A Current Look at the GARP FPGA CPU Hybrid

Steven Stanek Robert El-Soudani

May 12, 2005

1 Introduction & Background

We compare an updated implementation of a reconfigurable array mated to a CPU core with the original implementation in the GARP design. We examine the motivations for a reconfigurable array, briefly discuss the original implementation, mention why we choose to re-examine it and discuss various design points for an implementation following the original GARP model. Then we examine the relative performance of our proposed implementation when compared with modern processors.

1.1 Motivations

FPGAs promise the ability to create applications specific circuits, which can offer incredible speedups over traditional CPUs for many problems. However, programs are often too large and ill-suited to be implemented directly on FPGAs, so a combination of CPU and FPGA seem appropriate for executing many modern applications. The FPGA can be used to run specially optimized critical portions of programs while the CPU executes the remainder of the program and engages in activities such as I/O and system calls while controlling the array itself. Placing the FPGA and CPU on the same die allows them to exploit the same memory hierarchy and gives the CPU ISA level control over the reconfigurable array.

It should be noted that programming techniques and models differ by application. Some algorithms are well suited to hardware implementations or even designed to run quickly in hardware. These algorithms could easily be implemented on a reconfigurable array in their entirety. Other algorithms may involve large loops, issue many system calls or have unpredictable or erratic behavior. For such programs, more cleaver programming models and compilation techniques are needed to reap the benefits of reprogrammable logic.

1.2 Original GARP

Slightly less than ten years ago, John R. Hauser and John Wawrzynek proposed the GARP reconfigurable microprocessor. The GARP design consisted of a MIPS-II microprocessor mated with an reconfigurable array of logic blocks. The array for the GARP design consisted of a modest 24 rows and 32 or more columns. Each logic block could perform 2-bit operations on up to four 2-bit inputs, be the receiver or sender in memory transactions, and could store values in clocked registers. Blocks in the array were connected via wires in an asymmetric configurations so that operations across rows were optimized over those in columns.

1.3 Why We Re-examine GARP?

It has been more than eight years since the inception of the GARP during which time large strides have been made in reconfigurable logic and CPU design. Indeed, a modern $9.00 FPGA from Xilinx contains many times the number of reconfigurable blocks as proposed in the original GARP paper and more expensive models contain nearly 1,000 times that number.

CPUs too have evolved. GARP was conceived in the era of the original Pentium and CPU clock speed increases have outpaced FPGAs. Modern CPUs have better support for instruction level parallelism and much larger cycle time advantage than they did when the GARP paper was published.

Our purpose is to leverage the additional hardware resources available on modern chips to modernize the GARP design and then compare it to the modern CPUs.

2 Design

Generally, our design mirrors GARP with changes in array size, interconnect and block design. We increase the size of the array and add horizontal interconnect to accommodate the wider size. These changes also required some corresponding changes in block design so we could maintain a 64-bit configuration size for each block.

Based on examinations of hardware datasheets from 1996 to 1997 processors, transistor counts have increased to approximately 64 (2^6) times since the original GARP paper. There are many potential areas in which this additional hardware can be invested which we describe below. We describe, the original GARP design and our modifications to leverage these additional transistors.

2.1 Width & Height

As originally introduced in the Dynamic Instruction Set Computer paper [4, 5], GARP encourages a one dimensional programming model in which rows are used to implement individual operations with execution moving from the top to the bottom of the array. Thus, increasing the width of the array provides more bit-level parallelism or parallel operations while increasing the height allows the array to represent larger amounts of sequential logic.

We chose to increase the width of the array to 72 columns and set a height of 128 rows. The original GARP array, though 24 blocks (48 bits) wide, only used to center 16 blocks

(32 bits) for memory transactions. In order to generate the 64 bit addresses which are used on many modern processors, a minimum of 16 additional center blocks were required. After examination of certain applications, especially in cryptography, we felt that an operating width of 64 blocks was justified in many applications. Our choice of the 128 row height was simply a result of available hardware resources and our examination of the logic required for applications.

2.2 Interconnect

Logic blocks must be able to communicate information effectively through vertical and horizontal 2-bit wire pairs in the array. If direct connections are not possible, intermediate blocks must be used to connect wires, a waste of clock cycles and hardware resources. The GARP paper provides details for scaling interconnect to accommodate vertical growth, but not horizontal. Horizontal growth is significant as some operations such as shifts and permutations require enormous quantities of interconnect to implement. Some changes to interconnect also require changes to legal paths in the blocks.

