Improving Functional Density Using Run-Time Circuit Reconfiguration

Michael J. Wirthlin and Brad L. Hutchings

Abstract—The ability to provide flexibility and allow fine-grain circuit specialization make field programmable gate arrays (FPGA’s) ideal candidates for computing elements within application-specific architectures. The benefits of gate-level specialization and reconfigurability can be extended by reconfiguring circuit resources at run-time. This technique, termed run-time reconfiguration (RTR), allows the exploitation of dynamic conditions or temporal locality within application-specific problems. For several applications, this technique has been shown to reduce the hardware resources required for computation. The use of this technique on conventional FPGA’s, however, requires additional time for circuit reconfiguration. A functional density metric is introduced that balances the advantages of RTR against its associated reconfiguration costs. This metric is used to justify run-time reconfiguration against other more conventional approaches. Several run-time reconfigured applications are presented and analyzed using this approach.

Index Terms—Adaptive computing, performance tradeoffs, reconfigurable computing, reconfigurable systems, run-time reconfiguration (RTR).

I. INTRODUCTION

FIELD programmable gate arrays (FPGA) are increasingly used for computation within many digital systems. Computing machines based on FPGA’s, termed custom computing machines (CCM), are used to provide efficient and high-performance solutions for many computationally intensive problems. In many cases, a modest array of FPGA devices out perform high-end work stations and even super computers [1]. In fact, one analysis has shown that for many computations, unit-silicon resources of FPGA’s provide more computation than other general-purpose alternatives [2].

A. Configurable Computing Machines

Improvements in efficiency and performance are achieved by spécializing a computing architecture to a single computation or narrow problem domain. Architectural specialization techniques include the optimization of the processing elements, storage, and interconnect to a very limited problem domain. Limiting an architecture to a narrow application area allows hardware resources to be used more efficiently than possible with other more general-purpose architectures.

Many CCM application examples demonstrate significant improvements in performance by customizing a computing architecture to a specific and often narrow application area. For example, a genetic database search engine developed within the SPLASH-2 CCM outperforms a super computer by two orders of magnitude [3]. This architecture achieves such high levels of performance by replicating 248 special-purpose character matching units throughout the reconfigurable resource. As another example, consider the RSA cryptography circuit designed within the DEC Perle-1 CCM; it demonstrates the fastest decoding speed of any technology at the time of its development [4]. This system achieves such fast decoding rates by utilizing custom long-integer multipliers and modulus exponentiation.

In traditional systems, architectural specialization is achieved at the expense of flexibility. Computing architectures designed for one application area are invariably too inefficient or inappropriate for other application areas. The lack of flexibility of such special-purpose systems inhibits their more widespread use [5]. In FPGA-based systems, however, architectural specialization does not sacrifice flexibility. By allowing the reconfiguration of circuit resources, a single FPGA device may operate as a wide variety of application-specific computing architectures. This flexibility makes a CCM system an attractive alternative for many application-specific computations.

The wide variety of computations solved by the DEC PeRLe CCM demonstrates this flexibility. The application examples demonstrating exceptional levels of performance by this CCM include: long multiplication, RSA cryptography, data compression, string matching, Laplace equations, Newton’s mechanics, 2-D convolution, Boltzmann machine, 3-D geometry, and the discrete cosine transformation [6]. Achieving a high level of performance on such a wide variety of computations is not possible with a fixed-architecture system. A custom ASIC designed to compute all of the applications executed on PerLe, for example, would most likely offer much lower levels of performance.

B. Run-Time Reconfiguration (RTR)

The ability to reconfigure hardware resources as needed offers advantages beyond flexibility. Not only can FPGA resources be reconfigured after application execution, but these hardware resources can be reconfigured during application execution. This technique, called run-time reconfiguration or
RTR, allows circuit resources to be specialized during the computation of a problem. Run-time reconfiguration increases the efficiency of CCM systems by providing additional opportunities for specialization.

One way of improving the efficiency of a computation using RTR is to replace idle or inactive hardware with other more usable circuitry. This run-time optimization of the circuitry allows a computation to take place with fewer hardware resources. An artificial neural-network demonstrates this approach by reconfiguring FPGA resources between three temporally exclusive network training steps [7]. Removing the circuitry associated with idle network training phases provides a 500% improvement in neuron density.

