Published in *Proc. 12th Reconfigurable Architectures Workshop (RAW). Denver, 2005.*

Accelerating Scientific Applications with the SRC-6 Reconfigurable Computer: Methodologies and Analysis

Melissa C. Smith
Oak Ridge National Laboratory
Oak Ridge, TN, USA 37831
smithmc@ornl.gov

Jeffery S. Vetter
Oak Ridge National Laboratory
Oak Ridge, TN, USA 37831
vetterjs@ornl.gov

Xuejun Liang
Department of Computer Science
Jackson State University
xuejun.liang@jsums.edu

Abstract

Reconfigurable computing offers the promise of performing computations in hardware to increase performance and efficiency while retaining much of the flexibility of a software solution. Recently, the capacities of reconfigurable computing devices, like field programmable gate arrays, have risen to levels that make it possible to execute 64b floating-point operations. SRC Computers has designed the SRC-6 MAPstation to blend the benefits of commodity processors with the benefits of reconfigurable computing. In this paper, we describe our effort to accelerate the performance of several scientific applications on the SRC-6. We describe our methodology, analysis, and results. Our early evaluation demonstrates that the SRC-6 provides a unique software stack that is applicable to many scientific solutions and our experiments reveal the performance benefits of the system.

Keywords: Reconfigurable Computing, Scientific Applications, Performance Analysis

1. Introduction

Although initially proposed in the late 1960s, reconfigurable computing is a relatively new field of study. The ability to customize the architecture to match the computation and the data flow of the application has demonstrated significant performance benefits when compared to general-purpose architectures. Its key feature is the ability to perform computations in hardware to increase performance, while retaining much of the flexibility of a software solution. The decades-long delay had mostly to do with a lack of acceptable capacities in reconfigurable hardware to accommodate 64b floating-point operations. Only recently have these devices, namely FPGAs, reached gate densities making them suitable for high-end scientific applications, which require 64b floating-point operations. With an anticipated doubling of gate densities every 18 months, the situation will only become more favorable from this point forward.

ORNL is working with SRC Computers, in an effort to understand how their reconfigurable computing technology can benefit ORNL's scientific workloads. SRC Computers has designed the SRC-6 as a general-purpose computing system with reconfigurable direct execution logic processors (using FPGAs), called MAP processors (Multi-Adaptive Processor), for high end computing. The SRC-6 uses standard programming languages and models, and the software environment turns user applications and algorithms into direct execution logic (DEL) for MAP processors. Successful implementations with this system exist for application genres such as DSP and cryptography; this evaluation is investigating the relevance of this architecture for more general scientific applications.

2. Related Work

A significant role of reconfigurable computing (RC) in the present and near future includes improving the performance of scientific and signal processing applications. Current research highlighting work with the SRC-6 for accelerating signal-processing applications includes the Hyperspectral imagery work by El-Araby, El-Ghazawi, et al [15]. They achieved an order of magnitude speedup gain for on-board preprocessing of hyperspectral imagery with an implementation of the automatic wavelet dimension reduction algorithm on the SRC-6. Other application efforts related to the SRC-6 system include an implementation of the DARPA Boolean equation benchmarking suite [16], implementation studies of Triple DES [17], and algorithm implementations for an elliptic curve cryptosystem [18] and a generic wavelet filter [19]. Other research highlighting the system and architecture of the SRC-6 includes a thorough discussion by El-Araby et al. on the optimizations used in designing for RC, specifically the

SRC-6E [20]. Finally, a paper by Fidanci et al. discusses the performance and overhead in the SRC-6 [21].

3. SRC-6 Overview

3.1. Hardware Architecture

The SRC-6 platform consists of two general-purpose microprocessor boards and one Multi-Adaptive Processor (MAP®) board. Each microprocessor board is based on two 2.8GHz Pentium 4 microprocessors. Figure 3.1 shows the hardware architecture of the SRC MAP processor, which consists of two user-programmable Xilinx® Vertex II XC2V6000™ FPGA devices [3], six 4MB banks of On-Board Memory (OBM), and a control FPGA. This configuration affords the SRC-6 platform a 1:1 microprocessor to FPGA ratio. The microprocessor and MAP boards are connected via SNAP® cards which plug into the DIMM slot on the microprocessor motherboard [1].

