Matching and Searching Analysis for Parallel Hardware Implementation on FPGAs*

Pablo Moisset, Pedro Diniz and Joonseok Park
University of Southern California / Information Sciences Institute
4676 Admiralty Way, Suite 1001, Marina del Rey, California 90292
{pmoisset, pedro, joonseok}@isi.edu

ABSTRACT

Matching and searching computations play an important role in the indexing of data. These computations are typically encoded in very tight loops with a single index variable and a simple search/matching predicate. Their inherent sequential nature, either because of data dependences but more often because of very strong control dependences, makes it impossible to apply existing data dependence and parallelization analysis to exploit significant levels parallelism on traditional architectures. This paper describes a class of searching and matching computations and describes a mapping strategy to map these computations to hardware. We have developed a compiler analysis in SUIF using array data dependence analysis and implicit loop unrolling analysis to expose more parallelism for the parallel evaluation of these computations. Our compiler generates parallel hardware specifications in VHDL. The resulting parallel hardware yields significant performance improvements when these kernel operators are repeated over shifted portions of the input data on FPGA-based computing architectures.

1. INTRODUCTION

Matching and searching computations play an important role in data indexing and searching in large arrays. Typically these computations are encoded in very tight loop nests that sequentially iterate over multidimensional arrays. The basic computation compares a key with the items in the array for a specific value/sequence in the data array and return its index if successful. Often, these loop nests repeat the same computation over shifted versions of the input data. For example, a pattern matching algorithm can be encoded as a search of a given pattern over every subsequence of the same length of an input string.

These small searching and matching loops often have control instructions such as *break* and *continue* to abort the inner loop once the result is known. These control dependences and the fact that the loop body is typically a single comparison operation, render inapplicable most data dependence and loop parallelization analyses [10]. In addition processor techniques such as predication and speculation offer little help to effectively increase performance on traditional processor architectures.

* Funded by the Defense Advanced Research Project Agency under contract number F30602-98-2-0113

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FPGA'01, February 11-13, 2001, Monterey, CA, USA.
Copyright 2001 ACM 1-58113-341-3/01/0002...$5.00.

The way to map these computations to hardware is to allow the hardware to evaluate, in parallel, a sequence of predicates corresponding to a sequence of loop iterations. For example, when matching a given string, hardware can match several characters in parallel and perform a logical OR operation for the results of the individual matching. If at least one of the comparisons fails, the whole matching must fail. Our proposed scheme carries out the computation in a parallel fashion even though in some instances the resulting hardware might execute more work that the original sequential implementation.

This approach of mapping matching and searching computations to parallel hardware is even more appealing when the compiler can specify the actual datapath of the parallel implementation, as is the case with existing FPGA-based computing architectures. For these architectures, the compiler can automatically unroll the inner loops of these computations to increase the number of array accesses and realize that most of the accessed data can be stored in tapped delay lines for use in subsequent computations. A computation that repeatedly performs the same operation over shifted versions of the input data can simply reuse most of the data already in a tapped delay line, thus substantially reducing the number of memory accesses.

We developed a compiler analysis and code generation scheme for automatically detecting and exploiting the parallelism available on kernel matching and searching computations. The analysis uses the SUIF [8] representation format and generates VHDL specifications for the parallel hardware that can carry out the these kernel computations. This analysis uses input data dependence analysis to extract the opportunities for data reuse in tapped delay lines over successive iterations of the computation and determines how the results of the parallel evaluation of the searching and matching are to be combined. In the case where break and continue statements exist, the analysis determines, with the help of priority encoding hardware, which are the correct results to be generated by the computation. We have mapped the generated VHDL specifications to FPGAs and evaluated the performance of the generated designs.

This paper makes the following contributions:

- Identifies a subset of tight loop nests as searching and matching computations. These loops have *break* and

continue statements that elude previous data dependence and parallelization analysis.

- Describes a compiler analysis algorithm to capture these computations. The algorithm combines loop unrolling and control flow analysis. The analysis also exposes data reuse, among array elements, and across successive iterations of the loop nests.
- Presents a code generation strategy to implement matching and searching computations in parallel hardware. These storage structures, derived by the compiler analysis, are aimed at implementing data reuse across iterations, therefore reducing the number of memory accesses.
- Presents experimental results of the automatic application of the compiler analysis and code generation for a set of computation kernels. It also presents performance results for the generated parallel implementation for these kernels on an FPGA-based computing architecture.

With the increasing number of available transistors on a die architects now have available the option of incorporating reconfigurable and more flexible datapaths with traditional architectures [13,14,15]. In this context the analysis presented in this paper would allow a compiler writer to explicitly target specific computations to be mapped to the reconfigurable fabric of the new processors and serve as a basis for more aggressive compilation/mapping strategies.

The rest of this paper is organized as follows. In the next section we describe via an example the mapping of a searching and matching computation to parallel hardware. In section 3 we describe the compiler analysis algorithm and its implementation. In section 4 we present our experimental results. Section 5 describes related work and we conclude in section 6.

2. EXAMPLE

Figure 1 below presents an example of a matching computation over strings of characters written in the C programming language.

```
for(i = 0; i < STRING_SIZE - PATTERN_SIZE; i++){
  res[i] = '1';
  for(j = 0; j < PATTERN_SIZE; j++)
    if(pattern[j] != string[i+j]){
      res[i] = '0';
      break;
    }
}
```

Figure 1. String Matching Computation Example.

