Optimal Runtime Reconfiguration Strategies for Systolic Arrays

Arpith C. Jacob
Jeremy D. Buhler
Roger D. Chamberlain

Arpith C. Jacob, Jeremy D. Buhler, and Roger D. Chamberlain, "Optimal Runtime Reconfiguration Strategies for Systolic Arrays," in *Proc. of 19th Int'l Conf. on Field Programmable Logic and Applications* (FPL), August 2009, pp.162-167.

Dept. of Computer Science and Engineering
Washington University in St. Louis

and

BECS Technology, Inc.

OPTIMAL RUNTIME RECONFIGURATION STRATEGIES FOR SYSTOLIC ARRAYS

Arpith C. Jacob∗, Jeremy D. Buhler∗, Roger D. Chamberlain∗†

∗Department of Computer Science and Engineering
Washington University in St. Louis
†BECS Technology, Inc.
St. Louis, Missouri
Email: {jarpith,jbuhler,roger}@cse.wustl.edu

ABSTRACT

Many computation kernels that analyze large data streams can be accelerated by converting their recurrences to parallel systolic arrays. Application domains such as bioinformatics seek to minimize the total time to analyze a large set of discrete small inputs. While traditional methods for array synthesis produce a single most efficient array design, modern computational platforms support fast runtime reconfiguration that can choose among a collection of arrays optimized for different input characteristics, such as input size.

In this work, we give dynamic programming algorithms to efficiently select a few array implementations from a large set of candidates so as to minimize total execution time on a dataset with a known distribution of input sizes. We apply our methods to accelerate the Nussinov RNA folding algorithm on a Xilinx Virtex 4 FPGA. Using runtime reconfiguration among five array instantiations, we are able to process a database of 2.7 billion RNA bases in 72 seconds, which is 48% faster than using a single array and $252\times$ faster than comparable software. We demonstrate substantial efficiency benefits even when the input length distribution is biased toward low-throughput arrays, when reconfiguration time is as large as half a second, and when only a small number of distinct arrays may be used.

1. INTRODUCTION

Numerous computational tasks on sequential or time-series data can be described by a system of recurrence equations. To accelerate the computation of such recurrences, one can derive a fine-grained parallel *systolic array* using the formal tools of space-time analysis [1]. Systolic array designs can be implemented efficiently by a variety of hardware platforms; they are commonly targeted to VLSI devices or field-programmable gate arrays (FPGAs).

The authors thank Berkley Shands for extensive support in the use of the Exegy infrastructure. This work was supported by NIH award R42 HG003225 and NSF awards DBI-0237902 and ITR-427794. R.D. Chamberlain is a principal in BECS Technology, Inc.

Classic systolic array design [2] seeks an array that firstly minimizes the latency of computation, i.e. the time to completely process one input, and secondly uses the fewest possible processing elements to achieve this optimal latency. Such a design also minimizes total computation time when the input is a continuous stream of data, as in many signal-processing applications.

However, one may instead seek to process a large collection of *discrete* inputs. In such a case, array design can explicitly minimize total execution time by maximizing computational throughput [3], processing inputs in a pipelined fashion. Streams of discrete inputs arise naturally in the domain of bioinformatics, where the input may consist of many short sequences such as high-throughput sequencing reads or of probabilistic sequence models such as hidden Markov models. This work seeks to deal efficiently with such streams of discrete inputs.

Once an array design has been chosen, it must be instantiated on the target device with a fixed array size and hence a fixed input size. If all inputs to the array are the same size, there is a naturally most efficient array size; otherwise, for any fixed array size, smaller inputs must be padded out to the array's input size, while larger ones (if supported at all) must be split and processed in multiple passes. Existing work has focused largely on selecting a single array design that yields good performance over a range of input sizes [2]. This approach is natural for VLSI synthesis, in which the design cannot change to respond to variations in input size.

Other target platforms such as FPGAs, however, have the flexibility to handle a range of input sizes without sacrificing efficiency. Because these platforms can be reconfigured quickly (less than a second) with a new array design, one can efficiently alter the array to accommodate runs of larger and smaller inputs. For smaller inputs, a smaller array instantiation eliminates the need for input padding and the associated useless computation. Moreover, because higher-throughput arrays often require substantially more computational resources, it may be possible to exploit pipelining at smaller sizes while reverting to more resource-efficient but

978-1-4244-3892-1/09/$25.00 ©2009 IEEE

slower arrays at larger sizes.

