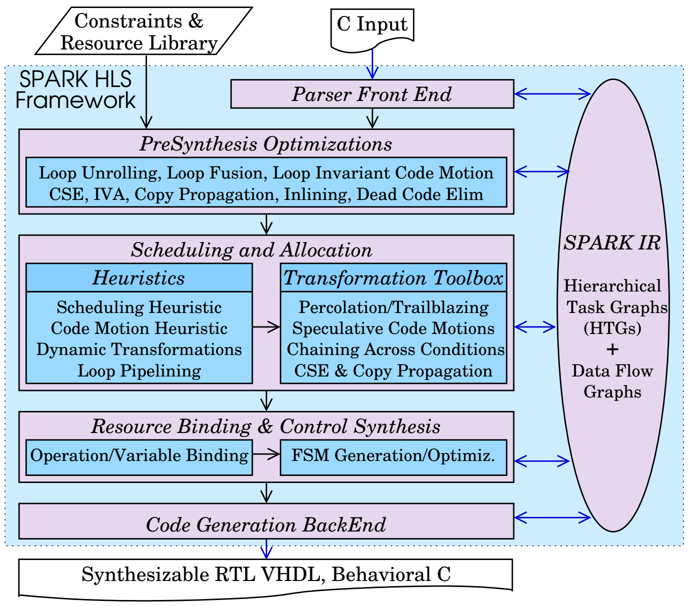
Constraints & Resource Library
C Input
SPARK HLS Framework
Parser Front End
PreSynthesis Optimizations
Loop Unrolling, Loop Fusion, Loop Invariant Code Motion
CSE, IVA, Copy Propagation, Inlining, Dead Code Elim
Scheduling and Allocation
Heuristics
Scheduling Heuristic
Code Motion Heuristic
Dynamic Transformations
Loop Pipelining
Transformation Toolbox
Percolation/Trailblazing
Speculative Code Motions
Chaining Across Conditions
CSE & Copy Propagation
Resource Binding & Control Synthesis
Operation/Variable Binding
FSM Generation/Optimiz.
Code Generation BackEnd
SPARK IR
Hierarchical Task Graphs (HTGs)
+
Data Flow Graphs
Synthesizable RTL VHDL, Behavioral C

Figure 2. **An overview of the *Spark* Parallelizing High-Level Synthesis Framework.**

4 The *Spark* Parallelizing High-Level Synthesis Methodology

We have developed a high-level synthesis methodology that incorporates parallelizing compiler and compiler transformations into the traditional high-level synthesis framework, both during a *pre-synthesis* phase and the *scheduling* phase. An overview of this methodology is shown in Figure 2. This methodology has been implemented in the *Spark* high-level synthesis framework. *Spark* takes an input of the design description in the "C" high-level language (with some restrictions on pointers, irregular control flow jumps, and recursive functions)and captures it using an intermediate representation that maintains the code structure in hierarchical task graphs (HTGs) [7].

A set of source level transformations are applied to the input design description in a *pre-synthesis phase*. These transformations, such as common sub-expression elimination (CSE), copy propagation, dead code elimination and loop-invariant code motion aim to reduce the number of operations executed and remove redundant and unnecessary operations. Also, we use coarse-level loop transformation techniques such as loop unrolling to restructure the code. This increases the scope for applying parallelizing optimizations in the scheduling phase that follows.

The scheduling phase employs an innovative set of *speculative, beyond-basic-block code motions* that reduce the impact of the choice of control flow (or programming style) on the quality of synthesis results [7]. These code motions enable movement of operations through, beyond, and into conditionals with the objective of maximizing performance by extracting the inherent parallelism in designs and increasing resource utilization. Since these speculative code motions often re-order, speculate and duplicate operations, they create new opportunities to apply additional transformations "dynamically" during scheduling, such as dynamic common sub-expression elimination (CSE) and dynamic copy propagation and dead code elimination.

Dynamic CSE eliminates common sub-expressions between the operation that has been scheduled and the other operations that are ready to be scheduled by taking advantage of the movement (and possible duplication) of the scheduled operation [19]. *Dynamic branch balancing* is another transformation that dynamically inserts scheduling steps in conditional branches during scheduling (without increasing the schedule length) in order to enable code motions, particularly those involving code duplication [20].