The computation is organized as a doubly nested loop. The innermost loop checks the *pattern* array variable against a substring of the *string* array variable. For every substring of the same length the computation checks if there is any character in a correspondingly aligned position in the *pattern* variable that differs. If there is at least one such character that differs, the computation sets the value of the result to'0'. The outermost loop simply restarts the matching for a new starting point of the *string* variable. The computation generates a set of result values stored in the *res* array.

Despite its inherent sequential nature it is possible to execute all of the iterations of the innermost loop in parallel. The key observation is that, for each iteration of the outermost loop the result of its computation is monotonic. The result is initially set to be'1' and then possibly reset to'0'. If all of the iterations of the innermost loop were to execute in parallel and more than one update to the variable *res* were to execute concurrently the semantics of the computation would have been preserved regardless of which of the iterations would contribute to the final result of the matching.

Figure 2 below illustrates a possible implementation of this computation using parallel hardware. The computation of the innermost loop is explicitly unrolled and each comparison of the characters from the *pattern* variable against the *string* variable are performed in parallel using comparators. The implementation merges the outputs of the comparators using a logical OR gate and selects the result to be assigned to the variable *res* using a multiplexor.

This example also reveals another source for performance improvement. Given the variable reference *string[i+j]* the compiler can determine that by unrolling the *j* loop, successive iterations of the outermost *i* loop will use data locations corresponding to successive and overlapping locations for the *string* array variable. This means that the parallel hardware can retain most of the *string* variable characters in a *tapped delay line* and reuse them across iterations of the *i* loop, therefore substantially reducing the number of memory accesses the computation performs.[1]

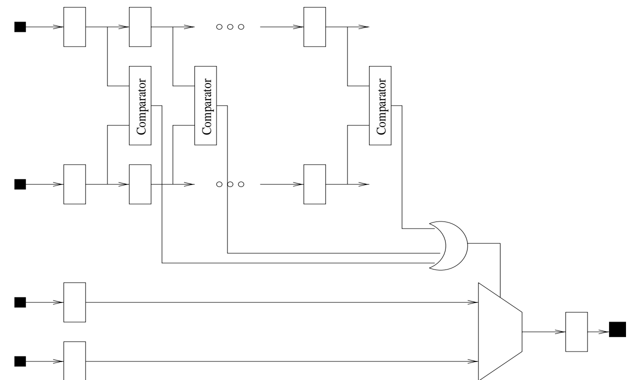

Figure 2. String Matching Parallel Implementation.

This example illustrates the set of opportunities our compiler analysis is designed to handle. First it concentrates on tight loops that perform matching and searching operations. These loops typically have *break* and *continue* statements and manipulate multi-dimensional array variables. Next the analysis selects which loops to unroll and which loop to remain rolled. Finally the analysis determines the opportunities for storing values across iterations of the loop using *tapped delay lines* for potentially reducing the number of memory accesses.

1 For convenience the storage designed to hold the values of the *pattern* variable is also a tapped delay line. However, this tapped delay is only activated to shift values during set up and remains constant afterwards.

3. COMPILER ANALYSIS

We now describe the compiler analysis algorithms and code generation strategy. We start by describing the specific format of the loop nests our analysis handles. Next we describe the compiler algorithms that extracts the predicates used in the parallel evaluation of the iterations of the loop. Finally we describe the code generation alternatives for the particular cases of searching and matching computations.

3.1 Targeted Loop Nests

We focus our attention on *for* loop nests with explicit index variables. We impose the constraint that the loop be of the generic form outlined in Figure 3. While the core of the loop must be perfectly nested we allow assignment statements between the core of the perfectly nested loop and outermost loops. The fundamental constraint is that the compiler analysis will not be able to unroll any loops outwards of the assignment statements.

```
assignment_0;
for (var_1 = const; var_1 rel_op expr; var_1 += const){
  ...
  for (var_k = const; var_k rel_op expr; var_k += const){
    loop nest body;
  }
  ...
}
assignment_n;
```

Figure 3. Generic Loop Format.

In terms of dependences we impose the loop not to have any loop-carried data/control dependences in its body other than through scalars as long as the scalars can be determined to be part of an associative operator.

It is possible to relax these constraints through the use of existing analyses and transformations (*e.g.*, scalar replacement [3]).To allow the compiler to analyze the impact of loop unrolling in the generated parallel implementation, we require the loop bounds on the innermost loop to be known at compile-time. Outermost loops, however, may have unknown compile-time bounds.

The analysis algorithm handles *if* statements in the loop body as they are common in searching and matching computations, as a convenient way to defined loop exit conditions. The body of the loop may also contain any number of *break* and/or *continue* statements. For implementation convenience, however, arbitrary control structures such as *goto*, *return* and *switch* statements are disallowed. As a consequence of the stated constraints, the control flow graph for the body of the loop is acyclic and has a single entry point. For simplicity, we also assume there is no unreachable code consisting of dead instructions located after a *break* or *continue* statement.

3.2 Datapath Generation Primitives

The ultimate goal of the compiler is to generate parallel hardware capable of carrying out the computation in the body of the loop nest, possibly with the unrolling of the innermost loop. To this effect we have defined an internal datapath representation using discrete functional blocks, connected in a data-flow like fashion. We now address the key elements in the mapping of high-level programming constructs to this datapath representation.

