1 An Investigation into the Applicability of Distributed FPGAs to High Performance Computing

John P. Morrison, Padraig O'Dowd and Philip D. Healy †
Department of Computer Science,
University College Cork, Ireland.

1.1 INTRODUCTION

FPGAs (Field Programmable Gate Arrays) are silicon chips that can be continuously reprogrammed with application specific logic configurations. This interesting property gives them many advantages over ASICs (Application Specific Integrated Circuits), which contain fixed hardware configurations and cannot be altered to implement different algorithms or updated if a bug is found. The first, simple, FPGAs were manufactured in 1986 and since then they have increased considerably in logic density and in clock speed (however, FPGA clock speeds are not increasing at the same rate as microprocessor clock speeds, see below). As FPGA logic density and clock speeds increased, the field of *Reconfigurable Computing* - i.e., using the reprogrammable aspect of FPGAs to implement different algorithms directly in hardware - grew in popularity.

In order to avoid confusion it is instructive to clearly distinguish between the similar but distinct fields of Reconfigurable Computing in Embedded Computing and Reconfigurable Computing in High Performance Computing . FPGAs have found widespread adoption in the Embedded Computing field as devices for prototyping ASIC designs, creating a steadily growing, multi-billion dollar industry. Despite the much greater complexity of FPGAs in comparison with ASICs, the economies of scale achieved during their production has led to them becoming a viable alternative to ASICs in many situations. With the release of the new Spartan 3 FPGA [1] it is expected that FPGAs will push even further into high volume applications, offering

†The support of IRCSET (through the Embark Initiative) and Enterprise Ireland (through grant no. IF/2001/318) is gratefully acknowledged

a low cost alternative to ASICs. FPGAs have also eroded the Embedded Microprocessor market, as they can offer the advantage of reprogrammability but at increased execution speed. Indeed, FPGAs can be seen as blurring the distinction between hardware and software by merging the functionality of ASICs and microprocessors while maintaining the advantages of both. FPGAs containing Embedded Microprocessors, such as the Virtex II Pro [2], have become available, allowing applications to be created that are partitioned between hardware and software on the same logic device. Since embedded microprocessors and FPGAs operate at similar clock speeds (currently around 400MHz), significant application speedups are easily attainable through the use of Reconfigurable Computing in embedded computing devices.

The goal of Reconfigurable Computing in High Performance Computing is to decrease application execution time by utilizing FPGAs as co-processors in desktop computers. There is a marked difference, however, between the clock speeds of desktop microprocessors and FPGAs. Relatively few applications can provide enough parallelism for the FPGA to exploit in order to close this performance gap. Despite this, applications do exist that can benefit greatly from FPGA implementation. This paper discusses the experiences and lessons learned from trying to apply Reconfigurable Computing techniques to High Performance Computing. The work execution times of several applications on FPGAs (placed on PCI boards) and general-purpose desktop machines were compared (high performance workstations or SMP machines were not considered). Comparisons of various cost factors are considered when comparing FPGAs to microprocessors, including speedups, ease of programming and financial cost.

Unfortunately, the field of Reconfigurable Computing lacks clear benchmarks for comparing FPGAs to microprocessors. Such comparisons should include complete descriptions of the FPGA and microprocessor. For example, it is not useful to state that an FPGA can offer a five-fold decrease in application execution time over a microprocessor without specifying the relative clock speeds of the devices used. The development board on which the FPGA is placed is also of great importance. Since FPGAs can access several memory banks in parallel, the type and number of banks present on the board can have a significant effect on application performance. Also, the number of FPGAs placed on the board can obviously make a big difference. Boards with multiple FPGAs allows fine grain parallelism between the FPGAs and hence may execute an algorithm much faster then a single FPGA alone.

The remainder of this paper is organized as follows: High Performance Computing with Cluster Computing is discussed in Section 1.2. In Section 1.3, the history of Reconfigurable Computing with FPGAs is discussed including the different types of architectures used in Reconfigurable Systems. The Distributed Reconfigurable Metacomputer (DRMC) project is discussed in Section 1.4, the goal of which is to apply distributed FPGA acceleration to High Performance Computing applications. Also discussed is the type of FPGAs used and the tools used to program them. Sections 1.5 and 1.6 present applications that were implemented on DRMC/FPGAs. The first shows how an application benefited from implementation on DRMC/FPGAs, while the second illustrates that not all applications benefited from implementation on DRMC/FPGAs with respect to application speedup, financial cost and ease of

programming. Section 1.7 presents conclusions of the DRMC project and speculates on the future of FPGAs in High Performance Computing.

1.2 HIGH PERFORMANCE COMPUTING WITH CLUSTER COMPUTING

A cluster can be defined as "a collection of computers which are connected over a local network and appear as a single computer system". Cluster Computing has the goal of gathering many computers together to work as one high performance computer.

Interest in Cluster Computing has grown rapidly in recent years, due the availability of powerful desktop machines and high-speed networks as off-the-self cheap commodity components. Clusters now provide a cheap alternative to supercomputers and offer comparable performance speeds on a broad range of High Performance Computing applications. Today, free software can be downloaded from the Internet to allow groups of networked desktop machines to be easily combined into Beowulf-type PC clusters [3]. For some, Cluster Computing is now seen as the future of High Performance Computing [4].