These compiler transformations are integrated with the standard high-level synthesis techniques such as resource sharing, scheduling on multi-cycle operations and operation chaining. Also, since our methodology targets mixed control-data flow designs, often operations have to be chained across conditional boundaries.

Although the pre-synthesis and scheduling transformations significantly improve circuit performance, these techniques also increase the complexity of the interconnect (multiplexers and associated control logic). We minimize the complexity of the interconnect by means of a resource binding methodology that binds operations to functional units and variables to registers such that the number of inputs and outputs to the units are minimized [21].

A control synthesis and optimization pass then generates a finite state machine (FSM) controller for the scheduled and bound design. This controller executes the operations as per the timing specified by the schedule and generates control signals to guide the data through the interconnect as specified by the resource binding. The *Spark* framework then employs a back-end code generator that generates RTL VHDL. *Spark* can thus interface with standard logic synthesis tools. This enables the evaluation of the effects of several coarse and fine-grain optimizations on logic synthesis results.

4.1 Role of Parallelizing Compiler Transformations in High-Level Synthesis

The use of compiler optimizations in high-level synthesis has focused on operation level transformations such as algebraic transformations (tree height reduction, exploiting the commutativity and distributivity of operations) [4, 5, 22] and so on. Few language (or source) level transformations have been explored. In contrast, “parallelizing” transformations are important not only at the source-level but also during operation scheduling and resource binding since these transformations provide opportunities for trade-offs of parallelism against control/area size constraints.

Parallelizing transformations are particularly essential when synthesizing code with complex control flow such as nested conditionals and loops. This is because the number of operations within a basic block is typically limited and thus, to increase resource utilization and performance, we have to exploit parallelism across basic block boundaries. We found that a range of parallelizing transformations that have been explored in the compiler community are useful for high-level synthesis. These include are transformations and techniques that exploit instruction-level parallelism, such as speculative code motions, Percolation scheduling, and Trailblazing [6], and coarse grain transformations such as loop transformations (loop unrolling, loop pipelining, loop invariant code motion, loop induction variable analysis).

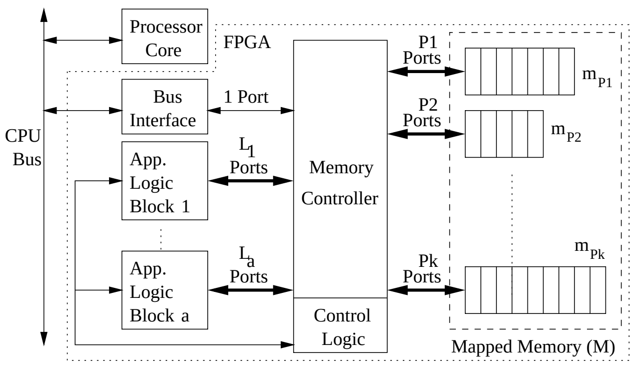

Figure 3. **The hardware interface architecture**

We found that these parallelizing transformations coupled with basic compiler transformations, such as common sub-expression elimination, copy propagation, dead code elimination, et cetera, are essential to improve the quality of high-level synthesis results – particularly, when synthesizing from high-level languages such as behavioral VHDL, C and C++ (and variants). When describing behaviors in high-level languages, designers use programming constructs such as conditionals and loops for programming convenience often with no notion of how these constructs may affect synthesis results.

In our work, we have adapted these parallelizing compiler optimizations for high-level synthesis by introducing notions of hardware concurrency and resource sharing. The contribution of our work has been in designing heuristics that guide these optimizations so that circuit performance is maximized without adversely affecting circuit area.

5 Interface Synthesis

Multimedia and image processing applications process large amounts of data. After hardware-software partitioning of these applications, the hardware component has to operate on the same data that the software operates on. Thus, the hardware component needs to store this data in memory banks on the FPGA. Our interface synthesis approach utilizes these memory banks for communication between the software and hardware. Thus, we generate a hardware interface that consists of data (variables and arrays) mapped to memory banks, a memory controller that directs the data to and from these memory banks, and a bus interface that communicates with the CPU bus.