In this work, we present an algorithmic framework for deciding which of a large set of possible systolic array designs to use so as to minimize total computation time on a known distribution of input sizes. Our algorithms select a set of designs (either bounded or unbounded) and indicate when to switch among them, assuming that the inputs are sorted monotonically by size. We demonstrate the utility of our approach for the bioinformatics domain by accelerating an FPGA implementation of the Nussinov algorithm for RNA folding [4]. We briefly describe three families of arrays for this problem from our prior work, then apply our algorithm to select arrays from these families for both real and synthetic size distributions and estimate the resulting speedups.

Finally, we demonstrate a realization of runtime array reconfiguration on a Xilinx Virtex-4 LX100-12 FPGA device and measure the performance impact of reconfiguration on a database containing tens of millions of short sequences to be folded. Our runtime implementation is 48% faster than the best single array for our dataset and $252\times$ faster than a baseline software implementation. We demonstrate that our runtime reconfiguration scheme confers substantial efficiency benefits even when the input length distribution is biased toward low-throughput arrays, when reconfiguration time is as large as half a second, and when only a small number of distinct arrays may be used.

2. SELECTING AN OPTIMAL SET OF ARRAYS

Suppose we are given a large collection of input items with lengths in the range $1 \ldots M$. We assume that the collection has been analyzed offline to determine the number $C(i)$ of inputs of each size i, and that it has been sorted in nondecreasing order of input size. These assumptions are reasonable in, for example, bioinformatics data sets, which are typically generated and formatted offline and then stored in a database for analysis. In the case of an online search, sequences may be binned by size and buffered upon receipt from clients.

Let A be a set of candidate systolic array designs. Each design $a \in A$ is actually a family of instantiations $a(i)$ parametrized by input size i. The largest feasible input size $S(a)$ for a is determined by the resource limits of the target device. Array $a(i)$ has a *block pipelining period* $\beta_{a(i)}$, which is the required delay in cycles between successive inputs; the reciprocal of β is the array's throughput.

Let $E(i)$ be the minimum time required to process all inputs of size 1 to i inclusive, using some combination of designs from A. Our goal is to compute $E(M)$. A dynamic programming recurrence for $E(i)$ is given by

$$E(i) = \min_{1 \le a \le |A|} \min_{1 \le j < i} \{ \; E(j-1) + \rho + \delta_a(j,i) \; \}$$

$$\delta_a(j,i) = \left\{ \begin{array}{ll} L_a(i) + \sum_{j \le k \le i} C(k) \times \beta_{a(i)} & \text{if } i \le S(a) \\ \infty & \text{otherwise} \end{array} \right\} \quad (1)$$

where ρ is the reconfiguration time needed to load a new array on the target device, and $\delta_a(j,i)$ is the time to execute all inputs of length $j \ldots i$ on array $a(i)$. $\delta_a(j,i)$ is computed as the sum of the pipelining periods of the inputs and the latency of execution, $L_a(i)$, of the last input on the array. This latency, however, is negligible compared to the rest of the sum and the reconfiguration time, so we ignore it hereafter. All times are expressed in cycles but could be converted to seconds to combine designs with different clock speeds.

To compute the optimal execution time for inputs of length $1 \ldots i$, the recurrence considers all possible reconfiguration locations $1 \le j < i$. Sequences of length $j \ldots i$ are executed on array $a(i)$. This is followed by reconfiguration and the optimal execution of the remaining inputs. We must consider all possible locations for reconfiguration and every candidate family of arrays in A. Correctness follows from the optimal substructure of the optimization problem.

The initialization condition is $E(0) = -\rho$, and the optimal execution time $E(M)$ over all inputs can be computed bottom-up in i. We can retrieve the optimal sequence of array instantiations using a standard traceback procedure.

