A Fast Algorithm for Prime Number Generation using Dynamically Reconfigurable Logic

Michael Saleeba*
and **Ronald Pose****
Department of Computer Science
Monash University
Clayton, Vic 3168, Australia
Phone: +61 3 9905 5194
Fax: +61 3 9905 5146

Abstract. This paper describes an algorithm which uses the runtime reconfiguration features of dynamically reconfigurable logic arrays to generate prime numbers at high speed. The algorithm is not only useful in its own right, it also serves to demonstrate some powerful techniques which can be employed in such architectures.

1 Introduction

This project was designed to demonstrate the power of self-reconfiguring logic, and in particular to test the supporting features offered by the *DRAMA* [Saleeba, 1995] architecture. The *DRAMA* architecture is a research tool which offers a number of special features for efficient runtime logic array reconfiguration. As well as creating a sizable and potentially useful application for the *DRAMA* architecture as proof of concept, the project explored the performance advantages of *DRAMA*'s architectural features.

Prime number generation was chosen as it did not have a bias towards floating point or other computation better suited to special-purpose coprocessors. Similar algorithms could find useful application in cryptography, where prime numbers are heavily used in creating code keys [Rivest et al, 1979]. The algorithm presented here could also be adapted to factorisation, and hence codebreaking.

1.1 Logic array architecture

Programmable logic devices have evolved dramatically in the last few years. One of the more recent developments is field-programmable gate arrays (FPGAs) which can have their logic function programmed while in-circuit. FPGAs are more than just a rapid-prototyping circuit element though, they can be used as a computing element in their own right [Grey and Kean, 1989; Gokhale et al, 1991; Bertin et al, 1989].

* zik@cs.monash.edu.au
** rdp@cs.monash.edu.au

Dynamic reconfiguration takes the reprogrammability of FPGAs a step further by allowing seamless reconfiguration during operation of the logic array. This provides better utilisation of the reconfigurable logic. An algorithm can use this feature to select what functions might be most useful to it at any given point during execution. This can be a big advantage in large programs which change computational characteristics frequently.

Special architectures are required to support dynamic reconfiguration. Some method must be provided to quickly modify the contents of the configuration store, and various architectures to support this have been presented. [Saleeba, 1993; Van den Bout et al, 1992; Athanas and Silverman, 1991; Athanas, 1992]

Partial reconfiguration is another useful feature. This allows sections of logic to be reconfigured while others remain the same. This can allow a double-buffered approach - one section of the array is filled with a new configuration while another is being used for computation. Partial reconfiguration can also offer the ability to reconfigure on a finer-grained basis. If two configurations are being used which have a great deal in common it allows the change from one to the other to be made without reloading the entire configuration store.

An alternative method for fast switching between configurations is to use multiple configuration stores. Logic arrays suffer from the problem that most of their die area is devoted to interconnect rather than logic function. This imbalance has led to designs where a relatively small number of quite powerful logic blocks sit in a sea of interconnect. Since the logic blocks use roughly one order of magnitude less die area than the interconnect it makes sense to increase the complexity of the logic blocks if there is any performance advantage in doing so. By duplicating the configuration store available in each logic block, perhaps several times, provides multiple configurations which can be quickly selected between. Multiple configuration stores greatly reduce the number of data transfers necessary to keep a logic array "fed" with configurations in a dynamically reconfigurable system and allow much faster switching between configurations, reducing "dead time" while the system waits on a configuration becoming available.

A step beyond the idea of dynamic reconfiguration is "self reconfiguration". If the multiple configurations of a logic array can be selected between by the circuit itself, the amount of preconfigured logic which spends most of its time waiting for the rare circumstance where it is needed can be dramatically reduced. Hardware triggers can be used determine which configuration is expressed, allowing the circuit itself to instantly select between any of those available. Designing circuits which can use this "bank-switched" technique requires some unconventional programming techniques, but can yield large performance gains in some situations.

Another method of self-reconfiguration is to allow outputs of the logic array random access to the configuration store, allowing it to rewrite its configuration in more detailed ways. This implies a high level of computational complexity in the logic array itself, but has the potential to not only increase the utilisation of conventional designs, it could allow complete machines to be built from dynamic

logic alone. These machines would be quite different from conventional machines, and would require entirely new forms of programming language and operating system to fully exploit their potential.

DRAMA [Saleeba, 1995] was designed as an abstract architecture to act as a platform for experiments with logic array architectures having the above features. The algorithm presented in this paper was designed to run on *DRAMA*, but should prove adaptable to platforms with similar architectural features.

