Microarchitectural Transformations Using Elasticity

MARC GALCERAN-OMS, eSilicon Corporation
ALEXANDER GOTMANOV, Intel Corporation
JORDI CORTADELLA, Universitat Politècnica de Catalunya
MIKE KISHINEVSKY, Intel Corporation

Elasticity is a paradigm that tolerates the variations in computation and communication delays. By applying elastic transformations that allow varying the original timing, circuits can be optimized beyond the conventional rigid transformations that do not modify the external timing.

Pipelining is one of the classical techniques to improve the throughput of a circuit. This article reveals how elasticity can be effectively and practically used to derive pipelined circuits by using correct-by-construction transformations that can be fully automated. Two designs, one of them industrial, are used to demonstrate how the area-performance trade-off can be explored using elasticity.

Categories and Subject Descriptors: B.1.1 [**Control Structures and Microprogramming**]: Control Design Styles; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Pipeline processors*

General Terms: Design, Performance

Additional Key Words and Phrases: Elastic circuits, pipelining, asynchronous circuits, latency insensitivity

ACM Reference Format:
Galceran-Oms, M., Gotmanov, A., Cortadella, J., and Kishinevsky, M. 2011. Microarchitectural transformations using elasticity. ACM J. Emerg. Technol. Comput. Syst. 7, 4, Article 18 (December 2011), 24 pages.
DOI = 10.1145/2043643.2043648 http://doi.acm.org/10.1145/2043643.2043648

1. INTRODUCTION

The increasing complexity of VLSI systems has stimulated an intense effort in design automation to make circuit production scalable and economically viable. In the front-end flow, logic synthesis has been the area in which designers have delegated most of the responsibility for circuit optimization to EDA tools. But logic synthesis works under a very strict behavioral constraint: keeping cycle-accurate equivalence.

Circuits can be further optimized when cycle-accuracy can be transgressed. This is the case of pipelining that has been one of the fundamental techniques for increasing the throughput of microprocessors for several decades. While the advantages of pipelining are clear, they come with the cost of certain disadvantages that affect the complexity and correctness of the circuits.

This work has been supported by grants from Intel Corporation, the project FORMALISM (TIN2007-66523) and FI from Generalitat de Catalunya.
This work was performed while Marc Galceran-Oms was at Universitat Politècnica de Catalunya.
Authors' addresses: M. Galceran-Oms, eSilicon Corp., Barcelona; A. Gotmanov, Strategic CAD Labs, Intel Corp., Moscow; J. Cortadella, Universitat Politècnica de Catalunya, Barcelona; M. Kishinevsky, Strategic CAD Labs, Intel Corp., Hillsboro.

DOI 10.1145/2043643.2043648 http://doi.acm.org/10.1145/2043643.2043648

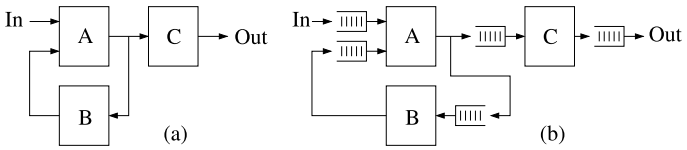

Fig. 1. (a) Original circuit; (b) Equivalent elastic circuit.

Pipelining requires additional memory components to store the temporary in-flight information involved in multiple concurrent operations executed simultaneously. The existence of causal dependencies between operations requires extra control to avoid hazards that could produce incorrect results. Forwarding and stalling are different techniques to guarantee correctness in pipelined circuits.

The design of a pipeline is a difficult and error-prone task. Defining the number of stages, the location of the extra memory, the control for stalling the circuit, the forwarding paths to avoid data hazards, and so on, are essential decisions that have a direct impact on the performance and complexity of the circuit.

Even though some academic work has been done in the last few years, optimizations transgressing the observable cycle accuracy are not handled by the mainstream EDA flows. These optimizations are explored manually by the designers at microarchitectural level.

1.1. Elasticity

Elastic circuits [Carmona et al. 2009] have emerged as a paradigm to relax the cycle accuracy constraints in the design of digital circuits. The concept of elasticity has been largely used in asynchronous circuits. For example, the term micropipeline was proposed [Sutherland 1989] to denote event-driven elastic pipelines.

When elasticity was discretized to work with synchronous systems, the term Latency Insensitivity was coined [Carloni et al. 1999]. In these systems, the handshaking is produced at the level of a cycle with events that are synchronized with the clock. Different variants of synchronous elasticity were proposed later [Cortadella et al. 2006a; Jacobson et al. 2002; Vijayaraghavan and Arvind 2009].

With elasticity, the sequences of valid data items are observed and preserved as if one had inserted FIFOs with nondeterministic delays in the communication channels between modules.

Figure 1 shows an example of transforming a circuit into an elastic form. The insertion of FIFOs must be done in such a way that the functionality of the system is not affected. This goal is achieved by using an elastic protocol that synchronizes the read/write operations of the FIFOs and preserves the order of the transferred valid data in every channel regardless of the computation time of the components.

The reader is referred to Carmona et al. [2009] for an extended discussion on different forms of elasticity, either synchronous or asynchronous.

1.2. Elasticity and Pipelining: A Quick Overview

The primary goal of pipelining is to increase performance. This article will show how elasticity can contribute to exploration of different pipelines at a microarchitectural level by applying behavior-preserving transformations. More interestingly, this exploration can be automated.

A fundamental transformation for pipelining is memory bypassing[1] [Kogge 1981]. When bypassing is combined with retiming in the presence of elasticity, the forward

[1]In this context, register files are considered a particular case of memories.

paths for avoiding data hazards are implicitly created. Early evaluation is essential to automatically generate the control for the forward paths with an elegant solution based on the concept of antitoken. Elasticity allows inserting bubbles (registers with no valid data) to reduce the cycle time when long combinational paths appear.

All the previous transformations can be automatically explored by using advanced heuristics [Galceran-Oms et al. 2010] and mathematical models to perform retiming and recycling with early evaluation [Bufistov et al. 2009].

To get an initial idea of the power of elasticity in improving the throughput of the system, the reader may look at Figure 8, where an unpipelined circuit is progressively transformed into a pipelined one by introducing elastic transformations. Another notorious example is the DLX microprocessor, showing how the initial circuit (Figure 11) can be transformed into a pipelined microprocessor (Figure 12), in which the classical load-store buffer for memory is automatically inferred. The details will be discussed in their corresponding sections.

1.3. Organization of the Article

This article reviews the basic concepts of elasticity and the behavior-preserving transformations using elasticity that have been proposed in the literature (Section 2). The article also gives a general view of the transformations, extending them to various forms of elasticity (Section 3). Even though the examples selected in the article are synchronous, the transformations are equally applicable for asynchronous versions of the same circuits obtained by desynchronization [Carmona et al. 2009; Cortadella et al. 2006b].

Section 4 discusses how different pipelines can be automatically explored by systematically applying elastic transformations. Section 5 presents a complete design experiment to evaluate the effectiveness of the elastic transformations. Finally, Section 6 discusses advanced transformations that can be automated in future design flows.

2. ELASTIC SYSTEMS

An elastic system is a collection of elastic modules and elastic channels. Elastic channels contain FIFOs that store the data being transferred between the sending and the receiving modules. The FIFOs are controlled by an elastic protocol that keeps track of the availability of data (e.g., FIFO not empty) and the ability to receive more data (e.g., FIFO not full). The simplest protocol needs two handshake signals that are typically called *valid* (in the forward direction) and *stop* (in the backward direction). When elasticity is asynchronous, these signals are usually called *request* and *acknowledge*.

The *stop* signal is the one that controls the back-pressure and prevents a global stall when any of the modules cannot accept input data. The storage provided by the input FIFOs and the handshake protocol allow distributing the stall conditions along the elastic channels [Carloni 2006].

2.1. Elastic Buffers

Elastic channels must have a minimum storage of two items to achieve the maximum throughput of the system, assuming that the forward and backward latencies of the elastic channels are one clock cycle [Carloni et al. 1999]. This is a necessary condition, but not sufficient. In some cases, the capacity needs to be increased to achieve the optimal throughput [Lu and Koh 2003].