RTR can also be used to partition large, special-purpose computing architectures onto limited FPGA resources. Instead of using a static architecture designed to perform all computational variations within an algorithm, several special-purpose architectural partitions can be used to solve the problem with greater efficiency. An image coding system demonstrates this approach by partitioning the coding algorithm into three sequential steps: the discrete wavelet transform, quantization/run-length encoding, and entropy coding [8]. By reconfiguring the FPGA resources with a special-purpose circuit for each of these steps, the complete algorithm can be solved using one third of the resources required by a completely static system. Other applications exploiting RTR include target-recognition [9], general-purpose image processing [11], application-specific processors [12], Huffman coding [13], stereo vision [14], and sequence comparison [15].

C. Configuration Overhead

The improvements in efficiency provided by run-time reconfiguration are not available without cost. Additional time and memory bandwidth are required to transfer circuit configuration bits from off-chip storage into the device configuration memory. In some cases, this extra time obviously mitigates the advantages of run-time specialization. For other cases, however, the tradeoff between increased efficiency and added configuration time is not so clear.

Although many run-time reconfigured systems demonstrate or suggest improvements in efficiency, few consider or justify the added cost of circuit reconfiguration. In practice, RTR will only be used if an advantage over more traditional static approaches can be shown. To date, no general quantitative method has been suggested to justify this added cost of configuration. This paper will address this issue by introducing a general method of balancing the advantages of run-time reconfigured systems against the added cost of configuration.

To the author’s knowledge, this is the first published effort to justify the use of RTR within conventional technology in an application independent manner.

Functional density, an area-time metric introduced in Section II, will form the basis of this analysis. Section III will extend the metric by including the cost of reconfiguration time. This allows the advantages of reduced area and time to be balanced against the addition of reconfiguration time. With a method of measuring functional density for both static systems and run-time reconfigured systems, Section IV will discuss the analysis approach used to compare run-time reconfigured systems against statically configured alternatives.

To reinforce the use of this analysis approach and demonstrate its utility on real systems, Section V will apply the functional density metric to several existing applications. For each application, the paper will identify the potential benefits of RTR, attempt to justify RTR against a statically configured alternative, and investigate the overhead imposed by reconfiguration.

II. FUNCTIONAL DENSITY

The primary advantage of run-time reconfiguration is the ability to provide greater architectural specialization within a circuit. As suggested earlier, more specialized circuits require less hardware and often run faster than a general-purpose alternative. A functional density metric will be used to measure the composite benefits of any given specialization technique.

Functional density \( (D) \) is defined in terms of the cost of implementing the computation in hardware. For VLSI circuits, the cost of a computation is traditionally measured as the product of circuit area and execution time, or \( C = AT \) [16], [17]. This cost measure applies to those systems in which throughput is more important than latency [18].

Functional density \( (D) \) measures the computational throughput (operations per second) of unit hardware resources and is defined as the inverse \( AT \) cost of a computation

\[
D = \frac{1}{C} = \frac{1}{AT}.
\]  

(1)

\( T \) is the total operation time of the computation and includes time required for execution, control, initialization, and data transfer. \( A \) is the total area required to implement the computation in hardware.

The functional density metric of (1) will be used to compare statically configured circuits against run-time reconfigured alternatives. Specifically, the functional density metric will be used to identify the conditions in which a run-time reconfigured circuit provides more functional density than its statically configured alternative.

The improvement in functional density of some run-time reconfigured circuit \( (D_r) \) over its conventional statically configured alternative \( (D_s) \) is computed as the normalized difference between \( D_r \) and \( D_s \) as follows:

\[
I = \frac{\Delta D}{D_s} = \frac{D_r - D_s}{D_s} = D_r - 1.
\]  

(2)

The “percentage” improvement is measured by multiplying (2) by 100.

III. CONFIGURATION TIME

The purpose of the functional density metric is to compare the overall advantage (or disadvantage) of run-time reconfiguration over other more conventional techniques. In order to make such a comparison, the functional density metric of (1) must be augmented for run-time reconfigured systems.

The major difference between run-time reconfigured systems and their static counterparts is the added cost of circuit
reconfiguration. Current devices require that circuit configuration and execution occur separately. This forces the addition of configuration time to the total operational time of a run-time reconfigured system. For run-time reconfigured systems, the total operational time $T$ includes both the total execution time ($T_e$) and configuration time ($T_c$), or $T = T_e + T_c$. Substituting $T$ into (1) provides a functional density measure that includes the added cost of configuration

$$D_r = \frac{1}{A(T_e + T_c)}, \quad (3)$$

It is clear from (3) that configuration time reduces functional density. During circuit reconfiguration, the hardware resources being configured are essentially idle and do not contribute to the computation. As reconfiguration time increases, the functional density of the system decreases. Although all RTR systems will incur some configuration overhead, those with lower configuration time will have greater functional density.

