String Matching on Multicontext FPGAs using Self-Reconfiguration

Reetinder P. S. Sidhu
sidhu@halcyon.usc.edu

Alessandro Mei
amei@halcyon.usc.edu

Viktor K. Prasanna
prasanna@ganges.usc.edu

Department of EE-Systems, University of Southern California,
Los Angeles CA 90089

Abstract

FPGAs can perform better than ASICs if the logic mapped onto them is optimized for each problem instance. Unfortunately, this advantage is often canceled by the long time needed by CAD tools to generate problem instance dependent logic and the time required to configure the FPGAs.

In this paper, a novel approach for runtime mapping is proposed that utilizes self-reconfigurability of multicontext FPGAs to achieve very high speedups over existing approaches. The key idea is to design and map logic onto a multicontext FPGA that in turn maps problem instance dependent logic onto other contexts of the same FPGA. As a result, CAD tools need to be used just once for each problem and not once for every problem instance as is usually done.

To demonstrate the feasibility of our approach, a detailed implementation of the KMP string matching algorithm is presented which involves runtime construction of a finite state machine. We implement the KMP algorithm on a conventional FPGA (Xilinx XC 6216) and use it to obtain accurate estimates of performance on a multicontext device. Speedups in mapping time of $\approx 10^6$ over CAD tools and more than 1800 over a program written specifically for FSM generation were obtained. Significant speedups were obtained in overall execution time as well, including a speedup ranging from 3 to 16 times over a software implementation of the KMP algorithm running on a Sun Ultra 1 Model 140 workstation.

1 Introduction

By exploiting the reconfigurability of FPGAs, significant performance improvements have been obtained over other modes of computation for several applications. However, there are two serious problems that prevent FPGAs from being utilized to their fullest potential:

- long mapping time;
- long reconfiguration time.

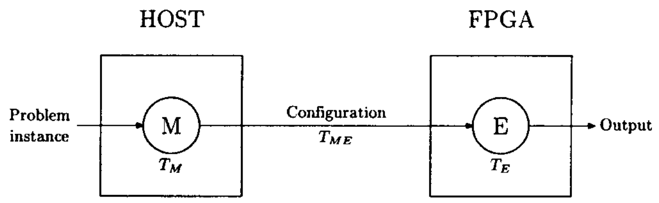
Mapping time refers to the time to compile, place and route the logic to be used on the FPGA; reconfiguration time is the time needed to load the configuration data into the FPGA. Mapping computation onto FPGAs is typically done using CAD tools. It is a time consuming process and can take anywhere from a few minutes to a few days. In order to take advantage of the reconfigurability of FPGAs, a new mapping should be created for every problem instance. As a result, the mapping time becomes very critical and it is extremely important to reduce it.

The time required to completely reconfigure an FPGA is typically about 1 ms. Since reconfiguration time needs to be amortized over computation time, frequent run-time reconfiguration is not possible. It should be noted that even partial reconfiguration is not a complete solution to this problem. Since reconfigurability is the key advantage of FPGAs over other modes of computation, reduction of reconfiguration time is very important.

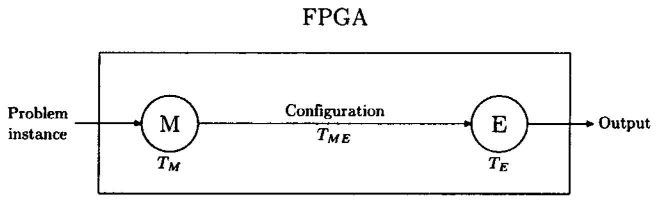
In this paper we show how to significantly reduce both mapping and reconfiguration times through *self-reconfiguration*. By self-reconfiguration we mean that not only does the FPGA load the configuration information itself, but also that it generates the configuration. We show how self-reconfiguration can be efficiently implemented using *multicontext FPGAs* (FPGAs having more than one configuration context on-chip). Although, such devices have been primarily designed to reduce reconfiguration times, we show how they can be used for self-reconfiguration as well.

Self-reconfiguration reduces mapping time because all logic to be configured is generated by previously configured logic. The mapping logic is designed to generate highly specific mapped logic and is therefore much simpler than general purpose CAD tools. Also, it executes on an FPGA. For these reasons, the mapping time is considerably lesser than mapping via software running on a host machine. Self-reconfiguration reduces reconfiguration time because configurations are generated and stored on-chip which is much faster than loading it from an external source. Also, multicontext FPGAs can very quickly switch between stored configurations. As a result of these improvements, self-reconfiguration allows runtime generation of logic and its use to be interwoven in ways that would be impractical otherwise. We demonstrate this power and flexibility by a string matching algorithm implementation. Even though our early results are very promising, a deep investigation is needed to fully understand what can be achieved by using this approach

This work was supported by the DARPA Adaptive Computing Systems Program under contract DABT63-96-C-0049 monitored by Fort Hauchuca. Alessandro Mei is with the Department of Mathematics of the University of Trento, Italy. This work has been done during his stay at the USC.



(a) Structure of a typical application for reconfigurable devices.



(b) Mapping and execution on a self-reconfigurable device.

Figure 1:

to reconfigurable computing, and it seems to be a challenging and wide open research area.

In the first part of the paper, we introduce self-reconfiguration and its advantages (Section 2) and how it is achieved using multicontext FPGAs (Section 3). In the second part, we introduce the KMP algorithm (Section 4) and present detailed implementation description and performance analysis (Section 5). The conclusion is in Section 6.

2 Introduction to Self-Reconfiguration

2.1 Problem instance dependence and hardware compiling

The effectiveness of reconfigurable computing is better exploited by building hardware solutions for each single instance of a given problem. That essentially means that a good application for reconfigurable devices should read the input of the problem (the *instance*), compute *instance dependent* logic, i.e. logic optimized for that particular instance, and load it into a reconfigurable device to solve the problem. Applications which produce *instance independent* logic to be loaded onto a reconfigurable device are simply not exploiting the power of reconfiguration. In that case the logic mapped is static, depends only on the algorithm used, and is not conceptually different from ASIC approach.

