# Hardware/Software Partitioning of Floating Point Software Applications to Fixed-Pointed Coprocessor Circuits

Lance Saldanha, Roman Lysecky
Department of Electrical and Computer Engineering
University of Arizona
{saldanha, rlysecky}@ece.arizona.edu

## ABSTRACT

*While hardware/software partitioning has been shown to provide significant performance gains, most hardware/software partitioning approaches are limited to partitioning computational kernels utilizing integers or fixed point implementations. Software developers often initially develop an application using built-in floating point representations and later convert the application to a fixed point representation – a potentially time consuming process. In this paper, we present a hardware/software partitioning approach for floating point applications that eliminates the need for developers to rewrite software applications for fixed point implementations. Instead, the proposed approach incorporates efficient, configurable floating point to fixed point and fixed point to floating point hardware converters at the boundary between the hardware coprocessors and memory. This effectively separates the system into a floating point domain consisting of the microprocessor and memory subsystem and a fixed point domain consisting of the partitioned hardware coprocessors, thereby providing an efficient and rapid method for implementing fixed point hardware coprocessors.*

## Categories and Subject Descriptors

C.3 **[Computer Systems Organization]** Special-Purpose and Application-Based Systems - *Real-time and Embedded Systems*.

## General Terms

Design, Performance.

## Keywords

Hardware/software partitioning, floating point to fixed conversion, floating point, fixed point.

## 1. INTRODUCTION

Field programmable gates arrays (FPGAs) are becoming increasingly popular, having moved from primarily being utilized for hardware prototyping and debugging to being incorporated into many computing domains and consumer electronics. An FPGA can implement any hardware circuit simply by downloading bits for the hardware circuit, much in the same way that microprocessors can execute any software program simply by downloading a new application binary. With the continuing evolution of FPGAs, new devices have emerged that integrate one or more microprocessors and configurable FPGA resources onto a single chip with support for fast communication. Such devices, available from Xilinx, Altera, and Atmel, are ideally suited for hardware/software partitioning.

Hardware/software partitioning is the process of dividing an application between software executing on a microprocessor and hardware implemented within an FPGA or ASIC. Significant previous research has demonstrated the performance and energy benefits of hardware/software partitioning over software execution alone with typical performance gains of 10-100X [1][4][11][14][17][19]. Hardware/software partitioning can also provide reductions in energy consumption of up to 95% [9][18][20].

Most hardware/software partitioning approaches are typically limited to partitioning computational kernels utilizing integers or fixed point computations. During initial software development, single and double precision floating point representations are often utilized to represent real numbers due to the convenience provided by built-in support within most programming languages, including C/C++/Java. However, directly implementing floating point operations within hardware requires significant area resources and potentially large power requirements. In addition, to provide acceptable performance, hardware implemented floating point implementations are typically very highly pipelined, thereby requiring multi-cycle latencies. On a related note, many embedded microprocessors do not provide hardware support for floating point operations and must rely on slow software based floating point calculations.

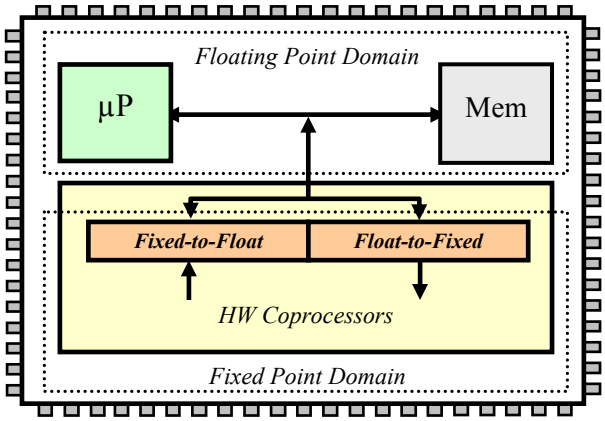
Fortunately, many applications do not need floating point representations to support real numbers. Instead, a fixed point representation is a viable alternative that can be implemented efficiently in hardware. A floating point number is stored in what can be referred to as a binary scientific notation using a sign-magnitude format that allows for a *floating* position of the radix point. For example, an IEEE 754 [10] single precision floating point number consists of a sign bit (*S*), an 8-bit exponent (*E*), and a 23-bit mantissa (*M*). The resulting decimal value can be computed as follows:

$$Value = (-1)^S * 1.M_2 * 2^{E-127}$$

For both single and double precision floating point representations, the mantissa provides the number's precision and the exponent controls the number's range. In comparison, a fixed point representation is directly stored as a two's complement binary number with a *fixed* radix point. Unlike floating point arithmetic, fixed point operations can be very efficiently performed using integer operations. Fixed point additions directly map to integer additions, and fixed point multiplication can be performed using an integer multiplication followed by a fixed shift by the radix position. For a fixed point representation all bits directly affect the number's precision, but the range of the number is determined by the fixed radix position. While a fixed point representation can provide greater precision than a floating point representation, a


*CODES+ISSS'08*, October 19–24, 2008, Atlanta, Georgia, USA.

**Figure 1:** Overview of proposed microprocessor/coprocessor architecture with configurable *Float-to-Fixed* and *Fixed-to-Float* converters.




floating point representation provides a much greater dynamic range of numbers that can be represented. For example, the smallest possible positive single precision floating point number is 1.2e-38, whereas the smallest possible positive number that be represented by any 32-bit fixed point representation is only 2.3e-10.

For those applications that do not require the dynamic range supported by floating point representations, a fixed point implementation is often the best approach. However, one of the biggest challenges of using fixed point arithmetic is that of software development. Again, due to the lack of built-in support for fixed point representations in most programming languages, software developers will often initially develop an application using floating point representations and later convert the application to a fixed point representation.

In an attempt to ease such development efforts, several research efforts have provided tools to assist in converting software programs using floating point values into fixed point implementations [2][5][12][15][16]. FRIDGE [12] is a system level fixed point design methodology in which a designer initially provides localized annotations for the critical fixed point operands using the *fixed-C* extension to the C programming language followed by simulation to verify correctness and automatically determine the fixed point representation needed throughout the remainder of the software application. The *fixify* environment, presented in [3], initially utilizes simulation to determine the ideal fixed point representation followed by a design space exploration step that analyzes the tradeoffs between numeric degradation, cost, and performance. As digital signal processors (DSPs) typically incorporate dedicated support for fast fixed point computations, in [15], researchers present an automated methodology for optimizing the application performance executing on a DSP while meeting a designer specific accuracy constraint. These existing automated conversion approaches are primarily targeted at the compilation phase of software development, for which the resulting output is a fixed point software implementation in which all floating point operations have been converted to a fixed point implementation.

Similar challenges are encountered within hardware/software partitioning. For applications using floating point computations, a designer must minimally convert the floating point operation to a fixed point implementation for the critical kernels of the application that will be partitioned to hardware. While dedicated hardware for floating point operations may be needed for some applications, the significant area requirements and multi-cycle latencies preclude its use for most. Whether a designer manually converts the critical kernels to a fixed point representation or relies on an automated methodology, the end result is that the initial software application must be modified before partitioning.

In this paper, we present an automatable hardware/software partitioning approach for floating point software applications that eliminates the need for developers to rewrite software applications utilizing fixed point implementations before partitioning. Instead, the proposed approach incorporates efficient, configurable floating point to fixed point and fixed point to floating point hardware converters at the boundary between the hardware coprocessors and memory. This effectively separates the system into a floating point computing domain consisting of the microprocessor and memory subsystem and a fixed point computing domain consisting of the partitioned hardware coprocessors. Within the proposed approach, no modifications are needed for the software application. Furthermore, the hardware coprocessor development, whether automated or manually performed, can treat floating point computations much in the same way as integer computations, thereby providing excellent hardware performance with minimal hardware requirements. We examine the performance improvements and hardware requirements of the proposed hardware/software partitioning approach for several embedded applications from the MiBench [8] and MediaBench [13] benchmark suites.