1.2 Primes

The generation of prime numbers tends to reduce to a related problem - determining if a given number has any factors. A great deal of work has been done on factorisation algorithms over the years, and Knuth gives a good survey of these [Knuth, 1981]. An early method was devised by Fermat [Dickson, 1952], and is the basis for many of the other methods.

Probabilistic techniques for primality testing are also possible. A Monte Carlo-type method for factorisation was described by Pollard [Pollard], and a probabilistic primality test [Solovay and Strassen, 1977,1978] was found to be fast compared to other methods. While it did not produce deterministic results, the chance of error could be shown to be vanishingly small.

A continued fractions approach is well suited to finding large factors of large numbers [Lehmer and Powers, 1931; Brilliart and Morrison, 1975]. This method produces factors very quickly, with $O(N^{\epsilon(N)})$, where $\epsilon(N) \rightarrow 0$ as $N \rightarrow \infty$.

Sieve techniques have the advantage of not requiring complex maths, which makes them uniquely suited to hardware implementation. A survey of sieve methods is given in [Wunderlich, 1967]. A sieve machine is described in [Lehmer, 1933]. A 1965 hardware delay-line implementation was capable of processing one million numbers per second. A similar technique is also adopted for this paper as sieves provide the greatest effect for the least hardware.

A simple sieve requires one bit of storage for every number tested. The algorithm used here modifies the algorithm so that the sieve values are generated on the fly rather than stored.

2 Algorithm

The algorithm used in this device uses a type of parallel-sieve technique and relies on the fact that nearly all potential primes will be eliminated very quickly by an initial, limited sieve. In fact, on average the algorithm eliminates close to two numbers as primes per clock cycle. The trickle of numbers which are not immediately eliminated must be exhaustively checked for primality at considerably greater expense.

One advantage of the algorithm is that it is able to exploit the primes already produced to check for factors in later numbers. The availability of these primes drastically reduces the search space when looking for factors.

The algorithm is split into two main stages. These stages do not go through their sequence only once - the algorithm is designed to cycle between the stages indefinitely as more and more primes are generated. In fact the algorithm has no termination condition - results will be generated until the storage space for primes is exhausted. Even then, the algorithm can conceivably continue producing results to a stream device up to the square of the last prime stored.

Stated simply, the algorithm generates primes by checking each successive integer for divisibility by each prime number less than or equal to its square root. Any integer with no prime factors is added to the list of primes.

The reason for splitting the algorithm into two stages is that the issues involved in creating the first-guess "candidate" primes are different from those involved with checking all the possible factors of a candidate. Since the vast majority of numbers can be eliminated in the first phase it makes sense to optimise this case as heavily as possible.

The first stage is generates an initial set of candidate primes by testing each successive number and eliminating the primes from two upwards as factors. The upper limit on number of primes that can be checked is determined by the size of the available logic array. This stage produces a list of candidate primes in a buffer external to the logic array. These candidate primes have been checked for divisibility by low-valued primes, but may not have had all possible prime factors checked.

The second stage uses specially created logic array configurations to quickly check batches of higher-valued prime numbers against the lists of candidate primes. The results are stored in a second external buffer. When all the candidates have been checked against the primes which a configuration has been designed to check for, a new configuration is created to check for the next batch of possible factors. The candidates are passed back through the logic array to the first buffer, eliminating more potential primes in the process. This stage can repeat until all the potential factors have been exhausted, and the contents of the final buffer are declared prime.

2.1 Candidate generation

Candidate generation is the first stage of the algorithm. The candidate generator is capable of generating a candidate prime every few clock cycles. This candidate has already been checked for divisibility by a number of low-valued primes.

In essence the procedure is to count upwards through the integers, testing a new integer for primality on each clock cycle. As an optimisation only the odd integers are checked. These checks are carried out with a high degree of parallelism, with a number of factor checks occuring every cycle.

To qualify as a candidate prime, an integer must not have a prime factor in a range from 2 upwards. The upper limit is determined by the size of the logic array since each additional factor checked incurs an expense in logic elements. Given infinite circuit resources the first stage could generate all the primes itself, obviating the need for the latter stage. Unfortunately, realistic limits on the number of available array elements necessitate the use of a second stage.