Various schemes have been proposed to implement elastic channels. In Carloni et al. [1999], relay stations based on flip-flops were initially proposed. In asynchronous design, latch-based designs have been typically used (e.g., as in Micropipelines [Sutherland 1989]). Based on these schemes, latch-based implementations were also

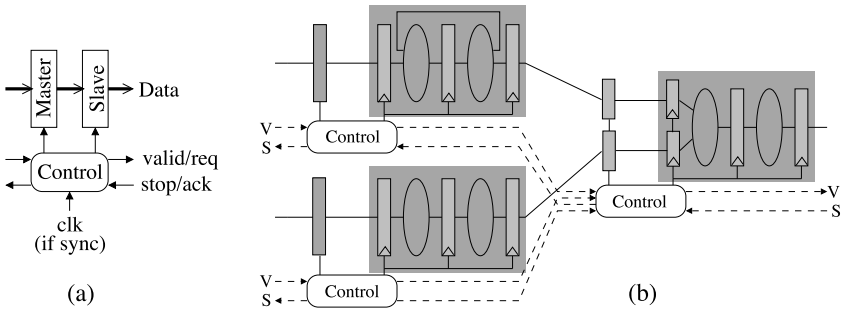

Fig. 2. (a) Elastic buffer; (b) Coarse-level control of functional blocks.

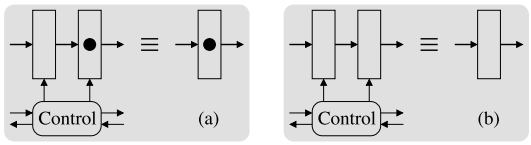

Fig. 3. Graphical representation of EBs: (a) with token, (b) without token (bubble).

proposed for synchronous elasticity [Cortadella et al. 2006a; Jacobson et al. 2002]. In this article, we will generically refer to the storage components used in elastic channels as *Elastic Buffers* (EB), regardless the specific details of their implementation.

At a fine level of granularity, EBs can be implemented by substituting flip-flops for pairs of latches (master/slave) with independent control. This is the strategy also proposed to desynchronize circuits [Cortadella et al. 2006b]. Figure 2(a) depicts a block diagram of an EB with two latches. This scheme is valid either for synchronous or asynchronous elasticity, depending on whether the events of the handshake signals are synchronized with the clock.

At a coarse level of granularity (see Figure 2(b)) elasticity can be implemented by simply adding one latch for each of the inputs of an elastic module. As in the previous case, the input latch plays the role of master and handles the back-pressure, whereas the internal registers play the role of slaves. With this scheme, the handshake protocol is identical and the modules are not touched. This is a practical approach to incorporate elasticity without modifying any of the existing IPs in the system.

In the initial state, every EB may have either one valid data item (representing the initial value of the register in a nonelastic system) or no valid data (empty). We will call these configurations *token* and *bubble* respectively, as illustrated in Figure 3. For simplicity, every EB will only be represented as one box with token or bubble depending on its initial state. Note that EBs can have at most one token in their initial state. The storage of more tokens will only occur dynamically as the result of handling back-pressure from the receiving module.

2.2. Early Evaluation

The execution model of conventional elastic systems is based on *strict evaluation*: a computation is initiated only when all input data are available. This requirement can be relaxed if early evaluation is used. Consider a 2-input multiplexor that can be modeled using the following statement.

$$z = \texttt{if } s \texttt{ then } a \texttt{ else } b.$$

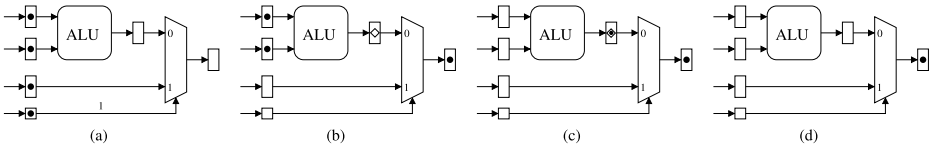

Fig. 4. Movement of tokens and antitokens for early evaluation.

If s is available and its value is *true*, there is no need to wait for the arrival of b. Then, the result can be produced as soon as a arrives. A similar situation occurs when s is *false* and b arrives.

Early evaluation contributes to increasing the performance of elastic systems at the expense of additional control for guaranteeing the correct execution. In particular, the spurious enabling of functional units must be prevented when the late inputs arrive after the completion of the computation. These late inputs must be ignored.

One of the mechanisms to discard late inputs is the use of *negative tokens*, also called *antitokens*. Each time an early evaluation occurs, an antitoken is generated at every non-required input in such a way that it annihilates when it meets a valid data item (*positive token* [Cortadella and Kishinevsky 2007]).

Figure 4(a) depicts a circuit in which early evaluation contributes to improving performance. The circuit contains valid data in all registers with token (●). The 0-input of the multiplexor has no valid data, however the control signal indicates that the 1-input must be selected. In this situation, there is no need to wait for the late data produced by the ALU.

Figure 4(b) shows that the 1-input has been sent to the output of the multiplexor. Additionally, an *antitoken* (◇) has been sent backward to the 0-input to annihilate the pending data from the ALU. When the result from the ALU arrives, as shown in Figure 4(c), the valid data and the antitoken meet (◈) and cancel each other, thus disregarding the nonrequired data, as shown in Figure 4(d).

Early evaluation is an essential mechanism to generate a correct control for bypassing schemes or to implement speculation techniques (e.g., branch prediction). Even though the concept of early evaluation can be applied to general combinational logic, it becomes useful mainly in multiplexors.

The idea of antitokens was initially used in Kishinevsky et al. [1994] to handle OR causality in Petri nets. Various implementations exist both in synchronous (e.g., Casu and Macchiarulo [2007]) and asynchronous (e.g., Reese et al. [2005]) circuits. Among the different implementations of antitokens, two main classes have been distinguished: passive antitokens, that statically wait for the arrival of tokens, and active antitokens, that move backward to meet tokens. Hybrid approaches combining active and passive antitokens are also possible.

Our experience demonstrates that a good practical choice is to use passive antitokens as a default design option, because the controller becomes simpler, and to use active antitoken locally in the regions of the system where passive antitokens become a bottleneck—a very rare case. In most design examples we have observed that the performance gain by using active antitokens is not significant, even though a synthetic counterexample can always be constructed. A choice of the type of antitokens can be done during fine-tuning of the microarchitecture.

3. BASIC TRANSFORMATIONS

One of the major features of synchronous elastic systems is their tolerance to latency changes. Such tolerance can be used to introduce novel correct-by-construction transformations enabling the exploration of new microarchitectural trade-offs [Kam et al.

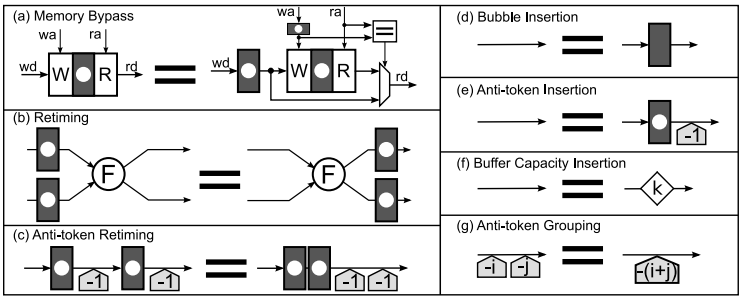

Fig. 5. Correct-by-construction transformations [Kam et al. 2008].

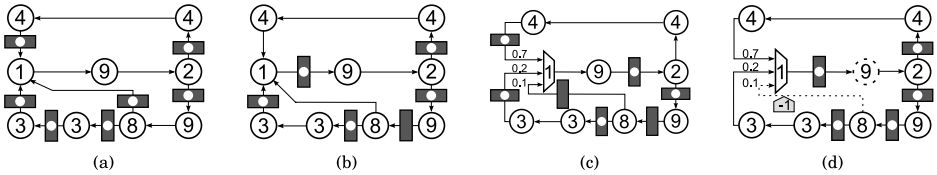

Fig. 6. Design after, (a) retiming, (b) retiming and recycling, (c) early evaluation, (d) antitoken insertion.