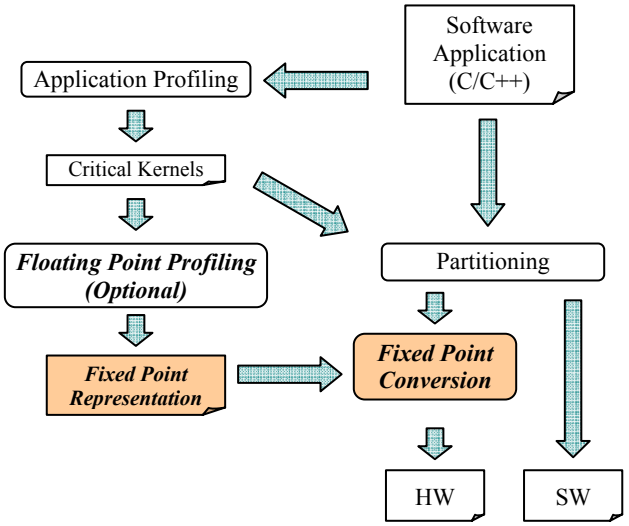
## 2. HARDWARE/SOFTWARE PARTITIONING OF FLOATING POINT APPLICATIONS

Figure 1 provides an overview of the proposed microprocessor/coprocessor architecture for hardware/software partitioning of floating point applications to fixed point hardware implementations. The partitioned application execution is separated into a floating point computing domain and a fixed point computing domain. The floating point domain encompasses the normal software execution and includes the microprocessor, caches, and memory subsystems. After partitioning the application between hardware and software, the software portion may continue to utilize floating point numbers. As such, all real numbers stored within the memory will utilize single or double precision floating point.

The fixed point domain consists of the hardware coprocessors implemented using a fixed point representation and configurable *Float-to-Fixed* and *Fixed-to-Float* hardware converters that interface between the hardware coprocessors and microprocessor or memory. All floating point values read from memory will be converted by the *Float-to-Fixed* converter to the target fixed point representation. Similarly, all fixed point values written back to memory will be converted by the *Fixed-to-Float* converter to a floating point representation.

By separating the partitioned application execution into floating point and fixed point domains, the software implementation does not need to be developed using, or converted to, a fixed point implementation. Instead, floating point operations may be used as needed by the application, requiring no modification to the initial floating point application, thereby reducing designer effort. After partitioning, the software partition will continue to utilize floating point operations. On the other hand, the hardware partition will utilize a fixed point representation in which arithmetic operations are only marginally more complex than their integer counterparts, providing fast, small, and power efficient hardware support for real numbers.

**Figure 2:** Hardware/software partitioning methodology for floating point applications.




We note that while technically possible, converting between floating point and fixed point numbers in software – a process requiring tens to hundreds of instructions for each such conversion – incurs considerable overhead that would severely impact performance, even leading to performance slowdowns.

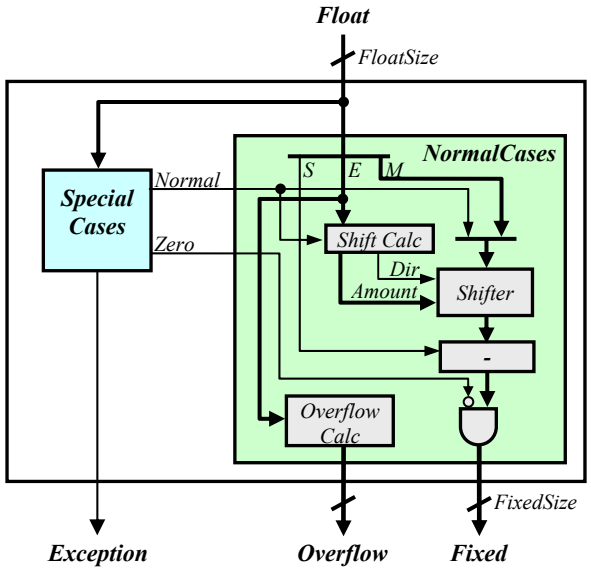
Figure 2 presents an overview of the proposed hardware/software partitioning methodology. Initially, the software application is profiled to determine the application's critical kernels that are the potential candidates for hardware implementation. The partitioning process will then determine which critical kernels to implement in hardware, creating an initial annotated hardware description for the partitioned kernels.