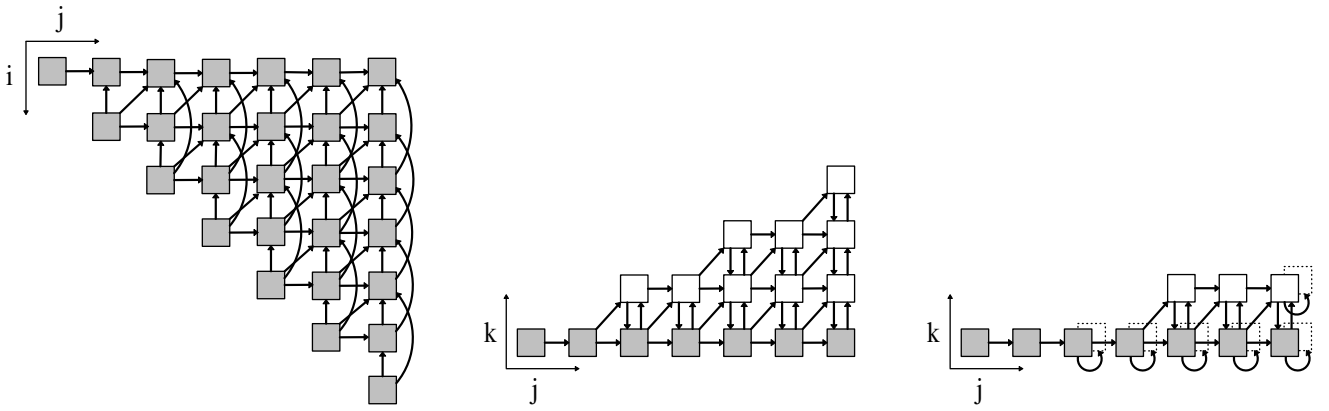
The full algorithm solves M subproblems, each requiring maximization over $O(M|A|)$ cases. One may precompute and store all required δ values in time $O(M^2|A|)$, making each case constant-time and yielding total computation time of $O(M^2|A|)$. This running time is practical provided that the size range M is restricted or that the actual set of input sizes is sparse. If it is large, we could compute faster at some cost to result quality by quantizing the set of sizes considered.

3. APPLICATION TO RNA FOLDING

In this section we give a brief introduction to the RNA folding recurrence and its systolic arrays. These arrays have been described in detail in our previous work [5].

Small RNA molecules carry out diverse functions in living cells. An RNA is a linear sequence of *bases* from the alphabet $\{A, C, G, U\}$ whose function is determined by its *secondary structure*. This structure is the folded shape that results from pairing of complementary bases (mainly A-U and C-G) within one sequence. Determining this structure is key to analyses that identify and assign functions to RNAs.

The simplest secondary structure prediction algorithm is due to Nussinov [4]. Given RNA sequence S of length N, the dynamic programming algorithm computes the largest



(a) The GKT array has the highest throughput but uses the most resources.

(b) GJQ array

(c) Clustered version of GJQ array, which can fold the longest sequences but has the lowest throughput

Fig. 1. Hardware arrays generated to accelerate the Nussinov recurrence. Clustered and resource-intensive processing elements are shown by the dashed and solid boxes respectively.

Table 1. Parallel arrays for Nussinov recurrence. β: block pipelining period. PEs: processing elements.

Array	β	**# PEs**	**Max.** N
GKT	$\frac{N-1}{2}$	$\frac{N(N+1)}{2}$	49
GJQ	$N-2$	$\frac{N(\frac{N}{2}+1)}{2}$	81
GJQC	$2N-4$	$\frac{N(\frac{N}{2}+1)}{4}$	97

number of paired bases $X(i,j)$ in any folded structure of subsequence $S_i..S_j$ as follows:

$$X(i,j) = \max \begin{cases} X(i+1,j) \\ X(i,j-1) \\ X(i+1,j-1) + \gamma(S_i, S_j) \\ \max_{i<q<j} [X(i,q) + X(q+1,j)]. \end{cases} \quad (2)$$

The score γ evaluates to 1 if bases S_i and S_j may pair or 0 otherwise. The variable X is defined over the domain $1 \leq i < j \leq N$; the score of the best structure for the whole sequence is $X(1,N)$. We are interested only in the score (i.e. whether the RNA is likely to fold), not the actual structure.

