Exploiting Program Branch Probabilities in Hardware Compilation

Henry Styles and Wayne Luk

Abstract—This paper explores using information about program branch probabilities to optimize the results of hardware compilation. The basic premise is to promote utilization by dedicating more resources to branches which execute more frequently. A new hardware compilation and flow control scheme are presented which enable the computation rate of different branches to be matched to the observed branch probabilities. We propose an analytical queuing network performance model to determine the optimal settings for basic block computation rates given a set of observed branch probabilities. An experimental hardware compilation system has been developed to evaluate this approach. The branch optimization design space is characterized in an experimental study for Xilinx Virtex FPGAs of two complex applications: video feature extraction and progressive refinement radiosity. For designs of equal performance, branch-optimized designs require 24 percent and 27.5 percent less area. For designs of equal area, branch optimized designs run up to three times faster. Our analytical performance model is shown to be highly accurate with relative error between 0.12 and 1.1×10^{-4}.

Index Terms—Automatic synthesis, dataflow architectures, queuing theory.

1 Introduction

Reconfigurable architectures such as Field Programmable Gate Arrays exhibit two properties. First, reconfigurable architectures permit rapid specialization through the mechanism of runtime reconfiguration. A popular modern FPGA such as the Xilinx Virtex-II can be reconfigured in approximately 20-50ms and other coarse-grain architectures [1], [3] can be partially reconfigured in tens of cycles. Second, reconfigurable architectures present a massively extended design space and permit a higher degree of specialization when compared to software. These properties point toward reconfigurable systems which exploit information on program behavior and specialize operations both before and during execution.

Such systems differ from conventional fixed designs in two ways.

1. First, design tools must be capable of generating multiple designs, ideally from a single high level source, which are specialized for different program behavior patterns.
2. Second, a management system must be devised which monitors program behavior and initiates appropriate actions to maintain design specialization over time.

This paper contributes to tackling both of these challenges. The primary contribution of the work is toward meeting challenge 1. A suitable basis for specialization is identified in program branch probabilities. A consequence of the property of program phase [7] is that branch probabilities can be predicted from historical measurement. In computer programs which include flow control, the execution frequency of each basic block is controlled by the probabilities of the interconnecting conditional branches. Branch probability information can therefore be used to specialize operations by guiding the allocation of resources between different basic blocks. This principle is used to specialize operations in several aspects of existing computer systems. Software profilers collect accurate execution profiles. This information is used by software engineers to target optimization efforts toward computational bottlenecks. Embedded systems engineering tools use execution profiles to adjust hardware allocation to program behavior [18]. High-level synthesis tools make use of profiling information to optimize the partitioning and optimization of multiprocess hardware systems [36]. Multiconfiguration microprocessors [9] collect branch probability information at runtime. The information is used to aid dynamic resource allocation by specializing aspects of the architecture such as cache line size.

This paper explores the analogous use of branch probability information to optimize resource allocation in hardware compilation. The scheme proposed in this paper differs from the systems mentioned above in that it optimizes a fundamental building block of complex reconfigurable computing systems: hardware pipelines. A compilation scheme is proposed that automatically maps programs written in a subset of C to a set of hardware designs which are optimized for different branching probabilities. Our compilation scheme employs a tag-token and is closely related to dataflow machine architectures. This control flow mechanism confers a suitable mechanism for design specialization by allowing different basic blocks to operate at different rates. Resource allocation is specialized on the basis of branch probability information by slowing down underutilized blocks and using the saved resources to speed up blocks which are bottlenecks. The effectiveness of the proposed compilation path is explored in experiments involving two large case study applications: video feature extraction and radiosity.

• *The authors are with the Department of Computing, Imperial College London, 180 Queen's Gate, South Kensington Campus, London SW7 2AZ, UK. E-mail: {hes2, wl}@doc.ic.ac.uk.*

Manuscript received 1 Dec. 2003; revised 28 June 2004; accepted 1 July 2004. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-0250-1203.

A property of our tag-token scheme is the ability to parallelize and pipeline nested loop carry dependencies. A loop carry dependency is a loop variable which is carried across loop iterations. Loop carry dependencies are currently a barrier to both the quality and applicability of reconfigurable systems—to the authors' knowledge, no existing hardware compilation system is able to parallelize and pipeline this class of operations. The benefits of our new approach are illustrated through the examples of fractal set generation and Newton-Raphson equation root finder.

A further contribution of this work is in management of specialization (challenge 2). A system to manage and control specialization over time would consist of 1) collecting program behavior patterns, 2) predicting future program behavior patterns, and 3) scheduling a series of optimal reconfigurations to preserve specialization. The problem of collecting and predicting branch probability information (subtasks 1 and 2) in hardware has been considered by other authors [9] and is beyond the scope of this paper. This paper contributes to the problem of determining an optimal reconfiguration schedule given information on future program behavior (subtask 3). For this optimization task to be completed successfully, the performance of the different possible configurations must be compared for the period of predicted program behavior. This paper introduces an analytical method, based on queuing networks, for elucidating the properties of the proposed compilation procedure. The modeling system is capable of delivering fast and accurate performance estimates for the different designs and is suitable for runtime operations in such a management system.

The rest of the paper is organized as follows: Section 3 describes our basic framework for hardware compilation, consisting of two compilation phases: dependency analysis and circuit synthesis. Section 4 then describes the branch-optimized compilation path and explains how designs are specialized for different branch probabilities. Section 5 describes how our compilation path copes with feedback systems. Section 6 deals with the models for studying the analytical properties of this compilation procedure and is followed by Section 7 which describes three experiments to test the three central contributions of the paper. The first experiment assesses the effectiveness of the branch-optimized compilation path against an existing system and control experiment which do not consider branch probabilities. The second experiment illustrates the effectiveness of the proposed scheme in pipelining and parallelizing a class of loop carry dependencies. The third experiment evaluates the effectiveness of the proposed analytical model in deriving accurate performance predictions for compiled designs. Section 8 examines the results of these experiments. Finally, Section 9 summarizes our current and future research.

2 Related Work

Hardware compilers are increasingly recognized as the key to raising the productivity of hardware designers. Many hardware compilers have been reported recently and they cover various source descriptions including C [4], Handel-C [2], Streams-C [35], Matlab [12], StReAm [23], Cantata [26], and Java [13]. These systems compile a single design which is optimized for all runtime conditions.

A second class of related design tools and methods is devoted to specialization of reconfigurable circuits at runtime. These tools focus on enabling large fixed designs to be accommodated in small devices [19], [30] and can also permit improved performance by constant folding [5]. A more general structural model for dynamic reconfiguration is presented in [5]. Structural design environments have been extended to generate partially reconfigurable cores [21], [31], however, there has been limited work into synthesizing and controlling reconfigurable design sets from behavioral descriptions [20].

Section 4 describes our tag-token flow control scheme. The mechanism for flow control and synchronization in related hardware compilers ranges from nonblocking scheduling [23], [26] to blocking protocol [4], [2], [12], [13]. Our flow control scheme uses blocking as in [4], [2], [12], [13], but also employs a tag-token as seen in dataflow machines architectures [33] and rate-smoothing FIFOs from queuing network literature [24].

The queuing network model proposed in Section 6 is based on simple, analytically solvable Markovian queuing networks [24], [15]. In the past, queuing theory has been applied in the modeling of various aspects of computer systems, including computer networks [16], and in performance models of time-sharing and multiaccess systems [16], [24]. The closest related application to that described in this paper is in processor-memory performance modeling, for uniprocessor [11] and multiprocessor [10] systems. Research into performance modeling in the context of hardware compilation has concentrated on static models of execution time without branch probability information. In [6] and [14], Cottet et al. propose a static method for area and performance estimation of regular and data-intensive FPGA circuits used for applications such as multimedia, telecommunications, and cryptography. Park et al. [29] propose an analogous set of static performance estimates for reconfigurable designs under optimizing loop transformations.

3 Compilation

Our compilation procedure consists of two phases: dependency analysis and circuit synthesis.