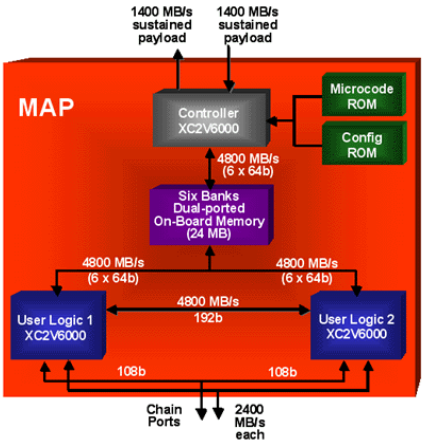

Figure 3.1: Hardware Architecture of the SRC MAP Processor with indicated bandwidths for P4 systems. *(Image courtesy of SRC Computers [2])*

3.2. Programming Model

The programming model for the SRC-6 is similar to that for a conventional microprocessor-based computing system, with the additional task of producing logic for the MAP processor. Two types of application source files are needed to target the microprocessor and MAP boards as shown in Figure 3.2. Source files intended for execution on the Intel platform (software only code) are compiled using the traditional microprocessor compiler. All source files containing functions that call hardware macros (designated with an .mf extension for FORTRAN or .mc extension if written in C) and thus execute on the MAP are compiled by the MAP compiler.

The SRC programming model has two levels of source files for the MAP processor. At a high level, the user describes the functions designated for hardware using a high-level language such as FORTRAN or C. Optimized macros for the hardware are included as a bundled library with the SRC system and can be called from the sources written for the MAP processor. This library includes functions such as: DMA calls, accumulators, counters, etc. Additionally, at a lower level, the MAP compiler allows users to integrate their own custom VHDL/Verilog functions or macros to extend the built-in set which are included with the SRC-6 platform.

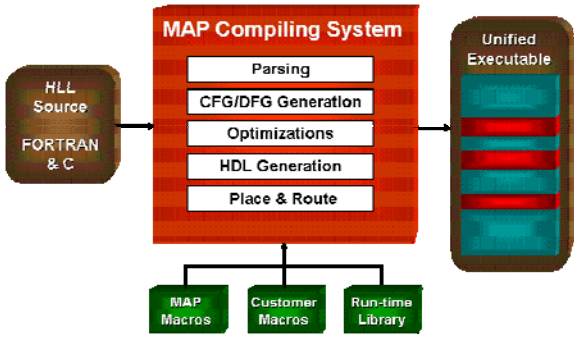

Figure 3.2: SRC Compilation Process (adapted from [1]) *(Image courtesy of SRC Computers [2])*

3.3. MAP Compiler Architecture and Code Optimization

The SRC MAP complier translates the user's MAP function, written in a high-level programming language, from source code to FPGA firmware, allowing designers with little knowledge of the hardware architecture to target the MAP processor. However, some specific code optimizations are very critical in order to achieve good performance for the generated firmware. To this end, it is also helpful to know what optimizations are performed by the MAP compiler.

In a typical execution of the firmware generated by the MAP compiler, one code block is active at a time. In the current SRC compilation environment, the parallelism is explored from the following three aspects.

(1) Operations within a basic block that have no data dependencies can execute concurrently. Therefore, the user should try to avoid data dependencies within a basic block. For example, so that the assignments are independent and can be executed in parallel, these three assignment statements can be re-written as shown:

Original	Re-written
T2 = IC T3 = T2 + INC T4 = T3 + INC	T2 = IC T3 = IC + INC T4 = IC + 2*INC

(2) All parallel sections in a parallel region will start executions simultaneously when normal execution

reaches the parallel region; the execution of a parallel region will be complete only when all parallel sections in the parallel region are complete. This programming feature can be used in a MAP function and is realized by using some compiler pragmas to mark a code segment as a parallel region and mark the designated code segments in the parallel region as parallel sections.

(3) The MAP compiler will pipeline every innermost loop in a MAP function. Therefore, the user should try to avoid loop-carried dependencies in an innermost loop in order to achieve optimum performance of a pipelined loop. Moreover, merging a nested loop into a single loop is certainly desirable since a larger loop is pipelined after merging.

Aside from the parallelism, the MAP compiler will go through a series of operation constructs to look for logic compaction: multiply by 0.5 or a power of 2, square, a*b + c*d, multiply accumulate (MAC), accumulate (ACUM), and so on. For the MAP compiler to recognize these constructs and instantiate the appropriate macro, users must write the constructs appropriately using the library of macro calls. For example, the expression e = a*b + c*d could be written as something like MMADD (a, b, c, d, & e), where MMADD is a macro that does the evaluation of the above expression.

SRC provides many user callable macros some of which can be used to optimize the user code of the MAP function. For example, the delay queue can be used to reduce redundant on-board memory accesses in a loop computation so as to increase the throughput, if that loop is innermost and thus pipelined.