In [5], we used formal synthesis techniques to derive two systolic array families for the Nussinov recurrence: the GKT and GJQ arrays. While both arrays have the same latency of $2N-4$, their throughput differs. The block pipelining period β (reciprocal of throughput) of the GKT and GJQ arrays is respectively $4\times$ and $2\times$ lower than the latency. Unfortunately, increased throughput also results in a larger array; the GKT array uses twice as many processing elements as the GJQ array for the same input size. Table 1 summarizes these results and shows the maximum size of RNA sequences that can be folded on arrays synthesized on a Xilinx Virtex-4 LX100-12 FPGA device. All arrays run at 80 MHz; refer to [5] for details on the hardware implementation.

New to this work, we have included a third array family, a clustered version of GJQ that we call GJQC. The processing elements of the GJQ array are only 50% efficient, i.e., they are active in only one of every two clock cycles. We can increase the array's efficiency [6] by pipelining two sequences simultaneously to achieve $\beta = N-2$. Alternatively, we may cluster two processing elements into one; we call this the GJQC array. The throughput of the GJQC array is reduced by half compared to GJQ, but clustering improves space efficiency and so increases the size of the largest RNA molecule that can be folded. The three arrays are illustrated in Figure 1; the clustered processing elements in the GJQC array are shown by dashed boxes.

Another source of parallelism for array design comes from the discrete nature of the input stream. Because each input sequence can be processed independently, we can instantiate multiple copies of a small array to increase throughput for small input sizes. For example, the GJQ array can be instantiated on the target FPGA to fold sequences of length up to 81 and has $\beta = N-2$. Using the formula in Table 1, we calculate that 1680 processing elements for the GJQ array can be synthesized on the target FPGA. If we equally divide these processing elements among two GJQ arrays of the same size, we can effectively reduce the block pipelining period to $\frac{N-2}{2}$. We can estimate the maximum size of k identical GJQ arrays running in parallel by finding the positive solution to the equation $\frac{N}{2}(\frac{N}{2}+1) = \frac{1680}{k}$ (in general this equation is not quadratic). For $k=2$, we can instantiate two copies of the GJQ array of size $N = 56$. In our optimization, we consider parallel instantiations of each array

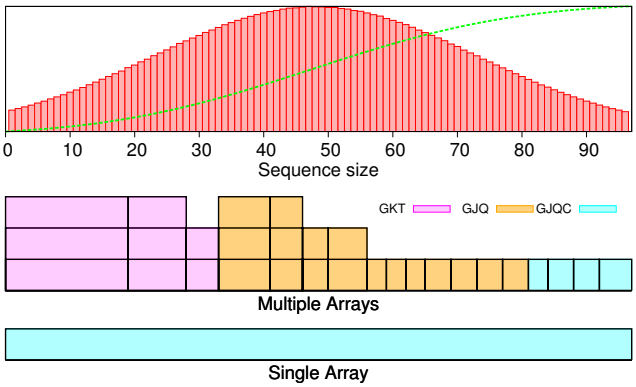


Fig. 2. Optimal selection of Nussinov arrays to fold synthetic sequences with normally distributed lengths. Top: histogram and cumulative frequency of sequence lengths. Middle: design with reconfigurations produced by our algorithm. Bottom: best single array supporting all input lengths (up to 97 bases). Size of each array instantiation is given by length of longest sequence it processes.

type as another family of arrays available for use.

4. RESULTS

To gauge the performance impact of reconfiguration, we first tested our algorithm on synthetic data. We generated sequences whose length was distributed according to geometric ($p = 0.05$), normal ($\mu = 48$, $\sigma = 25$), and Pareto (order 0.6) distributions. We generated 1 billion bases in each case, and all sequences were at most 97 bases in length. The synthetic data represents ideal inputs with lengths biased toward higher-throughput array sizes. We used the three arrays described previously. We permitted a maximum of three parallel instantiations of each array running simultaneously on the target FPGA. We assumed a reconfiguration time of 20 ms, as reported in [7].