3.2.1 Handling If Statements

The key to implement conditional flow constructs in hardware is to execute in parallel all possible execution paths in hardware and select the correct set of side-effects. A side-effect for a given program variable is the last assignment to that variable on each execution path. If only *assign* and *if* statements are present in the loop body then at most one multiplexor per variable per *if* statement is required as illustrated in Figure 4.

```
a = b ;
if (p)
  a = c ;
```

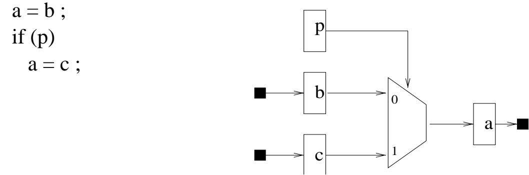

Figure 4. Implementation of If Statements.

3.2.2 Handling Break and Continue Statements

If the loop body contains *break* or *continue* statements the set of side-effects is determined by the predicates that control the execution of the *break* and *continue* statements. In the example in Figure 5 (left) the last assignment to the variable *pos* corresponds to the first iteration in which the predicate *(str[i]==value)* holds. To select the correct loop output values the generated hardware can use a *Priority Encoder (PE)* logic block, as schematically presented in Figure 5 (right). A *PE* block selects and encodes the most significant (given some predefined convention) predicate that is *true*. A *PE* consist of combinatorial logic with N binary inputs, named $IN_1...IN_n$, and one output. The output is the binary encoded *m* value iff inputs $IN_1...IN_{m-1}$ are zero and IN_m is one. The output is zero iff all inputs are zero.

```
/* Find the position of first matching character*/
pos = -1 ;
for (i = 0 ; i < LEN ; i++){
  if ( str[i] == value ){
    pos = i ;
    break ;
  }
}
```

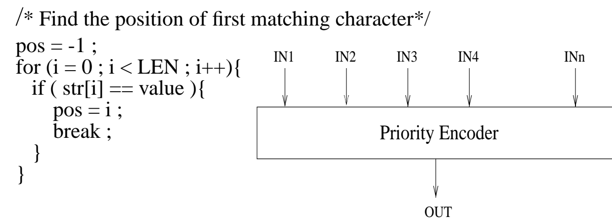

Figure 5. Character Searching and Priority Encoder.

3.2.3 Datapath Manipulation Functions

In addition to the basic structures outlined above we have also implemented a set of datapath manipulation functions for creating, and replicating datapath basic blocks in the context of loop unrolling transformations.

3.3 Compiler Analysis Algorithms

We now describe the analysis algorithms the compiler currently uses to evaluate the predicates associated with *break* and *continue* statements in the body of loop nests. We start with some basic definitions.

- **Control Flow Graph (CFG)**: Each vertex v in V(CFG) represents a single statement in the body of the loop nest. Edges represent control flow in the body of the loop. We also use **TopologicalSort**(CFG) as a list of statements in the CFG sorted in topological order.

- **Breaks(L) and Continues(L)**: Describe, respectively, the set of nodes in V(CFG) corresponding to *break* and *continue* statements in the source code.
- **Predicate(s)**: Predicate of an *if* node *s* is the condition of the *if* statement corresponding to *s*.
- **s.pred**: For every statement *s*, s.pred is the predicate that represent the condition to execute *s*.

3.3.1 Algorithm to Handle Continue Statements

The goal of this algorithm is to extract the binding of control-flow predicates associated with each *continue* statement in the loop and the corresponding set of side-effects when that *continue* statement is executed.

The analysis algorithm works in two steps as outlined in Figure 6. First the algorithm extracts for each *continue* statement the symbolic predicate that corresponds to the control flow reaching that statement. This is accomplished by symbolically tracing the control flow paths from the entry of the CFG to the *continue* statement, and is equivalent to findings the lexical nesting predicate of that statement. In the second step the algorithm traverses the CFG to symbolically extract the side-effects corresponding to the *continue* statement.

```
ComputeSymbolicPredicates ( StatementList list, CFG ) is
for all t  in IfNodes(CFG) do
 p = 'true';
 for all statements s in list do
  if ( t is an IfNode and
      (s is reachable from t through 'then' branch xor
       s is reachable from t through 'else' branch) ) then
    if ( s is reachable from t through 'then' branch ) then
       p = p + "and " + predicate(t);
    else
      p = p + "and not" + predicate(t);
    endif
  endif
 endfor
  s.pred = p;
endfor

datapath AugmentDatapath (Datapath datapath, StatementList list) is
  for all statements s in list do
     add logic implementing s.pred to the datapath;
  endfor
  return datapath;
end
```

Figure 6. Algorithms to Extract Symbolic Execution Predicates and to augment a Datapath with the predicates attached to statements.