During this partitioning process, all floating point data types are treated as integers – single precision as a 32-bit integer, double precision as a 64-bit integer – in creating the partitioned hardware coprocessors. However, as the floating point values will eventually be replaced by the required fixed point representations, the resulting hardware description includes synthesis annotations indicating which registers and computations are floating point operations that must be converted to a fixed point representation.

Floating point profiling can then be performed to determine the fixed point representation required by each critical kernel. However, if the required precision and dynamic range are already known, a designer can directly specify the required fixed point representation. Otherwise, automated fixed point profiling tools, such as those previously described, can be used to determine the required fixed point representation. Finally, the fixed point conversion process will convert the annotated hardware implementation into the final fixed point implementation by replacing the annotated floating point registers and operations with the provided fixed point representation and utilizing the configurable *Float-to-Fixed* and *Fixed-to-Float* converters.

Because different application kernels may require varying fixed point representations, the floating point profiling is performed for all candidate kernels considered for partitioning to hardware. The resulting hardware implementation may require a *Float-to-Fixed* and a *Fixed-to-Float* converter for each distinct fixed point representation needed by the hardware coprocessors. Although, if only a single fixed point representation is needed for all hardware coprocessors, a single set of *Float-to-Fixed* and *Fixed-to-Float* converters is sufficient.

**Figure 3:** Hardware architecture for radix point parameter implementation of *Float-to-Fixed* converter.




## 3. FLOATING POINT TO FIXED POINT CONVERSION

The *Float-to-Fixed* converter is a combinational logic design that converts the incoming floating point values into the defined fixed point representation. The *Float-to-Fixed* converter is implemented as a configurable Verilog hardware description that can be configured through a set of parameters to specify both the input floating point representation and the output fixed point representation. The configurable parameters include:

- *FloatSize* – specifies the total number of bits within the floating point representation. A single precision floating representation requires 32 bits, whereas a double precision floating point representation requires 64 bits.
- *MantissaBits* – specifies the number of bits within the floating point representation allocated to the mantissa. A single precision floating point representation uses 23 bits for the mantissa and a double precision representation uses 52 bits.
- *ExponentBits* – specifies the number of bits within the floating point representation allocated to the exponent. The exponent is specified as 8 bits for a single precision floating point representation and as 11 bits for a double precision representation.
- *FixedSize* – specifies the total number of bits within the fixed point representation.
- *RadixPointSize* – specifies the number of bits needed to represent the *RadixPoint*.
- *RadixPoint* – specifies the fixed location of the radix point. The fixed radix position also corresponds to the number of bits allocated for the fractional portion of a

fixed point number. As such, the integer portion of a fixed point representation can be calculated as *FixedSized* minus *RadixPoint*.

Figure 3 presents an overview of the *Float-to-Fixed* converter design consisting of a *NormalCases* component for handling normal floating point to fixed point conversion and a *SpecialCases* component for detecting several special representations defined within the IEEE 754 standard, including positive zero, negative zero, denormalized numbers, positive infinity, negative infinity, and not-a-number (NaN).

A fixed point representation does not support representing special cases for infinity or NaN. Thus, if a floating point input is infinity or NaN, the *SpecialCases* component will assert an *Exception* signal indicating that an exception has occurred. On the other hand, if the floating point input is the special case for representing either a positive or negative zero, the *SpecialCases* component will assert the *Zero* output signal. Finally, if the floating input is the special case for a denormalized number, the *SpecialCases* component will deassert the *Normal* output signal. A denormalized floating point number is a floating point number in which mantissa directly specifies the corresponding value, for which the resulting decimal value can be computed as follows:

$$Value = (-1)^S * 0.M_2 * 2^0$$

*The NormalCases* component converts the input floating point value to the target fixed point representation assuming the input is not a special case and that no overflow will occur. In addition, the *NormalCases* component will also calculate an *Overflow* output that corresponds to the numbers of additional bits needed within the fixed point representation to avoid an overflow for the current conversion. Given the specified fixed point representation, incoming floating point representation, the exponent of the floating point input, and the *Normal* input from the *SpecialCases* component, the *NormalCases* component computes the required shift direction and amount needed to align the floating point number's mantissa with the fixed point representation. Once aligned, the *NormalCases* component will calculate the two's complement of the aligned value if the sign bit indicates the input is negative. Finally, all bits are and'ed with the inverse of the *Zero* input from the *SpecialCases* component to generate an output of zero if the input is one of the special cases for zero.