3.4. Operation Modes

The SRC system also has three modes of operation: debug, simulation, and hardware. The debug mode allows the developer to develop the code and test data movement without executing the lengthy process of place and route for the MAP processor (thus does not require MAP hardware to run). The code compiles fast and executes on the microprocessor fast enough to conduct algorithm testing. The simulation mode is used by functional unit designers when developing user macros. The hardware mode is the normal mode of operation where the compiled binaries are run on the microprocessor and MAP hardware.

4. Applications

4.1. Molecular Dynamics Kernel (MD)

In the broadest sense, molecular dynamics is concerned with molecular motion. The goal of molecular dynamics algorithms is to determine how the state of a molecular system evolves over time. Given a molecular system with a set S of n particles, the state of each particle i at time t can be described by its mass m_i, its charge q_i, its position $x_i(t)$, and its velocity $v_i(t)$. The force applied to a particle i at time t is the vector sum of the pairwise interactions of particle i with all other particles in the system. These pairwise interactions can take several varieties. Typically, they can be divided into two main groups: *intra-molecular* forces and *inter-molecular* forces. Intra-molecular forces occur between close-by particles in the same molecule, while inter-molecular forces occur between any pair of particles. When considering that several of these forces decay rapidly with distance, it is enough to consider only particles inside a sphere centered at particle i. The molecular dynamics kernel calculates the interactions of the particles using Newton's equations.

4.2. Matrix-Matrix Multiplication Kernel (MM)

The matrix-matrix multiplication kernel is a significant component of many scientific algorithms. Many higher-level calculations, such as transforms, rely heavily on multiplications of large matrices and the kernel studied here will be used in block matrix decomposition algorithms for these large matrices. The abundant resources currently provided by FPGAs provide new opportunities to improve the performance of these calculations for scientific applications [14].

The first implemented algorithm employs a linear array architecture in which multiple floating-point operations are performed concurrently. Multiple MAC units are instantiated and the rows and columns of data are allowed to propagate across and down the array in a pipeline-like fashion. The resultant matrix is available once the data has propagated through the linear array.

The second algorithm uses a delay pipeline to read the matrix data once and the SRC compiler is allowed to instantiate the floating-point MACs.

4.3. Climate Modeling Kernel: PSTSWM

Parallel Spectral Transform Shallow Water Model (PSTSWM) is a message-passing application and parallel algorithm testbed that solves the nonlinear shallow water equations on a rotating sphere using the spectral transform method [4-7]. It is a parallel implementation of STSWM, developed by J. J. Hack and R. Jacob at the National Center for Atmospheric Research (NCAR) and used to generate reference solutions for the shallow water test cases described in [4]. PSTSWM was developed to evaluate parallel algorithms for the spectral transform method as it is used in global atmospheric circulation models.

The spectral transform method used in the shallow water model is comprised of a sequence of transformations between the physical domain and the spectral domain. The spectral transformation from the physical coordinate to the spectral coordinate involves performing a real fast Fourier transform (FFT) for each line of constant latitude, followed by integration over latitude using Gaussian quadrature, approximating the Legendre transform (LT), to obtain the spectral coefficients. The inverse spectral transformation involves inverse LTs and inverse real FFTs.

5. Migrating applications to the SRC-6

5.1. Matrix-Matrix Kernel Implementation

In the first algorithm implementation, we instantiate an array of 16 independent MACs (maximum number for available FPGA resources) and propagate the data through the array as shown in Figure 5.1 [9]. We explicitly encoded 16 MACs and the SRC compiler was allowed to implement the 16 MACs. The MACs are numbered from left to right. The *jth* MAC receives the input data from the *(j-1)th* MAC. The final elements of the output matrix are transferred from right to left.

Figure 5.2 demonstrates how the first matrix multiplication algorithm is implemented for the MAP processor. This implementation achieved a 4.8X speedup over an equivalent serial non-MAP version running on the SRC-6 host processor. The performance measurements include data transmission and computations times.

In the second algorithm, a delay pipeline was constructed to read the matrices from on-board memory once and the SRC compiler was allowed to determine how to compute the matrix multiplication in parallel [1]. Figure 5.3 shows how matrix B is stored in on-board BRAM and the current row of matrix A is stored in registers. The pseudo code segment in Figure 5.4 demonstrates how the matrix multiplication algorithm is implemented for the MAP processor. This implementation is not as efficient as the first algorithm and achieved only a 2X speedup.

Each algorithm implementation is 64b floating point and uses only one of the two MAP FPGAs. The device utilization and performance results for both designs are shown below in Table 1. Future work will utilized the second FPGA to populate more MACs for improved performance and processing bandwidth.