Traditional High Performance Computing applications are diverse and are found in the scientific, engineering and financial domains. To compete in these areas, much research has been done in the area of cluster design. The goal of the DRMC project was to take the idea of Cluster Computing (with desktop machines) one step further, by adding Reconfigurable Computing hardware to form a Reconfigurable Cluster using only off-the-self components. A Reconfigurable Cluster (see Fig. 1.1) retains all the advantages of a traditional cluster, but also allows algorithms to be implemented directly in hardware (on FPGAs), with the aim of improving application execution speeds.

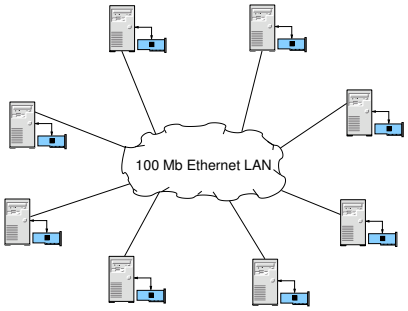

Fig. 1.1 Cluster containing reconfigurable computing boards

1.3 RECONFIGURABLE COMPUTING WITH FPGAS

As of August 2003, state of the art FPGAs like the Xilinx Virtex II [5] had clock speeds of 400 MHz and up to 10 million gates. As FPGA capabilities increased and numerous early Reconfigurable Computing research projects reported that FPGAs could provide enormous performance gains, research activities in the area surged [6] [7]. Even after years of research, Reconfigurable Computing remains in the research domain with very few examples of commercial use (one notable exception being [8]). The main market for FPGAs is still ASIC prototyping and low-cost ASIC replacement.

Many different approaches to exploiting Reconfigurable Computing exist. Some of the most common are as follows:

1. Adding a Reconfigurable Logic Unit [9] as a functional unit to a microprocessor (similar to an ALU). The microprocessor still executes algorithms in the traditional von Neumann style, but parts of some algorithms can be implemented in hardware and executed using the Reconfigurable Logic Unit. This requires building a new microprocessor from scratch and so is an expensive option.

2. Developing of custom machines with a specialized motherboard containing some number of FPGAs [8]. Algorithms are mapped onto the different FPGAs for execution. This approach is very different to the traditional von Neumann style. This type of reconfigurable architecture is suited to fine grain parallelism between the FPGAs.

3. Adding reconfigurable computing boards to general desktop machines using the PCI bus. This is the approach taken by the DRMC project [10] [11]. Drivers running on the host processors allow the FPGAs to be configured with chosen algorithms and data can be transferred to and from the reconfigurable computing boards over the machines' PCI buses. Other projects such as [12] use the more unusual approach of mixing the algorithm's logic with the networking logic in order to try and speed up execution speed even further. This requires a specialized reconfigurable computing board connected directly to the network.

An extensive list of Reconfigurable Computing projects and their target application areas can be found in [13] [14] [15]. The experiences described in this paper are based on the DRMC approach, although undoubtedly particular approaches are more suitable to implementing some applications than others. It may therefore be claimed, that specialized architectures result in the optimum implementation for certain problems (e.g., architectures that allow fine grain parallelism between several FPGAs). In the DRMC project no specialist tuning was attempted, so applications that are not suited to DRMC may execute better on other reconfigurable architectures.

Reconfigurable computing boards such as the RC1000 contain one or more FPGAs, a number of external memory banks and PMC connectors are also usually present to allow other hardware to be connected to the board, if desired.

A significant factor mitigating against the increased adoption of reconfigurable hardware is the fact that FPGA clock speeds are increasing at a far lesser rate than microprocessor clock speeds. 1n 1991, top of the range microprocessors ran at 33 MHz, but FPGAs (Xilinx XC3090) could run as fast as reported in [17]. In June 2001, top of the range microprocessors ran at 1.7 GHz, but FPGAs could only reach speeds of 200 MHz. In August 2003, top of the range microprocessors ran at 3.08 GHz, but FPGAs (Xilinx Virtex II Pro [2]) could only reach maximum speeds of 400 MHz.

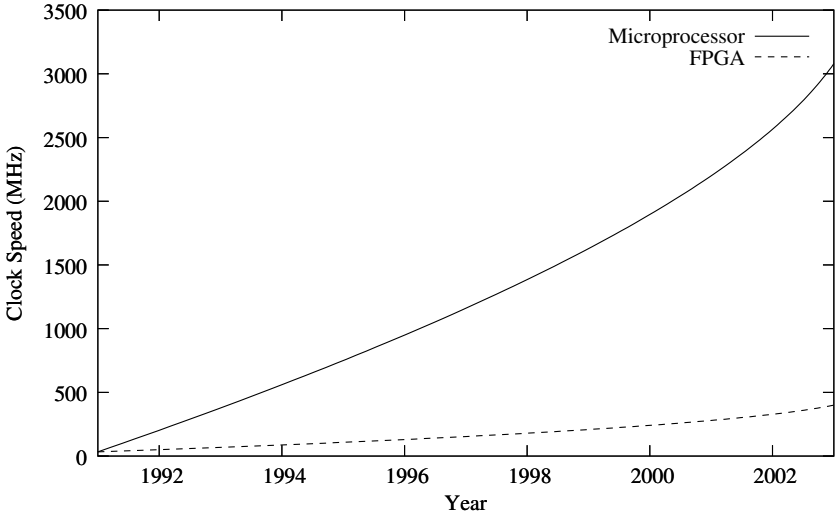

Fig. 1.2 Graph comparing the rates at which FPGA and microprocessor clock speeds are increasing