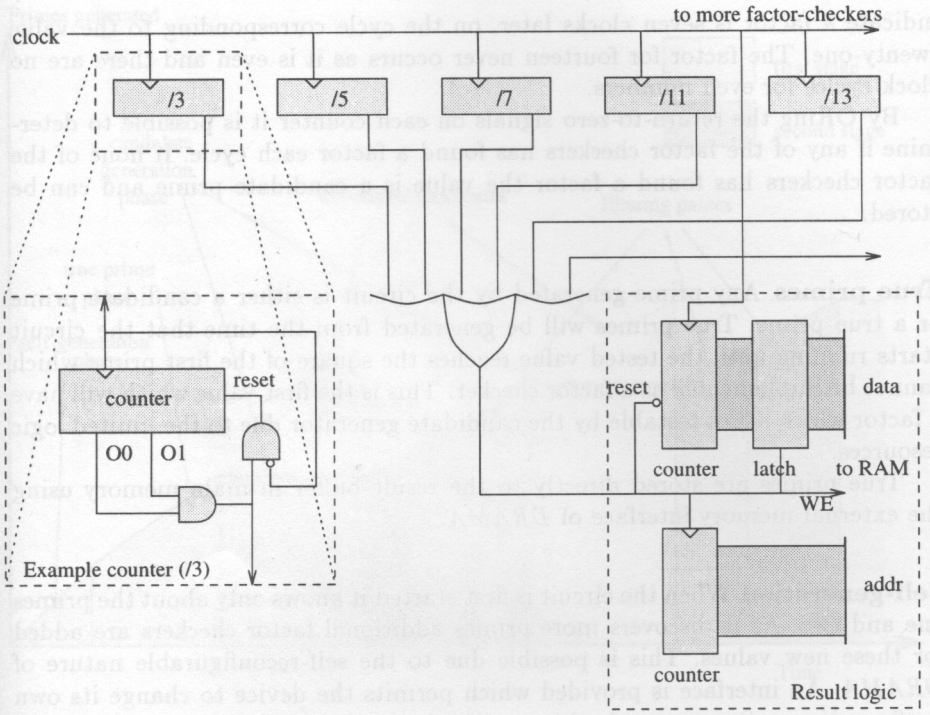

Fig. 1. Candidate generator

Every single clock cycle another number is checked for primality. All the factors which have factor checkers in hardware are checked in parallel, simultaneously. If on any given clock cycle all the factor-checking units fail to find a factor, the number is saved on the list of candidate primes.

Ideally as many factors as possible should be checked before a number is declared a candidate and passed to the next stage. Section 3 gives simulation results describing the relationship between the number of factors checked by this first stage and the overall performance of the system.

Counting factors The factor-checking circuits are very simple. Essentially each one is a divide-by-n counter, where n is the prime factor to be tested. Each counter starts at zero and increments up to n-1, after which it returns to zero. Since the period of the counter is the same as the value of the factor, each zero output of the counter corresponds to another number divisible by that factor. Figure 1 shows a partial circuit of the first stage.

Interestingly, the fact that each clock indicates a step by two rather than one does not affect the *period* of the counters, only the point at which they are initialised. In other words the if counter for prime factor seven is reset to zero on the clock cycle which corresponds to the value seven, then the next time it will

indicate a factor is seven clocks later, on the cycle corresponding to the value twenty-one. The factor for fourteen never occurs as it is even and there are no clock cycles for even numbers.

By ORing the return-to-zero signals on each counter it is possible to determine if any of the factor checkers has found a factor each cycle. If none of the factor checkers has found a factor the value is a candidate prime and can be stored.

True primes Any prime generated by the circuit is either a candidate prime or a true prime. True primes will be generated from the time that the circuit starts running until the tested value reaches the square of the first prime which cannot be implemented as a factor checker. This is the first value which will have a factor which is not testable by the candidate generator due to the limited logic resources.

True primes are stored directly to the result buffer in main memory using the external memory interface of *DRAMA*.

Self-generation When the circuit is first started it knows only about the primes one and two. As it discovers more primes additional factor checkers are added for these new values. This is possible due to the self-reconfigurable nature of *DRAMA*. An interface is provided which permits the device to change its own configuration as it operates. In this case the feature is used to create a new factor checker containing a divide-by-n counter where n is the value just checked.

The algorithm for creating the factor checker is simple enough but expensive of logic array space. For this reason it is stored in a separate configuration, as will be described shortly. The first thing it does is find an appropriate area of free logic to configure. This is easy on *DRAMA* as elements can be allocated linearly. A configuration for a simple ripple counter is created, with the reset function being determined by the bit pattern of the factor being checked. The generator's main OR function is modified to include the output of the new circuit.