The input language for the compiler is a streaming subset of the C language in which access to memory by arbitrary pointers is not supported. Each input program specifies the body of a single loop, with flow control specified by *if..then..else* and *do..while* nested loop constructs.

A simple example program, shown on the left of Fig. 1, will be used to illustrate our compilation procedure in the following sections.

The dependency analysis phase constructs a two-level data flow graph from the input program. The data flow graph for our simple example program is shown on the right of Fig. 1. It includes a numbered direct acyclic graph (DAG) for each basic block. Flow control between DAGs is represented by BRANCH and MERGE nodes with firing rule semantics as described in the data flow computing literature [33]. Reads and writes to vector variables at the start and end of the data flow graph are mapped to READ and WRITE nodes.

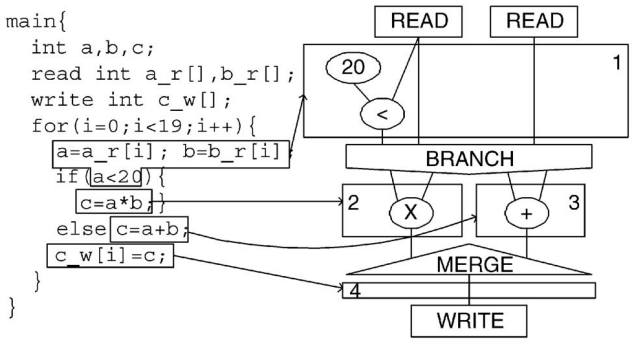


Fig. 1. A simple example program and its two-level data flow graph. The four basic blocks map to four numbered DAG subgraphs. The *if..then..else* maps to BRANCH and MERGE nodes. Array accesses map to READ and WRITE nodes.

The circuit synthesis phase transforms the dependency graph into a unidirectional pipeline captured in structural VHDL. It consists of module selection, scheduling, binding, and instantiation of appropriate flow control circuits. Common facilities are provided for circuit synthesis. The *initiation interval* or *period* of a library block is equal to the number of cycles between each successive output. The term *initiation interval* is commonly used in literature on software pipelining [34]. An XML library block database specifies the initiation interval, latency in cycles, and area of available library blocks. A static pipelined list scheduler [8] is provided for basic block scheduling.

Circuit synthesis is specialized to form two compilation paths: the control study compilation path and the branch-optimized compilation path. The control study compilation path is inspired by the StReAm [23] compiler. It creates pipelines which perform equally well under all branching conditions. Designs are parameterized with a global initiation interval parameter $b_{pipeline}$, which sets a uniform rate of processing for all basic blocks in the pipeline. The control study compilation path circuit with $b_{pipeline} = 1$ for our simple example program is shown on the left of Fig. 2. Under the control study compilation scheme, the BRANCH node simply fans data out to both branch target basic blocks. The true and false branch target DAGs, labeled "2" and "3" in Fig. 2, are scheduled independently with the initiation interval set at $b_{pipeline}$.

In Fig. 2 after scheduling, the true branch target basic block, containing the multiplier, has a latency of three cycles and the false basic block, containing the adder, has a latency of one cycle. A second scheduling pass adds register chains to the false basic block so that processed data and the conditional result arrive simultaneously at the MERGE node. As a result of this scheduling pass, all paths through the pipeline have equal latency. An extra register chain is created to propagate the conditional result across the branch target computation. The MERGE node then simply multiplexes the correct processed data to their outputs based on the propagated conditional result.

Recall that our current prototype compiler does not support access to memory by arbitary pointers. The aim of this restriction is to preclude programs which manipulate large amounts of internal state. Our hardware compilation scheme generates pipelines. As such, and like Dataflow Machines [33] and all pipeline hardware compilation systems, our compilation scheme suffers from limitations in executing programs with large internal state. Each stage in the hardware pipeline must have access to the live variables of its

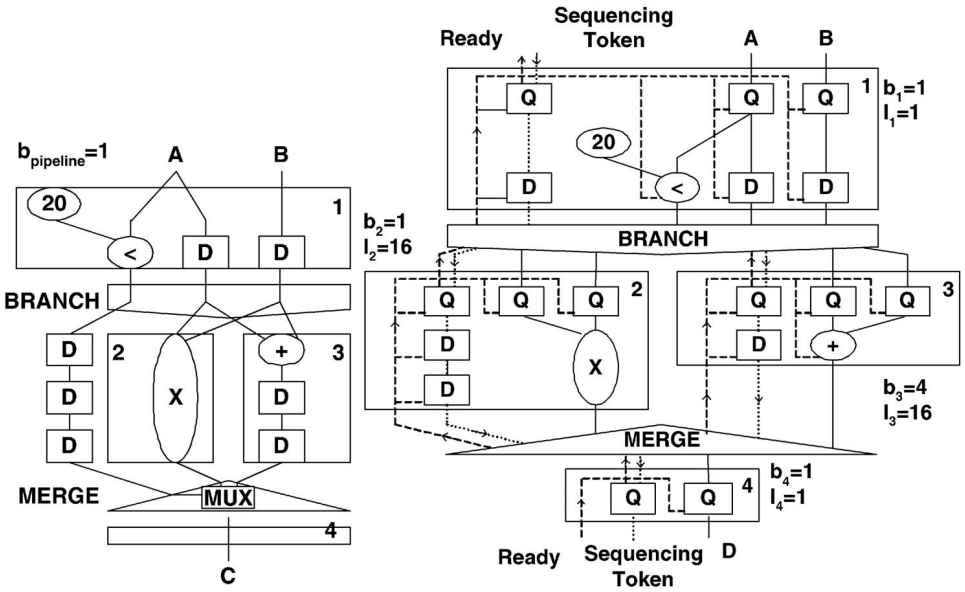


Fig. 2. Uniform (left) and multiple (right) rate circuits for the simple program of Fig. 1. For the control study compilation path, additional flip-flops (D) are inserted to synchronize data flow. For the branch-optimized compilation path, rate-smoothing queues (Q) and sequencing token (dotted line) are inserted to synchronize data flow. The add and multiply modules are pipelined and the flow can be halted by the *ready* control signal (dashed line) shown as an additional input to these components. The uniform rate circuit is annotated with the global initiation interval parameter $b_{pipeline}$. For this example, the parameter is set to run at one result per cycle. The multiple rate circuit is annotated with initiation interval parameter b_i and FIFO length parameter l_i for each basic block. This example is a multiple rate system in which resources are weighted to optimize the left path through the branch, with resources saved on the right path through the branch. All basic blocks, except block three, are set at a initiation interval (b_i) of one result a cycle. FIFO length parameter l_i is set manually to deliver maximum tolerance to burstyness at the branch target basic blocks ($l_2 = 16$ and $l_3 = 16$).

operations, hence, for concurrent operation of all pipeline stages, the complete set of live variables must be present at each pipeline. If a program has excessive internal state, the total cost of storing the live variables at each pipeline stage becomes prohibitive. Currently, we do not provide a locking and arbitration system for sharing state between pipeline stages and, therefore, in this instance, an alternative sequential system would be more appropriate.

Our prototype compilation system does not currently include a system of automatic source to source optimizing transformations. Optimizations involving code motion [18], [27] such as speculation [8], software pipelining, vectorization [34], strip mining, and loop factorization must be manually applied to the input program. In the future, we intend to augment our compilation system with an SUIF [17] front end to automate these tasks.

4 Branch-Optimized Compilation Path

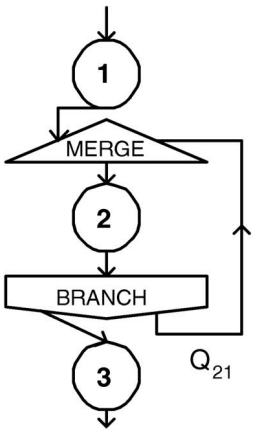
In this section, we introduce a new compilation scheme which promotes efficiency in the presence of branch probability information. A branch-optimized circuit for the simple example program is shown on the right of Fig. 2. The branch-optimized compilation scheme transforms the data flow graph into a set of hardware configurations in which different basic blocks run at different initiation intervals. From this set, a configuration can be chosen in which the resources assigned to different branches match the observed computational load. The branch-optimized compilation scheme creates designs with the following characteristics.