Fig. 1.2 shows these comparisons and illustrates the different rates at which the clock speeds of both FPGAs and microprocessors are increasing. If this trend continues into the future, FPGAs will have a very hard time competing with microprocessors in the general-purpose High Performance Computing arena. If FPGAs are to outperform microprocessors, they need applications that can offer tremendous amounts of parallelism. This challenge is compounded by the Place and Route tools currently available. These determine if a design can fit on an FPGA and at what clock speed the design can run. Very often designs end up with very poor clock speeds, since it is virtually impossible to realize the maximum clock speed of an FPGA. Trying to increase the clock speeds of designs beyond what the Place and Route tools initially reports can become a very laborious process, requiring significant knowledge of the underlying hardware by the designer. As a result, the speed at which the clock runs on an FPGA is often far slower than the theoretical maximum.

Notwithstanding the limitations outlined above, those applications that do exhibit the requisite level of parallelism will continue to see significant performance increases in the future. Fig. 1.3 compares the rate at which FPGA and microprocessor densities

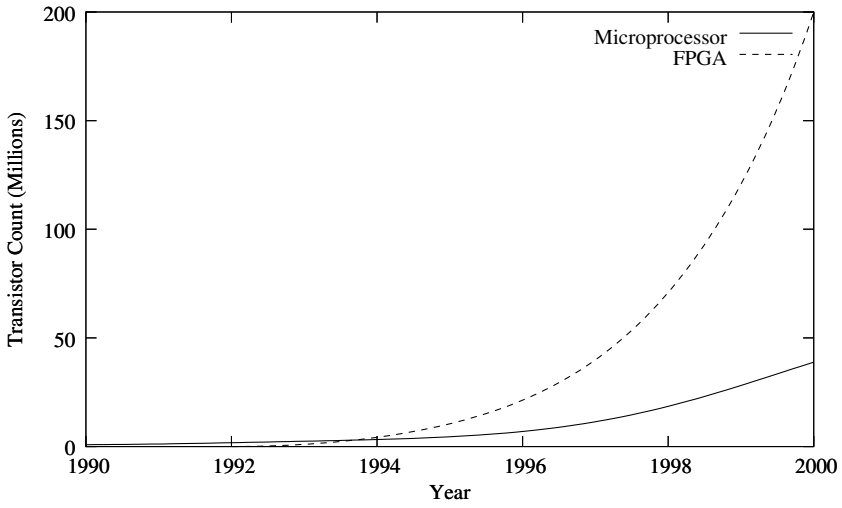

Fig. 1.3 Graph comparing the rates at which the transistor count of FPGAs and the Intel Pentium family of microprocessors increased through the 1990s.

have been increasing in recent years. If FPGA densities continue to increase at such a dramatic rate, those applications already suited to acceleration using FPGAs will see greater and greater speedups using Reconfigurable Computing techniques as time goes on. However, due to the limitations imposed by the relatively low clock speeds of FPGAs, there are many classes of application that are unlikely to benefit from FPGA implementation in the foreseeable future.

1.4 DRMC: A DISTRIBUTED RECONFIGURABLE METACOMPUTER

The Distributed Reconfigurable Metacomputer (DRMC) project [10] [11] provides an environment in which computations can be constructed in a high-level manner and executed on clusters containing reconfigurable hardware. DRMC is unique in that applications are executed on clusters using the Condensed Graphs Model of Computing [18]. The DRMC system is comprised of several components: a metacomputing platform containing a Condensed Graphs engine capable of executing applications expressed as graphs, a Condensed Graphs compiler, a control program for initiating and monitoring computations, and a set of libraries containing components that simplify application development.

1.4.1 Application Development

A DRMC application consists of a set of graph definitions (expressed as XML, in a scheme similar to the one outlined in [19]) and a set of executable instructions. Instructions are implemented either in C or as FPGA configurations.Executable instruc-

tions are represented by object code (contained in `.o` files) or FPGA configurations (contained in `.bit` files).

The Condensed Graphs Compiler compiles the set of definition graphs and links them with the set of executable instructions to produce a shared object (`.so`) file ready for dynamic linking and execution by the metacomputer. Any FPGA configurations required by the computation are loaded separately by the metacomputer as needed. Currently, application components are created manually, although tools to automate this process are under development.

1.4.2 Metacomputer Overview

The metacomputer is a peer-to-peer UNIX application composed of a daemon and, when an application is being executed, a multi-threaded computation process. The daemon is lightweight and runs on each cluster node, listening for incoming messages. At an appropriate signal from the control program, the daemon spawns a computation process. The computation process essentially consists of a number of threads that exchange instructions and results (see Fig. 1.4). At its core is the scheduler, responsible for routing instructions and results between the various modules.

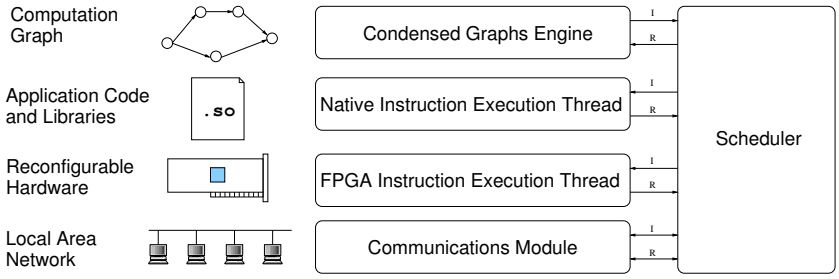

Fig. 1.4 An overview of the various components comprising a DRMC computation process, along with the resources managed by each. Arrows indicate the flow of instructions (I) and results (R).