A large class of applications developed for reconfigurable devices can thus be modeled in the following way (see Figure 1(a)). A process M reads the input problem instance. Depending on the instance a logic E, ready to be loaded, is computed such that it is optimized to solve that single problem instance. This process is usually executed by the host computer. Let T_M denote the time to perform this.

After reconfiguring the device, E is executed. Let T_{ME} denote the time to reconfigure. The time T_E required for the execution includes the time needed for reading the inputs from the memory and producing the output. Therefore, the time required by the execution of a single iteration of the computation described above is $T_I = T_M + T_{ME} + T_E$.

The actual execution time on the reconfigurable device is T_E. It is often very low compared with the time needed to solve the same problem by using a software solution, due to hardware efficiency. This has been used to claim that very high speed-up can be achieved by reconfigurable computing. It should be clear that this is not a fair way to compare the performance of reconfigurable systems. However, this is frequently done. We believe that all times involved in computing the solution to a given instance of a problem should be taken into account.

The time T_M required by M varies considerably among applications, and usually ranges from a few minutes to several hours, and, for some particularly complex logic, even days! The reason lies in the fact that usually CAD tools are used. CAD tools are very powerful and general applications, but their flexibility is obtained at the expense of large computing time. In fact, what is actually done, is to *compile*, using a CAD tool, each single instance to derive the logic E to be used to solve the problem.

The fact that T_M is usually large limits the effectiveness of reconfigurable computing. In [1], for example, a shortest paths algorithm is implemented. In that case, the execution time T_E for a problem instance is order of microseconds, while the mapping time T_M is order of hours. Also in [15], the proposed algorithm for SAT usually takes hours to be mapped. SAT is NP-complete, and thus a good candidate to make T_M affordable since T_E is usually very high. In spite of that, when mapping time is taken into account, only modest speedups are obtained. The time T_{ME} depends on to the device used. For FPGAs, for example, it is typically around 1 millisecond, and it is related to the bottle-neck represented by the bus connecting the host computer to the FPGA board. Even if the reconfiguration time T_{ME} is often much lower than the mapping time, it can still be unacceptable for most real-time applications.

Some efforts have been made to overcome these problems. For example, in [7] CAD Tools are used only once to compute a generic skeleton logic. Then, for each problem instance, some limited changes are made by the host computer to build an instance dependent circuit and load it into the FPGA board. This is an interesting technique that can be useful to lower the mapping time T_M, but cannot avoid the bottle-neck represented by the bus connecting the host computer to the FPGA board. In [7], $T_M + T_{ME}$ is around 3 seconds, still too high for a large class of applications.

This paper presents a novel approach to reconfigurable computing which is able to dramatically reduce T_M and T_{ME}. Since M has to be speeded up, what we propose is to let fast reconfigurable devices to be able to execute it (see Figure 1(b)). In case a single FPGA is being used, this essentially means that the FPGA should be able to read from a memory the problem instance, configure itself, or a part of it, and execute the logic built by it to solve the problem instance. Evidently, in this case M is itself a logic circuit, and cannot be as complex and general as CAD tools are.

Letting FPGA system execute both M and E on the same chip gives the clear advantage that CAD tools are used only *once*, in spite of classical solutions where they are needed for computing a logic for each problem instance. This is possible since the adaptations, needed to customize the circuit to the requirements of the actual input, are performed dynamically by the FPGA itself, taking advantage of hardware efficiency.

Another central point is that the bus connecting the FPGA system to the host computer is now only used to input the problem instance, since the reconfiguration data are generated locally. In this way, the bottle-neck problem is also handled.

These ideas are shown to be realistic and effective by presenting a novel implementation of a string matching algorithm. String matching is one of the most important problems in Computer Science, both from a theoretical and from a practical point of view. In Section 5, a detailed implementation is described, and $T_M + T_{ME}$ is shown to be around 28μs, for patterns 16 character long, achieving a dramatic speed-up over classical FPGA computations.

2.2 Self-reconfiguration

The main feature needed by an FPGA device to fulfill the requirements needed by the technique shown in the previous section is self-reconfigurability. This concept has been mentioned few times in the literature on reconfigurable architectures in the last few years [6][5]. In spite of that, to the best of our knowledge not only no one devised an application that actually used that feature, but no one even investigated to understand how self-reconfiguration could be used to achieve superior performance.

The concept of self-reconfiguration was earliest presented in [6], where a small amount of static logic is added to a reconfigurable device based on an FPGA in order to build a self-reconfiguring processor. Being an architecture oriented work, no application of this concept is shown.

The recent Xilinx XC6200 is also a self-reconfiguring device, and this ability has been used in [5] to define an abstract model of virtual circuitry, the Flexible URISC. This model still has a self-configuring capability, even though it is not used by the simple example presented in [5].

All these devices are potentially capable of self-reconfiguring, and are thus able of implementing the ideas presented in this paper. However, moving the process of building the reconfigurable logic into the device itself requires a larger amount of configuration memory in the device with respect to traditional approaches. For this reason, multi-context FPGAs seem to answer better to these requirements, since they have been shown to be able to store a large amount of different contexts (see [12], for example, where a self-reconfiguring 256-context FPGA is presented).

3 Multicontext FPGAs

As described in the Introduction, the time required to reconfigure a traditional FPGA is very high. To reduce the reconfiguration time, a device having more than one configuration context was proposed in [4]. Several such *multicontext* FPGAs have been recently proposed [13][11][14] [8][3].

These devices have on-chip RAM to store a number of configuration contexts, varying from 8 to 256. At any given time, one context governs the logic functionality and is referred to as the *active* context. Switching contexts takes 5–100 ns. This is several orders of magnitude faster than the time required to reconfigure a conventional FPGA ($\approx$1 ms).

For self-reconfiguration to be possible, the following two additional features are required of multicontext FPGAs:

- The active context should be able to initiate a context switch—no external intervention should be necessary.
- The active context should be able to read and write the configuration memory corresponding to other contexts.