3.3.2 Algorithm to Handle Break Statements

The main difference between the analysis algorithm for *break* statements and the analysis for *continue* statements described earlier is that the analysis assumes the loop to be completely unrolled. Given the loop unrolling, the analysis annotates each of the *break* statements with a set of predicates, one for each of the loop iterations. The compiler then uses these predicates to generate a parallel implementation that choose the appropriate side-effects of the loop corresponding to the first iteration for which the corresponding predicate is *true* using *PE* blocks. The number of inputs of each *PE* block is the number of *breaks* in the original body times the loop unrolling factor.

```
datapath HandleBreaks(CFG) is
 B = Breaks(TopologicalSort(CFG));
 ComputeSymbolicPredicate(B, CFG);
 datapath = CreateDatapath ( IgnoreBreaks(CFG) );
 datapath = AugmentDatapath (datapath, B);
 datapath = UnrollDatapath ( datapath, NumIterations);
 priority_encoder = NewPriorityEncoder (Length(B)*NumIterations);
 for i=1 to Length(B)*NumIterations do
   Connect(b[i].pred, priority_encoder. input[i]);
 endfor
 for all output variables v do
  mux = NewMultiplexor ( Length(B)*NumIterations+1 to 1);
  t = last assignment to v in datapath when no break is taken;
  Connect(t, mux.input[0]);
  for i=1 to Length(B)*NumIterations do
    t = last assignment to v in datapath before break b[i];
    Connect (t, mux.input[i]);
  endfor
  Connect ( priority_encoder.out, mux.control );
 endfor
end
```

Figure 7. Algorithm to Extract Symbolic Execution Predicates and Generate datapath in the presence of Break Statements only.

In the presence of both *break* and *continue* statements the algorithm is as outlined in Figure 8.

```
Datapath HandleContinues(CFG, datapath) is
 C = Continues(TopologicalSort(CFG));
 ComputeSymbolicPredicates({C}, CFG);
 datapath = AugmentDatapath (datapath, {C});
 priority_encoder = NewPriorityEncoder(Length(C));
 for i=1 to Length(C) do
  Connect( c[i].pred, priority_encoder.input[i] );
 endfor

 for all output variables v do
  mux = NewMultiplexor ( Length(C)+1 to 1 );
  t = last assignment to v in datapath ignoring breaks/continues;
  Connect ( t, mux.input[0] );
  for i=1 to Length(C) do
   t = last assignment to v in datapath before c[i];
   Connect ( t, mux.input[i] );
  endfor
  Connect ( priority_encoder.out, mux.control );
 endfor

Datapath HandleBreaksContinues(CFG) is
 datapath = CreateDatapath(CFG);
 datapath = HandleContinues(CFG, datapath);

 B = Breaks(TopologicalSorting(CFG));

 ComputeSymbolicPredicates({B}, CFG);
 datapath = AugmentDatapath (datapath, {B});
 datapath = UnrollDatapath (datapath, NumIterations);

 priority_encoder = NewPriorityEncoder (Length(B)*NumIterations);
 for i=1 to Length(B)*NumIterations do
   Connect(b[i].pred, priority_encoder. input[i]);
 endfor

 for all output variables v do
  mux = NewMultiplexor ( Length(B)*NumIterations+1 to 1 );
  t = last assignment to v in datapath ignoring breaks/continues;
  Connect ( t, mux.input[0] );
  for i=1 to Length(B)*NumIterations do
   t = last assignment to v in datapath before b[i];
```

```
      Connect ( t, mux.input[i] );
    endfor
    Connect ( priority_encoder.out, mux.control );
  endfor
end
```

Figure 8. Algorithm to Generate Datapath in the presence of Break and/or Continue Statements.

3.4 Special Cases - Searching and Matching

When the computation in the loop generates a single value for each of the variables it updates, then we say the computation performs a matching / searching computation. Under these conditions it is legal to replace the priority encoder for each variable with a OR logic gate. This OR gate controls the multiplexor that assigns the two only possible values for each variable, respectively the value before the loop and the value if any assignment in the loop is executed. The string matching computation in Section 2 is an example of such particular case. We illustrate a more sophisticated example in Figure 9.

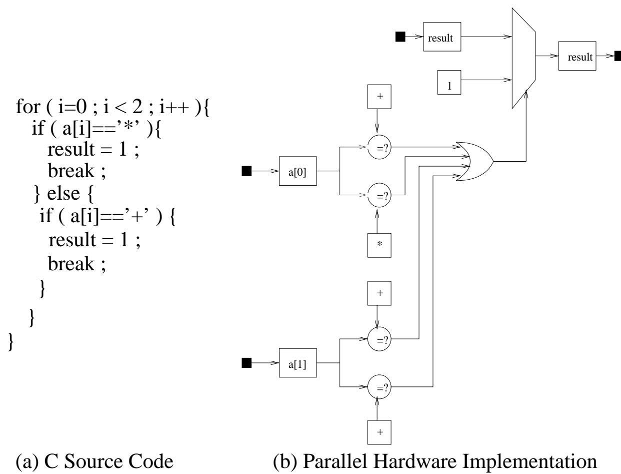

Figure 9. Special Matching and Searching Example.

To handle the opportunities exposed by the conditions of these searching and matching computations we modified the algorithm that handles the *break* statements as show below.

```
datapath HandleSingleValueBreaks (CFG) is
  B = Breaks(CFG);
  ComputeSymbolicPredicates({B}, CFG);
  datapath = CreateDatapath ( CFG );
  datapath = AugmentDatapath ( datapath, {B} );
  datapath = UnrollDatapath ( datapath, NumIterations);
  p = All unrolled versions of b.pred combined using an 'OR' operator;
  for all output variables v do
    mux = NewMultiplexor ( 2 to 1 );
    t = last assignment to v in datapath ignoring breaks;
    Connect(t, mux.input[0]);
    t = last assignment to v in datapath before b;
    Connect ( t, mux.input[1] );
    Connect ( p, mux.control );
  endfor
end
```

Figure 10. Special Algorithm for Single Value Breaks.