Instructions may arrive either from the Condensed Graphs Engine or from the communications module. The scheduler sends native and Condensed Graph instructions to the Native Instruction Execution Thread. Likewise, FPGA instructions are sent to the FPGA Instruction Execution Thread. Some instructions may have implementations in both software and hardware, in which case the scheduler is free to decide which thread is most appropriate. Instructions accumulate in the scheduler while awaiting execution. The scheduler will delegate instructions to other cluster nodes if this is deemed to be more expedient than waiting for an execution thread to become available.

Results arrive from the execution threads or, in the case of instructions executed remotely, the communications module. Results for instructions that initiated on the local machine are sent to the Condensed Graphs Engine, progressing the computation. Results for instructions that originate remotely are sent to the appropriate machines.

1.4.3 Hardware Setup

The current metacomputer utilizes a standard Beowulf-type cluster [3], consisting of eight nodes, each a commodity desktop machine running the Redhat Linux operating system. The nodes were connected by a 100Mb Ethernet switch.

A single Celoxica RC1000 reconfigurable computing board [20] was fitted to each cluster node (see Fig 1.5). These boards are PCI-based and incorporate a single Xilinx Virtex XCV2000E FPGA [21] as well as 4 banks (each 2MBs in size) of on-board memory. The four memory banks can be accessed in parallel by the FPGA. This model of FPGA contains over 2.5 million gates and has a max clock rate of 100 MHz.

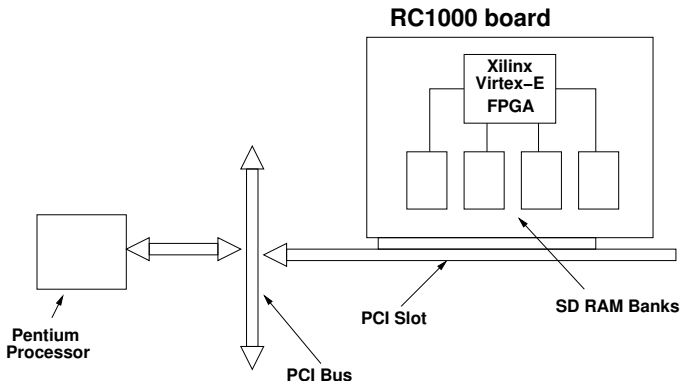

Fig. 1.5 Celoxica's RC1000 board

1.4.4 Operation

The execution of an application is initiated by sending the appropriate command from the control program to an arbitrary node in the cluster. This initiator node then spawns a computation process and broadcasts a message instructing the other cluster nodes to do likewise, specifying a shared directory containing the application code. Once the shared object containing the application code is loaded, a special registration function is called that informs the computation process of the instructions available and the libraries that the application depends on. The initiator node's computation process then commences execution of the application's top level graph, which is equivalent to a C `main` function.

As instructions become available for execution, they form a queue that is managed by the scheduler. Some instructions are executed locally by sending them to the computation process's native instruction execution thread or FPGA instruction execution thread. If these local resources are busy, some instructions will be sent for execution to other cluster nodes. Instructions corresponding to Condensed Graphs

may also be delegated to other cluster nodes, allowing parallelism to be exposed on remote machines.

Each cluster node regularly advertises its load to the others, allowing the schedulers to favour lightly loaded nodes when delegating instructions. If all the nodes are heavily loaded with long instruction queues, the computation is throttled, i.e., no more new instructions are generated until the backlog has eased.

The Control Program (CP) monitors the execution of a computation, providing the user with real-time information on the state of each machine. The CP is also responsible for the display of log messages as well as user interaction with executing applications. In the event that a computation process exits prematurely (e.g., as the result of a badly written instruction causing a segmentation fault), the DRMC daemon executing on the affected node sends an appropriate error message to the CP before broadcasting a message to halt the computation.

1.4.5 Programming the RC1000 board

The RC1000 boards were programmed using the Handel-C [22] language from Celoxica. Handel-C is a derivative of ANSI C specifically designed for translation to hardware. The language contains a number of extensions required for hardware development, including variable data widths and constructs for specifying parallelism and communications at the hardware level. Though Handel-C was chosen, other languages such as Hardware Description Languages (HDLs) like Verilog [23] and VHDL [24] could have been used, but these languages are extremely low level and are very time consuming to use compared to programming in C. Handel-C has the advantage of allowing software engineers with very little hardware knowledge to program FPGAs quickly.