2.2 Phases

The progress of the first stage can be split into three phases - firstly the self-generation phase (which also produces the low-valued true primes), next the true prime generation phase, and finally the candidate generation phase, which can recur indefinitely. See figure 2. It may seem overly complex to switch between phases this way, but in fact most of the circuit remains identical - the change in phase merely indicates that a small subsection of the array has switched from one configuration to another. This is an advantage of the "multiple configuration" feature of *DRAMA*, which permits arbitrary sections of circuit to change configuration instantly on receipt of circuit stimuli.

Multiple configuration is also used heavily during the self-generation phase. When a new configuration is to be created the normal clocking scheme and some of the supporting logic is switched out while the new factor checker is constructed

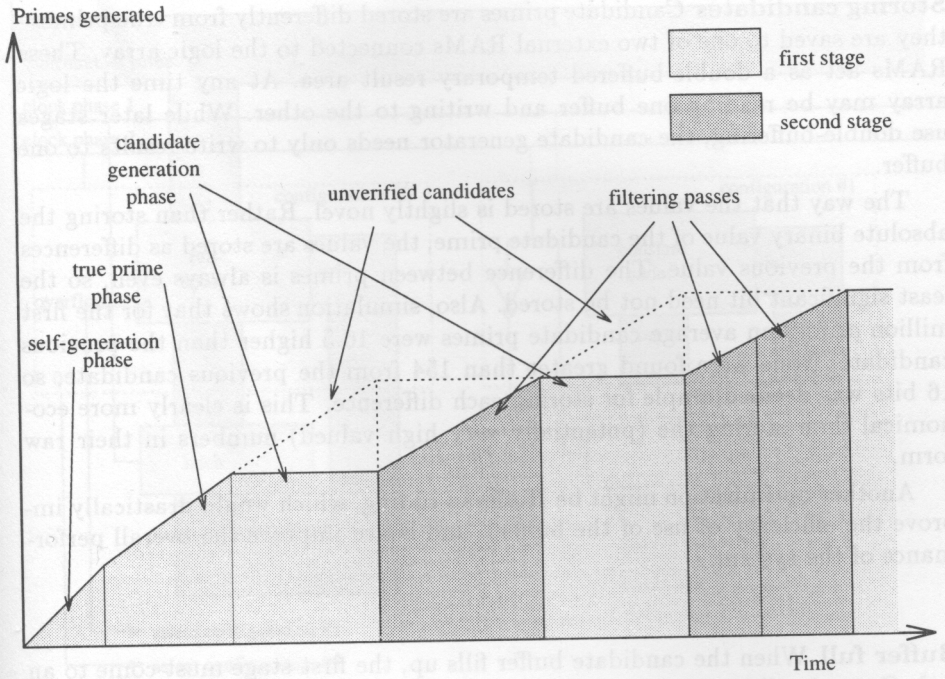

Fig. 2. A general outline of execution

by a different configuration. When this is complete the normal generator circuitry is switched back in and continues as if uninterrupted.

Calculating squares Yet another configuration is used for calculating squares. The square of the first prime which can't fit in the device as a factor checker is used to determine the transition from true prime generation to candidate prime generation. On the transition from the self-generation phase to the true prime phase the discovery of the first true prime causes the system to switch configurations, perform a simple shift/add multiply to square the current number, and then modify the configuration again.

The modified configuration sets an AND gate to monitor the current value being tested and inform the system when it's time to change to candidate generation phase. In non-reconfigurable systems this function would be performed by a comparator, at great expense in logic array elements. With self-reconfiguration we can use a simple AND gate and modify it to suit.

Realistically speaking it is not strictly necessary to start with a bare-bones candidate generator and bootstrap an initial candidate generator from there. It would be easy enough to generate a list of low-valued primes and create a configuration beforehand, however the algorithm described vividly demonstrates the power of self-reconfiguration.

Storing candidates Candidate primes are stored differently from true primes - they are saved to one of two external RAMs connected to the logic array. These RAMs act as a double-buffered temporary result area. At any time the logic array may be reading one buffer and writing to the other. While later stages use double-buffering, the candidate generator needs only to write results to one buffer.

The way that the values are stored is slightly novel. Rather than storing the absolute binary value of the candidate prime, the values are stored as differences from the previous value. The difference between primes is always even, so the least significant bit need not be stored. Also, simulation shows that for the first million primes on average candidate primes were 16.5 higher than the previous candidate. None were found greater than 154 from the previous candidate, so 16 bits was deemed ample for storing each difference. This is clearly more economical than storing the (potentially very high valued) numbers in their raw form.