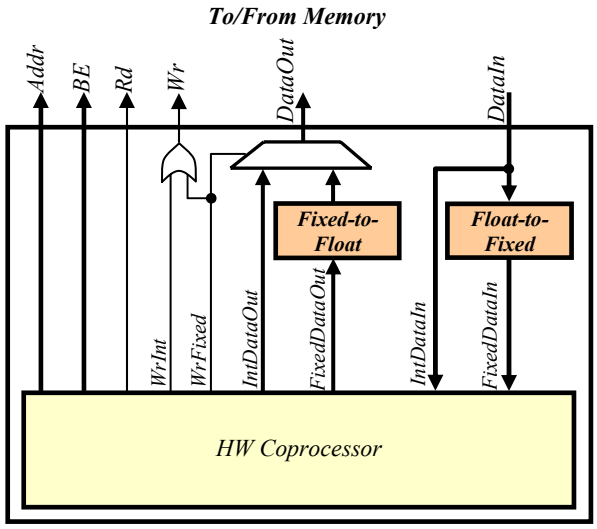
For designs that require multiple fixed point representations or designs for which the final fixed point representation may not be known *a priori*, the specification of the radix point for the fixed point representation can be provided as an input to the *Float-to-Fixed* converter, instead of as a parameter. With a radix point input, the modified *Float-to-Fixed* converter will incur some area and performance overhead compared to the parameter based implementation.

# 4. FIXED POINT TO FLOATING POINT CONVERSION

The *Fixed-to-Float* converter is a combinational logic design for converting a fixed point representation into the target floating point representation. The *Fixed-to-Float* converter can be configured through the same set of parameters used to specify the *Float-to-Fixed* converter. Since the fixed point representation does not support special cases for infinity or NaN, the *Fixed-to-Float* converter need only handle the special case for representing zero. If an input fixed point number is zero, the *Fixed-to-Float* converter will output the special floating point representation for a positive zero. Although a representation for negative zero is possible within the floating point representation, there is no such distinction in the fixed point representation. Hence, we chose to store the value of zero as a positive zero. If the fixed point input is not zero, the *Fixed-to-Float* converter first determines the sign of the fixed point number, outputting the numbers sign bit. If the number is negative, the converter will calculate the number's two's complement. The *Fixed-to-Float* converter then utilizes a priority encoder to determine the position of the most significant bit of the fixed point input whose value is one. The position of this leading one is used to determine the shift direction and amount required to shift the fixed point number into a binary scientific notation, thus obtaining the required exponent and mantissa for the floating point representation.

Again, for designs that require multiple fixed point representations or designs for which the final fixed point representation may not be known *a priori*, the specification of the radix point can be provided as an input to the *Fixed-to-Float* converter, instead of as a parameter.

**Figure 4:** Hardware coprocessor interface with configurable *Float-to-Fixed* and *Fixed-to-Float* converters.




# 5. FIXED POINT HARDWARE COPROCESSOR

The *Float-to-Fixed* and *Fixed-to-Float* converters allow for a straightforward fixed point coprocessor implementation. Figure 4 presents an overview of the hardware coprocessor interface to memory encapsulating the integration of the *Float-to-Fixed* and *Fixed-to-Float* converters. The memory interface utilized within this illustration includes a memory address (*Addr*) input, separate read (*Rd*) and write (*Wr*) control inputs, a byte enable (*BE*) control input which enables writing individual bytes in a word addressable memory, and separate data inputs (*DataIn*) and data outputs (*DataOut*).