1. Basic blocks can run at independent rates, with different degrees of sequentialization. The scalable rate of computation is the basis for performance gains and area savings throughout this work. A basic block with an initiation interval of one result a cycle is fully pipelined. A basic block with an initiation interval of one result every two, four, or eight cycles contains sequential library blocks which consume less area than the fully pipelined versions. For example, a simple 32-bit array multiplier, running at an initiation interval of one result per cycle, requires 32 adder blocks. A sequential multiplier, running at an initiation interval of one result every two cycles, can be constructed using 16 adder blocks. The rate of basic block i is controlled by the initiation interval parameter b_i.
2. Each basic block propagates a sequencing token downward, shown as a dotted line in Fig. 2. Different paths through the pipeline run at different rates and, so, computations may retire out of order. The sequencing token identifies the loop index associated with a set of results at the pipeline outputs, enabling the original ordering to be recovered. Typically, this is accomplished by writing the results to a memory. When a result exits the pipeline, the token value is used to control the address of a write to memory in which the result data are written to a memory bank. Over the course of execution, the memory bank is filled out of order. At the end of execution, the full set of results is available in order as defined by the memory addresses. The width of the sequencing token is determined from the upper loop bound and maximum length path through the compiled design. The sequencing token implemented in the branch-optimized compilation scheme is inspired by tagged-token [33] flow control in the dataflow architectures of the 1980s and 1990s. The implication on dependencies, principally concerning pointer memory access, of the out-of-order scheme is discussed in detail in Dataflow Machine literature [33] and is a motivating factor in the design of our restricted language syntax. It is used to recover the original ordering of computation, given that computations may retire out of order having traveled different rate paths through the pipeline.
3. Basic blocks have rate smoothing FIFOs, labeled Q in Fig. 2, for the sequencing token and data inputs. The FIFO length for basic block i is given by parameter l_i. The purpose of these FIFOs is to smooth out variations in arrival rate and bursty arrivals caused by periods of biased branching.
4. Each basic block propagates a ready signal back up through the pipeline, shown as a dashed line in Fig. 2. The ready signal allows basic blocks to stall incoming computation when input queues are full. For each basic block, the incoming ready signal fans out to the clock-enable input of all registers in the datapath. The ready signal is pipelined on entry to each basic block by oversizing the basic block rate smoothing FIFOs. This has the effect of localizing the ready signal to each basic block, minimizing the delay in combinatorial feedback. Our experiences in compiling circuits with this control method, presented in Section 8, confirm that the proposed control method has a limited impact on overall clock rate. In our current implementation, we adopt a fully synchronous design style. However, a globally asynchronous locally synchronous (GALS) design style could potentially be adopted in which each basic block operates in a separate clock domain and the ready signal is replaced with true asynchronous handshaking.
5. The BRANCH node routes sequencing token and data to the branch target specified by the branch condition. It receives ready signals from the two branch targets and blocks computation if the branch target set by the branch condition is not ready.
6. The MERGE node forwards data and sequencing tokens from true and false branch targets. If sequencing tokens arrive from both branch targets simultaneously, the MERGE node blocks the branch targets alternately in a round-robin fashion.
7. Resource sharing can occur between two mutually exclusive nodes, guarded by a BRANCH and MERGE pair, and with the same initiation interval parameter b_i and FIFO length l_i.

The experimental system generates a spectrum of hardware configurations which are specialized for different branch probabilities. The FIFO length parameter l_i for each

```
void main(void){
  int count,x,y,savex,savey;  //scalars
  int savex2,xsquared,ysquared;
  read int rx[],ry[]; //vectors
  write int fract[];
  for(i=0;i<2^20;i++){
    // setup
1   x=rx[i]; y=ry[i];
    savex=x; savey=y; count=0;
    do{
      // mandelbrot kernel
      savex2=x;
      xsquared=x*x;  ysquared=y*y;
2     x=(xsquared-ysquared)+savex;
      y=((savex2*y)<<1)+savey;
      count=count+1;
    }while( (xsquared+ysquared) <= 83886080 ) && ( count <= 256 ) ) ;
3   fract[i]=count;
  }
}
```

(a)



(b)

Fig. 3. Compiling nested loop carry dependencies in the Mandelbrot fractal generator. (a) Input program. (b) Top-level data flow graph. The loop carry inner loop in (a) corresponds to the feedback path in the data flow graph.

basic block in the design spectrum is set manually after compilation. FIFO length parameter l can be manually shaped to minimize the possible impact of burstyness on performance; however, automatic control of the FIFO queue length is beyond the scope of this paper. The basis of specialization between different configurations is the processing rate, or initiation interval b_i, of the individual basic blocks. The set of initiation interval parameterizations b in the spectrum of designs is determined through the following three steps:

1. Determine a set of valid initiation interval parameterizations b_i for each basic block. This is achieved by enumerating the initiation interval range of the contained library blocks, as specified in the library block database.
2. Enumerate valid initiation interval combinations. The set of initiation interval parameterizations for the complete design is formed from the combinations of the valid initiation intervals for each basic block.
3. Cull inefficient initiation interval combinations by applying static flow heuristics. Certain initiation interval combinations can be statically shown to be suboptimal. A design is eliminated if a BRANCH node exists where either output is of greater rate (lower initiation interval) than the input or a MERGE node exists where either input is of greater rate (lower initiation interval) than the output.

5 Pipelining Nested Loop Carry Dependencies

A loop carry dependency is defined as a true data dependency which crosses loop iterations. In programs containing for and while loops, loop carry dependencies are introduced when data are carried across loop iterations. Loop carry dependencies restrict the degree of parallelism which can be expressed through hardware compilation. The key difficulty is that loop carry dependencies force sequential execution of successive loop iterations. In a loop carry dependent loop, iterations n and n-1 cannot be executed by successive issues to a long pipeline as loop iteration and must retire from computation before loop iteration n. In existing hardware compilers, these restrictions preclude parallel execution and pipelining, which are central to achieving speedup over software in reconfigurable hardware. Instead, all existing hardware compilation tools create circuits in which loop iterations are run sequentially.

The tag-token control flow scheme described in Section 4 enables a subset of loop carry dependencies, nested loop carry dependencies, to be parallelized and pipelined in hardware. Compilation of nested loop carry dependencies will be illustrated by two simple examples: Mandelbrot fractal generation and repeated Newton-Raphson method. A fractal is an object or quantity which displays self-similarity on all scales. The Mandelbrot set is a fractal formed by the sequence of complex numbers. The Mandelbrot set is a fractal formed by the sequence of complex numbers

$$z(0) = c, z(n+1) = z(n) * z(n) + c, n = 0, 1, 2, \ldots,$$

where c is an initial coordinate in the complex plane. This can be displayed in graphical form by plotting the number of iterations required to "escape" an absolute bound (or circle about the origin) in the complex plane. The source code and top level dataflow graph for Mandelbrot fractal generation are shown in Fig. 3. The outer loop passes over different pixels, corresponding to different initial values of c. The inner loop-carry loop calculates the Mandelbrot sequence for a given pixel. The repeated Newton-Raphson example refines estimates on the roots of a fifth degree polynomial. It consists of a single nested loop with loop