We decided to radically increase the amount of available horizontal interconnect because the additional width made shift and permutation operations, which had already been clumsy with the previous scheme, extraordinarily time consuming with our three fold increase in overall width of the array. By providing a more capable wire network, we were able to painlessly implement some operations in one row that would have taken 4 or 5 using the original scheme. For the vertical scaling, we used the method suggested in the original paper which added only four additional wire-pairs. Also, we require that all wires be driven by a register, banning one of the original GARP clocking schemes. This convention results from our belief that the longer wires we added would make this scheme impractical.

2.3 Block Contents

Individual logic blocks have several different operating modes such as lookup tables (LUTs) that can apply arbitrary bitwise functions, multiplexors and 3-way adders in addition to logic determining what values are sent to memory and what is output and which wires to use for input or output. Hardware can be invested in adding additional computational resources to logic blocks or in space to store additional configuration data. In the latter case, programming time and memory accesses may also be affected.

We decided that maintaining a 64 bit configuration data size was vital to allow fast and easy configuration as it is a likely multiple memory bus width. Unfortunately, the increased counts of horizontal and vertical wires require that more bits be invested in selecting inputs and outputs for logic blocks so changes in the internal configuration of blocks were needed. Our changes were manifested in the elimination of certain logic (and certain logical combinations) which we felt were of limited utility but required precious configuration bits. This change means that our design is no longer block level compatible with the original GARP.

Settings for each block include:

- Specification of input and output wires
- Modes of Operation: LUTs, MUXs or 3-Adder
- LUT Contents (for LUTS)
- Optional Shifts and Inversions of some inputs (for MUXs, 3-Adder)
- Whether register values come from memory or are internally generated
- Memory bus output value

2.4 Configuration Techniques

An increase in the number of blocks in the array through changes in width or height results in more blocks which potentially need to be programmed. Certain compression techniques and configurations methods can be examined to increase initial programming speed but much of the speed of the array lies in its use of caches of recent configurations we call contexts. By increasing the number of these caches, the FPGA can refresh previous configurations quickly.

We did not consider contexts very much in our evaluation but we would encourage the investment of any remaining hardware resources in contexts as programming time for 9216 blocks is very long. We did consider some compression methods but time constraints prevented us exploring them.

2.5 Memory Access

Our memory access scheme is similar to original GARP approach. The left most block of the array can output values signaling memory requests. Requests use the memory output values of blocks 4 through 35 to generate a 64 bit memory address. Blocks 36 through 39 generate an 8 bit value to specify the row into which the memory request should be read to or written from. Reads will write into the specified row on the following array clock cycle, writes will access the value from the specified row on the same clock cycle.

Memory accesses go through the L1 cache, not directly to the memory system so the clock speed difference between the reconfigurable array and the CPU allows the L1 to potentially service many memory requests on a single array cycle. We do not fix this quantity but instead investigate its effects in our testing. In the event of cache misses or an excess of memory requests the entire array must be stalled until all outstanding memory accesses have been completed. Fortunately, the presence of the contexts, the cache of recent array configurations, allows the array to be switched to a different task on costly page faults.

2.6 Control

Unlike GARP, we did not specify a processor and ISA for the host processor. Instead, we assume that the control instructions introduced in the GARP paper also exist for controlling the our array. Instructions would include loading configurations, starting and stopping the array, transferring data into array registers and managing cached array configurations.

3 Evaluation

After examining a variety of potential algorithms and full software applications, we chose to investigate the full implementation of the Blowfish symmetric (private) key encryption algorithm and GZIP compression algorithm. We compared the relative performance of these algorithms on our proposed FPGA implementation with their actual performance on modern processors such as the Athlon XP and the PowerPC G4. Since the clock speeds of the array and the CPU differ, we commonly use metrics such as cycles per block so the actual ratio of speeds can be accounted for later.