The hardware coprocessor can directly interface with the memory's *Addr*, *BE, and Rd* inputs. Memory reads include both reading floating point values as well as reading integer values. To provide a clear distinction between floating point and integer data, separate inputs for integer and converted fixed point values are provided to the coprocessor. For reading integer data from memory, the coprocessor's *IntDataIn* input provides the unaltered value read from memory. Alternatively, for reading floating point values from

**Table I:** Area (*LUTs*) and delay (*nanoseconds*) requirements for *Float-to-Fixed* and *Fixed-to-Float* converters implemented using the Xilinx Virtex-5 FPGA.

| | Radix Point Parameter | | Radix Point Input | |
|---|---|---|---|---|
| | DELAY | AREA | DELAY | AREA |
| Float-to-Fixed (SP»12.20) | 4.56 | 357 | 5.04 | 401 |
| Float-to-Fixed (SP»21.30) | 5.12 | 386 | 5.62 | 421 |
| Float-to-Fixed (SP»17.47) | 5.38 | 421 | 5.85 | 468 |
| Fixed-to-Float (12.20»SP) | 4.81 | 251 | 5.60 | 206 |
| Fixed-to-Float (21.30»SP) | 5.74 | 418 | 8.01 | 342 |
| Fixed-to-Float (17.47»SP) | 6.38 | 571 | 9.03 | 417 |

memory, the *FixedDataIn* input is connected to the output of the *Float-to-Fixed* converter providing the converted value to the coprocessor.

Similarly, the hardware coprocessor interface includes separate data outputs for writing integer (*IntDataOut*) and fixed point (*FixedDataOut*) values to memory. The *WrFixed* control output is utilized by the coprocessor to indicate the current value being written to memory is a fixed point value that needs to be converted to floating point. The *WrFixed* controls a multiplexer that either connects the *IntDataOut* output or converted floating point output of the *Fixed-to-Float* converter to the memory's data input. The *WrInt* control output is utilized by the coprocessor to indicate the current value being written to memory is an integer value. The coprocessor's *WrFixed* and *WrInt* outputs are connected via an OR gate to the memory's *Wr* input.

# 6. EXPERIMENTAL RESULTS

We implemented the configurable *Float-to-Fixed* and *Fixed-to-Float* converters using the Verilog hardware description language, in which the radix point can be specified as a parameter within the converter description or provided as an input to the converters. Three implementations for the *Float-to-Fixed* and *Fixed-to-Float* converters are needed for the various software applications considered within this paper. The first set of converters *(12.20)* is needed for converting between a single precision floating point representation and a 32-bit fixed point representation with a radix point of 20. The second set of converters *(21.30)* converts between a single precision floating point representation and a 51-bit fixed point representation with a radix point of 30. Finally, the third set of converters *(17.47)* converts between a single precision floating point representation and a 64-bit fixed point representation with a radix point of 47. The *Float-to-Fixed* and *Fixed-to-Float* converters were synthesized to a Xilinx Virtex-5 FPGA using the Xilinx ISE 9.2 synthesis tools with speed as the main optimization goal.

Table I presents the area (*lookup tables (LUTs)*), and delay (*nanoseconds*) for the three *Float-to-Fixed* and *Fixed-to-Float* converter pairs described above. On average, the radix point parameter implementation of the *Float-to-Fixed* converter is 9% faster with 10% fewer LUTs than the input based implementation. On the other hand, the radix point parameter implementation of the *Fixed-to-Float* converter is 25% faster, but that increase in speed comes at the expense of increased area – requiring 30% more LUTs.

Using the *Float-to-Fixed* and *Fixed-to-Float* converters and the proposed hardware/software partitioning approach, we partitioned the *mpeg2dec*, *mpeg2enc*, and *epic* applications of the MediaBench benchmark suite [13] and *fft* and *ifft* applications on the MiBench benchmark suite [8], all of which required extensive floating point calculations. The target architecture includes a 250 MHz MIPS processor with support for floating point calculations and a Xilinx Virtex-5 FPGA executing at the maximum frequency as determined by synthesizing the resulting fixed point coprocessors using Xilinx ISE 9.2 for both the parameter based and input based converter alternatives.