The set of optimal arrays selected by our dynamic programming algorithm for the normally distributed sequences is shown in Figure 2 using three graphs. The top graph shows both a histogram and the cumulative frequency of all sequences. The traditional method selects a single array, in this case the slower GJQC array, at a fixed size large enough to fold all input sequences, as shown in the bottom graph. GJQC was selected because it was the *only* array type that fit on our FPGA given the sequence length requirement of 97. The reconfigured solution is shown in the middle graph. Our algorithm selects various instantiations of the GKT, GJQ, and GJQC arrays to process subsets of the sequences. The size of an array instantiation is the length of the longest sequence it executes. The number of stacked boxes indicate the number of parallel instantiations of the array synthesized on the FPGA. The number of columns denote the number of array instantiations selected by the algorithm, in this case eighteen. Our algorithm predicts a speedup of $4\times$ over the non-reconfigured solution. We obtained similarly encouraging results for the geometric ($20\times$) and Pareto ($2\times$) distributions.

4.1. Folding pyrosequencing reads

In the past decade, short (20-30 bases) noncoding RNAs have been discovered to be important regulators of eukaryotic genes [8]. One class of such RNAs is the microRNAs (miRNAs). An important biological problem is how to detect miRNA sequences in the genomic DNA from which they are copied. Currently, new sequences are scanned computationally to detect candidate miRNA precursor sequences, which are then experimentally validated [9]. An important feature of miRNA precursors is their distinctive secondary structure, which can be detected in part by looking for an unusually high number of paired bases when a short piece of DNA is treated as RNA and folded. We use the Nussinov algorithm to scan for high numbers of paired bases.

Large-scale genomic DNA sequencing today is done via *pyrosequencing*, a massively parallel sequencing technique. Pyrosequencing of a DNA sample can obtain short sequence fragments (100 bases or less) at a rate of tens of millions of bases per hour. While some pyrosequencing datasets are assembled to produce one long genomic sequence, others, especially environmental sequencing [10], sequence DNA from many organisms at once and so remain highly fragmented. We therefore investigated the impact of our methods on the rate at which the Nussinov algorithm could be applied for miRNA detection within the fragmentary DNA produced by pyrosequencing.

We folded pyrosequencing reads from 130 environmental samples[1]. The dataset contained 22.6 million reads totaling 2.7 billion bases, with an average read length of 121. For a software baseline, we wrote a C implementation of the Nussinov recurrence compiled using gcc 4.1.2 with flags *-O3 -march=nocona -fomit-frame-pointer*. The software was run on a single core of a 3 GHz Intel Core 2 Duo CPU; it folded all reads in the dataset in 18,215 seconds (excluding I/O time).

The Nussinov arrays of the previous section were written in VHDL and synthesized using the Xilinx toolchain Release 9.2.04i for a Xilinx Virtex-4 LX100-12 FPGA device connected via PCI-X to two dual-core 2.4 GHz AMD Opteron CPUs with 16 GB of memory, running 64-bit Linux. The development system was provided by Exegy Inc[2] and is based on the Mercury prototyping system [11]. For reconfiguration, our current system must load the FPGA configuration file from the host CPU's memory and so has a re-

[1] `http://scums.sdsu.edu/meta_overview.php`
[2] `http://www.exegy.com/`

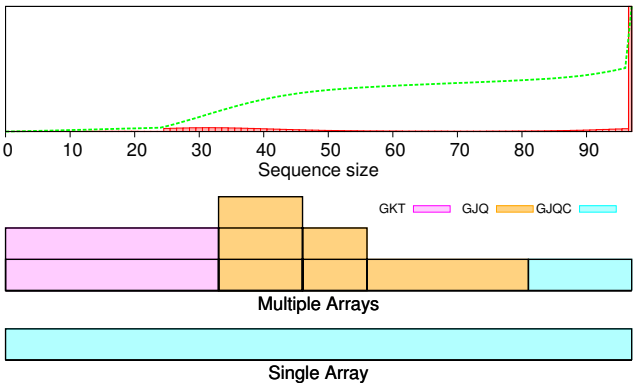


Fig. 3. Optimal selection of Nussinov arrays to fold pyrosequencing reads. Estimated speedup of the reconfigured solution over the single-array approach was 51%; direct measurement yielded 48%.