3.1 Blowfish

Blowfish is a publicly available block cipher created by *Applied Cryptography* author Bruce Schneier. The algorithm operates on 64 bit blocks. It consists of 16 iterations of an loop with each iteration performing various permutations and substitutions on the result of the previous iteration. Over 4 KB of subkeys are required for the algorithm--considerably too much to store in the array itself so they must be stored in memory and accessed over the memory bus. Every iteration requires five memory accesses for four S-box subkeys and one P-array subkey. As this is an encryption algorithm, the needed subkeys are not predictable so caching old values on the array or prefetching is not useful. [10]

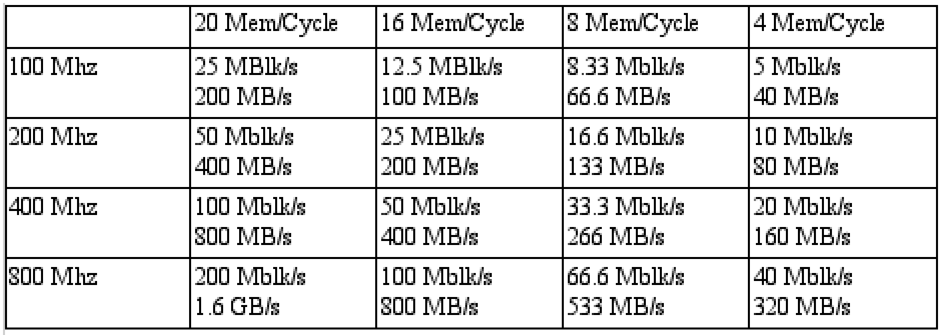
Our implementation of a single loop iteration requires 21 rows and allows for issuing all five memory requests. Additional memory requests would be needed to feed in unencrypted data and write out encrypted data. Since the array is 128 rows long, we can unroll the loop to run four iterations in hardware. The unrolled loop would then issue 20 memory requests for subkeys per clock cycle and if its pipeline is kept filled, could complete one block every 4 un-stalled array clock cycles.

It is highly doubtful that the result of one block every 4 clock cycles could be maintained due to the 20 memory requests that are placed. Thus completion rates for various FPGA memory service rates are shown:

- 20 Memory Requests per cycle: 1 completion every 4 cycles
- 16 Memory Requests per cycle: 1 completion every 8 cycles

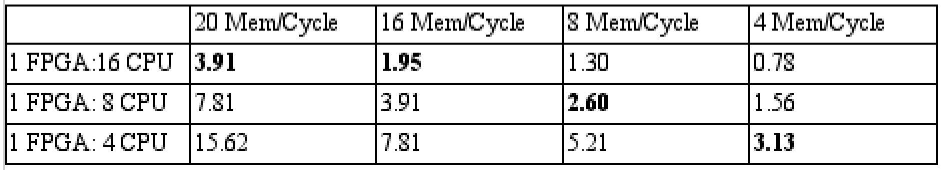
- 8 Memory Requests per cycle: 1 completion every 12 cycles
- 4 Memory Requests per cycle: 1 completion every 20 cycles

We also evaluated the Blowfish algorithm on both the PowerPC G4 and the Athlon XP CPUs. We choose to use the author's version of the source code which was compiled using gcc with all optimizations turned on. Randomly generated data was encrypted so as to avoid any L1 cache misses. Neither the Altivec short vector unit on the G4 or the SSE unit on the Athlon were explictly used, though the compiler may have made some use of SSE (Altivec support was left off). Our results indicate that the PowerPC can encrypt one block every 250 clock cycles while the Athlon can encrypt one block every 450 to 500 cycles. We believe the difference lies in the fact that the algorithm was designed for big-endian machines so the x86 based CPU must expend a large number of computations on endian conversion.



	20 Mem/Cycle	16 Mem/Cycle	8 Mem/Cycle	4 Mem/Cycle
100 Mhz	25 MBlk/s 200 MB/s	12.5 MBlk/s 100 MB/s	8.33 Mblk/s 66.6 MB/s	5 Mblk/s 40 MB/s
200 Mhz	50 Mblk/s 400 MB/s	25 MBlk/s 200 MB/s	16.6 Mblk/s 133 MB/s	10 Mblk/s 80 MB/s
400 Mhz	100 Mblk/s 800 MB/s	50 Mblk/s 400 MB/s	33.3 Mblk/s 266 MB/s	20 Mblk/s 160 MB/s
800 Mhz	200 Mblk/s 1.6 GB/s	100 Mblk/s 800 MB/s	66.6 Mblk/s 533 MB/s	40 Mblk/s 320 MB/s

Raw Array Throughput as a function of memory access rates and array clock speed



	20 Mem/Cycle	16 Mem/Cycle	8 Mem/Cycle	4 Mem/Cycle
1 FPGA:16 CPU	**3.91**	**1.95**	1.30	0.78
1 FPGA: 8 CPU	7.81	3.91	**2.60**	1.56
1 FPGA: 4 CPU	15.62	7.81	5.21	**3.13**

Speedup of FPGA over CPU for various memory access rates and ratios of FPGA clock to CPU clock (this takes stalls into account). Bold figures are the ones that are most practical.