A. Configuration Ratio

Although the absolute configuration time of a system is an important parameter of RTR systems, the relative configuration time to execution time is much more informative. The configuration ratio, $f = T_c/T_e$, defines this important system parameter. The total operating time of an RTR system can be expressed in terms of this ratio as $T = T_e(1+f)$. Substituting this time into the original functional density metric provides a functional density metric in terms of $f$

$$D_r = \frac{1}{AT_e(1+f)}, \quad (4)$$

As suggested in (4), long configuration times can be tolerated if followed by correspondingly longer execution time (i.e., small $f$). Systems that operate on large data-sets or exhibit a coarse granularity of reconfiguration (configure infrequently between major computation steps) have been shown to tolerate the relatively large configuration times of today’s devices [19].

In the limit, as $f \to 0$, the overhead imposed by configuration is negligible. Such systems approach the maximum functional density available by a RTR system. This maximum value ($D_{\text{max}}$) is calculated by ignoring the effects of configuration time (i.e., $D_{\text{max}} = \lim_{f \to 0} D_r = 1/AT_e$). Using this maximum functional density, the upper bound on the improvement ($I_{\text{max}}$) over a static system can be found. This important parameter suggests the maximum benefits of RTR and is a good indication of the appropriateness of RTR for a given application

$$I_{\text{max}} = \frac{D_{\text{max}}}{D_s} - 1, \quad (5)$$

B. Reducing Configuration Overhead

With the relatively long configuration times of today’s FPGA devices, most RTR systems will be very sensitive to $f$. One way of reducing configuration overhead is to increase the execution time between configuration steps. This is often accomplished by completing more computations or processing more data samples between configuration steps. Processing multiple computations before configuration allows the cost of configuration to be amortized over more than one computation.

The time to compute $n$ computations between each configuration step is $T_n = nt_e + T_c$, where $t_e$ is the time required to complete a single computation. The time required for a single computation is simply $T_e/n$

$$T_o = t_e + T_c/n = t_e(1 + f_e/n). \quad (6)$$

As suggested by this equation, the configuration time is amortized over the many computations performed between each configuration step. The configuration ratio ($f$) is correspondingly reduced by $n$. This reduces the configuration overhead associated with functional density as follows:

$$D_{r_n} = \frac{1}{At_e(1 + f_e/n)}. \quad (7)$$

RTR systems frequently increase the amount of computation performed between each configuration step to reduce the configuration overhead.

IV. ARCHITECTURAL ANALYSIS

The functional density metric allows the composite benefits of a run-time reconfigured system to be evaluated against a general-purpose statically configured counterpart. More importantly, it is necessary to understand the conditions in which the functional density of the run-time reconfigured circuit exceeds that of the static circuit, or when $D_r \geq D_s$. Using $A_r$ and $T_r$ to represent the area and execution time of the RTR circuit, this relation can be reduced as follows:

$$\frac{1}{A_r(T_r + T_c)} \geq D_s$$

$$\frac{1}{D_s} \left( \frac{1}{A_r T_r} \right) \geq 1 + \frac{T_c}{T_r}. \quad (8)$$

This relation can be simplified by substituting $D_{\text{max}}$ as follows:

$$\frac{D_{\text{max}}}{D_s} - 1 \geq \frac{T_c}{T_r}. \quad (9)$$

The left-hand side of the relation is the maximum improvement possible with RTR [see (5)]. Substituting $I_{\text{max}}$ into (9) produces the relation

$$I_{\text{max}} \geq f. \quad (10)$$

Equation (10) is an important result that describes the maximum allowable configuration ratio ($f$) of a given RTR system. This relation states that in order for a run-time reconfigured system to provide more functional density than a static alternative, the configuration ratio must be less than the maximum potential improvement ($I_{\text{max}}$) of the run-time reconfigured system.

As an example, consider a static circuit requiring circuit area $A$ and time $T$ to complete a given operation. Suppose a run-time reconfigured circuit performs the same computation in half the area ($A/2$) and two-thirds the time ($2T/3$). As summarized in Table I, the maximum improvement of the RTR circuit is 2 (200%). Using (10), this result suggests that the RTR circuit
TABLE I
SAMPLE CIRCUIT PARAMETERS

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>T</th>
<th>D_{max}</th>
<th>I_{max}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>a</td>
<td>t</td>
<td>\frac{A}{T}</td>
<td>0</td>
</tr>
<tr>
<td>RTR</td>
<td>\frac{a}{2}</td>
<td>\frac{t}{2}</td>
<td>\frac{A}{2T}</td>
<td>2</td>
</tr>
</tbody>
</table>

![Diagram: Feed-Forward → Back Propagation → Update]

Fig. 1. Three stages of the back propagation training algorithm.

will provide greater functional density than the static circuit so long as \( f < 2 \), or when the configuration time is less than twice the execution time.