The multicontext FPGAs described in [13][11][14] satisfy the above requirements and hence are capable of self-reconfiguration[1].

4 The KMP Algorithm for String Matching

The String Matching problem consists of finding all occurrences of a *pattern* P, of length m, in a *text* T, of length n, $m \leq n$, with P and T being strings over a finite alphabet Σ.

Besides being a fundamental problem in Computer Science from a theoretical point of view, String Matching is of paramount practical relevance. Important examples of its application can be easily found in the areas of Text Processing, Pattern Recognition, Image Understanding, Databases, and Biology, to name a few. In particular, applications of String Matching in Biology are of utmost importance, since finding patterns of DNA inside longer sequences is becoming central in the analysis of human genome.

A naive algorithm that can be used to solve String Matching consists in trying to match the pattern at each position in the text by a "brute force" search. Meaning that for each position i in the text, we perform a do-loop operation to check whether all m characters of P match m characters of T starting from position i. If we found a mismatch, say at position $i + h$, we can stop this search and try at position $i + 1$. This leads to a simple, but slow, algorithm, whose time complexity is $O(mn)$, in the worst case, and thus quite far from optimality.

It can be remarked, however, that if we find a mismatch at position i, it makes sense to try at position $i + 1$ only if the pattern is such that its first $h - 1$ characters, which are equal to the $h - 1$ characters starting at position i in the text, are exactly equal to the $h - 1$ characters starting at position 2 in the pattern itself. If this is not the case, we waste our time looking for a match at position $i + 1$; moreover, if this is the case, we also waste time comparing the first $h - 1$ characters of the pattern, from position $i + 1$ to position $i + h$ excluded in the text, since we already know that we are going to find all matches.

More generally, after finding a mismatch at position $i + h$, we can jump in the pattern at the end of the longest prefix that is also a suffix of the first h character in the pattern, and keep on comparing the character at position $i + h$ in the text. There is no way to find an

[1]The string matching implementation described later also requires configuration memory writes to take only a few clock cycles. At least one of the devices [11] allows this and others may also.

```
**Procedure** TextSearch($P, T$)

  $n = length(T)$;
  $m = length(P)$;
  $\pi$ = ComputePrefixFunction($P$);
  $q = 0; i = 0$;
  while ($i < n$) do
    if ($T[i] \neq P[q]$)and($q == 0$) then
      ++i;
    else if ($T[i] \neq P[q]$)and($q \neq 0$) then
      $q = \pi[q]$;
    else if ($T[i] == P[q]$)and($q \neq m - 1$) then
      $+ + i; + + q$;
    else if ($T[i] == P[q]$)and($q == m - 1$) then
      print "match found";
      $+ + i; + + q$;
    end if
  end while
```

```
**Function** ComputePrefixFunction($P$)

  $m = length(P)$;
  $\pi[1] = 0$;
  $i = 1; q = 0$;
  while ($i < m$) do
    if ($P[i] \neq P[q]$)and($q == 0$) then
      ++i;
      $\pi[i] = 0$;
    else if ($P[i] \neq P[q]$)and($q \neq 0$) then
      $q = \pi[q]$;
    else if ($P[i] == P[q]$) then
      $+ + i; + + q$;
      $\pi[i] = q$;
    end if
  end while
```

Figure 2: KMP algorithm Phase 2 (Text search) and Phase 1 (Prefix function Computation).

occurrence of the pattern before that point, and, at the same time, we can take advantage of internal symmetries of the pattern avoiding checking characters in the text more than needed.

This is the key idea of the Knuth-Morris-Pratt algorithm, which computes, for each position h in the pattern, the longest prefix that is also a suffix of the first h character of the pattern itself. This information is encapsulated in a function π such that $\pi[h] = j$ if and only if the first j characters of P are the longest proper prefix that is also a suffix of the first h characters of P. Note that π does not depend on the text, and can be thus precomputed by looking at the pattern only.

The KMP algorithm is a classical 2-phase computation. It takes in input the pattern P, performs a precomputation on P to get the function π, and then, in the second phase, uses π to speed-up the search inside the text. Using terminology introduced earlier, $T_M + T_{ME}$ is the time taken by Phase 1 while the time taken by Phase 2 is T_E. The algorithms used for Phase 1 (Prefix function computation) and Phase 2 (Text search) are shown in detail in Figure 2. The algorithms shown have been written such that they correspond closely to their hardware implementation. It can be proved that KMP is optimal, requiring $O(m + n)$ to perform both phases (see [2] for a proof and a detailed description of the KMP algorithm).

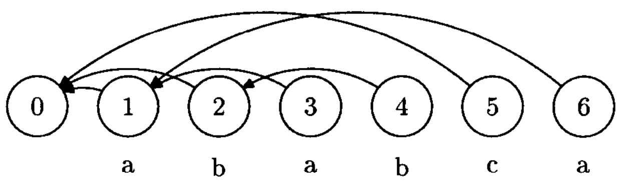


Figure 3: Example of π function for a pattern $p = ababca$. The index q of the algorithms in Figure 2 can be implemented as a pointer to a node, and an edge from the node h to the node j is present if and only if $\pi[h] = j$.

KMP seems to be an ideal candidate to be implemented on reconfigurable devices. Indeed, thanks to reconfigurability, the function π, depending on the input of each single instance of the problem, can be implemented in hardware, thus considerably speeding-up the searching phase. A good way to visualize the function π is given in Figure 3, where each node indicate a position in the pattern, and an edge is present between nodes h and j if and only if $\pi[h] = j$. In this way, the value of the index q in the the KMP-Matcher, shown in Figure 2, can be stored as a pointer to a node, and at each step of computation the pointer q moves either to the next node $q + 1$, if a match is found, or to the node $\pi[q]$ indicated by the edge starting from node q, otherwise. This behavior is very similar to that of a finite state machine, and it is well suited for hardware implementation, as will be shown in the next section.