5.1 The Hardware Interface Architecture

An overview of our hardware-based interface is shown in Figure 3. The bus interface generated is specific to the bus protocol used. The control logic contains memory mapped registers that can be used to reset or start/stop execution of any of the application logic blocks through software. It also

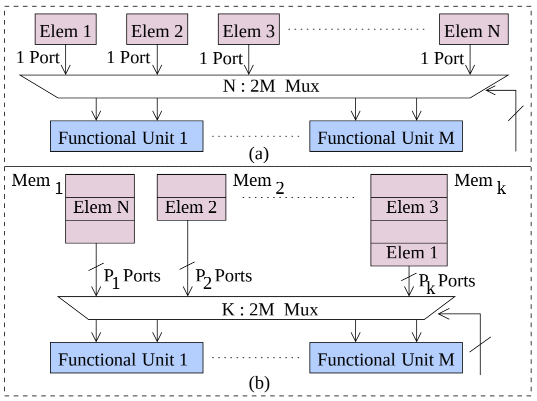

Figure 4. **(a) Unmapped Design: Registers for each data element (b) Mapped Design: Data elements or variables (*N*) mapped to memory banks (*K*) such that $K \ll N$.**

contains registers reflecting the execution status and logic for interrupt generation, masking, et cetera. The memory controller services all memory access requests for data residing in the mapped memory *M*. It is designed to give a higher priority to access requests by the application logic blocks. An access request by software is serviced only if a free port is currently available on the memory instance.

5.2 Memory Mapping

The chief challenge in our interface synthesis approach is mapping the large amounts of data that multimedia and image processing applications process to memory on the FPGA. The way this data is mapped to a memory has tremendous impact on the complexity of the multiplexers and the generated control logic. Ideally, we would store all data in a single large memory. However, such a memory would require as many ports as the maximum number of simultaneous memory accesses in any cycle [5]. This is impractical for programmable FPGA platforms, since they provide memories with only a limited number of ports [23, 24]. Consequently, memories with a larger number of ports have to be implemented using individual registers. This requires a large number of registers and complex, large multiplexers as shown in Figure 4(a).

In our memory mapping approach, we utilize scheduling information – available from the high-level synthesis tool – about data accesses and the cycles that they occur in. We can then map the data elements to memory banks, given constraints on the maximum number of ports each memory in the target FPGA can have. This approach eliminates the use of registers for storage, thus, saving a large amount of area. This way, we can also use much smaller and faster multiplexers in the data-path as illustrated in Figure 4(b).

Arrays and data structures are mapped to memories after being broken down into their basic constituents (variables). These can then be mapped in a way identical to regular variables. Consequently, these basic constituents might get mapped to non-contiguous memory addresses. This is explained in the next section.

5.3 The Software Interface

In the hardware interface shown in Figure 3, the memory *M* uses a contiguous address space. Hence, data declara-

```
struct example {                          struct example {
  int array1[256];                          int element1;
  int array2[256];      New Memory          int array1[256];
  int element1;            Map              int element3;
  int element2;        ---------->          int array2[256];
  int element3;                             int element2;
} *dm = 0xF800;                           } *dm = 0xF800;
```

Figure 5. **Modified address offsets in the software interface after memory mapping**

tions in the software code have to be reordered so that they conform to the order in which they were mapped to this address space. The example in Figure 5 illustrates this. Note that, when multiple arrays or data structures get sliced apart due to memory mapping, instead of modifying the application software, it is possible to perform address translations in the memory controller in order to abstract the memory architecture details from software.

The software can share data with the hardware using two schemes: (a) *Data transfer scheme*, whereby the shared data is copied from the processor (or main) memory to the mapped memory, the hardware is executed, and the results are copied back from the mapped memory to the main memory; (b) *Shared memory scheme*, whereby the hardware and software interface through a *shared memory* (i.e., *M* in Figure 3 is shared) [25, 26]. For processors with local storage, explicit instructions for flushing the local storage to memory have to be used.