Once an algorithm has been implement in Handel-C, tested by the simulator in the Handel-C IDE [25] and found to be correct it is compiled into EDIF [26] format. This EDIF design file is then passed to the Xilinx Place and Route tools [27] which produces a `.bit` (*bitstream*) file, which is used directly to configure the FPGA. Analysis with Place and Route tools reveals the longest paths in the resulting hardware design and thus the maximum clock speed at which a design can be run (sometimes the clock speed specified in the Handel-C code can not be met). If clock rate was not set in the Handel-C code, the Handel-C compiler sets it to 20 MHz by default. Through a process of iterative refinement, various optimizations can be performed until an acceptable level of speed/efficiency is reached - this means each time the clock speed is not met, the Handel-C code is modified and re-compiled and the resultant EDIF design is again passed through the Place and Route tools. Place and Route can be a long and cumbersome processes. When designs take up large amounts of the resources on the FPGA or/and have high clock speeds set, Place and Route can become a very long task. The example application in the next section took up large amounts of the logic of the FPGA, and it took well over a day to go through the process of going from the Handel-C code to a .bit file (on a 1.8 GHz Pentium 4 machine with 1 GB of RAM). Therefore, the iterative refinement to get the desired clock rate for large designs can be a slow process. As a result, the process

of application development on an FPGA is considerably more difficult than on a microprocessor.

1.5 ALGORITHMS SUITED TO THE IMPLEMENTATION ON FPGAS/DRMC

An algorithm suited to implementation on an DRMC/FPGAs is now presented. The chosen algorithm is a cryptographic key-crack application (of RC5), and is a good example of an embarrassingly parallel computation [28], i.e., it can be divided into completely independent parallel tasks that require no intercommunication.

The RC5 key-crack application running on DRMC is discussed in detail in [10]. The rest of this section presents a brief summary of that work.

RC5 is a simple and fast symmetric block cipher first published in 1994 [29]. The algorithm requires only three operations (addition, XOR and rotation), allowing for easy implementation in hardware and software. Data-dependent rotations are used to make differential and linear cryptanalysis difficult, and hence provides cryptographic strength. The algorithm takes three parameters: the word size (w) in bits, the number of rounds (r) and the number of bytes (b) in the secret key. A particular (parameterized) RC5 algorithm is denoted RC5-$w/r/b$, with RC5-32/12/16 being the most common. As 64-bit chip architectures become the norm, it is likely that 64 bit word sizes will increase in popularity. In that case it is suggested that the number of rounds be increased to 16. Variable length keys are accommodated by expanding the secret key to fill an expanded key table of $2(r + 1)$ words.

RC5 is extremely resistant to linear cryptanalysis, and is widely accepted as being secure (notwithstanding certain pathological examples that could yield to differential cryptanalysis and timing attacks) [30]. A brute-force attack (which this work focuses on), works by testing all possible keys in turn against an encrypted piece of known plain-text. This type of attack is feasible when key lengths are small and have been successfully mounted on a number of occasions using Networks of Workstations and distributed computing projects. For longer key lengths (128 bits or greater), the brute-force approach is totally inadequate, requiring millions of years to yield the key. Despite this, brute-force RC5 key-cracking is a good choice of application in order to compare the possible speed-up an FPGA can give over a traditional microprocessor implementation.

The RC5 application was implemented as a graph definition file, and a single `checkKeys` function implemented both as a native and an FPGA instruction - yielding a hardware and a software implementation. Other instructions required by the application were invoked directly from the DRMC libraries. The graph definition file was created using an XML editor. The computation graph is quite simple - it divides the key space into partitions (each containing 3 billion keys) that are then passed to instances of the checkKeys instruction. This instruction is responsible for encrypting the known plain-text with all the keys in the supplied key-space partition, and comparing the encrypted plain-text with the known cipher-text. If a match is found, the key is returned.

The software and hardware implementations of `checkKeys` are based on the RC5 implementation contained in RFC 2040 [31]. To create the native implementation, this code was augmented with an extra function interfacing with DRMC to perform type conversions. The compiled object code and the graph definition file were passed to the Condensed Graphs Compiler to create a shared object capable of being executed by the metacomputer.

The hardware implementation of `checkKeys` was created with Handel-C. The process of converting an ANSI C program to efficient Handel-C is relatively straightforward in comparison to traditional hardware design languages such as Verilog and VHDL. When the design was finished it ran at 41 MHz and consisted of three identical pipelines operating in parallel, each consisting of 8 stages. The longest stage took 72 clock cycles to execute, so the speed of the FPGA design is calculated as follows:

41MHz / 72 *3 = 1.708333 million keys per second.

The table below shows the results of the execution speed of the FPGA (at 41 MHz) compared to a Pentium II 350 MHz and a Pentium IV 2.4 GHz.

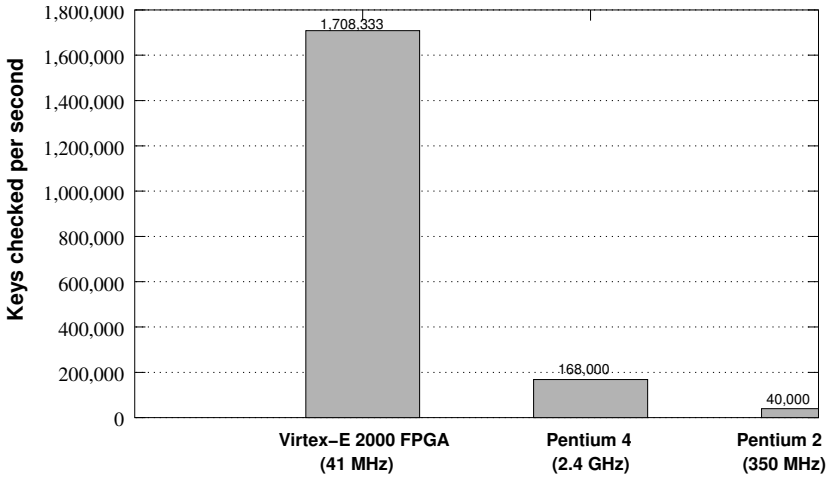

Fig. 1.6 Compares the speed differences between microprocessors and FPGAs for searching RC5 key spaces

As can be seen from Fig. 1.6 the FPGA provided more than a ten fold increase in speed over the Pentium IV and a 42 fold increase over the Pentium II. Newer FPGAs (such as the Virtex II) would offer even further speed-ups over the Virtex E FPGA used in this work as the algorithm could not alone benefit from increased clock speeds but also from higher logic densities as well. Not even a (top of the range) 3 GHz microprocessor could compete with an FPGA on this application.