Our implementation is devised to handle an *on-line* version of String Matching. Meaning that our FPGA system is able to read an incoming pattern, configure itself depending on it, and solve the problem on an incoming text. Moreover, it is possible to change the problem instance by furnishing a new pattern to the system. In this case, the FPGA reconfigures itself to optimize depending on the new pattern, and is ready to solve the new instance on an incoming text. All these operations (including reconfiguration) are performed inside the FPGA system itself, without involving the host computer.

5 Implementation of the KMP algorithm

In this section, we present the details of how the KMP algorithm exploiting self-reconfiguration would be implemented on a multicontext FPGA. Unfortunately, multicontext FPGAs are not commercially available. Therefore, we implement the logic on a conventional FPGA and simulate self-reconfiguration via software. We begin by describing in Section 5.1 how the algorithm is realized in hardware without discussing any FPGA specific features. Since the FSM is the most important component, its structure and runtime construction are described in detail. Section 5.2 presents the details of how it would be implemented on a multicontext FPGA. The actual implementation on a conventional FPGA (Xilinx XC6216) is presented in Section 5.3. Finally, performance is evaluated in Section 5.4.

5.1 Hardware Realization

We describe Phase 2 of the algorithm first. Logic is constructed at runtime in Phase 1 and used in Phase 2. Knowing what the constructed logic looks like and how it works makes it easier to understand the subsequent description of Phase 1.

5.1.1 Phase 2: Text Search

The datapath used for Phase 2 (see Figure 2) is shown in Figure 4. The text T is stored in external memory. The index i in the algorithm is essentially an address counter used to fetch the next text character. The entire pattern P is stored on-chip. The comparator is used to compare the appropriate text and pattern characters. The last major logic block implements the prefix function π .

The operation of the datapath can be easily understood by looking at Figure 4. Each clock cycle, the four **if** conditions are evaluated in parallel but only one of the statements is executed. The values of the signals `char_match`, `state_zero` and `state_final` determine which of the four paths is selected. The controller generates appropriate values for the signals `inc_i`, `inc_match`, `next_state` and `inc_state`. If `next_state` is 0, q remains unchanged for the clock cycle. Otherwise, $++q$ (`state_inc=1`) or $q = \pi[q]$ (`state_inc=0`) is performed. To improve performance, the implementation overlaps fetching $T[i]$ with datapath operation.

Prefix Function FSM As described in Section 4, the prefix function π can be implemented as a FSM. The FSM contains m states, 0 to $m-1$. The state corresponding to the value of q is the current state.

There are two standard techniques for implementing FSMs using programmable logic [9]. One way is using a LUT that stores the FSM states in a (typically binary) encoded form. As the FSM size increases, the speed decreases and area required increases because of the wider and deeper decoding logic and the associated routing. Also, in the our case two comparators would be required for generating the `state_zero` and `state_final` signals.

The other approach is to use the One-Hot Encoding (OHE) scheme—one flip-flop is associated with each state. At anytime exactly one flip-flop has a 1 bit signifying the current state. This approach is simpler and more efficient as it requires lesser decoding logic and suits the flip-flop rich architecture of FPGAs.

We exploit properties of π to develop a particularly compact and simple implementation of the FSM. There are exactly two possible transitions from each state. One of these is to the following state (forward edge) and the other is to one of the previous states (backward edge). These properties simplify the routing considerably. In addition, the signals `initial_state` and `final_state` are simply the outputs of the initial and final state flip-flops respectively, eliminating the need for any comparators.

5.1.2 Phase 1: Prefix Function Construction

As can be seen from Figure 2, Phase 1 is similar to Phase 2. Two minor differences are that i is initialized to 1 and the pattern P is compared with itself instead of text T. The only major difference is additional steps for constructing the prefix function π through assignments to $\pi[i]$. In terms of logic, these assignments translate to constructing the back edges of all the states of the FSM. Construction of the FSM at runtime and the logic required to do so are described below.

Online FSM Construction The FSM for the given pattern is constructed using a preconfigured *template*. The FSM template, shown in Figure 5 is independent of the pattern and constructed beforehand. Flip-outputs go to the next flip-flop (forward edges) and to horizontal wires (which runtime back edge construction described below). At any time during execution, only the flip-flop for state q has a 1-bit. The template also has storage for the pattern P with $P[q]$ available as the output of the rightmost mux.

At runtime, the first step is to customize the template for the input pattern size m. This is done by connecting the output of flip-flop for state $m-1$ to the horizontal wire that is the lower input to the state 0 flip-flop. This is followed by loading and storing the pattern on-chip. Next Phase 1 starts, and the execution of statements $\pi[i+1] = q$ and $\pi[i+1] = 0$ in the Phase 1 algorithm results in the construction a back edge from state $i+1$ to state q or state 0 respectively. As can be seen from Figure 6, this is only a matter of inserting an OR gate at the appropriate position. The piece of logic that constructs back edges takes q and i as inputs and computes the position (i^{th} row and q^{th} column) at which the OR gate is to be inserted. See Section 5.3 for implementation details of this logic.

In this manner, problem instance dependent logic is mapped within clock cycles, instead of minutes or hours that would be required if software was in the loop. Another interesting feature is that in FSM construction alternates with FSM use (whenever π is read in Phase 1). Such a fine grained interleaving would not be possible without self-reconfiguration.

5.2 Proposed Implementation on a multicontext FPGA

Before computation begins, the pattern P, pattern length m, text T and text length n are stored in external memory that can be accessed by the multicontext FPGA. The following logic is configured onto four contexts of the FPGA. Context 0 contains control logic that governs overall execution of the algorithm. Context 1 has logic for customizing the FSM for given m. Context 2 contains datapath for Phase 1 of the KMP algorithm as well as logic for runtime FSM construction. Hardwired into this logic are configuration bits for the OR-gate and its connections (referred to as *or_gate*). The number of configuration memory writes needed for OR-gate insertion is s_{or_gate}. The FSM is constructed on context 3 in Phase 1. During Phase 2 it includes the datapath required for Phase 2 as well.