Both these schemes have their advantages and disadvantages. The data transfer scheme is useful when the hardware-software partitioning can be done so that communication between the hardware and software is minimized, whereas the shared memory scheme is useful when the SOC platform has a soft processor core implemented on the FPGA fabric itself, such as in the Altera Nios [23] and Xilinx Vertex-II platforms [24]. In this way, the processor has quick access to the memory that is also on the FPGA fabric (rather than going over a system bus).

6 Experimental Setup and Results

In this section, we first evaluate the effectiveness of each set of code optimizations implemented in the *Spark* framework and then present a case study for the co-design of a multimedia application on to a FPGA platform.

6.1 Scheduling and Logic Synthesis Results for Code Optimizations in *Spark*

Table 1 lists the scheduling results produced by *Spark* for four designs derived from the MPEG-1 and MPEG-2 [27] multimedia applications and the GIMP image processing tool [28]. The results are in terms of the states in the finite state machine (FSM) controller and the cycles on the longest path through the design. The first row lists results for when no speculative code motions are enabled, i.e., operations can be moved within basic blocks and across conditionals and loops, the second row for when speculative code motions are enabled, the third row for when presynthesis optimizations (loop invariant code motion and

Transformation Applied	*MPEG*-1 *pred*1 Num of States	*MPEG*-1 *pred*1 Cycles on Long Path	*MPEG*-1 *pred*2 Num of States	*MPEG*-1 *pred*2 Cycles on Long Path	*MPEG*-2 *dpframe_est* Num of States	*MPEG*-2 *dpframe_est* Cycles on Long Path	*GIMP tiler* Num of States	*GIMP tiler* Cycles on Long Path
No Spec CMs	62	2726	123	6055	81	1036	86	8231
+Spec CMs	41(-33.9%)	1824(-33.1%)	86(-30.1%)	4187(-30.9%)	54(-33.3%)	672(-35.1%)	43(-50%)	3931(-52.2%)
+Pre-Synthesis	41(0 %)	1091(-40.2%)	76(-11.6%)	2575(-38.5%)	50(-7.4 %)	571(-15 %)	33(-23.3%)	2634(-33%)
+Dynamic CSE	38(-7.3%)	899 (-17.6%)	69(-9.2%)	2127(-17.4%)	48(-4 %)	563(-1.4 %)	32(-3 %)	2534(-3.8 %)
Total Reduction	-38.7 %	-67 %	-43.9 %	-64.9 %	-40.7 %	-45.7 %	-62.8 %	-69.2 %

Table 1. **Scheduling Results after applying speculative code motions, pre-synthesis transformations (loop-invariant code motion and CSE), and dynamic CSE on the four designs.**

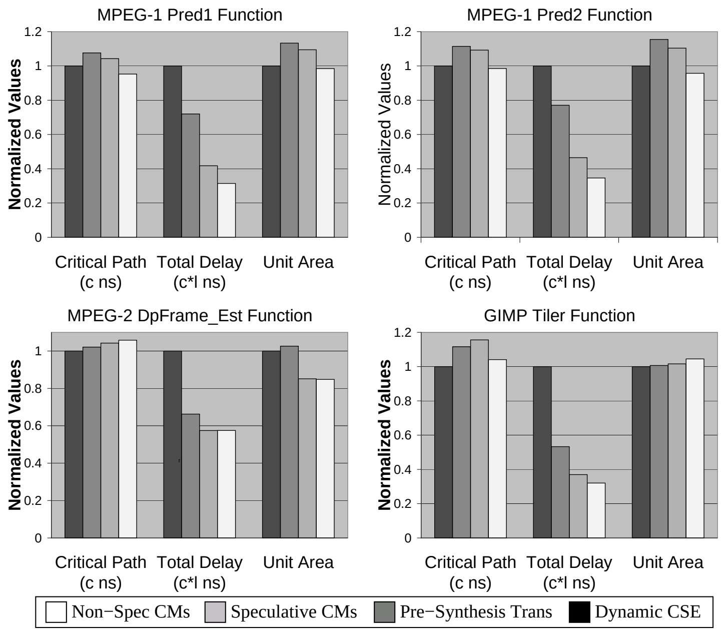

Figure 6. **Comparison of logic synthesis results after applying speculative code motions, pre-synthesis optimizations, and dynamic CSE over the case when no speculative code motions are employed.**

CSE) are also enabled, and the final row for when dynamic CSE is also enabled. The percentage improvement of each row over the previous row is given in parentheses. Note that the first row represents the results produced by a classical list scheduling heuristic [4].