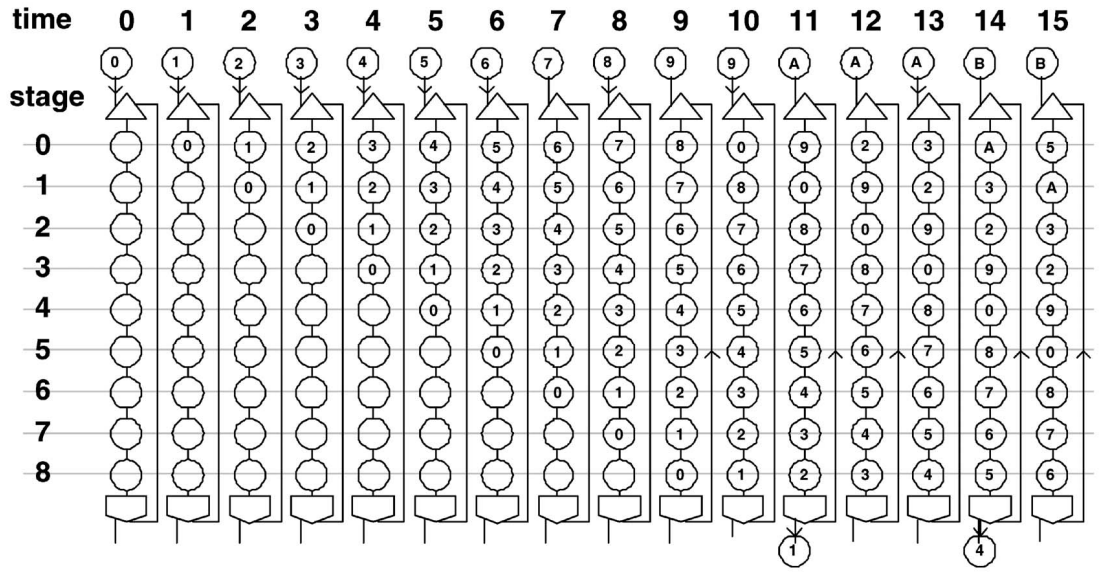


Fig. 4. Internal operations of Reconfigurable Dataflow Mandelbrot fractal generator over time. The dataflow graph for Mandelbrot is shown in Fig. 3a. In the above figure, each column of circles represents the Mandelbrot design at a given time period. Each circle represents a pipeline stage in the Mandelbrot design. Numbered pipeline stages indicate the outer loop iteration number (pixel number) for which calculations are in progress. This utilization pattern shows that, after the pipeline is filled, all pipeline stages are kept active in processing valid data.

carry dependencies that contains 16 multiplies, one divide, and 10 add/subs.

The proposed solution to parallelizing and pipelining recognizes that different iterations of the outer loop can be computed in parallel. In the fractal example, this corresponds to different pixels being calculated in parallel. A single pipelined implementation of the nested loop carry dependent loop is created (basic block 2). Fig. 4 illustrates the progress of tokens over time through the loop carry construct. Successive issues to inner loop pipeline are made until the MERGE block desserts the "ready" counterflow control signal, allowing different iterations of the outer loop (pixels) to be introduced over successive cycles. When an inner loop-carry loop iteration completes, one of two actions occurs. If the escape condition occurs, the result is propagated to one output of the BRANCH node and the output of the loop-carry loop construct. If the escape condition fails, the BRANCH and MERGE nodes recirculate the data to the input of the inner loop pipeline while blocking inputs from a successive outer loop iteration. The utilization pattern shown in Fig. 4 demonstrates that, after the pipeline is filled, all pipeline stages are kept active in processing valid data.

Table 1 shows the results of compilation for Xilinx Virtex series FPGAs, comparing area and speed against traditional sequential execution. For Mandelbrot, when cycle-count and clock period results are combined into total execution time, pipelining and parallelization in the tag token fractal generator accounts for 6.5 times speedup over sequential execution at a cost of 52 percent area. For Newton-Raphson, pipelining and parallelization in the tag token fractal generator accounts for 55 times speedup over sequential execution at a cost of 56 percent area.

6 Analytical Modeling

In this section, we describe analytical models of the area-throughput design space for the control study and branch-optimized compilation paths. These models are used to determine the best compilation path and parameterization from observed branch probability information. In the experimental study presented in Section 7, branch probability information is collected at compile time by profiling. In a future system, branch probability information could be collected and acted upon at runtime. Analytical techniques are of increasing importance, as severe time constraints on the optimization process would almost certainly preclude more complex modeling.

We model the cycle count throughput of branch-optimized designs using a queuing network model. Branch-optimized designs introduce finite queue lengths, blocking, and the possibility of correlated arrival rates. Queuing networks which model these properties are generally solved by simulation [28]. We adopt a simple analytical model based on an $M/M/1/\infty/FCFS$ queuing network with saturating external arrivals to node one [24]. This formulation models the behavior at steady state. The formulation does not model the effects of burstyness, instability, and intermittent blocking. In Section 7, we experimentally demonstrate that these limitations do not excessively compromise accuracy.

The model assumes knowledge of steady state branching probabilities. Given information about steady state branch probabilities, known variables in the model are:

1. The node initiation intervals vector $\vec{b} \in \Re^N$. In the model, element b_i is the exponentially distributed mean initiation interval of node i. b_i is set to the initiation interval of the $i\text{th}$ basic block.
2. The "routing matrix" $Q \in \Re^{N \times N}$. In the model, element Q_{ij} is the steady state probability that a

TABLE 1
Comparative Area-Throughput Results for Sequential and Branch Optimized Compilation Paths with Mandelbrot and Newton-Raphson Examples

Experiment	Area			Clk	To completion		Mean	
	(slice)	(FFs)	(LUTs)	(ns)	(cycles)	(ms)	(cycles/it)	(ns/it)
Sequential Mandelbrot	2120	3538	3356	10.781	50695713	546.55	9	97.029
Branch optimised Mandelbrot	3224	4027	3875	13.952	5632866	78.60	1	13.952
Sequential Newton-Raphson	4337	4915	1921	11.635	3510121267	40840.26	68	791.18
Branch optimised Newton-Raphson	6788	5605	2710	14.351	51623526	740.84	1.00001	14.352

job, completing node i, routes to node j. A BRANCH after node x to select between branch targets y and z is modeled with $Q_{xy} + Q_{xz} = 1$. The summation of probabilities is $q_i = \sum_{j=1}^{N} Q_{ij} \leq 1$, where $i = 1, 2, .., N$. If $q_i < 1$, then a job, on completing node i, exits the queuing network with probability $1 - q_i$. Q is filled with the known branch probability information.

To estimate performance, we determine the maximum sustainable external arrival rate to node one. In the model, the external arrival rates of the different basic blocks are captured in $\vec{\gamma} \in \Re^N$ where the ith element is the Poisson process mean external arrival rate for node i. In our compiled designs, input data from the outside environment arrive at basic block one only. All other blocks have no direct inputs from the outside environment. Hence, the vector γ is of the form $\vec{\gamma} = [\gamma_1, 0, 0, ..]$, where we name the unknown external arrival rate to the system as γ_1. The procedure to determine the maximum sustainable value of γ_1 is as follows:

1. Solve the traffic equations (1) to determine the net arrival rate at each node in terms of the external arrival rate at node one. The mean net arrival at each node is an element in $\vec{\lambda} \in \Re^N$. An equation is formed for each element λ_i in terms of γ_1.

$$\vec{\lambda}(I - Q) = \vec{\gamma}. \quad (1)$$

2. Determine the maximum arrival rate at node one given that the utilization of each node is less than or equal to one. In the model, the utilization of each node is an element in $\vec{\rho} \in \Re^N$. We maximize γ_1 subject to the utilization constraint (3).

$$\rho_i = \lambda_i b_i \quad (2)$$

$$\rho_i \leq 1 \qquad i = 1, 2, .., N. \quad (3)$$

Any design with $\rho_i = 1$, $i \neq 1$ for maximum λ_1 will exhibit steady state blocking.

The control study compilation path is parameterized with the global initiation interval $b_{pipeline}$. Designs sustain throughput $1/b_{pipeline}$ for all branching probabilities.