A cluster containing eight Pentium II processors and eight RC1000 boards provides enough computing power to search the entire key-space of a 40-bit RC5 key in less than 22 hours. That's over 350 times faster than if the eight Pentiums alone were used. This shows that FPGAs can offer significant speed-ups over traditional micro-

processors on this type of algorithm - highly parallel, easy to pipeline, little updating of a memory store (few memory accesses) and operations that a microprocessor does slowly.

Real world applications that are similar to the RC5 key-crack application (and could benefit from implementation on DRMC/FPGAs) include encryption, compression and image processing.

It must be noted that when creating the Handel-C design for the RC5 application, significant time was spent pipelining the design and that pipelining severely affected the readability of the Handel-C source code. As a result, unlike C code for a microprocessor, updating Handel-C code that has be extensively pipelined is a difficult process. Also, compiling the Handel-C code to a `.bit` file took a long time - over a day on a 1.8 GHz machine with 1 GB of RAM.

1.6 ALGORITHMS NOT SUITED TO THE IMPLEMENTATION ON FPGAS/DRMC

Some algorithms that were found not to be suited to implementation on DRMC's FPGA architecture are now presented. These include iterative matrix solvers [32] [33] - Jacobi iteration and Gauss-Seidel iteration. These applications differ from the RC5 example described above in that they require many memory accesses.

Iterative matrix solvers are used to solve systems of simultaneous linear equations, which are used extensively in scientific and engineering applications. The following presents a quick overview of the Jacobi and Gauss-Seidel algorithms, for a detailed discussion refer to [32] and [33]. The Jacobi iteration continuously applies the same iteration on a matrix until it converges. In each new iteration, the value of each entry in the matrix is calculated based on the values of its four neighbors in the previous iteration. As all the values calculated in iteration k depends on the values in the iteration $k-1$, each entry in the matrix could be processed in parallel. Therefore, the Jacobi algorithm is highly parallel. The Gauss-Seidel iteration is similar to the Jacobi iteration, except it uses data as soon as it becomes available. When calculating the values for each entry of the matrix on iteration k, the new values of any neighbor that has already been calculated in the iteration k are used instead.

As these iterative matrix solvers involve floating point math, the floating-point library [34] from Celoxica was used. The first problem encountered was that when these algorithms were implemented using the floating-point library, the Place and Route tools reported that the maximum clock rate could only be set as high as 20 MHz, even though the maximum clock speed of the FPGA used is 100 MHz. Although it is unrealistic to expect to get a design running this high, 20 MHz was rather a poor result.

The following three algorithms were implemented on the RC1000 boards and their execution times were recorded:

1. **Gauss-Seidel Iteration:** One large matrix was placed into the four memory banks of the RC1000 board. The banks are not accessed in parallel due to

the nature of the Gauss-Seidel algorithm. The algorithm reads enough data to compute the value of four entries in the matrix at a time and writes the results back into memory. Pipelining and parallelism were used as much as possible.

2. **Jacobi Iteration:** The Jacobi algorithm requires twice as much memory as the Gauss-Seidel algorithm so the matrix has to fit into just two memory banks. Due to the parallel nature of the Jacobi algorithm the matrix is partitioned in two to allow the FPGA to operate on two memory banks in parallel. For each iteration, the algorithm computes the values of the matrices for the next iteration and writes them to the two empty banks. When the iteration is complete the values on the boundaries of the two matrices are exchanged. On the next iteration, the data is read from the memory banks that were written to in the previous iteration and written to the banks that were read from in the previous iteration.

3. **Gauss-Seidel - Jacobi Hybrid:** This algorithm is based on the combination of the Gauss-Seidel algorithm and the Jacobi algorithm and is discussed in [32] (Page 142). The matrix is broken into four; each sub-matrix is placed in one of the memory banks on the RC1000 board. In each iteration, the FPGA performs a Gauss-Seidel iteration on each of the memory banks in parallel. At the end of iteration, values on the boundaries of the matrices are exchanged (Jacobi iteration). This hybrid algorithm does not require many more iterations then the Gauss-Seidel algorithm to converge - in [32], it is stated that with a 128*128 matrix decomposed into 8 sub-matrices, there is only a five percent increase in the number of iterations compared to performing the Gauss-Seidel algorithm on the entire 128*128 matrix.

Fig. 1.7 shows and compares the execution times of these three algorithms to the Gauss-Seidel algorithm running on 1.8 GHz microprocessor. Each algorithm was run on a 500*500 matrix for 1000 iterations. These measurements record the execution time of the algorithms on the FPGA. The time spent partitioning the matrix and transferring data over the PCI bus to the RC1000 board was not considered.