**Table II:** Application execution time and speedup for *mpeg2dec*, *mpeg2enc*, *epic, fft*, *and ifft* applications.

| | SW | Radix Point Parameter HW/SW | | | Radix Point Input HW/SW | | |
|---|---|---|---|---|---|---|---|
| | (s) | MHz | (s) | S | MHz | (s) | S |
| mpeg2dec | 1.02 | 101 | 0.31 | 3.3 | 77 | 0.34 | 3.0 |
| mpeg2enc | 17.02 | 101 | 5.52 | 3.1 | 77 | 5.66 | 3.0 |
| epic | 0.32 | 88 | 0.18 | 1.8 | 69 | 0.20 | 1.6 |
| fft/ifft | 2.88 | 82 | 0.35 | 8.2 | 74 | 0.39 | 7.5 |
| *Average:* | | | | **4.9** | | | **4.5** |

As determined by application profiling, the majority of execution time for *mpeg2dec* is spent computing the inverse discrete cosine transform (IDCT) that is implemented in software using a single precision floating point representation. Floating point profiling indicated that a 32-bit fixed point representation with a radix point of 20 provides the required dynamic range and accuracy to ensure the final results of the IDCT computation are identical to that of the floating point implementation.

For *mpeg2enc*, the partitioned critical kernels consist of the forward discrete cosine transfer (DCT) and a sum of absolute differences computation performed during motion estimation of the encoding process. While the DCT operation requires a 32-bit fixed point representation with a radix point of 20, the sum of absolute differences computation only consists of integer calculations. Thus, the resulting hardware implementation includes both integer and fixed point computations that can be efficiently enabled through the hardware coprocessor interface.

For the application *epic*, the execution is predominantly spent executing a convolution filter. The convolution filter internally utilizes a double precision floating point representation to compute the intermediate results, where the values read from and written to memory are stored using a single precision floating point representation. To provide the required dynamic range and accuracy needed for the calculation of intermediate results, floating point analysis determined that a 64-bit fixed representation with a radix point of 47 is required. We note that the *Float-to-Fixed* and *Fixed-to-Float* converters needed for this application do not need to convert from/to a double precision floating point representation because all values stored within memory are single precision floating point values.

Finally, both the *fft* and *ifft* applications of the MiBench benchmark suite resulted in almost identical hardware coprocessor implementations needed to compute Fast Fourier Transformation (FFT) and Inverse Fast Fourier Transformation (IFFT). The FFT and IFFT both utilized a double precision floating point representation to compute the intermediate results, where the values read from and written to memory are stored using a single precision floating point representation. To provide the required dynamic range and accuracy needed for the calculation of intermediate results, floating point analysis determined that a 51-bit fixed representation with a radix point of 30 is required.

Table II presents the application execution time before and after hardware/software partitioning and application speedup for the *mpeg2dec*, *mpeg2enc*, *epic, fft*, *and ifft* applications for both the radix point parameter and radix point input versions of the *Float-to-Fixed* and *Fixed-to-Float* converters. Utilizing the radix point

parameter converters, the proposed hardware/software partitioning approach achieves a speedup ranging from 1.8X to 8.2X, with an average speedup of 4.9X across all five applications. Alternatively, with the radix point input converters, the proposed hardware/software partitioning approach achieves an average speedup of 4.5X, with a maximum speedup of 7.5X for *fft* and *ifft*.

# 7. CONCLUSIONS

By integrating configurable *Float-to-Fixed* and *Fixed-to-Float* converters at the boundary between hardware and software, the proposed hardware/software partitioning approach for a floating point application provides application speedups of 4.9X on average without requiring any designer effort to re-implement software with a fixed point representation. The proposed approach provides an efficient method for partitioning floating point applications both in terms of design time and hardware requirements. In addition, the resulting hardware coprocessor interface provides a simple method for reading and writing both integer and floating point values from and to memory and can be efficiently integrated into the proposed design methodology.

By eliminating the need to re-implement floating point applications using a fixed point representation, the proposed approach can significantly reduce the development time for hardware/software partitioning. However, a designer must still profile the application to determine the appropriate fixed point representation. To further reduce the development effort, we are currently investigating a dynamically adaptable *Float-to-Fixed* and *Fixed-to-Float* approach in which the hardware coprocessor is implemented with an adaptive fixed point representation that can be adjusted at run time to avoid potential fixed point overflows by trading off accuracy.

# 8. REFERENCES


[1] Balboni, A., W. Fornaciari and D. Sciuto. Partitioning and Exploration in the TOSCA Co-Design Flow. In Proceedings of the International Workshop on Hardware/Software Codesign (CODES), pp. 62-69, 1996.