3.5 A Complete Example

To show how the algorithm *HandleBreaksContinues* (Sec. 3.3.3) works we present a complete nontrivial example which mixes both *break* and *continue* statements. The code we will use serves no real purpose but is complex enough to illustrate the incremental construction of the final datapath. The algorithm we will use is . The C code for this example is shown in Figure 11 (left).

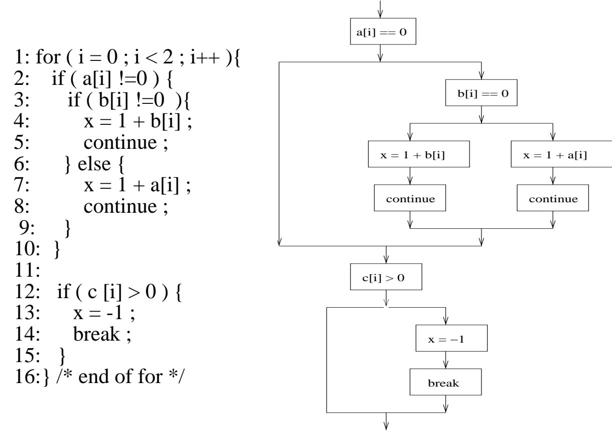

Figure 11. Control Flow Graph (CFG) for the Example.

The first step is to create the CFG. Both *break* and *continue* statements are treated as "no operations" at this point. The second step is to create the datapath that represents the loop body. At this point *break* and *continue* statements are still ignored. Only arithmetic expressions and *if* statements are relevant. The resulting datapath is shown in Figure .

At this point the analysis augments the datapath to handle the *continue* statements. *HandleContinues* determines the predicates that describe the conditions required to execute each *continue* statement, respectively as *(p[1] = "a[i] != 0 AND b[i] != 0")* and *(p[2] = "a[i] != 0 AND NOT b[i] != 0")* and where *p[1]* represents the condition to execute the *continue* in line 5 of the program while *p[2]* is the condition to execute the *continue* statement in line 8. After all predicates have been computed, the logic to evaluate is added to the datapath[2].

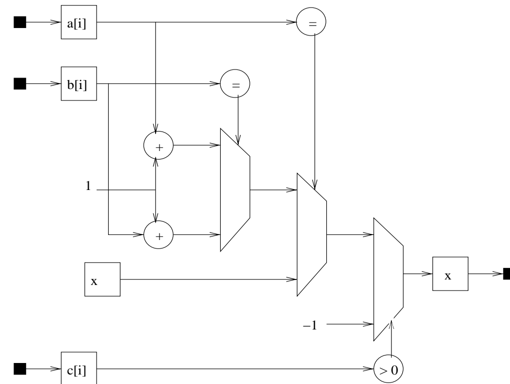

Figure 12. Generated Datapath for Example.

2 Since the predicates are composed by boolean expressions part of *if* statement conditions, most of the work was done during the generation of the initial datapath.

The second step of the *HandleContinues* algorithm adds a priority encoder and as many multiplexors as output variables in the loop. The example requires one multiplexor to select the proper value for variable ‘x’. Figure 13 shows the result of all transformations. The region inside the dashed line contains the logic added by *HandleContinues*.

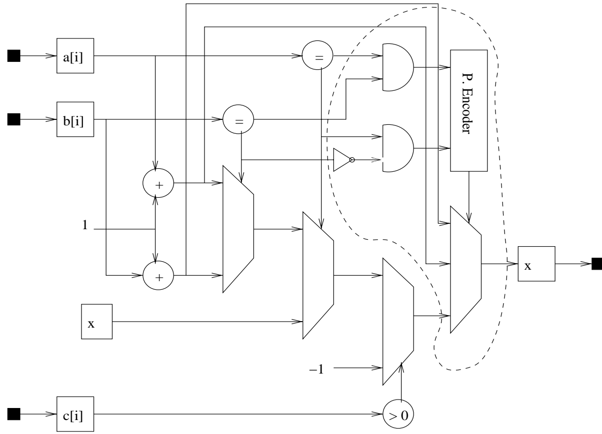

Figure 13. Augmented Datapath with Logic to implement Continues.

Next the algorithm handles the *break* statements. First it determines the predicates that describe the execution condition of the *break* statement in line 14 as *(“c[i] > 0”)*. At this point the datapath is unrolled, and the algorithm inserts the predicates corresponding to the unrolled constructs. In the example the unrolling factor will be 2 resulting in two copies of the loop body. The only loop output variable is ‘*x*’ and the last assignment to it is *(x = -1)* in line 13. After loop unroling there are three sets of possible expressions that can be assigned to the variable ‘*x*’:

1. The last assignment before the first instantiation of the *break*.
2. The last assignment before the second instantiation of the *break*.
3. The last assignment when no *break* is executed.

The algorithm adds one multiplexor and one priority encoder to implement this selection as shown in Figure 14.

3.6 Commutative and Associative Operators

Commutative and associative operators offer further opportunities for optimization and parallelization. Although not the focus of this paper, our compiler uses a simple pattern matching template approach to detect the following associative operators: Conditional accumulations; Minimum/Maximum over arrays; Array comparison, matching and searching.