Another optimisation might be Huffman coding, which would drastically improve the efficiency of use of the buffers, and hence improve the overall performance of the system.

Buffer full When the candidate buffer fills up, the first stage must come to an end. Control will be passed to the second stage by means of switching configurations. At some later stage when all the available candidate primes have been processed by the later stages, control will return to the first stage so that more candidates can be generated. This highlights an important feature of the way *DRAMA* handles its multiple configurations - when a configuration is switched out its *state* is stored with it. This means that restoring the candidate generator to the exact place where it left off is simply a matter of switching its configuration back in.

2.3 Clearing factors

The purpose of the second stage is to check the list of candidate primes and eliminate any with factors. Since the hardware is incapable of checking all the factors simultaneously, the algorithm divides the task into groups of factors to be checked and iteratively checks each group of factors against the list of candidates. On each iteration the candidates are read one by one from the source buffer, checked for factors, and candidates for which no factors have been found are written to the destination buffer.

After each pass a configuration with the next set of factors to check is established and the process is repeated. The only difference is that the source buffer and destination buffer swap their roles with each pass, performing a sort of "sifting" action on the candidates as they are "shaken" back and forwards between the two buffers.

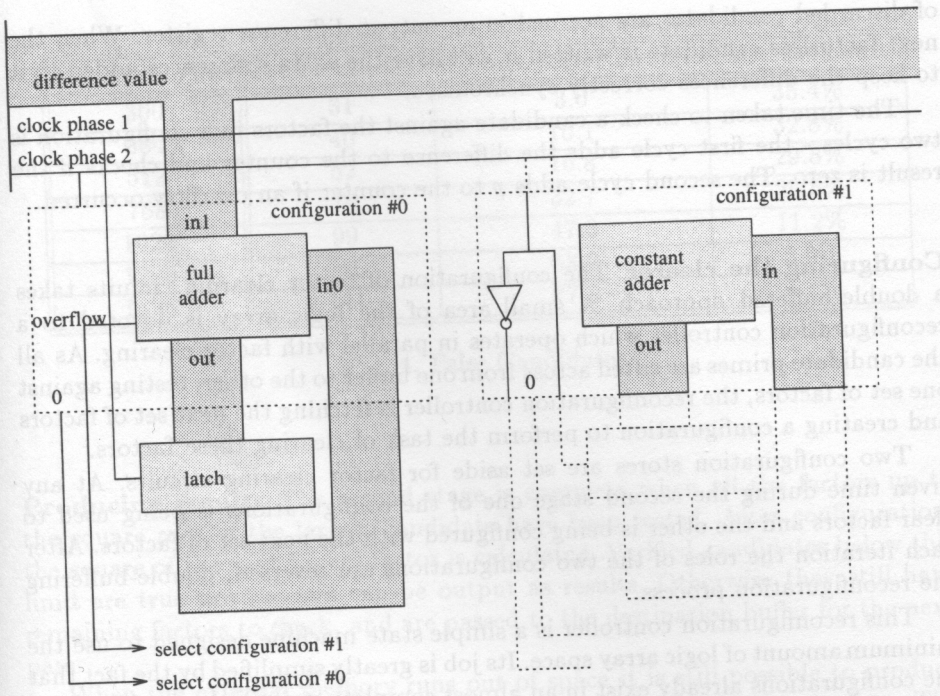

Fig. 3. Factor clearer

Factor clearing The process of checking factors in the second stage is similar in concept to that used in the first stage, though the procedure is slightly more complex. The checkers (called "clearing units" here to distinguish them from the factor checkers of the first stage) use the same technique of counting up the moduli of each of the factors being tested, however unlike stage one they can't count in predicatable steps.

The stream of candidate primes coming in from the source buffer is encoded as differences, so instead of incrementing the modulus counters by a fixed step as in stage one, the difference is *added* to the values of each of the counters.

As with stage one the counters have a period equal to the value of the factor being cleared. In this case though the counters count from a value p part way through the output range up to a state with all bits set. This approach is designed to minimise the circuit complexity and hence space that each clearing unit consumes. When a counter overflows p is added to the counter to return it to the appropriate range (figure 3). In this way the period of the counters is kept correct. p is simply the two's complement of the period.

When a counter overflows the output is quickly compared to zero before the addition takes place. If it was non-zero on all the clearing units, no factors were found and the candidate is written to the destination buffer. If on the other hand factors were found, then this candidate can be discarded. The difference values

of discarded candidates are accrued in an output difference register. When the next factorless candidate is written it will have the extra differences added to it to keep the differences correctly synchronised.