For algorithm 1, each MAC ideally can perform one floating-point addition and one floating-point multiplication every clock cycle. Hence the peak performance is expected as follows:

$$2 \times FPGA_{area} \times (FPMAC_{throughput} / FPMAC_{area})$$

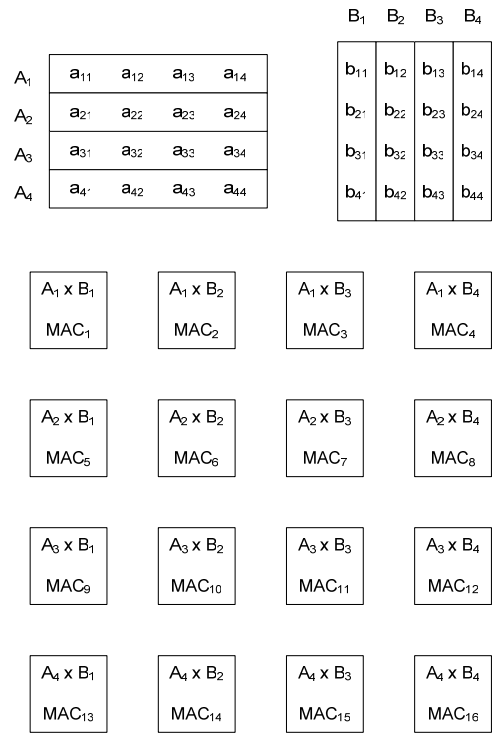

Figure 5.1: Matrix-Matrix Computation for Algorithm 1

```
Load matrix A and B to BRAM;
    For(i=0;MAC array size)
          c1(1)=c1(1)+( a1(i) * b1(i) )
          c1(2)=c1(2)+( a1(i) * b2(i) )
          c1(3)=c1(3)+( a1(i) * b3(i) )
          c1(4)=c1(4)+( a1(i) * b4(i) )
          c2(1)=c2(1)+( a2(i) * b1(i) )
        …
          c3(1)=c3(1)+( a3(i) * b1(i) )
        …
          c4(4)=c4(4)+( a4(i) * b4(i) )
```

Figure 5.2: Algorithm 1 pseudo code segment

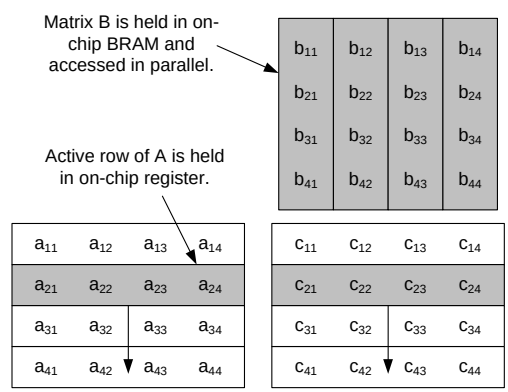

Figure 5.3: Matrix-Matrix Computation for Algorithm 2

```
Load matrix B to BRAM;
  For(j=0; rows)
    Load jth Row of matrix A into registers
    For(i=0;columns)
       c(i+((j-1)*n)) = (a1*b1(i)) +
                        (a2*b2(i)) +
                        (a3*b3(i)) +
                        (a4*b4(i));
```

Figure 5.4: Algorithm 2 pseudo code segment

Table 1: Matrix-Matrix Implementation Results

	MAP Runtime	FPGA Utilization		
		BRAM	Slices	MULTs
Algorithm 1	180 us	16%	88%	72%
Algorithm 2	440 us	6%	31%	27%

5.2. Molecular Dynamics Kernel Implementation

The core calculations of Molecular Dynamics Kernel consisted of calculating the forces acting on the each of the particles in the box for each time step. By moving these calculations into the MAP, we can achieve significant performance improvement.

As stated earlier, the calculations consist of multiplies and accumulates, nothing really novel. The performance advantage is achieved in the mapping of the data to the MAP architecture and utilizing the full capabilities of the hardware to perform the maximum number of calculations simultaneously.

The key to implementation was to determine how to best manage data movement and avoid "in-place" updates of data which would result in a "loop slow down" in the MAP. Multiple BRAM's on the FPGA were utilized to store the intermediate accumulation data and allow multiple calculations to occur concurrently without memory write contention. The method allowed the calculations to be pipelined by the compiler, further maximizing the hardware utilization and performance. The final step was to merge the accumulation data into the final arrays in the OBM.