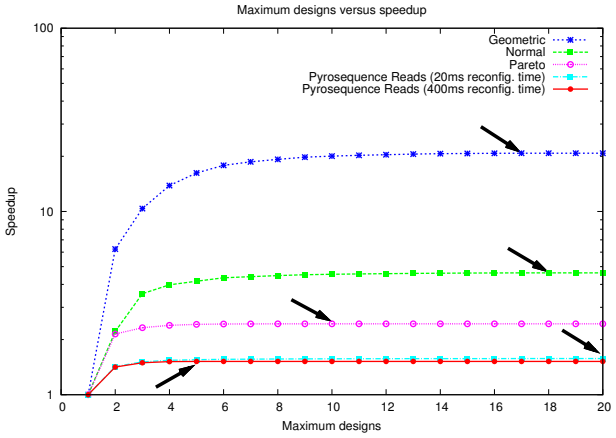


Fig. 4. Speedup of a reconfigured solution as a function of the maximum number of instantiations allowed. We may choose to limit the number of instantiations to the knee of the curve in order to reduce synthesis time. The y axis is shown in log scale.

configuration time of 400 ms, dramatically longer than that reported in [7].

Sorting and formatting the dataset for use by the hardware takes significant execution time (> 70 seconds) using a naive strategy. We may be able to accelerate this step to achieve an efficient implementation using in-memory sort for example. We have not included this cost in the execution time of the software or either hardware runs.

To process all the collected reads, we split reads greater than 97 bases into smaller chunks with an overlap of 25 bases. This increases the workload but enables us to fold all sequences while still being long enough to predict important features of the structure. Figure 3 shows the results of the experiment. Sequence lengths were heavily biased toward the largest size supported by the hardware, which requires our slowest array (GJQC); hence, we do not expect a large speedup. Moreover, our algorithm selected fewer instantiations of the arrays due to our hardware's comparatively longer reconfiguration time. As expected, the GJQC array was used for the longest reads, followed by the GJQ array. Though the GKT array can be selected for reads smaller than 50 bases, the algorithm preferred multiple units of GJQ, which has higher throughput. The GKT array was used only for sequences smaller than 34 bases. We predicted a speedup of 51% for the reconfigured over the single-array solution.

Using the single-array solution, we achieved a runtime of 107 seconds and a speedup of $170\times$ over the software. Runtime reconfiguration using our set of five arrays resulted in an execution time of 72 seconds and an improved speedup of $252\times$ over the software. Indeed, runtime reconfiguration resulted in 48% faster execution than the single latency-space optimal array, closely matching our prediction. We achieved significant speedups through runtime reconfiguration despite the input's bias toward long sequences requiring the low-throughput GJQC array (accounting for 64 of the 72 seconds of execution). Finally, we note that input bandwidth was not a limiting factor for performance.

4.2. Restricting the number of arrays

We continued our inquiry by investigating the performance of reconfiguration when restricted to a limited number of arrays. The number of arrays may be limited for two reasons. First, if optimization selects arrays that have not yet been synthesized, the cost of processing a data stream is increased by the substantial costs of synthesis for each new design used. Second, even if synthesis cost is discounted (say, because a library of designs will be reused over many datasets), the target platform may have limited storage to hold alternative array designs (e.g. bitfiles for FPGAs) and so may not support rapid reconfiguration among many designs. For example, the system described in [7] can reconfigure quickly from a set of configuration files kept in on-board non-volatile storage, but this storage holds fewer than six designs.

The algorithm of Equation 1 can easily be applied to a restricted set of pre-synthesized array designs, but this does not address the problem of selecting too many designs for the target's storage. To restrict the number of designs actually used in our reconfiguring solution, we modify Equation 1 to compute $E(i, n)$, the minimum time required to execute all inputs of length 1 to i using at most n array instantiations.

$$E(i,n) = \min \left\{ \begin{array}{l} E(i,n-1) \\ \min\limits_{1 \le a \le |A|} \min\limits_{1 \le j < i} \{ \; E(j-1,n-1) + \rho + \delta_a(j,i) \; \} \end{array} \right. \quad (3)$$

The revised recurrence adds a factor of n to the running time.

Figure 4 shows how the best estimated speedup for our real and synthetic datasets, compared to the single-array baseline, varies with the number of designs allowed for reconfiguration. The smallest number of designs that minimizes execution time (computed using Equation 1) for the experiments is shown by the arrows; solutions with more designs have the same speedup. We can achieve 90% of the full speedup available from reconfiguration using just two designs for the pyrosequencing reads and 3-8 designs for the synthetic data.

5. RELATED WORK

As far as we are aware, past work on space-time analysis only finds a single optimal array; our's is the first to take advantage of runtime reconfiguration to select multiple arrays optimal for different input subsets.