Reflecting on the poor execution times of the FPGA, its easy to see that even if the RC1000 board was to run at its maximum speed of 100 MHz (which would never be possible) it would not compete with the 1.8 GHz machine. Even though the FPGA can access the four memory banks in one clock cycle, for every clock cycle of the FPGA, the microprocessor can access its main memory 11 times. In addition, reconfigurable computing boards are considerably more expensive than general-purpose desktop machines. Any speed-up which might be gained from Reconfigurable Computing must always be offset by the relative cost of the hardware. In this case, the FPGA implementation took longer and was more difficult to program then the microprocessor and couldn't compete with the microprocessor for speed - a poor result overall.

Even though these iterative matrix solver algorithms are very simple, many other algorithms exhibit the same characteristics i.e., they require many memory accesses. If an algorithm exhibits a lot of parallelism and if the data needed for this parallelism is

1000 iterations on a 500*500 Matrix	Execution Time in Seconds
Gauss-Seidel on 1.8 GHz Pentium 4 with 1 GB of 233MHz RAM	11
Gauss-Seidel on a RC1000 board running at 20 MHz	134
Jacobi on a RC1000 board running at 20 MHz	56
Gauss-Seidel - Jacobi hybrid algorithm on a RC1000 board running at 20 MHz	43 (For 1050 iterations - 45)

Fig. 1.7 Comparison of the execution speed of FPGAs and microprocessors while executing iterative matrix solvers

all stored in the same memory bank, the FPGA can only access this data sequentially and slowly due to its relatively slow clock speed.

Although many general-purpose algorithms are similar to those discussed in this section, with far fewer exhibiting the characteristics of the RC5 algorithm discussed in the previous section. Except for a few specialized applications, when all factors are considered - cost, speed-up and ease of programming, it is hard to imagine that with current technology how FPGAs (based on the model of PCI-based reconfigurable computing boards) can be considered a cost effective general-purpose computing platform compared to top of the range desktop machines.

Ignoring the fact that several standard desktop machines could be purchased for the same price as one RC1000 board, combining multiple RC1000 boards to compete with one microprocessor is not efficient either, since the RC1000 board is connected to the PCI bus of the machine and fine grain parallelism between boards is very inefficient. The reconfigurable architecture of DRMC is best suited to coarse grain parallelism between the reconfigurable computing boards, in which the FPGAs execute algorithms at least as fast (but preferably faster) than the microprocessors of desktop machines. It is worth noting that other reconfigurable computing architectures would be a lot more suited to fine grain parallelism between FPGAs and thus more suited to the iterative matrix solver algorithms mentioned above. For example, consider the development of custom machines with a specialized motherboard containing a very large number of FPGAs. This architecture allows for fine grain parallelism between the FPGAs. Smaller (and cheaper) FPGAs could be used in these machines to reduce cost. This type of machine would be complicated to program and despite its high cost would potentially outperform any standard desktop machine while executing algorithms like the iterative matrix solvers. It is not being suggested that this type of machine would provide an efficient general computing platform since many sequential algorithms could not take advantage of the multiple

FPGAs. However, for standalone applications that expose much parallelism such an architecture would provide a high performance alternative.

1.7 SUMMARY

In this paper, experiences and lessons learned from trying to apply Reconfigurable Computing (FPGAs) to High Performance Computing were discussed. This work compared execution speeds of FPGAs to general desktop machines (high performance workstations or SMP machines were not considered). Top of the range desktop machines currently run at over 3 GHz, so the gap in clock speeds between these and FPGAs is significant. It was found that an application would have to exhibit significant parallelism for the FPGA (on a PCI board) to outperform a modern microprocessor. The clock speed of microprocessors is increasing at a far greater rate then FPGA clock speeds, and as a result big differences in clock speed between the FPGAs and microprocessors can diminish much of the FPGAs' advantages. This problem is not an issue in Embedded Computing as embedded microprocessors run at a similar clock rate to FPGAs.

Speeding up some of the applications discussed in this paper might be possible using some advanced features like multiple clock domains on the FPGA, creating caches on the FPGA (to try and reduce memory access to external RAM banks) and programming with hardware description languages. These require specialized skills on the part of the programmer. To be widely accepted, FPGAs need to be as easy to program as microprocessors. Also, other reconfigurable architectures that allow fine grain parallelism between multiple FPGAs would provide a better execution environment for certain algorithms.

FPGAs and their development boards are far more expensive than general desktop machines. Thus when comparing reconfigurable computing boards to general desktop machines the reconfigurable computing board would need to achieve a significant speedup to justify its cost. So, not alone is fine grain parallelism between reconfigurable computing boards inefficient because of communication over the PCI bus, but several desktop machines could be purchased for the price of one reconfigurable computing board.

There are many publications claiming significant speedups using FPGAs compared to microprocessor systems [13] [14] [15]. These claims need to be viewed in the correct context to ascertain absolute advantages. One thing is clear; proper benchmarks are needed when comparing the speed of FPGAs to microprocessors in order to paint a more accurate picture of their advantages and limitations.