3.7 Other Implementation Optimizations

The datapath generated by our algorithm can be further improved using the following two transformations:

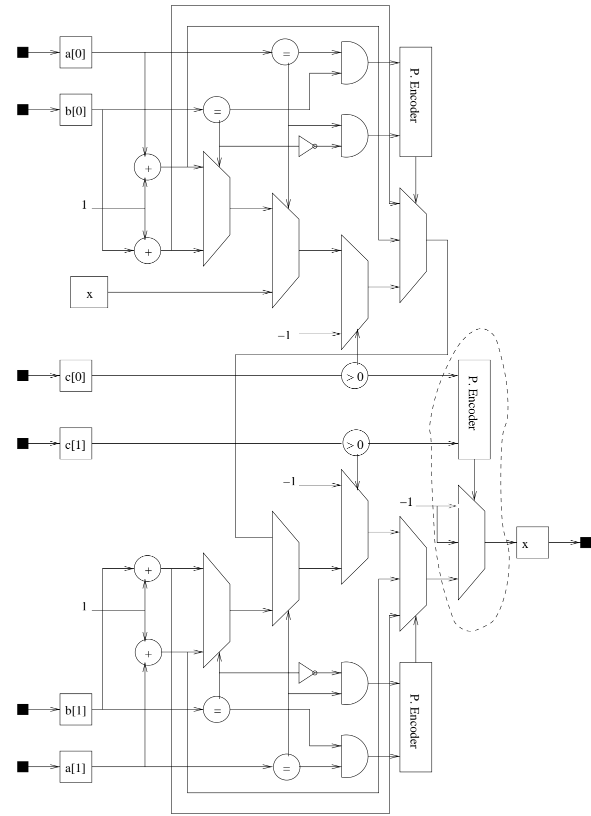

Figure 14. Final Datapath after unrolling and adding logic to handle *break* statements.

Pipelining: The throughput can be improved by pipelining the execution of the acyclic components of the datapath. As long as the critical path does not include cyclic components, the clock rate can be increased. Register insertion techniques are described in [9] and [10].

Data reuse: Whenever a data item is needed in more than one iteration there is an opportunity to avoid memory accesses.

```
for ( i=0 ; i < N ; i++ ){
  res[i] = 1;
  for ( j=0 ; j < LEN ; j++ ){
    if ( str[i+j] != pattern[j] ){
      res[i] = 0 ;
      break ;
    }
  }
}
```

In the example above only one element (*str[i+LEN-1]*) has to be read for a particular iteration of the outermost loop. Values *str[i]* through *str[i+LEN-2]* have been read in previous iterations an can be kept in registers. This approach avoids reloading items from memory several times as array elements can be stored in tapped delay lines. Further benefits are possible due to packing of array elements in basic memory transfer units. For example accessing consecutive 8-bit characters over an array can lead to a 4:1 reduction in the number of memory accesses using 32-bit memory transfer words. The analysis and implementation of this optimization is described in [5].

3.8 Code Emission

Given a loop nest with a matching or searching computation the compiler generates a parallel hardware implementation. Currently the compiler generates a structural VHDL design using a set of predefined building blocks. These blocks, described behaviorally, include tapped delay lines, comparators and priority encoders. The compiler maps the internal datapath representation to structural VHDL using these predefined blocks and links them to predefined and parameterizable control blocks and memory interfaces. The structural VHDL is then merged with vendor provided libraries that defined the actual memory interfaces. The final design can then be simulated and synthesized.

While the current designs work for a specific vendor logic synthesis, we have neither relied nor exploited any vendor specific VHDL style or library implementations. Clearly the data gathered by the compiler analyses described here would allow the compiler to exploit FPGA library component features by adequately choosing among a myriad possible implementations customized for each target architectures.

3.8.1 Priority Encoder vs. Cascading Multiplexors

Of particular importance for the handling of the search and index functions is the priority encoder module. This modules detects the lexico-graphical position of breaking conditions and can be implemented either as a monolithic block with multiple index entries, one per break condition in the loop, or by sequentially cascading multiplexors values corresponding to the exit values of each break condition. Figure below illustrates two possible implementations.

We found that the synthesis tool we have used in our experimental results, however, favors the cascading multiplexors implementation. We attribute this behavior to the fact that the monolithic implementation of the priority encoder inherently contains a sequential cascading of the conditions in the selection process. For significantly large number of inputs, which occurs due to the unrolling factor this monolithic implementation critical path can be described by the expression $T = A_1 \log(n+1)$. For the cascading implementation however the critical path is simply given by $T = A_2 (n+1)$. The FPGA synthesis tool used in our experiments yields best results for the cascading implementations, which we attribute to the fact that the tools can map better cascading behavioral if statements to the reconfigurable fabric. For other tools and target devices, however, results may vary.

At the time of this writing, the compiler automatically generates complete VHDL description that implement the computation of the body of the loop. However the generation of the complete design and loading onto an FPGA board has not been fully automated.

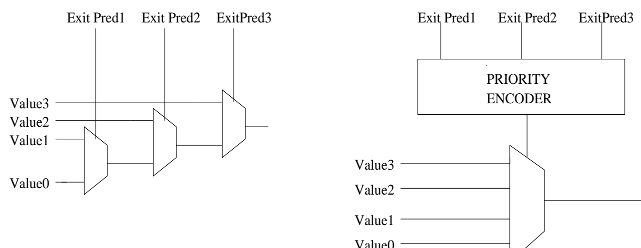

Figure 15. Monolithic vs. Cascading Multiplexor PE Implementations.

4. EXPERIMENTAL RESULTS

We now present results for the compiler analysis and automatically generated designs using a set of kernel codes.