Table 2: Molecular Dynamics Kernel Implementation Results

	FPGA Utilization		
	BRAM	Slices	MULTs
MAP 32b FP	48 (33%)	21,084 (62%)	33 (22%)

The current implementation is 32b floating point and utilizes only one of the two MAP FPGAs. The device utilization results for the 32b design are shown in Table 2. The 32b implementation achieved a 5X speedup over the serial non-MAP version running on the SRC-6 host processor. The dual chip version where a second identical computing engine is placed in the second FPGA is expected to achieve a 10X speedup over the serial non-MAP version. Dual chip and 64b implementations are in progress.

5.3. Climate Modeling Kernel Implementation

PSTSWM relies heavily on FFTs and thus computing these FFTs utilizing FPGA hardware is expected to provide significant performance advantages based on positive results with FFTs on FPGAs in real-time and DSP applications [8-12].

There are several potential design options for mapping the PSTSWM benchmark code onto the SRC-6 architecture. The SRC-6 contains four microprocessors and two MAP processors, each with two FPGA chips and six banks of on-board memory. The spectral transform can be partitioned onto the four microprocessors and the four FPGA chips in several ways. Two partitioning options are listed in Table 3 where the microprocessors execute any remaining calculations not suitable for the MAP processor FPGAs.

Table 3: Partitioning Options

	Option 1	Option 2
MAP1 FPGA1	Forward FFT	Forward FFT
MAP1 FPGA2	Forward LT	Forward FFT
MAP2 FPGA1	Inverse LT	Inverse FFT
MAP2 FPGA2	Inverse FFT	Inverse FFT

In Option 1, the spectral transform is mapped onto the MAP processor 1, with the forward FFT on one chip and the forward LT on the other chip, and the inverse spectral transform is mapped onto the MAP processor 2, with the inverse FFT on one chip and the inverse LT on the other chip. The remaining computations are executed by the microprocessors. For simplicity in determining performance, the tasks running on the microprocessors are assumed in serial mode, i.e., no parallel algorithms are used to achieve performance gain from multiprocessors.

In Option 2, two forward FFTs are implemented on one MAP processor, one with each FPGA chip, and two inverse FFTs are implemented on the other MAP processor, one on each FPGA chip. The computation task is easily portioned between the two FFTs on the two chips and since they are independent, each FFT can be performed simultaneously on different sets of vectors. Our research focuses on Option 2.

Mapping FFTs onto FPGAs has been extensively studied [8-12] and many commercial FFT IP cores are available [13]. In this research, the FFT FPGA implementation involves creating the MAP functions and since the PSTSWM code is written in FORTRAN, the MAP functions will also be written in FORTRAN.

In the PSTSWM code, there are both parallel and serial algorithms for the FFT. In this research, the serial algorithm is considered and, in order to maintain consistency with the PSTSWM code, the same in-place FFT algorithm is used to implement the FFT on FPGA. Additionally, owning to symmetry, we only consider the

mapping of the complex forward FFT onto FPGA (the CFFTF subroutine).

The CFFTF subroutine calculates in-place forward Fourier transforms of an array of complex vectors. Each vector is of length N, where N is a power of twos, threes, and fives. In this research, N is restricted as a power of 2 to simplify implementation and minimize area consumed in the FPGA. Since all problem inputs to the PSTSWM code satisfy this restriction it is not a limitation of the application code and is considered acceptable.

The inputs to CFFTF are a one-dimensional array TRIG, a two- dimensional array Y, and the sizes N, CJUM, and MVECS of the arrays, where N≤CJUM. The array Y is also used as the output.

The main computation in CFFTF can be formulated as a nested loop computation as shown in Figure 5.5.

```
For each column of Y (each vector)
    For each factor (4, or 2 and in this order) of N
        For each block (NBLOCK: number of blocks)
            For every 4 (or 2) elements with stride
              INCREM in a block
            Base-4 or Base-2 Butterfly Computation
```

Figure 5.5: CFFTF pseudo code for main computation segment

There are several notable features in this computation:

(1) The FFT computations for all vectors can be performed in parallel.

(2) If the two inner loops are for a factor 4 of N, then four elements are updated in the loop body, and the product of NBLOCK and INCREM is equal to N/4.

(3) If the two inner loops are for a factor 2 of N, then two elements are updated in the loop body, and NBLOCK is equal to N/2 and INCREM is equal to 1.

(4) The number of factor 2 of N is either 1 or 0.

Because of the obvious parallelism in the CFFTF computation, it is desirable to design a hardware unit that computes the CFFTF over a vector, and then to try to put as many copies of the hardware unit on a FPGA chip as possible. Such hardware unit design requires exploring the tradeoff between speed and area and balancing the utilization of various resources such as the on-board memory banks, the FPGA slices, the FPGA built-in multipliers, the FPGA block RAMs, and so on [22].