Runtime reconfiguration has been successfully used on FPGAs to improve the execution time of applications. Runtime customization responds to input to produce optimized designs, for example using constant propagation, precision variation, and branch optimization. In the case of constant propagation and precision variation, the circuit can be simplified when the input data item is known, improving performance and possibly reducing area requirements. In the case of branch optimization, a frequently executed branch case is optimized based on typical execution profiles. Specific application accelerators that use these techniques include SAT solvers, sequence alignment, and Viterbi decoding. Our work is intended to be generally applicable to any application specified as a recurrence. Runtime customization techniques can be used to build systolic arrays for the candidate list used by our dynamic programming algorithm.

Our array selection algorithm can be applied to improve the performance of existing accelerators through runtime reconfiguration. For example, a recently published accelerator for the Viterbi recurrence used in motif finding [12] expressly notes that runtime reconfiguration based on the input model length is required for acceptable performance, though the authors do not give an algorithm. In addition, it may be possible to select alternate, less resource-intensive designs for smaller input sequences when it can be guaranteed that the sequence contains only a single copy of the input model.

6. CONCLUSION

Exploiting the power of reconfiguration can result in significant performance improvements for systolic array implementation of recurrences. We have described systematic algorithms to select array designs and reconfiguration points so as to realize maximum performance. We have validated our approach empirically and have obtained speedups of $252\times$ over software and 48% over non-reconfiguring hardware for application of the Nussinov RNA folding algorithm to short nucleic acid sequences.

One important direction for further study is whether one can systematically exploit reconfiguration to improve performance based on criteria other than input size. For example, DNA sequence comparison algorithms in bioinformatics exhibit performance that is sensitive to the percentages of different DNA bases in each input. A second direction for improvement would couple our algorithms to a tool that automates formal synthesis and exploration of the space of possible array designs, so that alternative families like those we built for Nussinov can rapidly be generated for new computational problems of interest.

7. REFERENCES

[1] P. Quinton, "The systematic design of systolic arrays," in *Automata Networks in Computer Science: Theory and Applications*. Princeton University Press, 1987, pp. 229–260.

[2] D. Lavenier, P. Quinton, and S. Rajopadhye, "Advanced systolic design," in *Digital Signal Processing for Multimedia Systems*. CRC Press, 1999, pp. 657–692.

[3] J. Rosseel, F. Catthoor, and H. De Man, "An optimisation methodology for array mapping of affine recurrence equations in video and image processing," in *Application-specific Systems, Architectures and Processors*, 1994, pp. 415–426.

[4] R. Nussinov, G. Pieczenik, J. R. Griggs, and D. J. Kleitman, "Algorithms for loop matchings," *SIAM Journal on Applied Mathematics*, vol. 35, no. 1, pp. 68–82, July, 1978.

[5] A. Jacob, J. Buhler, and R. Chamberlain, "Accelerating Nussinov RNA secondary structure prediction with systolic arrays on FPGAs," in *Application-specific Systems, Architectures and Processors*, 2008, pp. 191–196.

[6] S. Y. Kung, *VLSI array processors*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1987.

[7] B. C. Brodie, R. D. Chamberlain, B. Shands, and J. White, "Dynamic reconfigurable computing." in *Military and Aerospace Programmable Logic Devices*, 2003.

[8] R. W. Carthew and E. J. Sontheimer, "Origins and mechanisms of miRNAs and siRNAs." *Cell*, vol. 136, no. 4, pp. 642–655, February 2009.

[9] E. Berezikov *et al.*, "Approaches to microRNA discovery." *Nature Genetics*, vol. 38 Suppl 1, June 2006.

[10] C. S. Riesenfeld, P. D. Schloss, and J. Handelsman, "Metagenomics: genomic analysis of microbial communities." *Annual Review of Genetics*, vol. 38, pp. 525–552, 2004.

[11] R. D. Chamberlain *et al.*, "The Mercury System: Exploiting truly fast hardware for data search," in *Proc. Int'l Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, September 2003, pp. 65–72.

[12] S. Derrien and P. Quinton, "Parallelizing HMMER for hardware acceleration on FPGAs," *Application-specific Systems, Architectures and Processors*, pp. 10–17, July 2007.