The time taken to check a candidate against the factors in a configuration is two cycles - the first cycle adds the difference to the counter and checks if the result is zero. The second cycle adds p to the counter if an overflow occurred.

Configuring the clearer The configuration of factor clearing circuits takes a double-buffered approach. A small area of the logic array is devoted to a reconfiguration controller which operates in parallel with factor clearing. As all the candidate primes are sifted across from one buffer to the other, testing against one set of factors, the reconfiguration controller is fetching the next set of factors and creating a configuration to perform the task of clearing these factors.

Two configuration stores are set aside for factor clearing circuits. At any given time during the second stage one of the configurations is being used to clear factors and the other is being configured with the next set of factors. After each iteration the roles of the two configurations are reversed, double-buffering the reconfiguration process.

This reconfiguration controller is a simple state machine designed to use the minimum amount of logic array space. Its job is greatly simplified by the fact that the configurations already exist in an almost complete state in the configuration store, and are designed so that only relatively few changes need be made to modify them to check for different factors.

Of course this same function could have been performed by statically-configured logic but reconfiguration has the advantage of allowing the size of the clearing units to be determined by necessity, allowing more to be squeezed into the available space.

A side effect of the approach taken here is that the same configurations are created again and again as each new set of candidates from stage one are checked against the same factors. This initially appeared to be an expensive choice, and storing the configurations off-chip was considered. The storage requirements of this would have been prohibitive however, and a simplified design of clearing unit helped reduce the complexity of the reconfiguration controller to a point where the cost of creating the configurations from scratch was acceptable.

Prepping the moduli Before a configuration can be used, each counter must be initialised with the correct modulus value for the starting candidate value modulo that factor. After the reconfiguration controller has finished its task for each factor, the same logic array area is replaced with a configuration for a simplified long divider. This divides the candidate by the factor, and stores the modulus in the appropriate counter by means of the self-reconfiguration interface.

When a configuration is first used, its state is such that one cycle of the "add p" state occurs, bringing the modulus into the correct range.

Logic array size	*Factor checkers*	*Average factor clearers*	*False candidates*
300	31	3.0	35.4%
384	40	10.1	32.8%
512	52	16.8	29.8%
768	76	32.7	25.5%
1024	99	47.0	11.2%
1280	120	63.0	5.9%
1536	141	77.0	3.1%

Fig. 4. False Candidates

Producing results The second stage is complete when all the factors up to the square root of the largest candidate have been tested. After configuration, the square of the next higher factor is calculated. Verified candidates below this limit are true primes, and can be output as results. Otherwise they still have remaining factors to check, and are passed to the destination buffer for the next pass.

When the external memory runs out of space it is still possible to produce results, as long as there is some type of stream device to output them to. The highest number which can be checked for primality is the square of the largest number stored in the result area. The result numbers are necessary as they are used in generating the second stage factor clearers. This maximum value is likely a very large number, and reaching it would probably require considerable patience on the part of the operators.

3 Simulation

The algorithm was converted to a form suitable for running under the *DRAMA* simulator. It was found that the design could comfortably fit in a multiple-configuration device with sixteen alternate configurations available. With careful placement it would almost certainly be possible to fit it in an eight-configuration device.

The effectiveness of the algorithm is highly dependent on the size of the available logic array. The more numbers eliminated in the first stage, and hence the less false candidates produced, the more effectively the external buffers will be used, and hence the faster the system will run. Also, the second stage has a considerable overhead in the size of the reconfiguration controller. The effect of this can only be minimised by use of a relatively large logic array.

Simulations were performed with various sizes of array. The results given in table 4 show the percentage of false candidates produced for various array sizes. These results were collected for a short run of primes up to one million. The proportion of false candidates is determined by the number of factor checkers.

This in turn is determined by the size of the logic array. For comparison the number of factor clearers from the second stage is also shown. Note that since the size of the factors change from pass to pass the number of them which can fit in the logic array varies. For this reason the number of factor clearers is expressed as an average.

The proportion of false candidate primes reduces gradually as the number of factor checkers is increased. The reason for the more marked drop towards the end of the sample space is that the 141 prime factors being checked for reach the value 811, most of the way to the square root of the maximum value being tested.

Figure 5 shows the rate of output of prime numbers graphed against the size of the logic array and the runtime elapsed. The first obvious feature of the graph is that the rate of production of primes is very irregular. In fact the graph approximates figure 2, with flat candidate generation phases alternating with steep factor clearing stages.