Intuitively, this result suggests that the greater the potential advantage of a run-time reconfigured circuit, the less stringent the configuration overhead limitations. Those run-time reconfigured circuits providing substantial improvements in functional density are less sensitive to configuration time than those providing only marginal improvements. Run-time specialization techniques that provide significant improvements in efficiency justify the use of RTR in light of the poor configuration performance of today’s devices.

This result also suggests that systems providing only modest improvements with RTR can still be justified so long as the configuration ratio is low. Systems executing for long periods of time relative to the configuration time can justify RTR under most circumstances.

V. APPLICATION ANALYSIS

This section will justify the configuration time of several RTR applications by applying the analysis approach introduced above. The applications reviewed in this section include an artificial neural-network, a template matching circuit, and a sequence comparison circuit.

A. Run-Time Reconfigured Neural Network (RRANN)

The run-time reconfigured neural network (RRANN) was one of the first applications to demonstrate improved performance and circuit density using RTR [19]. The RRANN architecture implements the well-known back propagation learning algorithm using FPGA hardware resources. As shown in Fig. 1, this algorithm is partitioned into three sequentially executing stages: feed-forward, back propagation, and update. Since only one stage is executing at a time, hardware can be conserved by reconfiguring the FPGA resources with a circuit executing the current algorithm stage.

As suggested by Fig. 1, the complete algorithm is executed by first configuring the FPGA resources with a specialized feed-forward neural processor and completing its computation. Next, the back-propagation circuit is configured and executed. Finally, the update circuit is configured and executed to complete a single iteration of the algorithm. This process of reconfiguration and execution continues until the training algorithm converges. Since this run-time reconfigured approach requires hardware for only one stage at a time, the entire algorithm can be executed with fewer resources than possible within a static, nonconfigured architecture.

To determine the amount of hardware saved by run-time reconfiguration, both a run-time reconfigured and static version of this algorithm were mapped to Xilinx 3090 FPGA’s [20]. The static approach, which must implement all three algorithm stages within a single circuit, provides only one neuron per FPGA. When run-time reconfiguration is used, a single FPGA provides six neurons for a 500% increase in neuron density. The ability to replace the idle control and arithmetic associated with inactive algorithm stages significantly improves the efficiency of the hardware.

To calculate the functional density of each system, the total operational time and area must be determined. The execution time and area of a neural-network executing on RRANN depends on the number of neurons within the network. As more neurons are added to the network, both the execution time and area grow linearly. Table II summarizes the circuit parameters of a 60 neuron, four layer network (i.e., 10 800 neuron connections) for both the static and run-time reconfigured system.

Specializing the neural processors within the RTR circuit provides a significant area advantage over the statically configured circuit. When reconfiguration time is ignored, the RTR circuit provides 435% more functional density than the static alternative (i.e., \( I_{max} = 4.35 \)). This result suggests that the run-time reconfigured system can tolerate a configuration time of over four times the execution time and still provide greater functional density than the static alternative.

To calculate the actual functional density of the RTR system, the configuration time must be known. Assuming the system configures at the maximum device reconfiguration rate, the three reconfiguration steps consume 19.8 ms. With a relatively short execution time of 1.93 ms, the configuration ratio, \( f \), is 9.9. Because this ratio is larger than \( I_{max} \), the RTR version provides less functional density than the statically configured circuit. As shown in Table II, the excessive reconfiguration time of the RTR system reduces the functional density by 52%.

Although the functional density for the run-time reconfigured network is lower than its static alternative, larger networks with longer execution time will overcome this configuration overhead. The longer execution times associated with larger networks reduce \( f \) and improves the relative performance of the system. To demonstrate the effects of increasing the network size, the functional density of the RTR circuit is plotted in Fig. 2 as a function of network size (neurons per layer). It has been shown that a run-time reconfigured network with 138 neurons executes long enough