Figure 7 shows the computation performed in each context (computation done in context 3 during Phase 1 and Phase 2 is shown as context 3a and context 3b respectively). At the end of each statement is the time required by the logic to execute it. The times are expressed in terms of t_{cm} (configuration memory read or write time), t_{em} (external memory read or write time), t_{clk} (one clock cycle time), t_{cs} (time required to switch contexts) and s_{or_gate}. Computation starts with context 0 switching to context 1 which customizes the FSM size. The FSM is constructed on a separate context since

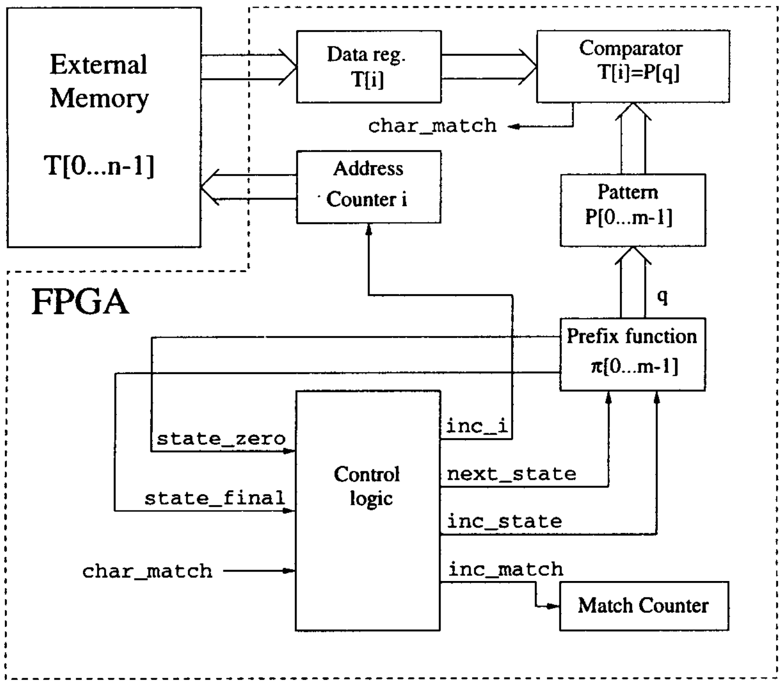


Figure 4: Datapath for Phase 2.

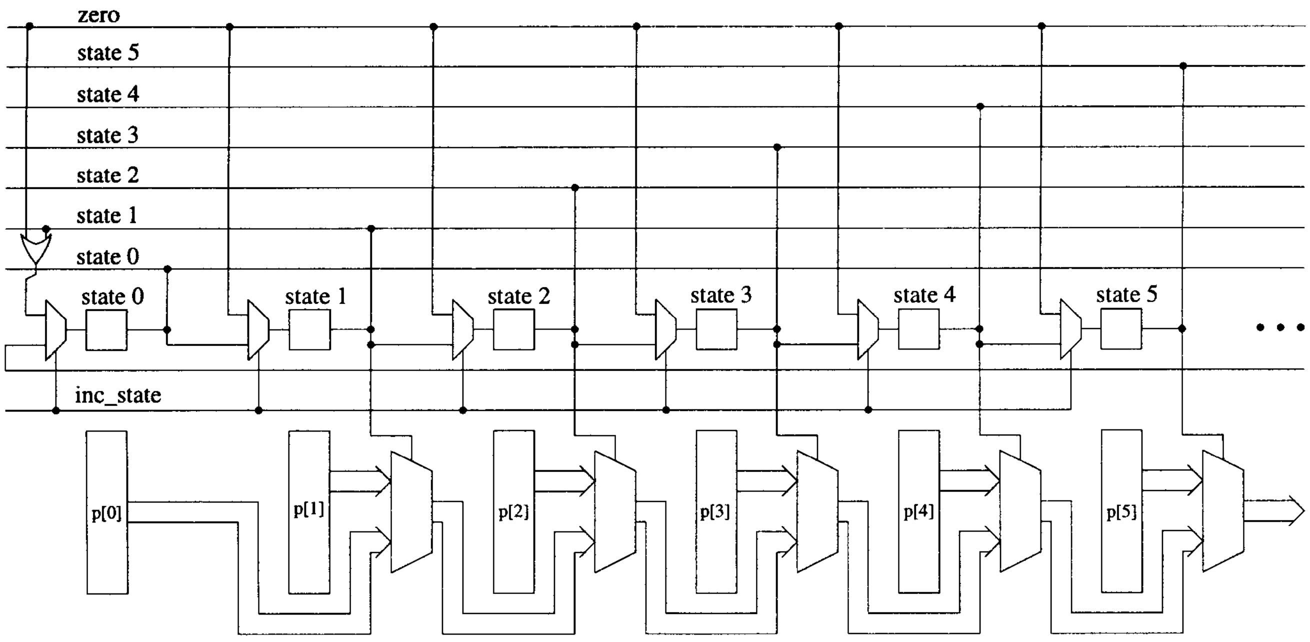


Figure 5: FSM template. The OR-gate implements $\pi[1] = 0$.