2008]. In some cases, the cycle time of the system can be reduced by increasing the latency of some operations. By properly balancing cycle time and throughput, the system with the optimal effective cycle time can be achieved.[2]

This section presents a set of transformations that can modify the latency of the communications and computations of an elastic system (see Figure 5). Verification using model checking shows that they preserve system functionality, i.e., the system after applying each of the transformations is latency equivalent [Krstić et al. 2006] to the original one.

3.1. Latency-Preserving Transformations

This section presents two transformations that do not change the latency of the computations and preserve the functionality of the design: bypassing and retiming.

At a certain level of abstraction, a register file or a memory can be represented by a monolithic memory element and additional logic to write (W) and read (R) data (see Figure 5(a)). The channels *wd* and *rd* represent data, whereas the channels *wa* and *ra* represent addresses.

Bypasses have been widely used since the late 50s [Bloch 1959], to resolve data hazards in processors [Hennessy and Patterson 1990]. Figure 5(a) shows a memory after a bypass transformation. An EB postpones the write operation by one cycle, and a forwarding path is added so that if the read address is equal to the write address of the previous operation (RAW dependency), the correct data value can be propagated. In an elastic system, the multiplexor selecting between the forwarded data and the memory data can be implemented with early evaluation. Multiple bypass transformations can be recursively applied to the same memory in order to create a bypass network.

Retiming [Leiserson and Saxe 1991] (see Figure 5(b)) is a traditional technique for sequential area and delay optimization. Figure 6(a) shows an example after an optimal

[2]The effective cycle time is a performance measure similar to the time-per-instruction, TPI, in CPU design. It captures how much time is required to process one token of information—the smaller the better.

retiming. The combinational nodes (shown as circles) are labeled with their delays. The boxes (labeled with dots) represent the elastic buffers with tokens of information (registers with valid data). The cycle time of this design is 17 time units.

3.2. Recycling

It is always possible to insert and remove an empty EB (a bubble) on any channel of an elastic system (for a formal proof see Krstić et al. [2006]). This transformation is shown in Figure 5(d). Bubble insertion is also known as recycling, and was initially introduced in Carloni and Sangiovanni-Vincentelli [2000]. The concept of inserting empty buffers for optimizing system performance was long known in asynchronous design [Manohar and Martin 1998; Williams 1994].

Figure 6(b) shows an optimal configuration combining retiming and recycling for the example from Figure 6(a). The cycle time has been reduced to 11 units. The throughput is determined by the slowest cycle. The token/register ratios for each cycle are 1, 4/5, and 2/3. Therefore, the throughput is 2/3, and the average number of cycles to process a token is 3/2. This provides an effective cycle time of 16.5 time units ($16.5 = 11 \cdot 3/2$). It means that a new token is processed on average every 16.5 time units—an improvement compared to the 17 units of the optimally retimed design.

Early evaluation can be used to further optimize an elastic system. Figure 6(c) shows the example from Figure 6(b) after applying early evaluation to the node, with delay 1, and adding a bubble on one of the input channels of the multiplexor. Each data input of the multiplexor has been assigned the probability of being selected by the control input, to enable throughput analysis. The example from Figure 6(c) has a cycle time of 10 time units, which is lower than the 11 units in Figure 6(b). Without early evaluation, its throughput would be 0.5, as determined by the slowest cycle. Then, the effective cycle time would be 20 units—worse than the 16.5 units obtained for the previous configuration. However, when early evaluation is introduced, the cycle with the worse throughput is only selected by the multiplexor in 10% of the cases. If the system is simulated using the given probabilities, the obtained throughput is 0.79. Thus, in this example, early evaluation allows one to reduce the effective cycle time to 12.65 units (10/0.79).

3.3. Antitoken Insertion

Antitokens are used to cancel spurious computations in early-evaluation nodes, but they can also be used to enable new retiming configurations. An empty EB is equivalent to an EB with one token of information followed by an antitoken injector with one antitoken (drawn as a pentagon), as shown in Figure 5(e). Antitoken counters can be retimed (as in Figure 5(c)) and grouped (as in Figure 5(g)). When retiming antitokens, care must be taken with the initial values of the registers so that functionality does not change.

Antitoken insertion can be often applied to enable retiming of EBs that initially contain a different number of tokens (e.g., a bubble with an elastic buffer that contains one token). Figure 6(d) shows a system where antitoken insertion has been applied to the dashed channel. Then, the new EB can be retimed through the multiplexor. This new configuration has a cycle time of 11 units, but its throughput is very high, 0.918, since there is only one cycle with a bubble (a sum of a token and an antitoken is equal to zero) as compared to Figure 6(c), where two out of the three cycles have bubbles. The resulting effective cycle time, which can only be achieved by using the antitoken insertion transformation, is 11.98 units.

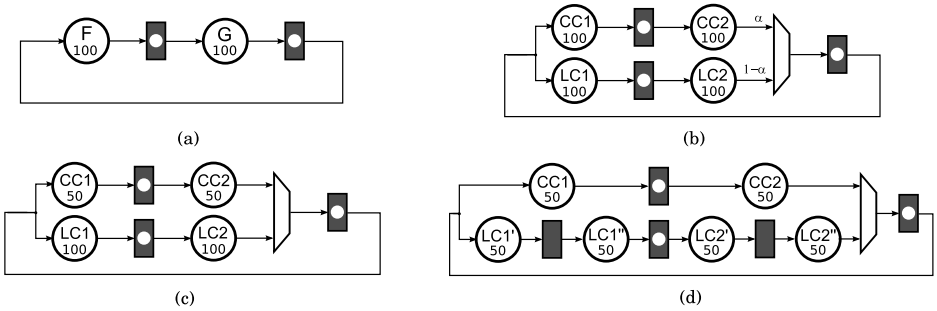

Fig. 7. (a) Pipeline with two functional blocks, F and G, (b) separation of Little Cares (LC) from Critical Core (CC), (c) optimization of the Critical Core branch, (d) insertion of bubbles into the Little Cares branch.

3.4. Variable-Latency Units

Variable-latency units (VLUs) can be handled in a natural way in synchronous elastic systems. A handshake with the datapath unit is required so that the control can keep track of the status of operation, as shown in Cortadella et al. [2006a].

For example, an ALU may spend one clock cycle to compute frequent operations with small operands (operands with few significant digits), and spend two clock cycles for rare operations involving larger operands. This is a typical example of a telescopic unit [Benini et al. 1999; Su et al. 2007]. VLUs can improve the performance by decreasing the overall cycle time, and they can also improve the area of the design by reducing the number of logic gates per cycle of operation.

In the example from Figure 6(d), the critical cycle is determined by the dashed node with delay 9 followed by the node with delay 2. Assume we can replace the dashed node with a variable-latency node that has a typical delay of 7 time units at the cost of spending an extra cycle (14 time units) in rare cases. Then, the cycle time of the system will drop from 11 to 9 units. Let us assume that the short operation can be applied 95% of the time. Then, the throughput of the system is 0.881, as estimated by simulating the controller. The resulting effective cycle time is 10.216 units (9/0.881), compared to the previous 11.98. Overall, correct-by-construction transformations have provided a 66% improvement in performance for this example.

3.5. Buffer Capacities

While buffer insertion in Figure 5(d) is formulated for the elastic buffer with capacity two, it holds for the elastic buffer of any capacity $k \geq 0$. Moreover, if the latency of the buffer is equal to 0 (implementable as a FIFO with a bypass), the performance of the design as measured by the throughput cannot decrease. The rhombus in Figure 5(f) stands for a 0-latency buffer (also called skid buffer) with capacity k.

3.6. Little Cares

VLUs illustrate a general principle of optimizing circuit performance for those cases that are expected to appear most frequently during computation. Due to their single server semantics, VLUs are most efficient for relatively small blocks, e.g. individual arithmetic units, pipeline stages, and so on. However, the idea of variable latency can be extended to larger design components containing entire pipelines, memories, finite-state machines, and so on.