TABLE II
CIRCUIT PARAMETERS FOR A NEURON NETWORK WITH 10980 CONNECTIONS

| Connections (n) | Static | RTR | RTR
cir

<table>
<thead>
<tr>
<th>Connections (n)</th>
<th>Static</th>
<th>RTR</th>
<th>Maximum</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>A (CLBs)</td>
<td>1260</td>
<td>2702</td>
<td>1260</td>
<td>2702</td>
</tr>
<tr>
<td>T_e (ms)</td>
<td>1.93</td>
<td>1.93</td>
<td>1.93</td>
<td>1.93</td>
</tr>
<tr>
<td>T_c (ms)</td>
<td>304</td>
<td>2079</td>
<td>19.8</td>
<td>19.8</td>
</tr>
<tr>
<td>( f )</td>
<td>N/A</td>
<td>1.99</td>
<td>1.99</td>
<td>1.99</td>
</tr>
<tr>
<td>( D )</td>
<td>0.38</td>
<td>189</td>
<td>0.38</td>
<td>189</td>
</tr>
<tr>
<td>( I )</td>
<td>4.28</td>
<td>-0.52</td>
<td>-0.52</td>
<td>-0.52</td>
</tr>
</tbody>
</table>
to overcome the configuration limitations and provide the same functional density as the static alternative [19]. Further increases in the network size increase the advantage of the RTR system over its static counterpart.

B. RRANN-II

Because of the long configuration times of the RRANN architecture and its extreme sensitivity to configuration time, a follow up architecture called RRANN-II was developed [21]. The basic structure of the RRANN architecture was re-designed to exploit the partial reconfigurability of the National Semiconductor CLAy FPGA [22]. Partial reconfiguration is used to reduce reconfiguration time by reconfiguring only a subset of the device resources. If only a small amount of hardware requires modification at each reconfiguration step, the configuration time can be substantially reduced.

As with RRANN, the RRANN-II project involved the design of both a general-purpose neural processor and a set of special-purpose processors to execute the back-propagation algorithm. RRANN-II differs from RRANN by exploiting the similarities between each of the neural processors. The three neural processors are designed to share as much hardware as possible to reduce the configuration data at each reconfiguration step [23]. The more circuitry shared by each processor, the less configuration data needed to convert one neural processor into the next neural processor.

Designing the partially reconfigurable RRANN-II system required considerable hand-layout and manual design. Common circuit functions were identified and manually mapped to the hardware. Functions unique to each stage were identified and designed to operate correctly with those static subcircuits remaining on the device during configuration. Modifications to the processors during configuration include the modification of arithmetic operator precision, fixed constants, and global control.

The hand-layout of this dynamic circuit produced a very efficient and highly optimized design. All of the circuits operate at 20 MHz and each achieve a very high utilization of the FPGA resources. The general-purpose neural processor, designed to compute the entire algorithm without configuration, contained three neurons within a single FPGA. The optimization of each special-purpose neural processor increased the neuron density from three to nine neurons per FPGA.

The circuit parameters of a 60 neuron network are listed in Table III for both the static and run-time systems. As shown, the special-purpose RTR circuit completes the algorithm with one-third the hardware needed by the static system. This reduction in hardware provides a maximum improvement in functional density of 168% over the static alternative.

To calculate the actual functional density of the RTR system, the configuration time must be known. The three reconfiguration steps required by the RRANN-II architecture consume 2.42 ms when global reconfiguration is used. Although this justifies the system with a configuration ratio of 1.62, the

---

**TABLE III**

<table>
<thead>
<tr>
<th></th>
<th>Static</th>
<th>Maximum</th>
<th>Global</th>
<th>Partial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connections</td>
<td>10980</td>
<td>10980</td>
<td>10980</td>
<td>10980</td>
</tr>
<tr>
<td>$A$ (cells)</td>
<td>50764</td>
<td>18904</td>
<td>18904</td>
<td>18904</td>
</tr>
<tr>
<td>$T_c$ (ms)</td>
<td>1.49</td>
<td>1.49</td>
<td>1.49</td>
<td>1.49</td>
</tr>
<tr>
<td>$T_e$ (ms)</td>
<td>N/A</td>
<td>0</td>
<td>2.42</td>
<td>1.16</td>
</tr>
<tr>
<td>$f$</td>
<td>N/A</td>
<td>0</td>
<td>1.62</td>
<td>.78</td>
</tr>
<tr>
<td>$D$</td>
<td>146</td>
<td>391</td>
<td>148</td>
<td>219</td>
</tr>
<tr>
<td>$I$</td>
<td>0</td>
<td>1.68</td>
<td>.014</td>
<td>.50</td>
</tr>
</tbody>
</table>
overall improvement of run-time reconfiguration is only 1.4%. By reconfiguring only the changes in the circuit using partial reconfiguration, the configuration time reduces to 1.16 ms. This reduction in configuration time has a significant impact on overall functional density. As summarized in Table III, the configuration ratio is reduced to 0.78 and the improvement in functional density increases to 50%.