4.1 Methodology

We used a set of complete kernels written in C for the evaluation of the compiler analysis and compiler generated designs. The applications are compiled using the SUIF v1.1.2 *scc* distribution tool. We then used the compiler algorithms described in Section 3 (approx. 8,000 lines of C++) to analyze and select a particular implementation for the loop nests of interest in each application. We then used the Xilinx Foundation Series V2.1i tool package to generate logic design data and bitstream files.

We report results for the logic synthesis running on a Pentium III PC running at 800 MHz and with 256 Mbytes of memory for the generation and evaluation of the compiler generated designs. In our synthesis experiments, we used the Virtex 1000BG560 FPGA series and a low effort, optimized for speed place-and-route (P&R) settings and a timing grade of ‘-6’. The timing results we report are extracted from the generated log files with timing analysis with complete (100%) path coverage.

4.2 Kernels

We now describe the set of C kernel programs used in our performance evaluation of the generated hardware designs. While these kernel computation appear to be simple the unconditional jumps in the presence of *break* statements poses a challenge to the application of traditional analysis techniques. Also, and although subtle, some kernels differ in the respect that the resulting computation does not depend on the actual iteration the inner loop exits but on the fact that it exists (matching versus searching).

Kernel 1 - Lookup Character Index in String

```
pos = -1;
for(i=0; i < SIZE; i++){
  if(str[i] == ch){
    pos = i;
    break;
  }
}
```

Kernel 2 - Search Character in 1D String

```
found = 0;
for(i=0; i < SIZE; i++){
  if(str[i] == ch){
    found = 1;
    break;
  }
}
```

Kernel 3 - String Equality Compare

```
res = 1;
for(i=0; i < SIZE; i++){
  if(a[i] != b[i]){
     res = 0;
     break;
  }
}
```

Kernel 4 - Lexicographical Precedence over Strings

```
result = 0;
for(i=0; i < SIZE; i++){
  if(a[i] != b[i]){
    result = (a[i] - b[i]);
    break;
  }
}
```

Kernel 5 - Pattern Recognition over String

```
for(i = 0; i < STRING_SIZE _ PATTERN_SIZE; i++){
  r = 1;
  for(j=0; j < PATTERN_SIZE; j++){
    if(pat[j] != str[i+j]){
      r= 0;
      break;
    }
  }
  res[i] = r;
}
```

4.3 Results

We now describe the performance results of our code generation strategy and quantify the simulated performance of the resulting generated designs. We begin this discussion by presenting the compilation and synthesis metrics.

Table 1 presents the compilation and synthesis results. For each of the kernels, we report the result of the compiler operator analysis and unrolling dimensions. Next we report the size of the generated structural VHDL source code, the number of distinct components and instances used. We also report on the compilation analysis and synthesis speed.

	Compiler Analysis		VHDL Code Metrics			Analysis & Synthesis Time		
Kernel	Operator	Unrolling	Code Lines	Num Comps	Num Inst	Analyzes Time	Emit Time	Synthesis Time
1	index	1D	390	5	28	<1 sec	<1 sec	75 sec
2	search	1D	390	5	28	<1 sec	<1 sec	75 sec
3	comp	1D	530	6	49	<1 sec	<1 sec	100 sec
4	comp	1D	400	6	49	<1 sec	<1 sec	115 sec
5	match	1D	255	4	12	< 1sec	1< sec	75 sec

Table 1: Compilation and synthesis results.

Table 2 shows the simulated performance results for the generated designs. It includes the overall simulated clock speed, the number of flip-flops and latches used as well as the number of LUTs and equivalent gate counts. Finally we report on the area used by the P&R tool for Virtex1000BG560 parts.

Kernel	Core Datapath Clock (MHz)	Number FF & Latch	Number LUTs	Equiv. Gates	Slices (CLBs)	Virtex1000 Area
1	42.1	112	295	2,930	150	1%
2	40.0	112	260	2,720	132	1%
3	54.3	192	340	3,840	186	1%
4	37.2	184	552	5,048	289	2%
5	46.3	129	166	2,028	102	2%

Table 2: Simulated target designs performance metrics.

4.4 Discussion

Table 2 shows that the generated kernel designs attain respectable clock rates. These results reveal the compiler is able to identify the opportunities of the parallelism across the conditional statements and properly generate complete VHDL designs automatically. The generated designs for all kernels are synthesized and routed fairly quickly due to their relative small size.

Since the compiler exploits aggressive loop unrolling for data reuse, the overall number of data memory accesses is substantially reduced. However, for the cases where the compiler unrolls loops that manipulate 2D array accesses and organizes the tapped delay lines as 1D delay lines, the number of data accesses per iteration can increase substantially. For example, the compiler unrolls the two inner most loops of the kernel, generating a large number of tapped delay lines which must be fed at each evaluation of the parallel hardware. This increased memory access rate can create an imbalance with the computation rate.

Clearly the compiler must negotiate a trade-off between the generated hardware memory access rate and the computation execution rate. We are currently investigating effective approaches to this problem, in particular by coupling the information the compiler extracts to higher-level compiler transformations such as partial unrolling and loop tiling. For our experimental set up, the compiler has always assumed the target FPGA area available was always enough to implement a fully unrolled version of a loop nest.

5. RELATED WORK

Markus Weinhard and Wayne Luk describe approaches to map generic computations in loop nests to hardware [11,12]. They address the specific issues of data dependences, memory mapping of arrays for the pipelined execution of the computations but have not addressed the data reuse and the presence of irregular control flow in the form of *break* and *continue* statements.