Consider a simple example of two pipeline stages in a loop in Figure 7(a). Suppose that the functional blocks F and G have a delay of 100 time units each. Effective cycle time (ECT) of the system would then be 100.

Suppose that there are two types of tokens flowing through the pipeline. Tokens of the first type occur with high probability, α, and tokens of the second type occur with low probability, $1-\alpha$. To improve the effective cycle time we can replace F and G with equivalent VLUs optimized for tokens of the first type. Note however that introduction of VLUs is done locally within each stage, while both types of tokens continue to share the same pipeline. Alternatively, we can completely decouple implementation of frequent and rare behaviors by separating the pipeline into two branches, as shown in Figure 7(b). We will informally refer to the frequent behaviors, induced by tokens of the first type, as the *Critical Core* (CC); and to the rare behaviors, induced by tokens of the second type, as the *Little Cares* (LC).

After the separation, we can optimize CC and LC pipelines independently using any applicable transformations. In practice, LC behaviors include many complex corner cases, which can now be ignored while addressing the performance-critical part of the circuit (CC). To obtain the pipeline in Figure 7(c), we focus on the CC branch and reduce timing of the functional blocks from 100 to 50 time units. Note that we can apply any sequential transformations like retiming, speculation, bypassing, and so on, which may be inefficient for the pipeline in Figure 7(a), but become practical for the CC branch (for concrete examples see Section 5). To take advantage of the optimized CC part, we need to reduce the cycle time of the LC branch, which is done by inserting empty elastic buffers into its timing-critical paths (see Figure 7(d)). The final circuit is effectively a pipelined variable-latency unit. Its effective cycle time depends on the value of probability α. For $\alpha = 0.0$, there is no speedup, because all computations are carried out by the slow LC branch. Maximum speedup of $2\times$ occurs at $\alpha = 1.0$. In the more realistic case of $\alpha = 0.9$, effective cycle time is improved by approximately $1.67\times$.

3.7. A Simple Pipelining Example

Starting with a functional specification graph of a design, it is possible to obtain a pipelined design by using the elastic transformations presented in this section.

Bypasses with early-evaluation multiplexors are essential for pipelining, since they introduce new EBs that can be retimed backwards. In order to pipeline a design, bypasses must be inserted around register files and memories of the functional model. Then, the graph is modified to enable forwarding to the bypass multiplexors. Finally, the system can be pipelined by retiming the EBs inserted with the bypasses and using other transformations such as recycling or antitoken insertion.

Figure 8(a) shows a specification of a simple design. The register file *RF* is the only state-holding block. *IFD* fetches instructions and decodes the opcode and register addresses. *ALU* and *M* are arithmetic units. The results are selected by the multiplexor for *RF* write-back. *M* has been divided into three nodes. The breaking up of logic to allow pipelining is a design decision that is typically considered in concert with pipelining decisions. Thus, the user may try to divide a functional block into several nodes and let the optimization algorithm decide the best channels to place the EBs.

In Figure 8(b), the bypass transform has been applied three times on *RF* to build a bypass network. Node *DD* receives all previous write addresses and the current read address in order to detect any dependencies and determine which of the inputs of the bypass multiplexor must be selected. The conventional use of bypasses is to forward data already computed to the read port of the bypassed memory element. In addition, this bypass network can be used as a data hazard controller, taking advantage of the underlying elastic handshake protocol with early evaluation to handle stalls.

The right-most multiplexor and the bypass EBs must be duplicated to feed each bypass path independently, enabling new forwarding paths, as shown in Figure 8(c). Once the forwarding paths have been created, the design can be pipelined by applying

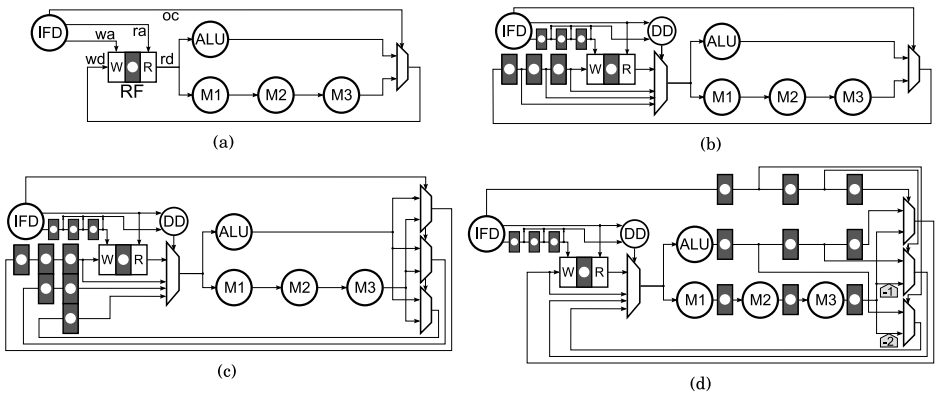

Fig. 8. (a) Graph model of a simple design, (b) after 3 bypasses, (c) duplicate multiplexor, enable forwarding, (d) final pipeline after transformations.

retiming and antitoken insertion, achieving the system in Figure 8(d). The final elastic pipeline is optimal in the sense that its distributed elastic controller inserts the minimum number of stalls. Furthermore the pipeline structure is not redundant, since there are no duplicated nodes. Therefore, this is as good as a manually designed pipeline.

Fast instructions that require few cycles to compute, like *ALU* in this example, use the bypass network to forward data avoiding extra stalls, while long instructions use the bypass network as a stall structure that handles data hazards. In this example, the only possible stalls occur when the paths with antitoken counters are selected by the early-evaluation multiplexors. This situation corresponds to a read-after-write (RAW) dependency involving a result computed by *M*, which needs three cycles to complete.

4. AUTOMATIC EXPLORATION OF PIPELINES

The example in the previous section is small enough to allow a manual exploration. However, manual exploration becomes complicated and error-prone for larger microarchitectural graphs. This section shows how pipeline exploration can be automated. A DLX pipeline is used to illustrate the proposed method.

To enable a quick analysis of the throughput of an elastic system, probabilities must be assigned to each of the data inputs of the early-evaluation multiplexors. Furthermore, it must be determined how often a data dependency can occur in order to map these probabilities to the bypass early-evaluation multiplexors. Such probabilities should be derived from profiling the benchmarks expected to be run on the microarchitecture. There are no known exact methods for the efficient throughput analysis of elastic systems with early evaluation. The method in Júlvez et al. [2010] returns an upper bound of the throughput using linear programming.

4.1. Retiming and Recycling Optimization

The retiming and recycling transformations can be combined in an optimization problem, as initially presented in Carloni and Sangiovanni-Vincentelli [2003]. This method can be extended to handle early evaluation and antitoken transformations [Bufistov et al. 2009]. The problem is formulated as a mixed integer linear programming problem, and it is solved by finding a set of designs that are Pareto points with respect to the cycle time, τ, and the throughput, Θ, of the design. The metric to optimize is the effective cycle time of the elastic system, $\xi = \tau/\Theta$. Thus, the designs found by retiming

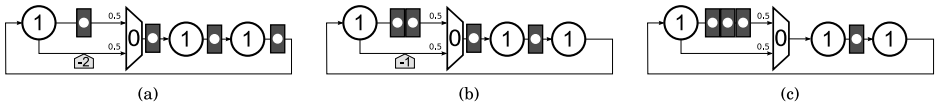

Fig. 9. Example with three nondominated Pareto-point configurations found by retiming and recycling. Their performance metrics are (a) $\tau = 1, \Theta = 0.5, \xi = 2$, (b) $\tau = 2, \Theta = 0.66, \xi = 3$, (c) $\tau = 3, \Theta = 1, \xi = 3$. τ: cycle time, Θ: throughput, $\xi = \tau/\Theta$: effective cycle time.