As suggested earlier, functional density is dependent on the execution time and increases with the network size. The functional density of both the global and partial configured systems is plotted against the static system in Fig. 3. It is clear from Fig. 3 that the improved configuration time of the partially configured system provides greater functional density than the globally configured system. In addition, the shorter configuration time reduces the “break-even” point in which the functional density of the run-time reconfigured system matches the functional density of the static system. Reducing the configuration time by a factor of two within RRANN-II reduces the break-even point from a network with 58 neurons to one with only 28 neurons.

C. Template Matching

Template matching is a computationally intensive processing step within many object recognition systems. This operation involves the correlation of an input image against a large database of templates. Completing this computation requires significant hardware resources and time. A special-purpose template matching circuit was designed within FPGA’s to demonstrate improvements in functional density using run-time reconfiguration [10]. Unlike the two RRANN systems which reconfigure between temporally exclusive circuits, this system reconfigures hard-wired constants within an array of special-purpose correlation processors.

A deeply-pipelined circuit is created by tiling conditional bit-serial adders into a 2-D array as seen in Fig. 4. Each conditional adder within the array corresponds to a pixel of the template image. These conditional adders operate in parallel and each correlate incoming pixels against its respective template pixel value.

Before each correlation computation, the conditional adders are programmed with the new template pixel values. Two methods of programming these adders were investigated. The first method involves the loading of template bits within a general-purpose conditional adder. This general-purpose circuit remains static throughout the computation and is capable of performing a correlation operation against any template image.

The second method involves the run-time reconfiguration of special-purpose correlation processors. Instead of providing
TABLE IV  
CIRCUIT PARAMETERS FOR 16 x 16 TEMPLATE MATCHING CIRCUIT

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Static (cells)</th>
<th>RTR Maximum</th>
<th>Global</th>
<th>Partial</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2304</td>
<td>1536</td>
<td>1536</td>
<td>1536</td>
</tr>
<tr>
<td>T_{sec} (ms)</td>
<td>2.17</td>
<td>2.05</td>
<td>2.05</td>
<td>2.05</td>
</tr>
<tr>
<td>T_{x} (μs)</td>
<td>N/A</td>
<td>0</td>
<td>808</td>
<td>51</td>
</tr>
<tr>
<td>f</td>
<td>N/A</td>
<td>0</td>
<td>.394</td>
<td>.025</td>
</tr>
<tr>
<td>D</td>
<td>2554</td>
<td>4055</td>
<td>2909</td>
<td>3995</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>.588</td>
<td>.139</td>
<td>.564</td>
</tr>
</tbody>
</table>

Both a static and run-time reconfigured correlation circuit were mapped to the CLAy FPGA for 16 x 16 templates. Specializing the run-time reconfigured circuit to a single template image provides two advantages: first, the correlation processing elements consume 50% fewer resources, and second, the correlation processors operate at a faster clock rate. As summarized in Table IV, the advantages of run-time circuit specialization provide a maximum 59% improvement in functional density over the more conventional, nonconfigured approach.

Although the maximum advantage of this system is much lower than that of RRANN, the reconfiguration overhead is modest and, for large enough images, provides sufficient execution time to break-even. As shown in Table IV, a globally configured FPGA requiring 808 μs for reconfiguration has a configuration ratio of 0.394. Since the configuration ratio is smaller than the maximum improvement, RTR is justified. With larger input images that require more execution time, the configuration overhead decreases and provides even greater improvements in functional density.

The benefits of RTR can be extended by exploiting partial configuration. Because only a subset of the special-purpose correlation PE’s change between each custom template circuit, the amount of hardware requiring configuration is significantly reduced. As shown in Table IV, partial reconfiguration reduces the configuration time by a factor of 16. This reduces the configuration ratio to 0.025 or to a point when changes in configuration time have limited impact on overall functional density.

Like all RTR systems, the functional density of the template matching circuit is dependent on the problem size. The functional density of both the global and partial configured systems is plotted against the image size in Fig. 5. The break-even point for the globally configured system is 7820 pixels (89 x 89 image) and 304 (18 x 18 image) for the partially configured system. As shown in Fig. 5, the reduction in configuration time using partial configuration allows the functional density to approach the upper-bound functional density of the system.