Given observed stable state branch probabilities and a target pipeline throughput, this model allows us to determine a constraint on the minimum service rate required at each node which will preserve a state of no persistent blocking. In effect, the constraint states which basic blocks can be slowed down and by how much these blocks can be slowed down without affecting the overall performance of the circuit. To minimize circuit area while meeting a circuit speed specification, we lengthen the initiation interval of different basic blocks subject to the minimum service rate constraint. Area savings are made possible as a slow, long initiation interval basic block circuit can be implemented in less logic than a fast, short initiation interval circuit.

7 Case Studies

In this section, we compare the performance of both compilation paths and evaluate the accuracy of analytical models for two case study applications. The input programs and their corresponding top-level data flow graphs for the case study applications are shown in Fig. 6 and Fig. 7. The test scenes are shown in Fig. 5.

Video feature extraction. The algorithm consists of edge detection, thresholding, and 3×3 sum-squared difference. A detailed description can be found in [378], where a multiple-FPGA implementation is reported. There are four basic blocks and one branch.

Progressive refinement radiosity. Radiosity algorithms simulate radiation of energy between surfaces. A full description of the system can be found in [32]. Up to 90 percent of processing time in software radiosity is spent in determining occlusion between surfaces in the environment. We consider hardware occlusion calculations using stochastic ray casting and the Moller-Trumbore ray-triangle intersection [25] test. Each intersection test consists of several vector cross product and vector dot products. There are 10 basic blocks guarded by three branches.

8 Results

For the purposes of the experiments, all designs have a uniform word length of 32 bits. All results use the Xilinx XCV3200E-8 device. Arithmetic library blocks are generated using Xilinx Core GENERATOR 5.1.02i, with $b_i = 1$,

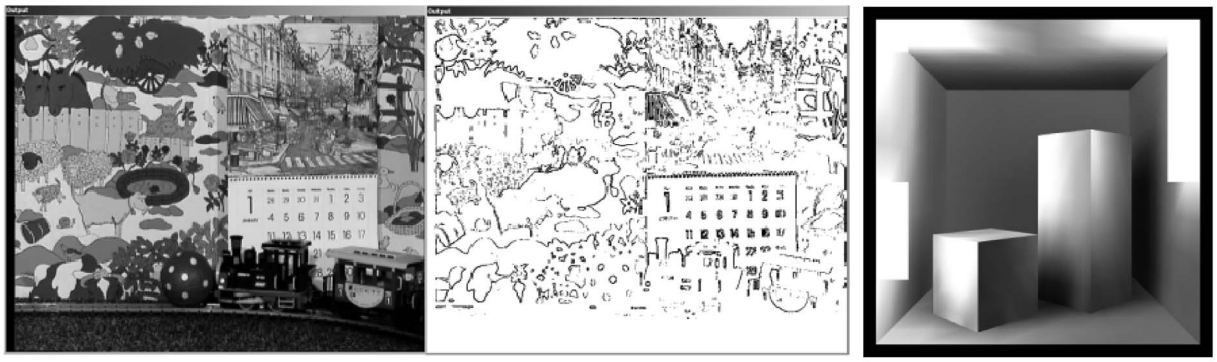


Fig. 5. Left and center panes show Video Quality Expert Group test sequence 10 (VQEG10) before and after video feature extraction. Right pane shows the radiosity test scene.

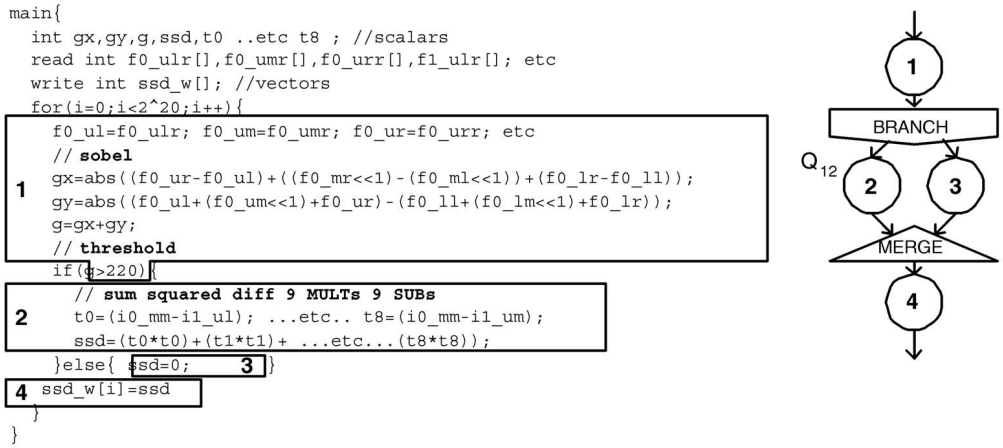


Fig. 6. Input program and the corresponding top-level data flow graph for video feature extraction.

2, 4, or 8 for multipliers and dividers. Other library blocks do not scale for initiation interval. VHDL output by the compiler is synthesized with Synplify Pro 7.1. Area and clock rate are collected from Xilinx 5.1i. Runtime basic block utilization and queue length behavior are observed by simulation using ModelSim SE Plus. A wrapper was constructed in Handel-C [2] to demonstrate designs on the RC1000-PP FPGA platform. The relative accuracy of the analytical model is calculated with the formula $(analytical\ \gamma_1 - observed\ \gamma_1)/observed\ \gamma_1$.

Branch probability information is collected by profiling. The software implementation of the video feature extraction case study is profiled with the test sequence VQEG for 100 frames. Routing table entries relating to Fig. 6 are $Q_{12} = 0.0891$. The software implementation of the radiosity case study is profiled for four refinements, involving approximately 800K ray-triangle intersection tests. Routing table entries relating to Fig. 7 are $Q_{12} = 0.520$, $Q_{23} = 0.164$, and $Q_{34} = 0.367$.

To illustrate the analytical model, the working for the radiosity case study application is given below. The routing matrix entries which relate to computation, as shown in Fig. 7, are

$$Q_{12} = 0.520$$
$$Q_{23} = 0.164$$
$$Q_{34} = 0.367.$$

The solutions to the traffic equations (1) are therefore

$$\lambda_1 = 1/\gamma_1$$
$$\lambda_2 = 0.520 * (1/\gamma_1)$$
$$\lambda_3 = 0.08528 * (1/\gamma_1)$$
$$\lambda_4 = 0.03129 * (1/\gamma_1).$$

The throughput-utilization constraint on each node (3) in terms of the network performance measure γ_1 is therefore

$$b_1 \leq \gamma_1$$
$$b_2 \leq 1.923 * \gamma_1$$
$$b_3 \leq 11.73 * \gamma_1$$
$$b_4 \leq 31.96 * \gamma_1.$$

Maximizing γ_1 for design EB1 in Table 5 with $b = [1, 2, 8, 8]$, we find that the second utilization constraint is limiting and γ_1, the performance of the design, can be estimated at 0.962 inputs processed per cycle.

The analytical and experimental results for both compilation paths and case studies are shown in Fig. 8 and Tables 2, 3, 4, and 5. Fig. 9 illustrates the effects of different probabilities on the performance of both compilation paths for the video feature extraction case study. The key results of the experimental study are as follows:

1. The branch-optimized compilation path automatically identifies the basic blocks that can benefit from

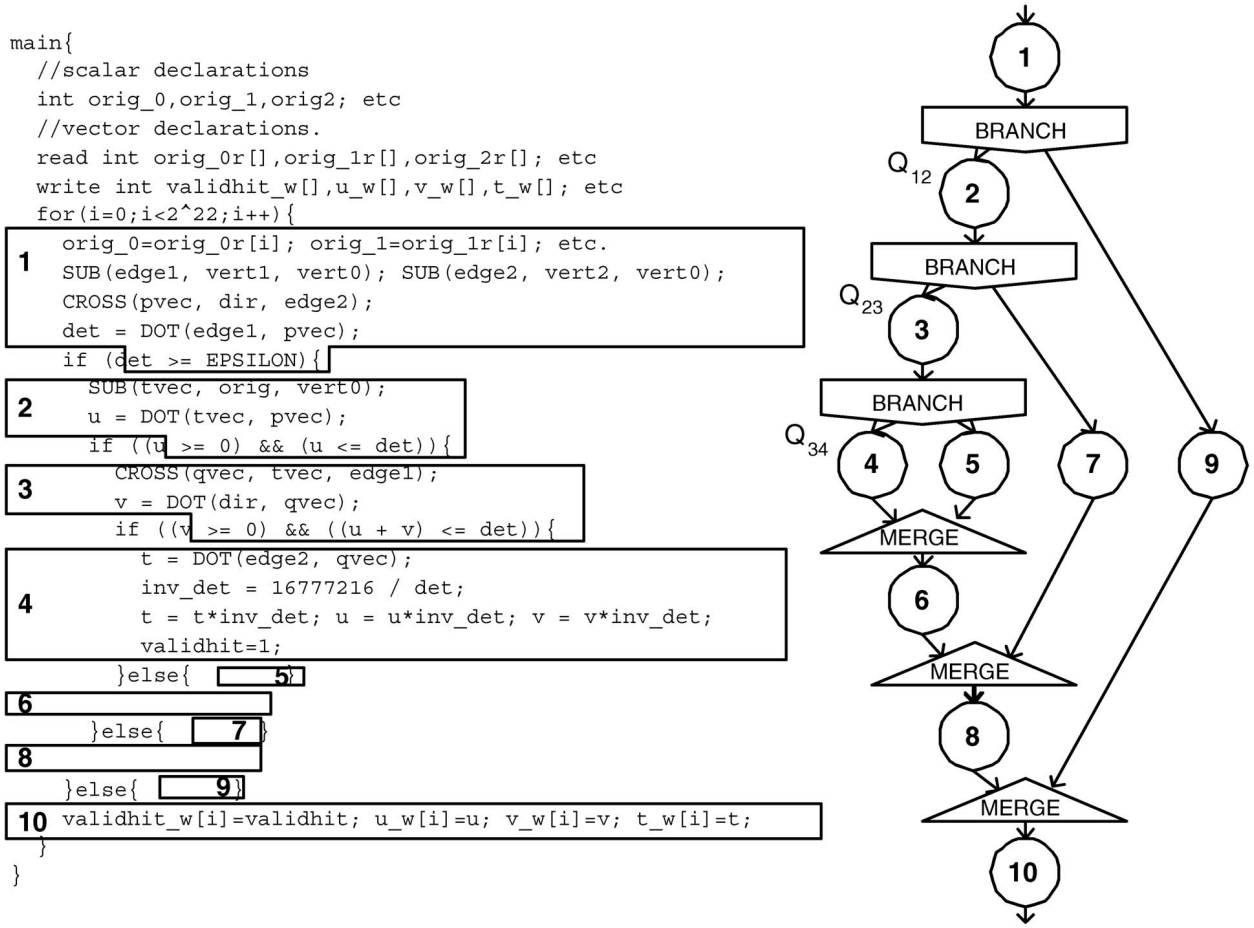


Fig. 7. Input program and the corresponding top-level data flow graph for radiosity Moller-Trumbore ray-triangle intersection. Empty boxes are used to denote basic blocks which contain no computation.

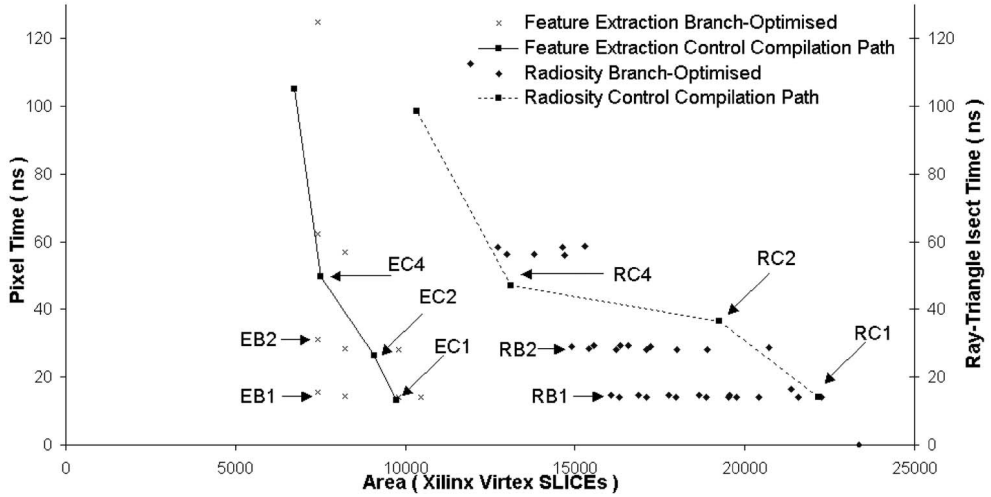


Fig. 8. Combined area-performance design space for video feature extraction (left) and radiosity (right) case study applications. The lines on the graph represent the control study compilation path designs, with $b_{pipeline}$ varying from 1 to 8. The clusters of points are different branch-optimized designs with different parameterizations of b. EC1, EC2, EC4, EB1, EB2, RC1, RC2, RC4, RB1, and RB2 correspond to the optimal designs for each compilation path under different performance constraints as shown in Tables 2, 3, 4, and 5.

branch probability information and produces designs with different parameterizations of b, the initiation interval vector. For the video feature extraction application, the compiler identifies basic block 2 and produces 10 different designs; for the progressive refinement radiosity application, the compiler identifies basic blocks 1, 2, 3, and 4 and produces 35 designs.

TABLE 2
Complete Area-Throughput Design Space with Control Study Compilation Path for Video Feature Extraction Case Study, with Input Scenes Shown in Fig. 5

	Area			Clk	Pixel Time	
$b_{pipeline}$	(slice)	(FFs)	(LUTs)	(ns)	(ns/pixel)	(Mpixels/sec)
1	9753	11137	14623	13.32	13.316	75.10 **EC1**
2	9097	10006	11392	13.23	26.46	37.79 **EC2**
4	7514	7003	8764	12.47	49.86	20.06 **EC4**
8	6733	5405	7469	13.13	105.05	9.52

Designs EC1, EC2, and EC4 are the smallest control study compilation path designs which meet performance constrains 64Mpixel/set, 32Mpixel/sec, and 16Mpixel/sec. These designs are labeled in Fig. 8.

TABLE 4
Complete Area-Throughput Design Space with Control Study Compilation Paths for Radiosity Case Study with Input Scene Shown in Fig. 5

	Area			Clk	Intersection Time	
$b_{pipeline}$	(slice)	(FFs)	(LUTs)	(ns)	(ns/I)	(MIsects/sec)
1	22182	34338	34,823	14.12	14.12	70.84**RC1**
2	19239	28638	25,238	18.27	36.55	27.36**RC2**
4	13116	18852	16,070	11.78	47.13	21.22**RC4**
8	10340	13657	11,809	12.32	98.53	10.15

Designs RC1, RC2, and RC4 are the smallest branch-optimized designs which meet performance constraing 70Mray-triangle intersections/sec, 33Mray-traingles intersections/sec, and 17.5Mray-triangle intersections/sec. RC1, RC2, and RC4 are labeled in Fig. 8.

2. For a given area, branch-optimized designs can often run significantly faster than non-branch-optimized designs. In Fig. 8, for instance, EB1 (7,411 slices) is slightly smaller than EC4 (7,514 slices) and, at 15.61 ns/pixel, is more than 3.2 times faster than EC4 at 49.86 ns/pixel. Similarly, while RB1 and RB2 are, respectively, 22.6 percent and 13.5 percent larger than RC4, they run 222 percent and 62 percent faster than RC4.
3. For a given performance, branch-optimized designs often require smaller areas than non-branch-optimized designs. In Fig. 8, for instance, at 64 Mpixels/sec, EB1 is 24 percent smaller than EC1 and, at 32 Mpixels/sec, EB2 is 18 percent smaller than EC2. Similarly, at 70 Mray-triangle intersections per second, RB1 is 27.5 percent smaller than RC1, while, at 35 Mray-triangle intersections per second, RB2 is 27.5 percent smaller than RC2.
4. The analytical performance model is shown to be accurate. For video feature extraction, the relative error varies between 0.12 and 2.4×10^{-5}; for progressive refinement radiosity, the worst-case relative error is smaller than 1.1×10^{-4}.
5. As the probability of a branch tends towards zero or one, the branch becomes more biased and branch-optimized compilation becomes more attractive. Fig. 9 shows that, for video feature extraction, branch-optimized compilation is favorable if branch probability Q_{12} is below a threshold of $Q_{12} < 0.41$. As Q_{12} tends toward zero, the performance gap between branch-optimized and non-branch-optimized designs increases.