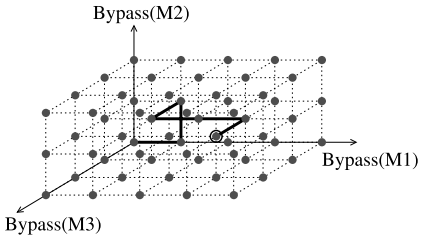

Fig. 10. Space of bypass configurations for three memories and navigation from the origin to find an optimal number of bypasses.

and recycling, RR, provide different trade-offs between the latency of the computations and the achievable cycle time for each of these latencies.

For example, Figure 9 shows an example that has three Pareto-point configurations. The first design is the optimal one, since its effective cycle time is 2, compared to 3 for the other two configurations. If the early-evaluation multiplexor had different probabilities at the inputs, the configurations might be different, or they might be the same with a different optimum in terms of effective cycle time. RR typically achieves better results when the system contains early-evaluation nodes.[3]

4.2. Exploration Algorithm

Capacity sizing and the number of bypasses to apply to each memory element are the only transformations that are not explored by the retiming and recycling algorithm. We now present an approach to automatically explore these parameters and find an optimal pipelined configuration of the system. During the exploration, it can be assumed that all buffers have enough capacity to maximize the throughput of the system. The optimal capacity for each elastic buffer and whether to add skid-buffers in any channel can be determined at the end of the exploration by running an ILP problem [Bufistov et al. 2008; Lu and Koh 2003].

The main question to answer now is: what is the minimal number of bypasses that must be included in each memory to achieve a maximum throughput? To illustrate the exploration strategy, we will use Figure 10, which represents the space of configurations that can be obtained for a system with three memories. Each point corresponds to a particular number of bypasses applied to each memory. The origin represents the configuration without any bypass.

We next present the sketch of a heuristic algorithm that explores a trajectory in this space for finding the optimal pipelining of the system. The algorithm iteratively adds bypasses to the memories and applies retiming and recycling until no further improvement is observed in the estimated effective cycle time.

Algorithm 1 shows a sketch of the procedure to find a set of interesting pipelines. At each iteration, a new bypass is added to a subset of memories of the design. The memory subset is chosen by a heuristic rule (see Galceran-Oms et al. [2010] for more

[3]See Bufistov et al. [2009] for a table of results that validates this statement.

ALGORITHM 1: Exploration for automatic pipelining

exploredPipelines := ∅ (*Set of interesting pipelines found during the exploration*)
ξ_{min} := ∞ (*Best effective cycle time found so far*)
improvement := true
while improvement **do**
  M := SelectSubsetMemories(G)
  for $m \in M$ **do**
    G.bypass(m) (*Add one bypass and forwarding paths to memory m*)
  end
  newPipelines := RetimingRecycling(G) (*Retiming and recycling optimization*)
  exploredPipelines := exploredPipelines ∪ newPipelines
  ξ_{best} := BestEffectiveCycleTime(newPipelines)
  if $\xi_{best} \geq \xi_{min}$ **then** improvement := false **else** $\xi_{min} := \xi_{best}$
end
for $P \in$ exploredPipelines **do**
  Simulate(P) (*Estimate the actual throughput by simulation*)
end
return exploredPipelines

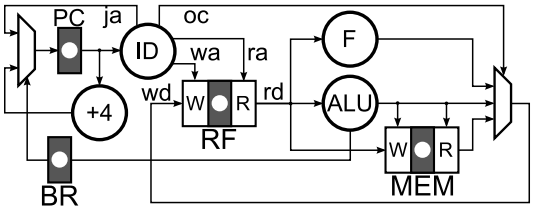

Fig. 11. DLX initial graph.

details). This corresponds to jumping to a new point in Figure 10. The resulting design has a certain number of bypasses and forwarding paths in each memory (e.g., the one in Figure 8(c)). Next, retiming and recycling is applied and a set of Pareto-point pipelines are returned (such as the one in Figure 8(d)). Since the throughput estimated by the retiming and recycling algorithm may not be exact, a set of promising designs is kept in a list during the exploration (`exploredPipelines`). The algorithm iterates as long as better pipelines are found. At the end of the exploration, the most promising designs stored in `exploredPipelines` are simulated to obtain a more accurate estimation of the throughput. Finally, a set of Pareto points with different area/performance/power values are delivered.

4.3. DLX Pipeline

Let us illustrate automatic exploration of pipelines on a simple microarchitecture similar to DLX, as shown in Figure 11 before pipelining. The execution part of the pipeline has an integer ALU and a long operation F. The instruction decoder ID produces the opcode, *oc*, that goes to the write-back multiplexor and a target instruction address, *ja*, that is taken from the previous ALU operation, as stored in the register BR. Table I shows approximate delays, latency, and area of the functional blocks of the example, taking NAND2 with fanout 3 as unit delay and unit area. In order to obtain these parameters, some of the blocks have been synthesized in a 65 nm technology library using commercial tools (ALU, RF, mux2, EB, and +4), and the rest of the values have been estimated. EB and mux2 delay and area numbers were taken for single bit units.

Table I.

Block	Delay (δ)	Area	Lat.	Block	Delay (δ)	Area	Lat.
mux2	1.5	1.5	1	EB	3.15	4.5	1
ID	6.0	72	1	+4	3.75	24	1
ALU	13.0	1600	1	F	80.0	8000	1
RF W	6	6000	1	RF R	11	-	1
MEM W	-	-	1	MEM R	-	-	10

The delay of bit-vector multiplexors and EBs is assumed to be the one shown in the table, while area is scaled linearly with respect to the number of bits. Multiplexors with a fan-in larger than two are assumed to be formed by a tree of 2-input multiplexors.

Even though a conventional DLX pipeline does not typically include this feature, F is considered to be a floating point unit. Its total delay is set to 80, around 5 times the expected cycle time of the design. Then, it may be partitioned in several stages to allow pipelining. It is assumed that the delay of each stage is $80/D$. It is possible to provide several possible partitions and perform design exploration on each one, thus determining the optimal pipeline depth.

The register file is 64 bits wide, with 16 entries, 1 write, and 2 read ports. The total footprint of the RF is 6000 units, (including both cell and wire areas). To account for wiring of other blocks, we assume that 40% of the space is reserved for it. Furthermore, we also need to consider the area overhead of elastic controllers. Based on experiments with multiple design points, a 5% area is reserved for the controllers. Given that $\text{Area}_{\text{Blocks}}$ is the area of the different combinational blocks plus the area of all the EBs, the total area of the design is $\text{Area}_{\text{RF}} + (\text{Area}_{\text{Blocks}} * 1.05)/0.6)$. The area of the initial nonpipelined design shown in Figure 11 is 23284 units.

The area of the memory subsystem is not taken into account (as it is roughly constant regardless of pipelining). It has a parameterized latency of L_{MEM} cycles for the read instruction, which is set to 10 in Table I. A latency of 10 clock cycles for read instructions is realistic if we consider it the latency of the L2 cache access. Read operations are considered to be nonblocking, i.e., a new token can be accepted even if the previous one is still in flight. Other parameters of the microarchitecture are the data dependency probability in both the register file and the memory ($\gamma_{\text{RF}}, \gamma_{\text{MEM}}$), the probability of a branch to be taken (p_{TBR}), and the probability of each instruction to be executed ($p_{\text{ALU}} + p_{\text{F}} + p_{\text{LOAD}} + p_{\text{STORE}} + p_{\text{BR}} = 1$). These probabilities are mapped to the early-evaluation multiplexors of the graph.

4.3.1. Pipelined Example. Figure 12 shows one of the best design points found automatically under the following design parameters: the F unit has been divided into three blocks, the memory data dependency probability is 0.5 ($\gamma_{\text{MEM}} = 0.5$), and register file data dependency probability is 0.2 ($\gamma_{\text{RF}} = 0.2$). The instruction probabilities are: ($p_{\text{ALU}} = 0.35$, $p_{\text{F}} = 0.2$, $p_{\text{LOAD}} = 0.25$, $p_{\text{STORE}} = 0.075$, $p_{\text{BR}} = 0.125$). Finally, the probability of a branch taken is 0.5. These values are based on the experiments found in Hennessy and Patterson [1990].