**D. DNA Sequence Matching**

The matching of DNA sequences using the popular edit-distance algorithm was one of the first CCM applications to demonstrate super-computer performance [3], [24]. The stan-
Standard approach for this algorithm, however, requires enough hardware for a systolic array representing the entire sequence. One approach investigates the use of run-time reconfiguration as a method of partitioning and scheduling a large edit-distance problem onto a limited resource platform [15].

This system investigates the use of RTR by comparing a statically configured systolic array based on general-purpose PE’s against a run-time reconfigured systolic array based on special-purpose constant propagated PE’s. Both systolic arrays are mapped to the same amount of hardware resources.

Unlike the other applications discussed above, run-time reconfiguration within this edit-distance example is not used to exploit the temporal nature of the problem. Instead, run-time reconfiguration is used to allow a CCM with insufficient resources to solve this problem using specialized circuitry. Two types of PE’s were mapped to the CLAy FPGA: a special-purpose PE designed to match only one character within the alphabet and a general-purpose PE designed to match any character. Propagating the constant within the special-purpose PE reduced the hardware requirements from 78 cells down to 45 cells. A secondary benefit is the slightly faster operating speed of the more efficient special-purpose PE.

Two systolic arrays were created—one based on the larger general-purpose PE and the other based on the more efficient special-purpose PE. When mapped to a four FPGA system, the general-purpose systolic array contained 160 PE’s while the special-purpose systolic array contained 276 PE’s. The additional PE’s within the special-purpose systolic array allows the computation to complete with fewer partitions. A 5000 character source sequence executed on the general-purpose PE’s requires 32 partitions while the special-purpose PE’s require only 19 partitions.

<table>
<thead>
<tr>
<th>n</th>
<th>Static (cells)</th>
<th>RTR (cells)</th>
<th>Maximum (cells)</th>
<th>Actual (cells)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>12544</td>
<td>12544</td>
<td>12544</td>
<td></td>
</tr>
<tr>
<td>T_{exec} (ms)</td>
<td>2.74</td>
<td>1.59</td>
<td>1.59</td>
<td></td>
</tr>
<tr>
<td>T_{c} (ms)</td>
<td>N/A</td>
<td>0</td>
<td>4.20</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>N/A</td>
<td>0</td>
<td>2.64</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>$705 \times 10^3$</td>
<td>$1260 \times 10^3$</td>
<td>$345 \times 10^3$</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>0</td>
<td>.79</td>
<td>-0.51</td>
<td></td>
</tr>
</tbody>
</table>

The area and time required to complete a 5000 character target sequence computation is shown in Table V. The more efficient constant propagated PE provides a maximum improvement in functional density of 79%. However, the overhead imposed by reconfiguration is greater than the maximum improvement. Even though partial reconfiguration is used within this application, the 19 reconfiguration steps required by the circuit partitions consumes over twice the time required by circuit execution. This excessive reconfiguration time reduces functional density by 51%.

Like the other applications exploiting RTR, the functional density of this system will improve with longer execution times. Specifically, longer target sequences will reduce the configuration ratio by amortizing the cost of configuration over more character matching operations. The functional density of this sequence comparison circuit is plotted in Fig. 6 as a function of target sequence length. Although this run-time reconfigured system does not provide more functional density than the static alternative at 5000 target characters, a target sequence of 17700 or more will justify RTR.
**TABLE VI**

<table>
<thead>
<tr>
<th>Application</th>
<th>$I_{\text{max}}$</th>
<th>$f$</th>
<th>$I_{\text{actual}}$</th>
<th>Break-Even</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRANN$_{60}$</td>
<td>4.28</td>
<td>9.9</td>
<td>$-0.52$</td>
<td>138 neurons</td>
</tr>
<tr>
<td>RRANN-II$_{\text{global}}$</td>
<td>1.68</td>
<td>1.62</td>
<td>0.014</td>
<td>58 neurons</td>
</tr>
<tr>
<td>RRANN-II$_{\text{partial}}$</td>
<td>1.68</td>
<td>0.78</td>
<td>0.50</td>
<td>28 neurons</td>
</tr>
<tr>
<td>Template$_{\text{global}}$</td>
<td>0.588</td>
<td>0.394</td>
<td>0.139</td>
<td>89 x 89 image</td>
</tr>
<tr>
<td>Template$_{\text{partial}}$</td>
<td>0.588</td>
<td>0.025</td>
<td>0.564</td>
<td>18 x 18 image</td>
</tr>
<tr>
<td>Edit$_{5000}$</td>
<td>0.79</td>
<td>2.64</td>
<td>0.51</td>
<td>17,700 chars</td>
</tr>
</tbody>
</table>

### E. Application Summary

Although the run-time reconfigured applications described above do not represent a particularly wide application range, they do provide insight into both the potential benefits and problems facing run-time systems. Table VI summarizes the results of these applications by including the upper-bound benefits of RTR ($I_{\text{max}}$), the configuration overhead ($f$), the actual improvement ($I_{\text{actual}}$), and the break-even point of each application described above.