A second control experiment is the unbiased branch-optimized design with equal resource allocation. For each of the case study applications, this is the design generated by the branch-optimized compilation path in which all basic blocks run at an equal initiation interval. These designs are marked as ENB and RNB in Tables 3 and 5. These results enable us to quantify the overheads in terms of area and clock rate of the branch-optimized control scheme over the non-branch-optimized control scheme. Tables 2 and 3 show that, for video feature extraction, a branch-optimized design with uniform initiation interval one result a cycle (ENB) requires 7.3 percent more area and runs at a 6.3 percent slower clock rate than the non-branch-optimized design of initiation interval one result a cycle (EC1). Tables 4 and 5 show that, for radiosity, a branch-optimized design with uniform initiation interval one result a cycle (RNB) requires

TABLE 3
Selected Area-Throughput Resuts for Branch-Optimized Compilation Path in the Video Feature Extraction Case Study with Input Scene Shown in Fig. 5

				Experimental Performance				Analytical Performance			
$\vec{b}$	Area			Utilization		Pixel Time		Utilization		Time	Relative
	slice	FF	LUT	ρ_2	γ_1	(ns/pix)	(Mpix/s)	ρ_2	γ_1	(ns/pix)	Error
1,1,1,1	10458	11594	15477	.089	1	14.17	70.57	.0891	1	14.17	2.4×10^{-5} **ENB**
1,8,1,1	7411	5562	8322	.635	.894	15.61	64.04	.7130	1	13.96	0.12 **EB1**
2,8,1,1	7411	5562	8322	.635	.447	31.22	32.03	.7130	0.5	27.92	0.12 **EB2**

Ten designs are automatically generated. Designs EB1 and EB2 are wth smallest branch-optimized designs which meet performance constraint 64Mpixel/sec and 32Mpixel/sec. Clock period for both designs is 13.96ns. ρ_1 can be calculated as $\gamma_1 \times b_1$. EB1 and EB2 are labeled in Fig. 8. Design ENB is branch-optimized with nonbiased resourse allocation.

TABLE 5
Selected Area-Throughput Results for Branch-Optimized Compilation Path, Radiosity Case Study with Input Scene in Fig. 5

			Experimental Performance					Analytical Performance					
$\vec{b}$	Area	Clk	Utilization				Itime	Utilization				Itime	Relative
	slice	ns	ρ_2	ρ_3	ρ_4	γ_1	ns	ρ_2	ρ_3	ρ_4	γ_1	(ns/I)	Error
1,1,1,1	23370	NA	.52	.085	.031	1	NA	.52	.085	.031	1	NA	NA**RNB**
1,2,8,8	16074	14.09	.99	.657	.241	.96	14.65	1	.654	.240	.962	14.64	5.3×10^{-4}**RB1**
2,4,8,8	14888	14.03	1	.328	.121	.48	29.16	1	.327	.120	.481	29.16	8.1×10^{-5}**RB2**
4,8,8,8,	12723	14.08	1	.164	.060	.24	58.51	1	.164	.060	.241	58.52	8.1×10^{-5}

Thirty-five designs are automatically generated. Designs RB1 and RB2 are the smallest branch-optimized designs with approximate performance 70Mray-triangle intersections/sec and 35Mray-triangle intersections/sec. ρ_1 can be calculated as $\gamma_1 \times b_1$. Processing rate can be calculated as 1/Itime(ns). RB1 and RB2 are labeled in Fig. 8. Design RNB is branch-optimized with nonbiased resource allocation.

5.3 percent more area than the non-branch-optimized design of initiation interval one result a cycle (RC1). These results show that the more complex control scheme adopted in the branch-optimized compilation scheme has limited overhead in terms of resource usage and clock rate.

Our performance gains are supported by three characteristics in the case study applications.

First, the branch probabilities for these applications are biased. Fig. 9 shows that, as branch probabilities become more biased, an uneven weighting of resources between branched becomes favorable. In the video feature extraction case study application, the branch has a probability of $Q_{12} = 0.0891$ and, so, execution is heavily weighted against basic block two. In the radiosity case study application, the combined branch probabilities of $Q_{12} = 0.520$, $Q_{23} = 0.164$, and $Q_{34} = 0.367$ mean that execution is increasingly weighted against execution of basic blocks two, three, and four.

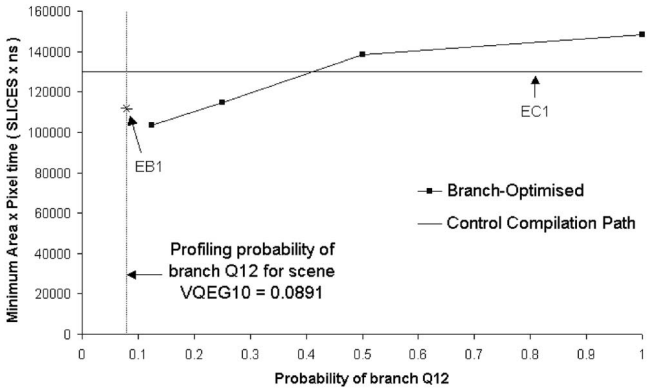


Fig. 9. Minimal area-time versus probability of branch Q_{12} for branch-optimized and control study compilation paths. Video feature extraction case study application is shown with performance constraint of 64Mpixels/sec. The observed probability for Q_{12} of 0.0891 is indicated with a vertical line through the graph. EC1 and EB1 correspond to the optimal designs for each compilation path as shown in Table 2 and Table 3. The trend line for branch-optimized compilation path with different probabilities is produced using our analytical model. The intersection of trend lines for the branch-optimized compilation path and control study compilation path shows that branch-optimized compilation is favorable when $Q_{12} < 0.41$. As the probability Q_{12} decreases, branch-optimized compilation becomes increasingly attractive. EB1 performs worse than the analytical model trend line due to intermittent blocking.

Second, significant computational resources are guarded by branches and those branches are biased against computationally expensive paths. In the video feature extraction case study, the branch is biased against the computationally expensive Sobel filter in basic block two. This enables significant resources to be saved in slowing down this basic block while preserving performance. In the radiosity case study, branches are biased against the computation in basic blocks two, three, and four. These blocks contain the majority of the computation involved in the Moller-Trumbore ray-triangle intersection and significant resources are saved by slowing down these blocks while preserving performance. This effect is boosted by the absence of conditions, as defined in Section 4 for resource sharing between basic blocks guarded by branches.

Third, basic blocks allow scalable performance. Where it is identified that particular basic blocks can be slowed without hindering performance, significant area savings are possible in doing so. In the case study applications, this is principally achieved by changing the degree of sequentialization and pipelining in multiplier circuits.

Clearly, any program which does not exhibit these properties will not stand to gain from branch-optimized compilation. Indeed, Fig. 9 shows that, if branch probabilities are unfavorable, a non-branch-optimized design is more competitive.

9 Conclusion

This paper explores using branch probability information to optimize hardware compilation. We demonstrate that this technique can result in significant improvements in area and performance. Future work will focus on extending the analytical model and compilation system. The analytical model will be extended to estimate the interaction of burstyness with finite buffer lengths. The compilation system will be extended to support array dependency analysis and global dependency analysis techniques. In addition, we wish to investigate compiling programs which exhibit large amounts of mutable state and have interspersed READ and WRITE memory operations. This will require an extension of the flow control scheme, which may

entail a separate memory unit with blocking or an implementation of I-structures [33]. We are currently evaluating the performance of our approach for tasks such as OpenGL rendering, which combine loop carry dependencies and biased branches. In the long term, we intend to develop a dynamically reconfigurable system in which branch optimization techniques are applied at runtime.