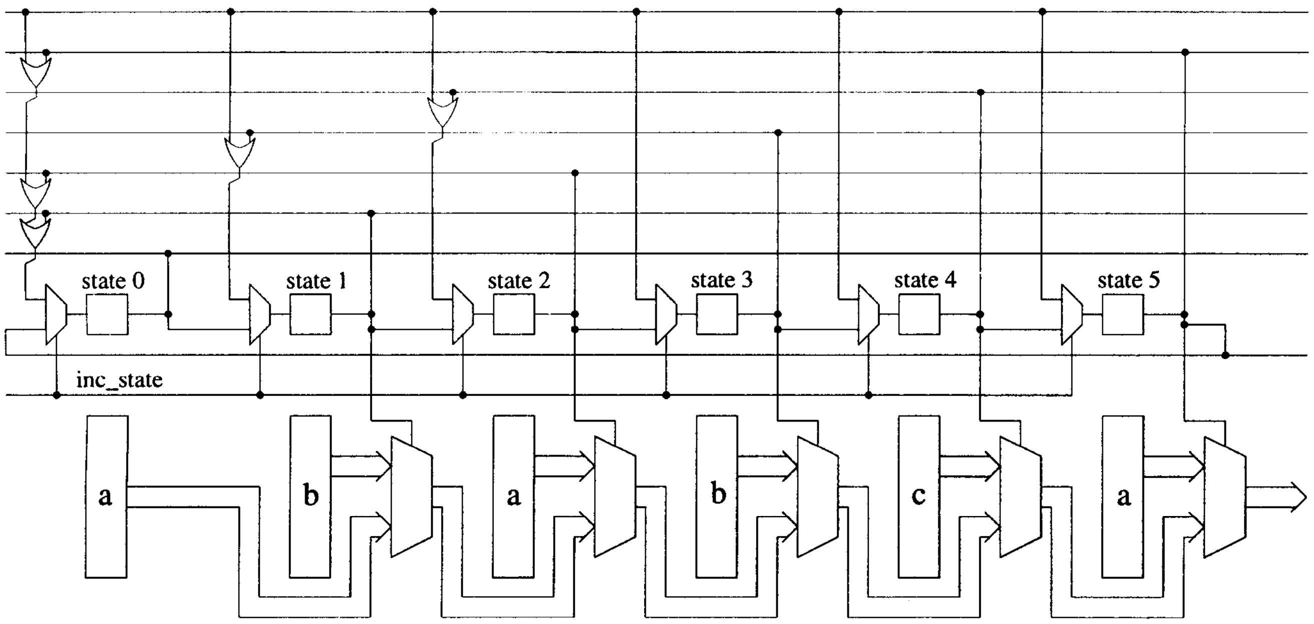


Figure 6: Back edges built through OR-gate insertion. Corresponds to FSM in Figure 3.

the currently executing context cannot modify itself. Doing do requires data sharing between contexts which is possible on multicontext FPGAs [13]. In Figure 7, *read_em* and *write_em* refer to an external memory access while *read_cm* and *write_cm* refer to a context memory access. Note that no external intervention by the host machine is required in constructing the FSM.

Next, the logic on context 2 performs Phase 1 of the KMP algorithm. Self-reconfiguration is performed via configuration memory writes to construct the appropriate back edges. Note how the FSM back edge construction alternates with use of the partially constructed FSM (by switching to context_3) alternates every few clock cycles. Finally context 0 connects the FSM to the text search datapath already present on context 2. Since their positions are fixed beforehand, the datapath can be interfaced with the runtime generated FSM to form the complete logic required for performing Phase 2 of the KMP algorithm.

The context switching is similar to context switching of processes on a uniprocessor. At a time only one of the FPGA contexts executes and switching to a context resumes its execution from where it had stopped earlier due to a context switch. This is possible because the state of the active context (bits stored in all the flip-flops) are saved before switching to a different context.

We now derive T_M, T_{ME} and T_E in terms of the times in Figure 7. T_{ME} is the time spent in write_cm operations. From the times in Figure 7,

$$T_{ME} = (m-1)s_{or\_gate}t_{cm} \qquad (1)$$

The remaining time spent in contexts 1, 2 and 3a is T_M, the time required to compute the FSM mapping and is given by [2]

$$T_M = (4m-2)t_{cs} + (m+1)t_{em} + (7m-4)t_{clk} \qquad (2)$$

Finally, the execution time T_E is the time spent in Phase 2 which is[3]

$$T_E = (2n - \frac{n}{m})t_{clk} \qquad (3)$$

A few remarks on how the above times were determined— read_em P and write_cm P are pipelined and take $(m+1)t_{em}$ time. In context 3b, only one if statement is executed each iteration taking t_{clk} time. Similarly context 3a also takes t_{clk} time. The execution time of context 2 depends upon the input pattern and the worst case occurs when all characters are identical and the last if statement is executed each iteration. The worst case time is used in T_m above.

5.3 Actual implementation on a conventional FPGA

We implement logic described for contexts 2, 3a and 3b in the previous section on a Xilinx XC 6216 device. From the implementation we determine t_{clk} and t_{em} and t_{cm}[4]. And by using a t_{cs} value based on published context switching times, we obtain using equations 1, 2 and 3, an accurate performance estimate of the KMP algorithm implemented on an abstract multicontext version of the XC 6216. The feasibility of such a device should not be in doubt since the extensions we assume have been demonstrated in various multicontext devices built so far.

The VCC Hotworks board was used for the implementation. Required logic was specified in structural VHDL and translated to EDIF format using velab. XACT 6000 was used for place, route and configuration file generation. For debugging and runtime support, XC 6200 Inspector and PCI Test were used. The 128 KB of SRAM (referred to as external memory henceforth) on the VCC board was used to simulate the configuration memory of a multicontext device.

[2]This is the worst case T_M which corresponds to a pattern containing all identical characters except the last one.

[3]This is the worst case T_E which corresponds to text containing m character repetitions in each of which the first $m-1$ characters match the pattern and the last one does not.