It is clear from Table VI that each application offers some potential improvement in overall functional density when configuration time is ignored ($I_{\text{max}}$ is positive). Some applications, such as RRANN, offer significant potential advantages. An improvement of almost 400% in functional density is a significant result that should not be ignored. Even a modest improvement of 58% for the correlation system is significant in large, cost-sensitive systems.

However, the advantages associated with RTR must be balanced against the corresponding configuration overhead. The relatively slow configuration times of today’s devices force a high configuration overhead for most of these applications. However, with a sufficiently large problem size, each of these applications can justify the use of RTR using devices available today.

Almost all of the applications listed in Table VI are sensitive to configuration time. Improvements to configuration time will significantly increase the functional density of each of these systems. In addition, reducing the configuration time allows the justification of RTR with smaller, more reasonable sized problems. Reducing partial configuration within the RRANN-II system, for example, reduced the break-even point for RTR from 58 to 28 neurons.

In summary, RTR provides advantages to digital systems that are available with today’s technology. Improvements in configuration time with future devices will maximize the benefits of RTR and justify its use within more computing environments.

### VI. Conclusions

This paper demonstrates how RTR improves functional density within custom computing applications. Specifically, several applications demonstrate a reduction in hardware and execution time by specializing circuit resources at run-time. Some applications exploit the temporal locality of a circuit by removing idle hardware from the FPGA resources. Other applications preserve the special-purpose nature of an architecture within a limited resource platform by reconfiguring special-purpose circuit partitions at run-time.

Although RTR may reduce the hardware requirements or execution time of a CCM application, additional time is required for circuit reconfiguration. In some cases, the added reconfiguration time eliminates any advantage of RTR. RTR should not be used unless some composite advantage can be shown over a more conventional approach. Without justification, RTR systems could unknowingly be designed to provide reduced levels of performance and require significantly more effort to develop.

This paper presents a method of comparing the relative advantages of RTR against conventional alternatives. Specifically, a functional density metric was introduced that balances the area and time improvements of an RTR system against the added cost of configuration. Although the use of this analysis technique requires the circuit size and speed of both the RTR circuit and the static alternative, the analysis approach can be used as a quick estimate of the suitability of RTR. Such a quick analysis would proceed as follows:

1. estimate the area and time provided by RTR;
2. compute $I_{\text{max}}$;
3. estimate the configuration time, $T_c$;
4. compute $f$ using $T_c$ and $T_e$.

If the configuration ratio ($f$) is less than the estimated maximum improvement of the RTR system, RTR will most likely be justified. Otherwise, RTR is not justified and should probably not be used within the application.

As demonstrated in this paper, the tradeoff between increased efficiency and configuration time of RTR systems can be balanced by using the functional density metric. Results from this paper show that the use of RTR can be justified using the relatively slow configuration times of today’s devices. This suggests that as configuration times of FPGA devices improve and innovative configuration techniques become available, RTR will increasingly become a more important design technique within configurable computing machines.

### ACKNOWLEDGMENT

The authors would like to acknowledge the work of several people within the Brigham Young University Configurable Computing Laboratory, Provo, UT, including J. Eldredge, J. Hadley, and J. Diether.

### REFERENCES


Michael J. Wirthlin received the B.S. degree in electrical and computer engineering in 1992 from Brigham Young University (BYU), Provo, UT. He is currently completing the Ph.D. degree at BYU, with a research emphasis in run-time reconfigured systems.

In 1997, he joined National Semiconductor’s Architecture Laboratory as a Staff Research Engineer. His research interests include reconfigurable computing, embedded systems, and application-specific processor architectures.

Brad L. Hutchings received the B.S., M.S., and Ph.D. degrees in computer science from the University of Utah, Salt Lake City, in 1984, 1987, and 1992, respectively.

He is currently an Associate Professor in the Department of Electrical and Computer Engineering at Brigham Young University, (BYU), Provo, UT. In 1993, he established and is Head of the Laboratory for Configurable Computing at BYU. His research interests are custom computing, embedded systems, FPGA architectures, CAD, and VLSI.