There are two levels of memory on the MAP processor that are used for array allocation. At the board level, six banks of on-board memory, each with 4 MB, can be accessed by both FPGA chips on the board. At the chip-level, 144 Block RAMs on the FPGA chip, each with at least 2 KB, are available to each individual chip. By default, the SRC compiler allocates arrays on the block RAM.

In CFFTF there are only two arrays. The one-dimensional array TRIG is for input and is reused over every vector CFFTF, and therefore is allocated to Block RAM. The two-dimensional array Y is for both input and output, is relatively large, and therefore is allocated to on-board memory (OBM).

As shown in Table 4, the first two designs use four OBM banks: two for inputs and two for outputs. Note that Y is a complex number array. So, two arrays are used for Y, one for the real part and one for the imaginary part. Because OBM is 64-bits wide and both real part and imaginary part of Y are 32-bits wide, the real part and imaginary part of Y are packed together in Design 3 using two OBM banks, one for input and one for output.

Table 4: Array Memory Allocations

Design	Array Memory Allocation			
	TRIG	Y	VA	VB
1	Block RAM	4 OBM Banks: 2 for input and 2 for output	None	None
2	Block RAM	4 OBM Banks: 2 for input and 2 for output	None	None
3	Block RAM	2 OBM Banks: 1 for input and 1 for output	None	None
4	Block RAM	1 OBM Bank: for both input and output	Block RAM	Block RAM

Table 5: FPGA Resource Utilization

	Slices (33,792 max)	Block RAMs (144 max)	18x18 MULTs (144 max)
Design 1	18,748 (55%)	4 (2%)	181 (125%)
Design 2	12,132 (35%)	2 (1%)	60 (41%)
Design 3	10,069 (29%)	2 (1%)	60 (41%)
Design 4	9,367 (27%)	6 (4%)	56 (38%)

Design 4 uses only one OBM bank for both input and output. In this case, two extra arrays VA and VB are needed and allocated to Block RAM. In this design, a vector (a column of Y) is loaded to VA first where VA is the input and VB is the output in the loop computation for each factor. VB is copied to VA before the next loop iteration. Finally, VB is written back to the OBM bank. Table 5 shows the FPGA resource utilization for the four designs previously discussed.

The next step is optimizing the MAP function. First, the two inner loops for a factor 4 of N can be combined as a single loop. Second, since the number of factor 2 of N is either 1 or 0, the loop computation for the factor 2 is eliminated. Also the two inner loops for a factor 2 of N become a single loop computation. Third, a multiplication operation ((L-1)*cjump) is removed. Fourth, re-write the code block:

From	To
TOFF2 = IC TOFF3 = TOFF2 + INCREM TOFF4 = TOFF3 + INCREM	TOFF2 = IC TOFF3 = IC + INCREM TOFF4 = IC + 2*INCREM

Re-writing avoids the data dependency and thus reduces the latency. This design is called Design 5.

All the previously discussed designs are implemented using integer operations. Now we focus on defining the two arrays of data in the MAP function as floating-point numbers. First, Design 5 is changed to Design 6 from 32-bit integer to 32-bit floating-point. The FPGA resource reports for Design 6 show that this design is too big to fit on the FPGA chip. Switching from integer to floating-point numbers caused a significant increase in the number of occupied slices and the design will no longer fit into the FPGA.

Next, Design 5 is changed to Design 7 where we use 64-bit floating-point numbers. An additional optimization is made to the design: only the base-2 butterfly computation is considered in CFFT computation without loss of generality. A factor 4 of N can be considered as two factors 2 of N. Therefore, the base-4 butterfly computation in CFFT is removed. As a result, the design is smaller and able to fit into the FPGA as shown in Table 6.

Table 6: Optimized FPGA Resource Utilization

	Slices (33,792 max)	Block RAMs (144 max)	18x18 MULTs (144 max)
Design 5	8,846 (26%)	6 (4%)	52 (36%)
Design 6	36,232 (107%)	6 (4%)	68 (47%)
Design 7	14,042 (41%)	12 (8%)	28 (19%)

As discussed, our implementation uses Option 2 in Table 3 and while all four FPGAs can implement this kernel in parallel, results thus far have only included one FPGA device. Additionally, it is conceivable that more FFT butterfly computations can be performed per FPGA device since there are unused FPGA resources (see Table 6). Finally, the current implementation is not integrated with the full PSTSWM application. A smaller version of the PSTSWM application which only contains the CFFTF FFT subroutine over an array of vectors is used to simplify analysis and performance measurements of the FPGA code and interface. Future work will include full versions of the PSTSWM application as well as maximum utilization of the MAP devices (all four FPGAs).