[4]We make the conservative assumption that $t_{cm} = t_{em}$.

```
context_0
/*Stage 1 of FSM construction.*/
switch context_1; $t_{cs}$
/*Stage 2 and Phase 1.*/
switch context_2; $t_{cs}$
/*Phase 2.*/
connect Phase 2 datapath; $t_{clk}$
switch context_3; $t_{cs}$
```

```
context_1
read_em $m$; $t_{em}$
/*Connect final state output to state 0 input.*/
connect flip-flop $m-1$; $t_{clk}$
/*Store pattern characters in pattern registers.*/
read_em $P$; $mt_{em}$
write_cm $P$; $mt_{cm}$
switch context_0; $t_{cs}$
```

```
context_2
$i = 1; q = 0;$ 0
read_em $m$; $t_{em}$
while ($i < m$) do
  /* One if statment executed every iteration.*/
  if ($P[i] \neq P[q]$)and($q == 0$) then
    ++i; $t_{clk}$
    /*Create back edge for $\pi[i] = 0$.*/
    compute OR gate insertion position; $t_{clk}$
    write_cm $or\_gate$; $s_{or\_gate}t_{cm}$
  end if
  if ($P[i] \neq P[q]$)and($q \neq 0$) then
    /*Switch to FSM context and perform $q = \pi[q]$.*/
    state_inc=0; $t_{clk}$
    switch context_3; $t_{cs}$
  end if
  if ($P[i] == P[q]$) then
    /*Switch to FSM context and perform $++q$.*/
    $++i$; $inc\_state = 1$; $t_{clk}$
    switch context_3; $t_{cs}$
    /*Create back edge for $\pi[i] = q$.*/
    compute OR gate insertion position; $t_{clk}$
    write_cm $or\_gate$; $s_{or\_gate}t_{cm}$
  end if
end while
switch context_0; $t_{cs}$
```

```
context_3a
if ($inc\_state == 1$) then
  if ($q == m-1$) then $q = 0$; else $++q$; $t_{clk}$
end if
if ($inc\_state \neq 1$) then
  $q = \pi[q]$; $t_{clk}$
end if
switch context_0; $t_{cs}$
```

```
context_3b
$i = 0; q = 0;$ 0
read_em $n$; $t_{em}$
while ($i < n$) do
  /* One if statment executed every iteration.*/
  if ($T[i] \neq P[q]$)and($q == 0$) then
    ++i; $t_{clk}$
  end if
  if ($T[i] \neq P[q]$)and($q \neq 0$) then
    $q = \pi[q]$; $t_{clk}$
  end if
  if ($T[i] == P[q]$)and($q \neq m-1$) then
    $++i; ++q$; $t_{clk}$
  end if
  if ($T[i] == P[q]$)and($q == m-1$) then
    /*Pattern match found.*/
    $++i; ++q; ++matches$; $t_{clk}$
  end if
end while
```

Figure 7: KMP algorithm implementation on a multicontext FPGA.

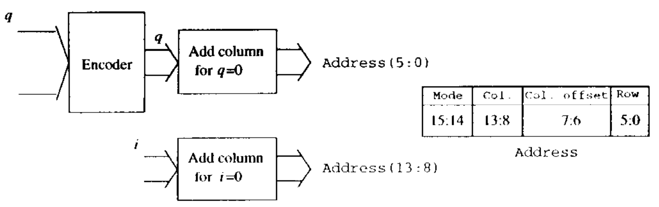


Figure 8: Generating configuration memory address for OR-gate insertion. Address bits 15:14 and 7:6 are constant and known beforehand.

For Phase 1 we implement on the XC 6216 the Phase 1 datapath, OR-gate construction logic and the FSM template. All this logic corresponds to contexts 2 and 3a in Figure 7. For each back edge, the address in configuration memory where the OR-gate is to be inserted is written out to external memory (in one clock cycle). This information is used to modify the configuration file which is used to reconfigure the FPGA for computing the next back edge. Knowing row and column of a logic cell, it is trivial to compute the corresponding configuration addresses since the row and column numbers directly form a part of the 6200 address. The logic for OR-gate computation is thus quite simple and is shown in Figure 8. Inserting the OR-gate and making the appropriate connections needs just 24 bits of configuration data which is embedded in the logic itself. Three separate writes are required however since each byte needs to be written to a separate address. Thus $s_{or\_gate} = 3$. For Phase 2 we implement logic corresponding to context 3b on the XC 6216. The logic searches through text stored in the external memory just as a multicontext FPGA would since no context switching is involved in this phase.

5.4 Performance Evaluation

From the implementation description in Section 5.3 it should be clear that $t_{cm} = t_{em} = t_{clk}$. Based on published literature, we make the conservative assumption that t_{cs} = 100ns. We determine t_{clk} as follows. For a given pattern size, we increase the clock frequency till any further increase makes the implemented logic stop working correctly. The corresponding clock period is the value of t_{clk}. t_{clk} increases somewhat with pattern size since the corresponding FSM is bigger and hence the critical path is longer. Plugging all the above values into equations 1, 2 and 3 for pattern size m varying from 4 to 16, and text size $n = 10^4$ characters, we obtain the results shown in Table 1.

m	t_{clk}	T_M	T_{ME}	T_E	Total time
4	81.6 ns	3.7 μs	0.7 μs	1428 μs	1432 μs
8	97.6 ns	9.0 μs	2.1 μs	1830 μs	1841 μs
16	129.6 ns	22.4 μs	5.8 μs	2511 μs	2539 μs

Table 1: Performance of the implementation for various values of m with $n = 10^4$.

We now compare the mapping time ($T_M + T_{ME}$) of the proposed multicontext FPGA approach with other approaches. Consider the case where CAD tools are used to perform the FSM construction. To find T_M for this approach, we determine the time taken to compile a structural VHDL description[5] for m = 8 using velab (4 s) and route it using XACT 6000 (68 s) giving T_M = 72 s. T_{ME} = 1 ms is the time required to download the configuration onto the XC 6216 via the PCI bus. To make T_M as small as possible, we explicitly specify placement of logic and use XACT 6000 only for routing. Even then, as can be seen from row 2 Table 2, the proposed approach is six orders of magnitude faster than the naive use of CAD tools. Of course a multicontext FPGA is needed to obtain the speedup. A smarter approach would be to write a program that directly modifies the binary configuration file based on the input pattern. This approach is essentially doing in software what we do on the FPGA itself. Row 3 of Table 2 shows the performance of this approach[6]. Although much faster than the CAD tools approach, it is still more than 1800 times slower than the proposed approach.

Table 3 shows the total execution time speedups over other approaches. We also compare the performance with a software implementation of the KMP algorithm running on a Sun Ultra 1 Model 140. As can be seen from row 4 of Table 3, reasonable speedups are obtained. A key point to note is that the multicontext FPGA is better than others for all values of n. This is in contrast to most reported results where the problem size must be very large to amortize the high mapping time.