ACKNOWLEDGMENTS

The authors would like to thank Danny Lee, David Thomas, Gabriele Figueiredo, Tim Todman, and Shay Seng for their comments and suggestions. The support of Celoxica Limited, Xilinx, Inc., and the UK Engineering and Physical Sciences Research Council (Grant number GR/N 66599 and GR/R 31409) is gratefully acknowledged.

REFERENCES

[1] Elixent d-fabrix and rap, http://www.elixent.com/products/white_papers.htm, 2003
[2] Celoxica Limited, *Handel-C Language Reference Manual,* version 3.1, document number RM-1003-3.0, 2002.
[3] "A Look into Quicksilver's Acm Architecture," http://www.qstech.com/pdfs/a_look_into_quicksilver.pdf, 2002
[4] M. Budiu and S.C. Goldstein, "Compiling Application-Specific Hardware," *Field-Programmable Logic and Applications,* pp. 853-863, Springer, 2002.
[5] W. Luk, N. Shirazi, and P.Y.K. Cheung, "Modelling and Optimising Run-Time Reconfigurable Systems," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines,* 1996.
[6] D. Cottet, R. Enzler, T. Jeger, and G. Tröster, "High-Level Area and Performance Estimation of Hardware Building Blocks on FPGAs," *Field Programmable Logic and Applications,* pp. 525-534, Springer, 2000.
[7] P.J. Denning, "The Working Set Model for Program Behavior," *Proc. ACM Symp. Operating System Principles,* pp. 15.1-15.12, 1967.
[8] G. De Micheli, *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1994.
[9] A.S. Dhodapkar and J.E. Smith, "Managing Multi-Configuration Hardware via Dynamic Working Set Analysis," *Proc. 29th IEEE/ACM Int'l Symp. Computer Architecture,* pp. 233-244, 2002.
[10] A.J. Field and P.G. Harrison, "A Methodology for the Performance Modelling of Distributed Cache Coherent Multiprocessors," *The State-of-the-Art in Performance Modeling and Simulation,* K. Bagchi, J. Walrand, and G.W. Zobrist, eds., pp. 55-92, Gordon and Breach, 1998.
[11] M.J. Flynn, *Computer Architecture: Pipelined and Parallel Processor Design.* Sudbury, Mass.: Jones and Bartlett, 1995.
[12] T. Harriss, R. Walke, B. Kienhuis, and E. Deprettere, "Compilation from Matlab to Process Networks," *Design Automation for Embedded Systems,* vol. 7, pp. 385-403, 2002.
[13] P.A. Jackson, J.L. Tripp, and B.L. Hutchings, "Sea Cucumber: A Synthesizing Compiler for FPGAs," *Field Programmable Logic and Applications,* pp. 875-885, Springer, 2002.
[14] D. Cottet, T. Jeger, R. Enzler, and G. Tröster, "The Performance Prediction Model," technical report, Electronics Lab, Swiss Federal Inst. of Technology (ETH) Zurich, 2000.
[15] L. Kleinrock, *Queueing Systems. Volume I: Theory.* John Wiley & Sons, Inc., 1975.
[16] L. Kleinrock, *Queueing Systems. Volume II: Computer Applications.* John Wiley & Sons, Inc., 1976.
[17] M.S. Lam, R.P. Wilson, R.S. French, and J.L. Hennessy, "Suif: An Infrastructure for Research on Parallelizing And Optimizing Compilers," *SIGPLAN Notices,* vol. 29, no. 12, pp. 31-37, 1994.
[18] R. Leupers and P. Marwedel, *Retargetable Compiler Technology for Embedded Systems: Tools and Applications,* Kluwer Academic, 2001.
[19] W. Luk, T.K. Lee, J.R. Rice, N. Shirazi, and P.Y.K. Cheung, "Reconfigurable Computing for Augmented Reality," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines,* 1999.
[20] X.-J. Zhang, K.-W. Ng, and W. Luk, "A Combined Approach to High-Level Synthesis for Dynamically Reconfigurable Systems," *Field Programmable Logic and Applications,* pp. 361-370, Springer, 2000.
[21] S. McMillan and S. Guccione, "Partial Run-Time Reconfiguration Using jrtr," *Field Programmable Logic and Applications,* Springer, 2000.
[22] K.N. McNall and A.E. Casavant, "Automatic Operator Configuration in the Synthesis of Pipelined Architectures," *Proc. 27th ACM/IEEE Design Automation Conf.,* pp. 174-179, 1990.
[23] O. Mencer, H Hübert, M. Morf, and M.J. Flynn, "StReAm: Object-Oriented Programming of Stream Architectures Using PAM-Blox," *Field-Programmable Logic: the Roadmap to Reconfigurable Systems,* pp. 595-604, Springer, 2000.
[24] I. Mitrani, *Probabalistic Modelling.* Cambridge Univ. Press, 1998.
[25] T. Moller and B. Trumbore, "Fast, Minimum Storage Ray-Triangle Intersection," *J. Graphics Tools,* vol. 2, no. 1, pp. 21-28, 1997.
[26] S. Ong et al., "Automatic Mapping of Multiple Applications to Multiple Adaptive Computing Systems," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines,* pp. 218-227, 2001.
[27] U. Prabhu and B.M. Pangrle, "Global Mobility Based Scheduling," *Proc. IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors,* pp. 370-373, 1993.
[28] R.O. Onvural, "Survey of Closed Queueing Networks with Blocking," *ACM Computing Surveys,* vol. 22, no. 2, pp. 83-121, June 1990.
[29] J. Park, K.R. Shesha Shayee, and P.C. Diniz, "Performance and Area Modeling of Complete FPGA Designs in the Presence of Loop Transformations," *Field Programmable Logic and Applications,* Springer, 2003.
[30] H. Schmit, "Incremental Reconfiguration for Pipeline Applications," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines,* 1997.
[31] S. Singh and P. James-Roxby, "Lava and jbits: From Hdl to Bitstream in Seconds," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines,* 2001.
[32] H. Styles and W. Luk, "Accelerating Radiosity Calculations Using Reconfigurable Platforms," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines,* pp. 279-281, 2002.
[33] A.H. Veen, "Dataflow Machine Architecture," *ACM Computing Surveys,* vol. 18, no. 4, pp. 365-396, 1986.
[34] M. Weinhardt and W. Luk, "Pipeline Vectorisation," *IEEE Trans. Computer-Aided Design,* vol. 20, no. 2, pp. 234-248, Feb. 2001.
[35] J.M. Stone et al., "Stream-Oriented Fpga Computing in the Stream-C High Level Language," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines,* 2000.
[36] W. Wang, A. Raghunathan, N.K. Jha, and S. Dey, "High-Level Synthesis of Multi-Process Behavioral Descriptions," *Proc. IEEE Int'l Conf. VLSI Design,* pp. 467-473, 2003.
[37] H. Ziegler, B. So, M. Hall, and P.C. Diniz, "Coarse-Grain Pipelining on Multiple FPGA Architectures," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines,* pp. 77-86, 2002.

Henry Styles is a PhD student in the Department of Computing, Imperial College London. His main research interests are compilation of level languages into hardware, runtime reconfiguration, performance modeling, and computer graphics.

Wayne Luk is a professor of computer engineering in the Department of Computing, Imperial College London and leads the Custom Computing Group there. His research interests include theory and practice of customizing hardware and software for specific application domains, such as graphics and image processing, multimedia, and communications. Much of his current work involves high-level compilation techniques and tools for parallel computers and embedded systems, particularly those containing reconfigurable devices such as field-programmable gate arrays.