In Figure 12, the cycle time is 29.817 time units, due to the F_1, F_2, and F_3 functional blocks. Three bypasses have been applied to RF and then EBs have been retimed to pipeline F. Note that an extra bubble has been inserted at the output of F_3. The reduction in the throughput due to this bubble, is compensated by a larger improvement in the cycle time (without this bubble the critical path would include the delay of the multiplexors after F_3). If the F operation had a higher probability, the design without this bubble might be better, since the bubble would have a higher impact on the throughput degradation. Such decisions are made automatically based on the

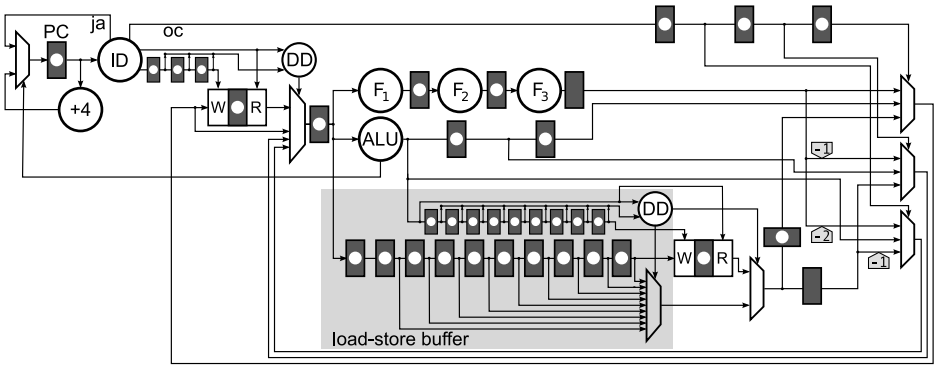

Fig. 12. Pipelined DLX graph (F divided into 3 blocks. RF has three bypasses and M 9).

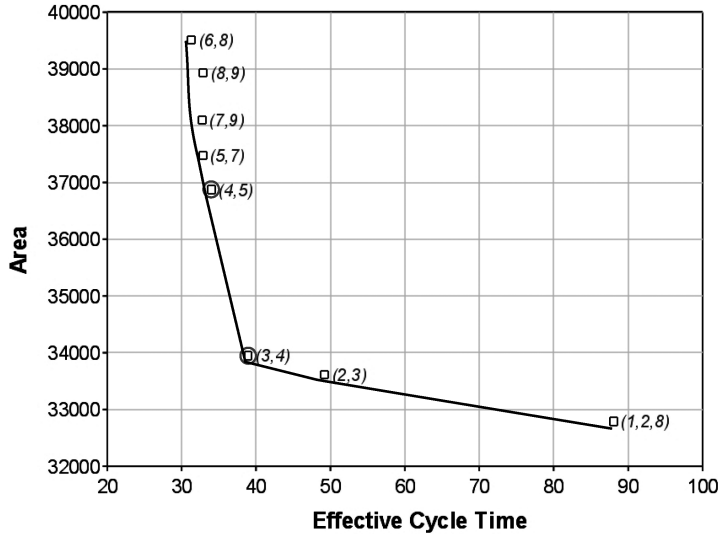

Fig. 13. Effective cycle time and area of the best pipelined design for different depths of F. (x,y) and (x,y,z) tuples represent the depth of F, the number of bypasses applied to RF and to MEM (z = 9 if omitted).

expected frequencies of instructions and data dependencies. An alternative way to avoid the bubble after F_3 is to add an extra bypass to RF. An extra bypass is indeed the optimal way to pipeline the design for this particular instance of the parameters. It might not be the case if the data hazard probability was higher.

The bypasses in the memory MEM are used to hide the memory latency via a *load-store buffer*, as shown in Figure 12. Such a structure can be substituted by a more efficient implementation: an associative memory. The need for a load-store buffer and its optimal size are detected automatically.

4.3.2. Optimal Depth. Figure 13 shows the effective cycle time and area of the best design points found on different partitions of F, forming a Pareto-point curve. The delay of F is the largest one, and as the number of nodes in which it is partitioned increases, the maximum node delay decreases, and so does the best possible cycle time in the graph. In the best design for the graph in which F is not partitioned (depth(F) = 1), the effective cycle time is 87.99, the area is 32791, 2 bypasses have been applied to

RF and 8 bypasses have been applied to MEM. The cycle time of this first point is 83.15 (δ(F) + δ(EB)), and the throughput is 0.94. Only back-to-back data dependencies involving F instructions require stalling of the pipeline.

As depth(F) increases, more bypasses are needed on the register file. The area of the design increases with more bypasses. For depth(F) = 2, the cycle time is 43.15, the throughput is 0.86 and the effective cycle time is 49.77. Until depth(F) = 6, the cycle time keeps decreasing because of the deeper pipeline (43.15, 29.81, 23.15, 19.26, 16.48), and so does the throughput because more data hazards must be handled. Overall, the effective cycle time improves with a deeper pipeline until 6 stages are reached.

For depth(F) = 7, the delay of each stage of F becomes lower than the delay of the ALU ($\delta(F_i) = 80/7 = 11.4$), and the cycle time cannot be further improved by increasing the depth of F. Thus, the best effective cycle time is achieved with F divided into 6 stages, 8 bypasses applied to RF and 9 to MEM. Design points (4,5) and (3,4) (circled in the figure) for 4 and 3 stages are simpler and may deliver a better design compromise.

Different parameters lead to different optimal design points. For example, when $\gamma_{RF} = 0.5$, bypasses can be used more often, and deeper pipelines perform better. Then, the optimal depth of F becomes 7 instead of 6. If the microarchitecture issues ALU instructions most of the time, the throughput is close to 1, since data dependencies can always use forwarding in order to avoid stalls. Besides, since F operations are never selected, fewer bypasses are applied and recycling is used in order to pipeline F. Because of the store buffer added by bypasses on MEM, memory operations can also be performed with a high throughput. In contrast, if F instructions are the most common ones, the throughput degrades faster, as this instruction always needs to stall the pipeline when a data dependency occurs.

5. DESIGN EXPERIMENT: H.264 CABAC DECODER

To study how elasticity can help optimizing nontrivial designs we conducted an experiment with the H.264 CABAC decoder used in Intel consumer electronics system on chip. Instead of designing an elastic decoder from scratch, we elasticized and optimized an existing design, which provided a clear point of comparison and permitted reuse of the validation environment and collaterals. In our studies we often relied on separation of Little Cares (see Section 3.6). It proved to be a useful practical tool for optimization and managing design complexity.

5.1. Video Coding with H.264 CABAC

CABAC, which stands for context adaptive binary arithmetic coding [Marpe et al. 2003], is an entropy codec used for lossless video compression in H.264 standard [H.264 2003]. It is also a performance bottleneck of H.264 decoding algorithm. As shown in Figure 14, the CABAC decoder receives a stream of encoded bits and decodes it into a stream of bins. Bins are sent to debinarizers, where they are translated into video symbols, like motion vector differences or discrete cosine transform coefficients. CABAC is an extension of binary arithmetic coding, which relies on context-based predictions to adapt to the statistical structure of the decoded video stream and achieve better compression ratios. For every decoded bin, the CABAC algorithm predicts the most probable symbol (MPS), i.e., whether the current bin is more likely to be 0 or 1, and the probability of MPS. The probability is encoded with a 6 bit integer state, which corresponds to a value in the [0.5, 1.0] range as specified by H.264 [2003]. The prediction varies at every cycle of decoding, based on the correctness of the previous prediction and the current context. The H.264 standard defines 449 different contexts based on the type of video symbol that is currently under decode, and the position of the current bin within the video symbol. A complete prediction information (a 7-bit

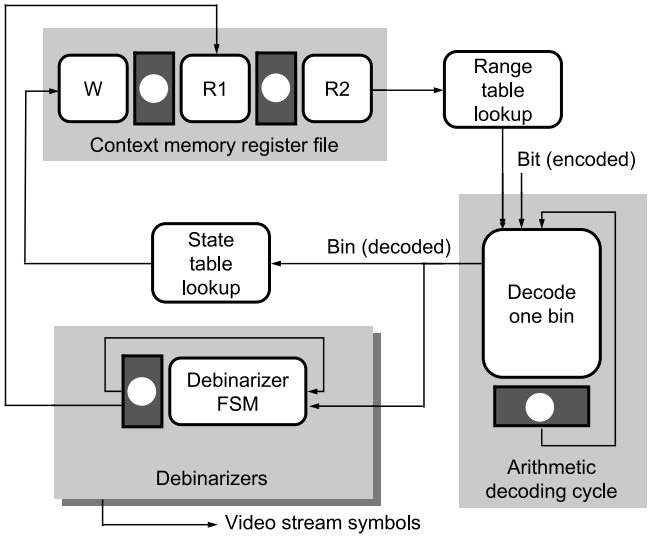

Fig. 14. Simplified microarchitecture of the CABAC engine.

state containing one MPS bit and 6 bits of the probability state) is represented as a word inside the context memory. The memory contains 449 words and is implemented as a register file with one write and one read port. Debinarizers compute current context identified by a number of context indices, which are then translated into addresses for the context memory.