The results in Table 1 demonstrate that the speculative code motions and pre-synthesis optimizations lead to the most significant improvements, at times by 30 to 40 %. Dynamic CSE is able to further improve the scheduling results from 1 % to up to 17 %. The total reductions achieved by our parallelizing code transformations over a design that is scheduled using a classical list scheduling heuristic with no speculative code motions enabled are in the range of 38 to 69 % (last row of Table 1).

We synthesized the VHDL generated by *Spark* corresponding to these experiments using the Synopsys Design Compiler. The results in terms the critical path length (nanoseconds), the total delay (cycles * critical path length) and the unit area are presented in the graphs in Figure 6. The bars in these graphs represent results with no speculative code motions (1st bar), with speculative code motions (second bar), with pre-synthesis optimizations as well (third bar), and finally, for when dynamic CSE is also enabled (fourth bar). All the metrics mapped are normalized with respect to the first bar.

These logic synthesis results show that we achieve improvements in results beyond scheduling results, i.e., in terms of circuit delay and area. We are able to reduce circuit delay by 50 to 75 % with fairly constant critical path lengths and circuit area. These results demonstrate that the heuristics in *Spark* are able to guide the parallelizing optimizations to achieve higher performance while keeping control and multiplexer costs in check.

6.2 Case Study: Co-design of the MPEG-1 Prediction Block

In this section, we present a case study using a multimedia application to evaluate the effectiveness of interface synthesis methodology coupled with *Spark*. For the case study, we mapped the *prediction block* from the MPEG-1 application to the Altera Nios development platform [23]. This platform consists of a soft core of the Nios embedded processor (no caches) to which various peripherals can be connected for customization. Once the system has been customized, it can be synthesized and mapped onto an FPGA based board provided by Altera. The user can use dedicated RAM block bits inside the FPGA to implement memories having two or fewer ports. The entire system, consisting of the Nios processor and its standard peripherals, the main memory, the CPU bus and the user-defined module operate at a frequency of 33.33 MHz. The user-defined module is connected to the CPU bus and contains the hardware interface and all the application logic.

To begin with, three computationally intensive kernels in the application were identified using profiling information and these were mapped to hardware using *Spark*. The VHDL generated by *Spark* was synthesized using the logic synthesis tool, Leonardo Spectrum. The resultant netlist was mapped to the FPGA using the Altera Quartus tool. The software component of the application was compiled and mapped onto the Nios processor using the software compiler suite from the Nios development toolkit.

In Table 2, we list the logic synthesis results for the hardware modules in terms of the area (logic cells), maximum frequency, and RAM bits. The first row in this table lists the results for the Nios embedded system (without the synthesized hardware module). The second row lists the results when the hardware module (all three kernels) is synthesized without using our memory mapping approach. Due to the high area of this design (9750 LCs), we could

HW Module	Area	Max Freq	RAM Bits
Nios System	2800 LCs	47 MHz	26496
Unmapped User-defined	9750 LCs	27 MHz	0
Mapped User-defined	2450 LCs	32 MHz	2048

Table 2. ***Logic synthesis results for an Altera APEX20KE EP20K200EFC484-2X FPGA target***

not integrate the design with the Nios embedded system since the capacity of the FPGA is 8320 logic cells (LCs).

Next, we applied our memory mapping algorithm to the data elements in the hardware component (all three kernels) and came up with a new memory configuration. Then, we made a new memory controller based on the new memory map and ran logic synthesis on the new hardware module. The third row in Table 2 lists the results for the module (including the memory mapped interface). As we can see from this table, we achieved a 75% reduction in area and 19% increase in operating frequency using our memory mapping approach. As a result, we were also able to integrate the hardware module with the Nios processor on the FPGA fabric. Note that, the maximum frequencies shown in the table are only estimates made by the logic synthesis tool and were found to be slightly conservative.