The larger logic arrays are able to produce considerably more primes in the initial spate - the number of primes they can generate without going to the second stage varies with the square of the number of factor checkers.

The flat stage indicates the candidate generation phase, and takes about the same time in each case. Of course the larger arrays produce a smaller proportion of false candidates.

The reconfiguration overhead in the second stage is large enough that a 300-element array is only just viable. Its performance is very bad compared to larger arrays, indicative of "thrashing". As the capacity of the arrays increases the number of primes which are produced in the second stage increases dramatically.

It is to be expected that for longer runs the rate of production of primes will decrease over time. Not only do primes become more rare the higher their values become, it is also more expensive to check them for primality.

Given a realistic clock speed of 10MHz and an array size of a thousand elements, the expected output is on the order of millions of primes per second for the first few seconds of operation.

One potential problem with the second stage occurs if the number of candidates in the buffer is small enough that they are processed more quickly than the next configuration can be loaded. Interlocks exist to prevent premature changeover of configurations, but if this situation was to occur often the efficiency of the process would be greatly reduced. Simulation showed that in practice this situation occurs rarely, if ever. In fact it might be considered worthwhile to reduce the length of the external buffers from the experimental size of 64k words if their expense is a significant consideration. This would increase the chance of premature completion, but this does not appear to be a limiting factor in operation.