The performance is first measured without using FPGAs where the application computes the FFT over 256 vectors. The execution times for different vector lengths on the SRC host computer are recorded in Table 7.

The performances of Design 5 (32b integer version) and Design 7 (64b floating-point version) of FFT over 256 vectors are also measured for different vector lengths as shown in Table 8.

Table 7: PSTSWM on Host Computer (ms)

Vector Length	256	512	1024	2048
32b Integer	2.380	5.676	11.41	25.76
64b Floating Point	3.906	7.812	15.62	31.25

Table 8: PSTSWM with MAP (ms)

Vector Length		256	512	1024	2048
32b Integer	Data Transfer	1.348	2.662	5.294	10.57
	Computation	8.858	18.16	39.21	82.59
	Total	10.21	20.82	44.50	93.16
64b Floating Point	Data Transfer	2.688	5.320	10.58	21.12
	Computation	14.17	28.86	60.59	129.0
	Total	16.86	34.18	71.17	150.1

It can be seen that the execution time with the FPGA is longer than the host processor alone, even without considering the FPGA configuration (60ms) and data transfer overhead.

For example, the 64b floating-point FFT implementation uses only one base-2 butterfly computation in one FPGA chip. The FFT for a vector of length 1024 needs to compute 10×512 base-2 butterfly computations. Each butterfly computation takes at least 2 clock cycles since two elements in a vector need to be updated and they are stored in one BRAM. Therefore, over the entire vector, the FFT takes at least 10×512×2=10240 clock cycles. For 256 vectors, the FPGA computation takes at least 10240×256 = 2621440 clock cycles, which is 2621440×10 = 26214400 ns ≈ 26 ms with the FPGA running at 100 MHz. (Note: this clock frequency is fixed on the SRC-6 computer.) Therefore, the best FPGA FFT design with only one base-2 butterfly computation implemented will be slower than the host program, whose execution time is 15.62 ms.

However, two copies of base-2 butterfly computation may be implemented per FPGA chip (see Table 6), and two FPGA chips can be used for the FFT computation. Therefore, the FFT implementation can hold four copies of base-2 butterfly computation, and the ideal computation time with full FPGA utilization will be 6.5 (=26/4) ms. Thus, this design will be faster than the host program, whose execution time is 15.62 ms.

6. Conclusions and Future Work

We have presented an early evaluation of the SRC-6 reconfigurable computer on several scientific applications. While our early evaluations utilize only one of the FPGA devices of the SRC-6, they still demonstrate the system's potential for solving our applications with improved performance, and with the additional benefit of power and space efficiency. The unique software stack provided with the SRC-6 provides users with the functionality to transform compliant C and FORTRAN functions directly into VHDL. On two of our application kernels (MD and MM), we were able to achieve a 5X performance improvement with only one FPGA in the MAP processor. Results are expected to improve to approximately 10X with the addition of the second FPGA for these kernels, provided data bandwidth does not become a bottleneck. The potential for additional

performance improvements exists in further algorithm optimizations specific to the SRC MAP architecture. For the other application kernel (PSTSWM), no speedup was achieved for the current implementation; however, analysis was given showing potential for speedup with full resource utilization (both MAP processors and full utilization of the FPGA resources). Furthermore, optimized versions of the FFT cores may provide some benefit. Future work will focus on utilizing both FPGA devices to their maximum potential.

Despite these advantages, the process of porting our applications to the SRC-6 could be improved by, for example, providing automated tools for identifying code regions to migrate to the FPGA. Complex functions still require manual intervention to complete the compilation process. SRC is in the initial stages of development of heuristics for the compiler to implement these tasks automatically which will go a long way toward improving the efficiency of implementing designs for these architectures. Further, the user has to manage much of the MAP memory manually, i.e. determining the best array configurations to maximize memory bandwidth and/or to remove memory bank conflicts. Given the positive results from our experiments, the expected improvements in FPGA technology, and the probable improvements from additional optimization, we anticipate continued success from the SRC-6.

7. References

[1] SRC-6 FORTRAN Programming Environment Guide, SRC Computers, Inc. 2004.

[2] SRC Computers, Inc., http://www.srccomp.com/, 2004.

[3] Xilinx, Inc., Virtex-II Platform FPGAs: Complete Data Sheet, June 2004.

[4] D.L. Williamson, J.B. Drake *et al.*, "A Standard Test Set for Numerical Approximations to the Shallow Water Equations in Spherical Geometry," *Journal of Computational Physics*, 192:211-24, 1992.