7 Conclusions and Future Work

We presented a C-based synthesis methodology that uses a parallelizing high-level synthesis framework called *Spark* and an interface synthesis approach to map applications to FPGA-based platforms. We demonstrated the effectiveness of a range of code optimizations implemented in *Spark* through scheduling and logic synthesis results on multimedia and image processing designs. Our interface synthesis approach is based on a novel memory mapping algorithm that uses scheduling information from *Spark* to map the data elements operated on by the software and hardware to shared memories. These shared memories, thus, form a hardware interface between the hardware and software components of the application. In future work, we want to validate our methodology for larger FPGA platforms and develop a generalized hardware-software synthesis framework that includes task analysis and partitioning.

References

[1] A. Sangiovanni-Vincentelli and G. Martin. Platform-based design and software design methodology for embedded systems. *IEEE D&T of Computers*, 2001.

[2] R.K. Gupta and G. De Micheli. Hardware-software cosynthesis for digital systems. *IEEE Design and Test of Computers*, September 1993.

[3] J. Henkel and R. Ernst. A hardware-software partitioner using a dynamically determined granularity. In *Design Automation Conference*, 1997.

[4] D. D. Gajski, N. D. Dutt, A. C-H. Wu, and S. Y-L. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic, 1992.

[5] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[6] S. Gupta, et al. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. In *Intl. Conference on VLSI Design*, 2003.

[7] S. Gupta, et al. Using global code motions to improve the quality of results for high-level synthesis. *To appear in IEEE Transactions on CAD*, 2003.

[8] P. Chou, et al. Interface co-synthesis techniques for embedded systems. *ICCAD*, 1995.

[9] J. Daveau, G.F. Marchioro, T. Ben-Ismail, and A.A. Jerraya. Protocol selection and interface generation for HW-SW Codesign. *IEEE TVLSI*, March 1997.

[10] F. Gharsalli, et al. Unifying memory and processor wrapper architecture for multiprocessor SoC design. *International Symposium on System Synthesis*, 2002.

[11] K. Wakabayashi and H. Tanaka. Global scheduling independent of control dependencies based on condition vectors. In *Design Automation Conference*, 1992.

[12] I. Radivojevic and F. Brewer. A new symbolic technique for control-dependent scheduling. *IEEE Transactions on CAD*, January 1996.

[13] G. Lakshminarayana et al. Incorporating speculative execution into scheduling of control-flow intensive behavioral descriptions. *DAC*, 1998.

[14] L.C.V. dos Santos and J.A.G. Jess. A reordering technique for efficient code motion. *DAC*, 1999.

[15] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. on Computers*, July 1981.

[16] M. Balakrishnan, et al. Allocation of multiport memories in data path synthesis. *IEEE TCAD*, April 1988.

[17] I. Ouaiss and R. Vemuri. Hierarchical memory mapping during synthesis in FPGA-based reconfigurable computers. In *DATE*, 2001.

[18] D. Karchmer and J. Rose. Definition and solution of the memory packing problem for field-programmable systems. In *ICCAD*, 1994.

[19] S. Gupta, et al. Dynamic common sub-expression elimination during scheduling in high-level synthesis. *Intl. Symposium on System Synthesis*, 2002.

[20] S. Gupta, et al. Dynamic conditional branch balancing during the high-level synthesis of control-intensive designs. In *DATE*, 2003.

[21] S. Gupta, et al. Conditional speculation and its effects on performance and area for high-level synthesis. In *International Symposium on System Synthesis*, 2001.

[22] M. Potkonjak and J. Rabaey. Optimizing resource utlization using tranformations. *IEEE TCAD*, Mar. 1994.

[23] The Altera Website. `http://www.altera.com`.

[24] The Xilinx Website. `http://www.xilinx.com`.

[25] M. Luthra, et al. Interface synthesis using memory mapping for an FPGA platform. *ICCD*, 2003.

[26] Z. Chen et al. Pilot - a platform-based HW/SW synthesis system for FPSoC. In *Workshop on Software Support for Reconfigurable Systems*, 2003.

[27] C. Lee, et al. UCLA Mediabench benchmark suite. http://cares.icsl.ucla.edu/MediaBench/.

[28] GNU Image Manipulation Tool. `http://www.gimp.org`.