[2] Banerjee, P., D. Bagchi, M. Haldar, A. Nayak, V. Kim, and R. Uribe, Automatic Conversion of Floating-point MATLAB Programs into Fixed-point FPGA based Hardware Design. In Proceedings of the Design Automation Conference (DAC), pp. 484-487, 2004.

[3] Belanovíc, P., M. Rupp. Automated Floating-point to Fixed-point Conversion with the fixify Environment. International Workshop on Rapid System Prototyping (RSP), 2005.

[4] Chen, W., Kosmas, P., Leeser, M., Rappaport, C. An FPGA Implementation of the Two-Dimensional Finite-Difference Time-Domain (FDTD) Algorithm. In Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA), pp. 97-105, 2004.

[5] Cmar, R., L. Rijnders, P.Schaumont, S. Vernalde, I. Bolsens. A methodology and design environment for DSP ASIC fixed point refinement. In Proceedings of the Design Automation and Test in Europe Conference (DATE), 1999.

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

[7] Guo, Z., Buyukkurt, B., Najjar, W., Vissers, K. Optimized Generation of Data-Path from C Codes. In Proceedings of the Design Automation and Test in Europe Conference (DATE), pp. 112-117, 2005.

[8] Guthaus, M., J. Ringenberg, D. Ernst, T. Austin, T. Mudge, R. Brown. MiBench: A free, commercially representative embedded benchmark suite. IEEE International Workshop on Workload Characterization (WWC), 2001.

[9] Henkel, J. A low power hardware/software partitioning approach for core-based embedded systems. In Proceedings of the Design Automation Conference (DAC), pp. 122-127, 1999.

[10] IEEE. IEEE 754: Standard for Binary Floating-Point Arithmetic. http://grouper.ieee.org/groups/754/.

[11] Keane, J., C. Bradley, C. Ebeling. A Compiled Accelerator for Biological Cell Signaling Simulations. In Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA), pp. 233-241, 2004.

[12] Keding, H., M. Willems, M. Coors, and H. Meyr. FRIDGE: A Fixed-Point Design And Simulation Environment. In Proceedings of the Design Automation and Test in Europe Conference (DATE), 1998.

[13] Lee, C., M. Potkonjak, W. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In Proceedings of the International Symposium on Microarchitecture (MICRO), pp. 330-335, 1997.

[14] Lysecky, R., G. Stitt, F. Vahid. Warp Processors. ACM Transactions on Design Automation of Electronic Systems (TODAES), Vol. 11, No. 3, pp. 659 - 681, 2006.

[15] Menard, D., D. Chillet, F. Charot, O. Sentieys. Automatic Floating-point to Fixed-point Conversion for DSP Code Generation. In Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES), pp. 270–276, 2002.

[16] Shi, C. R. W. Brodersen. Automated Fixed-Point Data-Type Optimization Tool for Signal Processing and Communication Systems. In Proceedings of the Design Automation Conference (DAC), 2004.

[17] Stitt, G., F. Vahid, G. McGregor, B. Einloth. Hardware/Software Partitioning of Software Binaries: A Case Study of H.264 Decode. In Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp. 285-290, 2005.

[18] Stitt, G., F. Vahid, S. Nematbakhsh. Energy Savings and Speedups from Partitioning Critical Loops to Hardware in Embedded Systems. ACM Transactions on Embedded Computing Systems (TECS), Vol. 3, No. 1, pp. 218-232, 2004.

[19] Venkataramani, G., W. Najjar, F. Kurdahi, N. Bagherzadeh, W. Bohm. A Compiler Framework for Mapping Applications to a Coarse-grained Reconfigurable Computer Architecture. In Proceedings of the International Conference on Compiler, Architecture and Synthesis for Embedded Systems (CASES), 2001.

[20] Wan, M., Y. Ichikawa, D. Lidsky, L. Rabaey. An Energy Conscious Methodology for Early Design Space Exploration of Heterogeneous DSPs. In Proceedings of the ISSS Custom Integrated Circuits Conference (CICC), 1998.