In [16] it is stated that "a good speedup candidate for FPGA implementation is complex, it requires operations the microprocessor does slowly, and it has few memory accesses". Relatively few applications in High Performance Computing meet these requirements. With current technology, when all factors are considered (execution speed, ease of programming and financial cost), the idea of using distributed FPGAs (PCI-based reconfigurable computing boards) in Cluster Computing as a general-purpose computing platform faces many obstacles. Before FPGAs become

a viable general-purpose alternative to microprocessors in High Performance Computing, there needs to be a reduction in their cost, better programming tools have to be developed and the clock speeds of FPGAs need to move closer to the clock speeds of microprocessors. The odds of these happening in the near future are small.

There are, however, many applications that are amenable to acceleration using FPGAs. These include cryptography, compression, searching, pattern matching, and image processing. As FPGA densities continue to increase at an impressive rate (much faster than microprocessor densities), the speedups attainable for these applications using Reconfigurable Computing will continue to increase dramatically. To sum up, although Reconfigurable Computing in High Performance Computing will offer better and better speedups over time to those applications to which it is already suited, it is unlikely to break into new application domains due to the reasons outlined above.

REFERENCES

1. Xilinx, Inc. *Spartan-3 1.2V FPGA Family: Complete Data Sheet*, 2003.

2. Xilinx, Inc. *Virtex-II Pro™ Platform FPGAs: Complete Data Sheet*, 2003.

3. T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U.A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995.

4. http://www.clustercomputing.org

5. Xilinx, Inc. *Virtex™-II Platform FPGAs: complete Data Sheet*, 2003.

6. R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proceedings of the Conference on Design, Automation and Test in Europe*, Munich, Germany, 2001.

7. B. Radunovic. An overview of advances in reconfigurable computing systems. In *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences, HICSS-32*, Hawaii, 1999.

8. http://www.starbridgesystems.com

9. Scott Hauck, Thomas W. Fry, Matthew M. Hosler, and Jeffery P. Kao. The Chimaera reconfigurable functional unit. In Kenneth L. Pocek and Jeffery Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 87–96, IEEE Computer Society Press, 1997.

10. John P. Morrison, Padraig J. O'Dowd and Philip D. Healy. Searching RC5 keyspaces with Distributed Reconfigurable Hardware. In *Proceedings of the*

2003 International Conference on Engineering of Reconfigurable Systems and Algorithms, pages 269–272, Las Vegas, 2003.

11. John P. Morrison, Philip D. Healy and Padraig J. O'Dowd. Drmc: A Distributed Reconfigurable Meta-Computer. In *Submitted to the International Symposium on Parallel and Distributed Computing ISPDC 2003*, Slovenia, 2003.

12. Keith Underwood, Ron R. Sass and Walter B. Ligon. Acceleration of a 2d-fft on an adaptable computing cluster. URL: citeseer.nj.nec.com/536950.html

13. http://www.io.com/ guccione/HW_list.html

14. http://www.eg3.com/WebID/soc/confproc/blank/univ/a-z.htm

15. http://www.site.uottawa.ca/ rabielmo/personal/rc.html

16. Ling-Pei Kung. *Obtaining Performance and Programmability Using Reconfigurable Hardware for Media Processing*. PhD thesis, Technische Universiteit Eindhoven, 2002.

17. P. Bertin, D. Roncin and J.Vuillemin. Programmable active memories: a performance assessment, Digital Paris Research Laboratory, 1993.

18. John P. Morrison. *Condensed Graphs: Unifying Availability-Driven, Coercion-Driven and Control-Driven Computing*. PhD thesis, Technische Universiteit Eindhoven, 1996.

19. John P. Morrison and Philip D. Healy. Implementing the WebCom 2 distributed computing platform with XML. In *Proceedings of the International Symposium on Parallel and Distributed Computing*, pages 171–179, Iasi, Romania, July 2002.

20. Celoxica Ltd. *RC1000 Hardware Reference Manual*, 2001.

21. Xilinx, Inc. *Virtex^{TM}-E 1.8V Field Programmable Gate Arrays Production Product Specification*, July 2002.

22. Celoxica Ltd. *Handel-C Language Reference Manual Version 3.1*, 2002.

23. Philip R. Moorby and Donald E. Thomas. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, May 1998.

24. Peter J. Ashenden. *The Designer's Guide to VHDL, 2nd Edition*. Morgan Kaufmann, May 2001.

25. Celoxica Ltd. *DK Design Suite Datasheet*, 2001.

26. Electronic Industries Assn. *Edif Electronic Design Interchange Format Version 200*. June, 1989.

27. Xilinx, Inc. *Development System Reference Guide - ISE 4*, 2001.

28. Gregory V. Wilson. *Practical Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, January 1996.

29. Ronald L. Rivest. The RC5 encryption algorithm. In William Stallings, editor, *Practical Cryptography for Data Internetworks*. IEEE Computer Society Press, January 1996.

30. B. Kaliski and Y. Yin. On the security of the RC5 encryption algorithm. *CryptoBytes*, 1(2):13–14, 1995.

31. R. Baldwin and R. Rivest. RFC 2040: The RC5, RC5-CBC, RC5-CBC-pad, and RC5-CTS algorithms, October 1996, URL: ftp://ftp.internic.net/rfc/rfc2040.txt.

32. Jianping Zhu. *Solving Partial Differential Equations on Parallel Computers - an Introduction*. World Scientific Publishing, 1994.

33. Gene H. Golub and Charles F. Van Loan. *Matrix Computations, Second Ed.* The John Hopkins University Press, 1989.

34. Celoxica Ltd. *Handel-C floating-point library manual, Version 1.1*, 2002.