Figure 1

In the figure 1, we show the throughput of the FPGA at certain clock speeds both in blocks per second and megabytes per second. We also show the speedup of the FPGA over the CPU for various ratios of CPU clock speed to FPGA speed and the number of allowed memory accesses per non-stall cycle. In the latter figure, the CPU assumed to be the G4, the CPU which did better in our benchmarks. We highlight results which we think are practical values for the speedups where the memory access per array clock cycle is approximately the same as the clock speed ratio between the CPU and the array.

Our results show that practical implementations of Blowfish on the reconfigurable array have speedups ranging roughly from 2 times to 4 times. Obviously a large part of this limitation is that the array spends an enormous number of cycles waiting for memory for every cycle of actual computation. With memory limitations, our results fall far short of the 24 time speedup realized by the original GARP paper using the similar DES algorithm and even extremely optimistic memory access rate fail to reach the 24 time speedup point.

3.2 Gzip

Gzip is an open source compression program commonly used in the Unix world. The bulk of computing that goes into this compression is the repeated sequential scanning of the input in order to find multiple instances of like strings. The longer the matching strings are, the more successful the compression algorithm is. During the compression of an arbitrary tiff image file of 3.75MB, our profiling tools show Gzip spends 90.4% of its processing time within the inner-loop that sequentially scans input for string matches. This inner-loop is therefore a prime candidate for implementation in hardware, as any resulting speedup will benefit the large majority of the program.

The inner-loop in question compares 4 bytes at one memory location to 4 bytes at another memory location repeatedly incrementing memory addresses until differences are found. In our hardware implementation, we use one array cycle to issue memory reads and simultaneously recalculating memory addresses for potential further memory accesses. To further exploit the parallelism offered by our array, we do comparisons in 8 byte increments. In our second array cycle, 8 bytes from each stream are compared, the length in bytes of the match is computed and stored in an array row register where the CPU can locate it. An array row register representing the array’s status is also updated every array clock. The two cycles are overlapped so that we have a 2-stage pipeline of the original loop that can compare 16 bytes per array clock. While the array is computing, the CPU will poll the array’s status and results registers. Once the length of the match is computed, the CPU will stop the array and continue with the rest of the compression algorithm. Implementing this in our array took only 8 rows due to their large width making it possible to do multiple operations in a single row. This implementation represents the most direct, naïve approach for this algorithm, as one would imagine that in the remaining 120 array rows, multiple searches could be instantiated or the entire algorithm could be redesigned to lend itself to the array hardware for far superior performance benefits, memory permitting.

Evaluation of our implementation was done on an AthlonXP 1.53Ghz machine. We used the x86 architectures provisions for counting cycles to get statistics on how long the CPU implementation takes to run Gzip on a collection of different characteristics. Our sample input consisted of large and small files as well as files with high, medium, and low compression ratios, taken both individually and together to get input-specific performance characteristics as well as a very general average. A C implementation of the array program described above was used to estimate the number of array cycles needed to complete the program on the same set of inputs. Programming of the array is estimated to take 142 array cycles considering the expected memory bandwidth and number of array blocks that need configuration (unconfigured blocks are left unused and disabled).

We found our hybrid CPU-FPGA implementation achieved an average speedup of 4 over the program running on the CPU alone. For large enough inputs in the tens of megabytes, the number of times the array was reprogrammed was negligible, although for small files in the tens of kilobytes, the speedup decreased sharply and after only 10 reprograms of the array all speedup was lost. Also, for highly compressible files, the hybrid implementation showed much more drastic speedup than for files which were not very compressible. However, for all inputs tested, there was a clear speedup even when assuming around 5 array reprograms. Figure 2 shows speedup numbers in greater detail for various input types, assuming a 1:8 CPU to array clock ratio for our hybrid implementation of Gzip versus the CPU implementation of the program being run on an AthlonXP alone.