The design of the CABAC decoder has several timing-critical paths including loops through the address, write, and read logic of the context memory, two lookup tables and the arithmetic decoding loop. Subsequent iterations of the decoding loop are interdependent as the value of the current bin determines the next context. Probability to correctly predict the next bin value is not always high. Therefore unrolling the loops to compute multiple bins in one cycle requires significant area overhead without reliable increase improvement in the performance. We explored a different avenue, based on case analysis following the design optimization flow explained in the next sections.

5.2. Divide and Conquer with Little Cares

Separation of Littles Cares (see Section 3.6), is a central part of our approach to optimization of the CABAC decoder. As shown in Figure 15, we start with profiling of the available models (e.g. functional model, performance model, or RTL). We systematically identify performance-critical computations (CC) and optimize them in isolation by ignoring all corner cases. The computations, which are not critical for performance (LC), are recycled to match the cycle time of the critical core. Finally, CC and LC branches are merged in a variable-latency unit in a provably correct way based on the elastic controller with early evaluation. The process is iteratively applied to different parts of the design and at different levels of granularity.

5.3. Elastic Redesign Flow

We created a design flow for converting existing RTL into elastic form and further optimizing it with transformations. In fact we found that most of the transformations can be faithfully modeled and evaluated directly in the original RTL before its elasticization. To check how transformations influence timing, area, and power of the design,

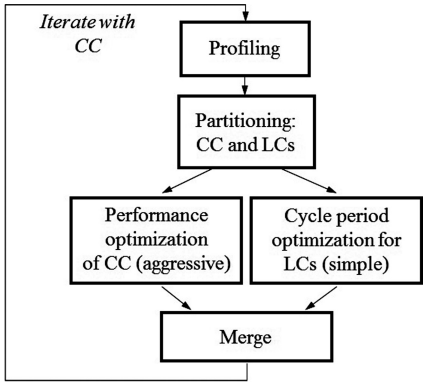

Fig. 15. Divide and conquer strategy using Little Cares.

we run a commercial synthesis flow. Synthesis checks provide measurable feedback to decide which transformations should be accepted and which should be considered next. After the desired timing/area/power trade-off is achieved on the original RTL we elasticize it by replacing registers with elastic buffers and adding an elastic controller.[4] The complete elastic RTL is again synthesized and compared with metrics of the original design. Finally, the new RTL is validated.

The design flow is partly automated and follows the steps described in the following.

(1) Components of the original RTL are grouped into elastic islands. Elasticity is preserved between the islands, but not within the islands. An automatic tool can take a description of groups provided by the user and generate a microarchitectural graph of the design. This step is not fully automated, as understanding of the design is critical for optimal partitioning. Based on performance analysis of the constructed microarchitectural graph we also decide which of the multiplexors (or other nodes) should be implemented with early evaluation.
(2) The microarchitectural graph of the partitioned design is used to perform quick experiments by applying the transformations described in Section 3. After performance analysis (done analytically or through simulation of the elastic controller) demonstrates sufficient progress, we reproduce the changes in RTL, such as inserting elastic buffers for pipelining or bubble insertion.
(3) Verilog RTL of the elastic controller is generated from the microarchitectural graph. The controller is merged with the transformed RTL, and the complete design is used for validation and synthesis.

Along with the Verilog of the elastic controller, we also generate a NuSMV model [Cimatti et al. 2002] for the verification of safety and liveness properties. While all elastic controllers are formally verified to be correct, it is still necessary to perform validation of the final RTL because formal verification abstracts away the reset phase; also (more importantly) manual changes that have been done to the original RTL can introduce errors.

We developed a practical method for dynamic validation of elasticized RTL. Note that testbenches available for the original synchronous block cannot be directly used

[4]In the case of the CABAC decoder, the elastic controller has very low overhead. The design can be partitioned into large islands of logic; each island contributes only a few latches to the controller, increasing total area and power by a small fraction. We expect the same to be true for other designs dominated by data processing. See also Casu [2011] for a discussion on the area and power overhead of elastic controllers.

with an elastic design due to the loss of synchronization. At the first step, we enforce static scheduling by removing all empty EBs. This is sufficient to eliminate all stalling cycles, because our design doesn't contain VLUs (we always use Little Cares to introduce variable latency). The simplified RTL can be simulated with original testbenches, which is useful for initial debugging of manual RTL transformations. At the second step, we simulate elastic design on a fast clock. The elastic design is fast forwarded relative to the original synchronous design, so that every large cycle of the synchronous design contains exactly n small cycles of the elastic design. At the edge of every large cycle we send new inputs to the elastic design and read back the results from the previous iteration, which can be directly compared with their synchronous counterparts. During the cycle, we assert all output *stop* signals and disassert all input *valid* signals. We will always succeed in sending input tokens and receiving output tokens, provided that the elastic design can completely process one batch of data in n or less small cycles. Note that such number n can always be found, because worst case performance of an elastic design is bounded by its most stringent cycle [Júlvez et al. 2010]. For CABAC, we determined n empirically. We applied this validation method to root-cause manually introduced bugs and obtained a high degree of confidence by testing on long video streams from the original regression suite.

5.4. Statistics of the H.264 Video Stream

Profiling of H.264 functional model on several video streams revealed significant imbalance between frequencies of different types of video symbols. The video stream data is dominated by ResCoeff symbols (blocks of residual DCT coefficients) followed by MVD symbols (motion vector differences). These elements occupy more than 90% of the encoded stream. Typically, the fraction of MVD symbols is below 5%, except for streams with heavy motion, for which it can go up to 30–40%. Video symbols of the other 9 types occur rarely and can be classified as Little Cares. According to the divide and conquer strategy described in Section 5.2, logic required for LC behaviors (including all debinarizers for symbols other than ResCoeff and MVD) is isolated and disregarded during optimization.

5.5. Design Transformations

Improvement in CABAC decoder performance is a combined result of a series of design transformations. We used multiple elastic (early evaluation, LC separation, and bubble insertion) and standard design transformations (register retiming, multiplexor retiming, bypassing, and precomputation). We discovered that sometimes a nonelastic transformation, although impossible to perform at the outset, might be enabled by an elastic transformation and vice versa. It is important to note that our optimizations cannot be repeated with the same results if the first step of identifying and isolating little cares is omitted. In this section we briefly mention some of the key transformations.