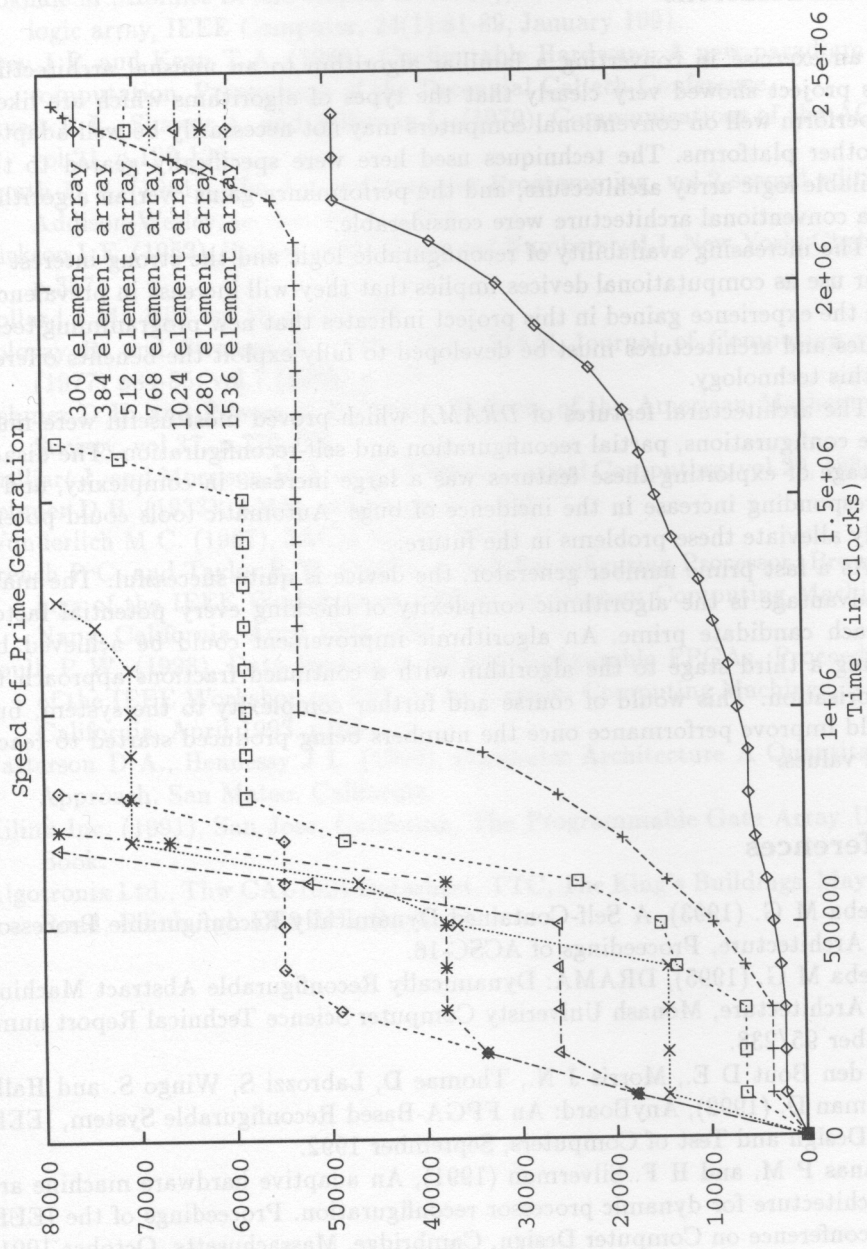

Fig. 5. Performance vs. Array Size

4 Conclusions

As an exercise in converting a familiar algorithm to an unusual architecture this project showed very clearly that the types of algorithms which are likely to perform well on conventional computers may not necessarily be well adapted to other platforms. The techniques used here were specifically geared to the available logic array architecture, and the performance gains over an algorithm on a conventional architecture were considerable.

The increasing availability of reconfigurable logic and the strong interest in their use as computational devices implies that they will increase in prevalence, and the experience gained in this project indicates that new programming techniques and architectures must be developed to fully exploit the benefits offered by this technology.

The architectural features of *DRAMA* which proved most useful were multiple configurations, partial reconfiguration and self-reconfiguration. The disadvantage of exploiting these features was a large increase in complexity, and a corresponding increase in the incidence of bugs. Automatic tools could potentially alleviate these problems in the future.

As a fast prime number generator, the device is quite successful. The main disadvantage is the algorithmic complexity of checking every potential factor of each candidate prime. An algorithmic improvement could be achieved by adding a third stage to the algorithm with a continued fractions approach to factorisation. This would of course add further complexity to the system, but would improve performance once the numbers being produced started to reach high values.

References

Saleeba M G. (1993), A Self-Contained Dynamically Reconfigurable Processor Architecture, Proceedings of ACSC-16.

Saleeba M G. (1995), DRAMA: Dynamically Reconfigurable Abstract Machine Architecture, Monash Univeristy Computer Science Technical Report number 95/238.

Van den Bout D E., Morris J N., Thomae D, Labrozzi S, Wingo S. and Hallman D. (1992), AnyBoard: An FPGA-Based Reconfigurable System, IEEE Design and Test of Computers, September 1992.

Athanas P M. and H F. Silverman (1991), An adaptive hardware machine architecture for dynamic processor reconfiguration. Proceedings of the IEEE conference on Computer Design, Cambridge, Massachusetts, October 1991.

Athanas P M. (1992), An adaptive machine architecture and compiler for dynamic processor reconfiguration, PhD thesis, Brown University, May 1992.

Bertin P, Roncin and Vuillemin (1989), Programmable Active Memories: A Performance Assessment, Technical Report #3, DEC Paris Laboratories.

Callahan P. (1988), Dynamic Instruction Set Coprocessors, MILCOM, p19.1.1-19.1.6.

Gokhale M., Holmes B. and Kopser A. (1991), SPLASH: A reconfigurable linear logic array, IEEE Computer, 24(1):81-89, January 1991.

Grey J P. and Kean T A. (1989), Configurable Hardware: A new paradigm for computation. Proceedings of the Decennial Caltech Conference.

Rivest R L., Shamir A. and Adleman L. (1979), Communications of the ACM, vol.21, p.120-126

Knuth D E. (1981), The Art of Computer Programming, vol.2 second edition, Addison-Wesley, section 4.5.4, p.364

Dickson L E. (1952), History of the Theory of Numbers vol.1, New York: Chelsea, p.357

Pollard J M., BIT vol.15, p.331-334

Solovay R. and Strassen V. (1977,1978), SIAM Journal of Computing vol.6 (1977) p84-85; vol.7 (1978) p.188

Lehmer D H. and Powers R E. (1931), Bulletin of the American Mathematics Society, vol.37, p.770-776

Brilliart J. and Morrison M A. (1975), Mathematical Computing, vol.29, 183-205

Lehmer D H. (1933), AMM, vol.40, pp.401-406

Wunderlich M C. (1967), JACM vol.14, 10-19

French P C. and Taylor R W. (1993), A Self-Reconfiguring Processor, Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, Napa, California, April 1993, p50.

Foulk P W. (1993), Data-Folding in SRAM configurable FPGAs, Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, Napa, California, April 1993, p163.

Patterson D A., Hennessy J L. (1990), Computer Architecture A Quantitative Approach, San Mateo, California.

Xilinx Inc. (1991), San Jose, California, The Programmable Gate Array Users Book.

Algotronix Ltd., Thw CAL1024 Datasheet, TTC, The King's Buildings, Mayfield Road, Edinburgh EH9 3JL, Scotland.