[5] PSTSWM (Parallel Spectral Transform Shallow Water Model), http://www.csm.ornl.gov/shammp/pstswm.

[6] P.H. Worley and B. Toonen, "A Users' Guide to PSTAWM," *ORNL Technical Report ORNL/TM-12779*, July 1995.

[7] T. Foster and P.H. Worley, "Parallel Algorithm for the Spectral Transform Methods," *SIAM J. Sci. Stat. Comput.*, 18(3), 1997, pp. 806-837.

[8] Xinzhong Guo and Jack S. N. Jean, Design Enumeration of Mapping 2D FFT onto FPGA Based Reconfigurable Computers, *Proceedings of International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, Nevada, USA, June 2004

[9] T. Sansaloni, A. Perez-Pascual and J. Valls, Area-effient FPGA-Based FFT processor, *Electronics Letters*, Vol.39, No 19, September 2003

[10] A. H. Kamaliza, C. Pan, and N. Bagherzadeh, Fast Parallel FFT on a Reconfigurable Computation Platform, *15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'03)*, Sao Paulo, SP-Brail, November, 2003

[11] I. S. Uzun, A. Amira, A. Bouridane, FPFA Implementations of Fast Fourier Transforms for Real-Time Signal and Image Processing, *IEEE International Conference on Field-Programmable Technology (FPT'03)*, The University of Tokyo, Japan, December, 2003

[12] Nabeel Shirazi, Peter M. Athanas, and A. Lynn Abbott, "Implementation of a 2-D Fast Fourier Transform on a FPGA-Based Custom Computing Machine", *5th International Workshop on Field Programmable Logic and Applications*, Oxford, United Kingdom, Sep 1995.

[13] Xilinx, Inc., Fast Fourier Transform v2.1 Product Specification, July, 2003

[14] Zhuo, Ling and Viktor K. Prasanna, "Scalable and Modular Algorithms for Floating-Point Matrix Mulitplication on FPGAs," *International Parallel and Distributed Processing Symposium (IPDPS)*, April 2004.

[15] El-Araby, Esam, Tarek E.-Ghazawi, Jacqueline Le Moigne, and Kris Gaj, "Wavelet Spectral Dimension Reduction of Hyperspectral Imagery on a Reconfigurable Computer," *2004 MAPLD International Conference in Washington, D.C.*, September 8-10, 2004.

[16] Buell, D.A., S. Akella, J.P. Davis, G. Quan, and D. Caliga, "The DARPA Boolean equation benchmark on a reconfigurable computer," *2004 MAPLD International Conference in Washington, D.C.*, September 8-10, 2004.

[17] Fidanci, Osman Devrim, Hatim Diab, Tarek El-Ghazawi, Kris Gaj, and Nikitas Alaxandridis, "Implementation trade-offs of Triple DES in the SRC-6E Reconfigurable Computing Environment," *2002 MAPLD International Conference in Laurel, Maryland*, September 10-12, 2002.

[18] Nguyen, Nghi, Kris Gaj, David Caliga, and Tarek El-Ghazawi, "Optimum Implementation of Elliptic Curve Cryptosystems on the SRC-6E Reconfigurable Computer," *2003 MAPLD International Conference in Washington, D.C.*, September 9-11, 2003.

[19] Taher, Mohamed, Esam El-Araby, Abhishek Agarwal, Tarek El-Ghazawi, Kris Gaj, Jacqueline Le Moigne, and Nikitas Alexandridis, "Effective Implementation of a Generic Wavelet Filter on a Hybrid Reconfigurable Computer," *2003 MAPLD International Conference in Washington, D.C.*, September 9-11, 2003.

[20] El-Araby, Esam, Mohamed Taher, Kris Gaj, Tarek El-Ghazawi, David Caliga, and Nikitas Alexandridis, "System-Level Parallelism and Throughput Optimization in Designing Reconfigurable Computing Applications," *18th International Parallel and Distributed Processing Symposium (IPDPS'04) - Workshop 3*, April 26 - 30, 2004, Santa Fe, New Mexico.

[21] Fidanci, Osman Devrim, Dan Poznanovic, Kris Gaj, Tarek El-Ghazawi, and Nikitas Alaxandridis, "Performance and Overhead in a Hybrid Reconfigurable Computer," *International Parallel and Distributed Processing Symposium (IPDPS'03)*, April 22 - 26, 2003, Nice, France.

[22] Liang, Xuejun, Jeffrey Vetter, Melissa C. Smith, and Arthur Bland, "Balancing FPGA Resource Utilities," *Submitted to ERSA05*.