Input type	Input size	Compression ratio	Number of Reprograms				
			1	5	10	50	100
Text	408B	1.73	3.14	1.06	.993	-	-
Image	11MB	2.17	3.88	3.88	3.88	3.88	3.88
Text	11MB	289.47	10.79	10.79	10.78	10.74	10.70
Mix	50MB	2.62	4.09	4.09	4.09	4.09	4.09

Figure 2 Speedup of Gzip program on CPU-FPGA relative to single CPU implementation

4 Conclusion

After analyzing the performance of real programs with respect to our updated design of the Garp processor, we are excited to see that the design still makes sense. There is definitely a handful of commonly used applications such as encryption or compression which permeates even the most casual computer user's workload that could readily benefit from a hybrid architecture such as the Garp processor.

The variation in speedup due to variations in input characteristics suggest that programming of such an architecture will require caution and care. Quick heuristics may be necessary on input data to determine whether to activate the array for a particular input or to do the equivalent on the CPU. Prefetching of array configurations as scheduled by a clever programmer can potentially offset the startup programming time of the array, and when combined with the multiple contexts provided by a cache of configurations, array programming can be made negligible.

It is important to note that here we employed the most simple approach to programming the array. Yet, even for this naïve, straightforward port from single CPU to hybrid CPU-FPGA implementation, a speedup of around 4 was achieved. With such architectures available, one must assume that potential performance increases will attract the attention of clever software engineers, and algorithms will be tailored for such architectures, potentially increasing performance speedup even more.

Even today this architecture remains very attractive. With proof that in the simplest case such an architecture can provide large performance increases, it should now be the point of further research to augment and mature the way the reconfigurable hardware is used. The rigid structure of the array design makes it possible to imagine relocatable configurations managed by the kernel much like the rest of the program is managed in virtual memory. Programming model changes for this architecture are not required to prove its worth, but it is expected that they are worth exploration. Also, programming the array by hand proves very difficult, perhaps even more so than directly programming x86 assembly. A key area of interest must be providing automatic compilation of code to configurable hardware. This is key for the success of such an architecture.

Currently we are at the end of a trend. We designers have gotten use of so many transistors that increased cache sizes are providing negligible performance and CPUs already exploit as much ILP as they can given their sequential structure. Furthermore, the clock speed of the CPU is not expected to continue ramping up as quickly as it has in the past due to physical limitations. Configurable hardware, on the other hand, is relatively young and FPGA development may provide an alternative avenue for performance increases in a hybrid chip. Speedup of future applications must come from a new direction, and a hybrid solution offers definitive results.

5 Resources

[1] David Andrews, Douglas Niehaus, Razali Jidin, *Programming Models for Hybrid FPGA-CPU Computational Components: A Missing Link*, University of Kansas, July 2004.

[2] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, Yury Markovskiy, John Wawrzynek, and AndrŽ DeHon, *Stream Computations Organized for Reconfigurable Execution (SCORE): Introduction and Tutorial*, extended version of the paper appearing in Conference on Field Programmable Logic and Applications, August 28--30, 2000.

[3] Micheal J. Flynn, *Very High Speed Computing Systems*, Proceedings of IEEE, vol 54, no. 12, December 1966.

[4] John R. Hauser and John Wawrzyneck, *Garp: A MIPS Processor with a Reconfigurable Coprocessor*, Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '97, April 16-18, 1997).

[5] John R. Hauser, *The GARP Architecture*, University of California Berkeley, Department of Electrical Engineering and Computer Science, October 1997.

[6] John Hennessey and David Patterson, *Computer Architecture A Quantitative Approach*, 3rd edition, Morgan Kaufmann, 2003 (Appendix G).

[7] Andre Hon, *DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century*, Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, April 1994, pp 31-39.

[8] Micheal J Wirthlin and Brad L. Hutchings, *A dynamic instruction set computer*, Symposium on FPGAs for Custom Computing Machines, April 1995, pp 99-107.

[9] R. Ramakrishna Rau and Joseph A. Fisher, *Instruction Level Parallel Processing: History, Overview & Perspective*

[10] Bruce Schneier, *Applied Cryptography: Protocols, Algorithms, and SourceCode in C*, 2nd edition, JohnWiley and Sons, 1996.