Comparison of the implementation with other FPGA based string matching implementations is unfortunately not possible due to differences in the FPGA architectures and the algorithms used. We note however, that in [7] T_M = 0.16s and T_{ME} = 3.05s. These times are for a naive string matching implementation on 16 CAL1024 FPGAs that runs at 20 MHz. Thus, in [7], speedups will be obtained only for very large problem sizes due to the high $T_M + T_{ME}$.

6 Conclusion

We have shown dramatic speedups in the time required to map logic at runtime onto FPGAs. This is done by the novel approach of developing logic that maps logic and putting the former on the FPGA itself. As a result CAD tools need to be used just once for each problem (to build logic that builds logic and some template logic) and not once for every problem instance as is usually done. The reduction in mapping time achieved is extremely important because FPGAs can do better than ASICs only if the mapping is problem instance dependent, which means that the runtime mapping time is a part of the overall execution time.

We show how self-reconfiguration can be performed using multicontext FPGAs and how to efficiently realize the above approach through self-reconfiguration. We demonstrate our approach by presenting a detailed implementation of the KMP string matching algorithm which utilizes the above approach to construct a FSM at runtime. An interesting feature of the implementation is that FSM construction and use of the FSM alternate every few clock cycles. Such a fine grained interleaving of mapping logic and using it would not be possible with software in the loop.

Finally, we implement the KMP algorithm on a conventional FPGA and use it to obtain accurate estimates of performance on a multicontext device. Our results show high speedups in mapping time

[5]We ignore the time required to generate the VHDL code for the given input pattern as it would be quite small. In any case, accounting for this time would only improve our speedup. The times are obtained on an IBM PC with a 200 MHz Pentium Pro and 64 MB RAM.

[6]The time T_M is for a C program running on a Sun Ultra 1 Model 140.

Approach	T_M	T_{ME}	$T_M + T_{ME}$	Speedup
Multicontext FPGA	9.0 μs	2.1 μs	11.1 μs	1.0
CAD tool mapping	76 s	1 ms	76 s	$\approx 6 \times 10^6$
Software mapping	20 ms	1 ms	21 ms	1892

Table 2: Speedup in mapping time (m = 8).

Approach	$T_M + T_{ME} + T_E$			Speedup		
	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^4$	$n = 10^5$	$n = 10^6$
Multicontext FPGA	1.8 ms	18.3 ms	183.1 ms	1.0	1.0	1.0
CAD tool mapping	76.0 s	76.0 s	76.2 s	$\approx 10^5$	$\approx 10^4$	$\approx 10^3$
Software mapping	21.8 ms	39.3 ms	204.1 ms	12.1	2.1	1.1
Sun Ultra 1	30 ms	80 ms	680 ms	16.6	4.4	3.7

Table 3: Speedups over other approaches for various values of n, with m=8.

and reasonable speedups in overall execution time over various existing approaches.

This work has been done as a part of the MAARC (Models, Algorithms and Architectures for Reconfigurable Computing) project. The MAARC project is developing a framework of algorithmic techniques for reconfigurable computing and exploiting this technology for embedded signal and image processing applications. Please see [10] for more information.

References

[1] BABB, J., FRANK, M., AND AGARWAL, A. Solving graph problems with dynamic computation structures. In *Proceedings of the SPIE - The International Society for Optical Engineering* (Boston,MA, 1996).

[2] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachussets, 1990.

[3] DEHON, A. Multicontext Field-Programmable Gate Arrays. http://HTTP.CS.Berkeley.EDU/ãmd/-CS294S97/papers/dpga_cs294.ps.

[4] DEHON, A. DPGA-coupled microprocessors: Commodity ICs for the early 21st century. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* (Napa, CA, Apr. 1994), D. A. Buell and K. L. Pocek, Eds., pp. 31–39.

[5] DONLIN, A. Self modifying circuitry - a platform for tractable virtual circuitry. In *Eighth International Workshop on Field Programmable Logic and Applications* (1998).

[6] FRENCH, P. C., AND TAYLOR, R. W. A self-reconfiguring processor. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* (Napa, CA, Apr. 1993), D. A. Buell and K. L. Pocek, Eds., pp. 50–59.

[7] GUNTHER, B., MILNE, G., AND NARASIMHAN, L. Assessing document relevance with run-time reconfigurable machines. In *Proceedings of the 4th IEEE Symposium on FPGAs for Custom Computing Machines* (Napa, California, Apr 1996).

[8] JONES, D., AND LEWIS, D. A time-multiplexed FPGA architecture for logic emulation. In *Proceedings of the 1995 IEEE Custom Integrated Circuits Conference* (May 1995), pp. 495–498.

[9] KNAPP, S. K. Accelerate FPGA macros with one-hot approach. *Electronic Design 38*, 17 (1990), 65–71.

[10] MAARC project. http://maarc.usc.edu.

[11] MOTOMURA, M., AIMOTO, Y., SHIBAYAMA, A., YABE, Y., AND YAMASHINA, M. An embedded DRAM-FPGA chip with instantaneous logic reconfiguration. In *1997 Symposium on VLSI Circuits Digest of Technical Papers* (June 1997), pp. 55–56.

[12] MOTOMURA, M., AIMOTO, Y., SHIBAYAMA, A., YABE, Y., AND YAMASHINA, M. An embedded DRAM-FPGA chip with instantaneous logic reconfiguration. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* (Napa, CA, Apr. 1998). Poster paper.

[13] SCALERA, S. M., AND VÁZQUEZ, J. R. The design and implementation of a context switching FPGA. In *IEEE Symposium on Field-Programmable Custom Computing Machines* (Apr. 1998), pp. 495–498.

[14] TRIMBERGER, S., CARBERRY, D., JOHNSON, A., AND WONG, J. A time-multiplexed FPGA. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* (Napa, CA, Apr. 1997), J. Arnold and K. L. Pocek, Eds., pp. 22–28.

[15] ZHONG, P., MARTONOSI, M., ASHAR, P., AND MALIK, S. Accelerating boolean satifiability with configurable hardware. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* (Napa, CA, Apr. 1998).