Research in the area of parallelizing compilers has targeted specific classes of kernel computations such as array reduction analysis [8,13]. These approaches focus on associative and commutative operators to exploit replication as the basic mechanism to reduce cache coherence traffic and increase parallelism for multiprocessors.

Other researchers [4,6,7] have addressed the issues of automatically mapping computations to FPGA-based computing engines by developing specific languages to facilitate the translation of high-level programs to hardware. An alternative route, as pursued by Banerjee et. al. [2] is to focus on predefined library functions and map them directly to hardware. Although these approaches yields the highest benefit in the short term, they are dependent on either a new language programmers must master or on the semantics of the library function for correctness.

Our work differs from these efforts in several aspects. First, we aim to detect opportunities for parallel hardware evaluation in a source code independent format - hence free of input language constraints and library semantics. Our approach focuses only on specific kernel computations, that due to control flow, cannot be easily handled by traditional data dependence analysis. Furthermore, we combine data value analysis and loop unrolling to expose more instruction level parallelism and data reuse.

6. CONCLUSIONS

We developed and successfully implemented a compiler analysis algorithm to identify and transform a selected range of matching and searching kernel computations. We have combined in our analysis loop unrolling to expose loop level parallelism and data

reuse so as to reduce the number of memory accesses. These kernels are very challenging as they contain break and continue statements that hamper traditional compiler data dependence analysis. Our results show the generated designs attain good performance for automatically generated hardware implementations for an FPGA-based target implementation.

7. BIBLIOGRAPHY

[1] A. Aho, R. Sethi and J. Ullman, “Compilers: Principles, Techniques, and Tools”, Addison-Wesley, 1986.

[2] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky, “A MATLAB Compiler for Distributed Heterogeneous Reconfigurable Computing Systems”, Int. Symp. on FPGA Custom Computing Machines (FCCM’00) IEEE Computer Society Press, Los Alamitos, Calif., 2000.

[3] S. Carr and K. Kennedy, “Scalar Replacement in the Presence of Conditional Control Flow", In Software - Practice & Experience 24(1), Jan. 1994

[4] D. Cronquist, P. Franklin, S. Berg and C. Ebeling, “Specifying and Compiling Applications for RaPiD”, In Proc. IEEE Symp. on FPGAs for Custom Computing Machines (FCCM’98), IEEE Computer Society Press, Los Alamitos, Calif., 1998, pp. 116-125.

[5] P. Diniz and J. Park, “Automatic Synthesis of Data Storage and Contol Structures for FPGA-based Computing Machines”, In Proc. IEEE Symp. on FPGA Custom Computing Machines (FCCM’00) IEEE Computer Society Press, Los Alamitos, Calif., 2000.

[6] M. Gokhale and J. Stone, “Napa C: Compiling for a Hybrid RISC/FPGA Architecture”,In Proc. IEEE Symp. on FPGAs for Custom Computing Machines (FCCM’98), IEEE Computer Society Press, Los Alamitos, Calif., 1998, pp. 126-135.

[7] M. Gokhale and B. Schott, “Data Parallel C on a Reconfigurable Logic Array”, Journal of Supercomputing, 9(3):291-313, 1994.

[8] “The Stanford SUIF Compilation System”. Public Domain Software and Documentation available at http://suif.stanford.edu.

[9] M. Wolf and M. Lam, “A Loop Transformation Theory and an Algorithm for Maximizing Parallelism”, IEEE Trans. on Parallel and Distributed Systems, Oct., 1991.

[10] M. Wolfe, *High-Performance Compilers for Parallel Computing*, Addison-Wesley, 1996.

[11] M. Weinhardt and W. Luk., “Pipelined Vectorization for Reconfigurable Systems”, In Proc. IEEE Symp. of FPGAs for Custom Computing Machines (FCCM’99), IEEE Computer Society Press, Los Alamitos, Calif., 1999, pp. 52-62.

[12] M. Weinhardt and W. Luk, "Memory Access Optimization and RAM Inference for Pipeline Vectorization”, In Proc. of the 9th Intl. Workshop on Field Programmable Logic and Applications (FPL'99), Springer Verlag LNCS Vol. 1673, 1999, pp. 61-70.

[13] A. Fisher and A. Ghuloum, “Parallelizing Comlex Scans and Reductions”, In Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’94), ACM Press, New York, 1994, pp. 135-146.

[14] M. Rinard and P. Diniz, “Commutativity Analysis : A New Analysis Framework for Parallelizing Compilers”, In Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’96), ACM Press, New York, 1996.

[15] S. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. Taylor and R. Laufer, “PipeRench: A Coprocessor for Streaming Multimedia Acceleration”, In Proc. of 26th Intl. Symp. on Computer Architecture (ISCA’99), ACM Press, New York, 1999.

[16] J. Hauser and J. Wawrzynek, “Garp: A MIPS Processor with a Reconfigurable Coprocessor”, In Proc. IEEE Symp. on FPGAs for Custom Computing Machines (FCCM’97), IEEE Computer Society Press, Los Alamitos, Calif., 1997.

[17] S. Hauck, T. W. Fry, M. M. Hosler, J. P. Kao, "The Chimaera Reconfigurable Functional Unit", IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, Los Alamitos, Calif., 1997, pp. 87-96.