Synthesis of the CABAC decoder RTL highlighted multiple timing-critical paths to and from the context memory register file (see Figure 14). As a first transformation, we isolated the write logic of the register file by a single bypass, effectively removing it from the critical path. Contexts for different video symbols are located in different regions of the context memory. Hence every debinarizer needs to access only a portion of the corresponding register file. Since profiling showed that only ResCoeff and MVD occur often, a large portion of the register file is owned by the LC debinarizers. Therefore, the initial LC separation of debinarizers can be taken one step further by partitioning the context memory into LC and CC parts and subsequently isolating the LC part from the critical paths by inserting bubbles (empty EBs). Regions allocated for ResCoeff and MVD can also be separated from each other and connected directly to the

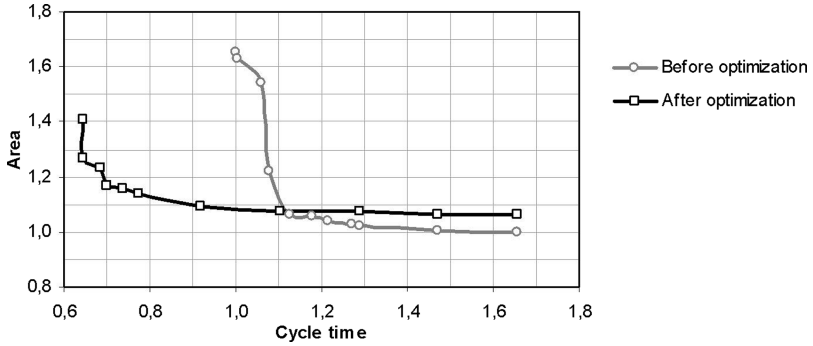

Fig. 16. Area-performance trade-off of the original and the optimized decoders.

corresponding debinarizers without intermediate multiplexing logic. As a result of this partitioning, delays in the second stage of the context memory read ($R2$ in Figure 14) were reduced, providing 10% reduction in the cycle time of the critical core.

The critical path that appeared after the register file partitioning, starts in the second stage of context memory read ($R2$), goes through the range table lookup and ends at a register in the arithmetic decoding cycle. Logic in the arithmetic cycle is well-balanced and optimized for delay, which makes it hard to improve. However, it is possible to remove the range table lookup from the critical path by precomputing the memory address (produced at debinarizer output) one cycle ahead [Hassoun and Ebeling 1996]. Then data can be read from memory one cycle earlier and range table lookup can be isolated from the arithmetic cycle by a retimed register. Potential correctness issue, with data loss due to late memory-write can be solved by adding another memory bypass or stalling reads for a cycle on the rare occurrence of data dependency of depth 2. Since context address depends on the value of the current bin, the precomputation can be done by reading two values from the CC portion of the register file (which requires an additional read port) and multiplexing the results after the second read stage when the actual bin value becomes available. This transformation corresponds to multiplexor retiming through the read logic of the register file and reduces cycle time of the critical core by another 10%.

The renormalization stage of decoding [Marpe et al. 2003] shifts two 9-bit vectors by a variable amount of bits. While the shift length can be in the range between 0 and 7, profiling demonstrated that long shifts by 3–7 bits are required in less than 5% of cases. This can be used for optimizing the shifting logic by deploying LC case splitting with two versions of renormalizers: a fast shifter by 0–2 bits (CC portion) and a slower shifter by 3–7 bits (LC portion). By applying the technique described in Figure 7, the slow shifter can be isolated with a bubble (an empty `EB`) removing it from all critical paths. This transformation reduced the overall delay of the arithmetic cycle by 15%.

Performance-area trade-off curves for the original (synchronous) and the optimized CABAC decoders are shown in Figure 16 with a cycle time shown along the X-axis and a total cell area along the Y-axis. The results are collected by sweeping different target cycle times of the designs. Each point in the graph represents normalized cycle time and total cell area after a run of the synthesis tool using a 32 nm library. Area and cycle time of 1.0 correspond to the minimal area and the cycle time of the original design. Figure 16 demonstrates up to 40% improvement in cycle time with equal area.

After evaluating probabilities at all early evaluation multiplexors in the optimized CABAC design, we ran a random simulation testbench, which produced a throughput estimate of 0.91, meaning that on the average, optimized CABAC outputs 91 data

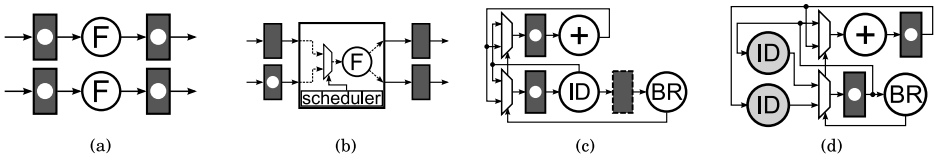

Fig. 17. (a) Logical view of a shared unit; F is considered a variable-latency unit, (b) physical view of a shared unit; the scheduler of the shared unit grants access to one of the channels. (c) branching subgraph in a simple microarchitecture, (d) same subgraph after applying speculation.

tokens in 100 clock cycles. Together with cycle time improvement from Figure 16, it gives up to 35% improvement in effective cycle time with equal area, compared to the original design (depending on the desired clock cycle point).

6. ADVANCED TRANSFORMATIONS

This section briefly discusses how some widely used microarchitectural techniques may be integrated in elastic systems. Resource sharing, speculation and multithreading are inherently elastic; they modify the latency of the computations and communications and require a control unit that alters and stalls the data flow. Furthermore, they may be added in an elastic system by applying correct-by-construction transformations. First, sharing and speculation are reviewed, and finally, as a further research topic, some possible strategies to implement multithreaded elastic pipelines are proposed.

6.1. Sharing of Functional Units

When a system includes early evaluation, the output of some computations is not always required, and hence, they can be delayed with no performance penalty or even canceled. As a result the actual utilization of some units can be way below 100%.

Different modules with the same behavior (e.g., two adders in the design) can be merged into a single module, which is then shared by the input channels competing for this resource. Sharing may provide a reduction of area and power, possibly with low (or zero) performance degradation. Using a shared module is like using a module followed by a buffer with unbounded but finite latency, since each data token may have to wait for a certain number of cycles until it is allowed to use the shared module.

In Galceran-Oms et al. [2009], a possible implementation of module sharing in an elastic system is proposed. A local scheduler decides, at each clock cycle, which input channel can use the shared resource. The performance variation compared to using unshared resources depends on whether the scheduler can distribute the load accurately among the different users. For better performance, the scheduler should take into account the elastic protocol: an invalid or a stalled channel cannot use the shared unit even if selected. For correctness, a scheduler should be fair and avoid starvation: every token that reaches the shared module must eventually be allowed to use it unless it is canceled by an antitoken.

For example, Figure 17(a) shows two channels, each one using a unit F, which compute exactly the same function. If both Fs are shared in a single physical entity, the logical view remains the same, although the latency of F becomes variable. Physically, a scheduler selects which channel can use F every clock cycle, as shown in Figure 17(b).

6.2. Speculative Execution

Sharing of functional units can be used to implement correct-by-construction speculative execution [Galceran-Oms et al. 2009]. Consider the example from Figure 17(c) showing a possible branch instruction in a microarchitectural graph. Each time a new

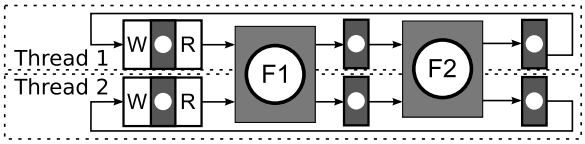

Fig. 18. Possible implementation for elastic multithreading support. Grey boxes are shared units.

instruction address must be generated, it is chosen from among the previous one plus a constant, the lower input in the multiplexors, or an address coming from the instruction decoder (ID). The selection depends on the value generated by the node named BR, which may look at some register in the ALU or may perform some operation indicated by the decoder stage.

Let us assume that ID and BR cannot be executed within one clock cycle because their total delay is too large. In Figure 17(c), the only way to cut this path is to add the bubble between ID and BR drawn with a dotted line. However, BR, ID, and one of the multiplexors form a cycle, and hence, adding this bubble limits the throughput of the design to 0.5. Early evaluation cannot help increasing the throughput in this example.

Given a multiplexor with several inputs, it is possible to move a functional block from the output of the multiplexor to its inputs using Shannon decomposition (viewed also as a multiplexor retiming) [Soviani et al. 2006]. The example from Figure 17(c) can be transformed into the design in Figure 17(d) by using multiplexor retiming and register retiming. In this second design, there is no critical cycle going through the control of the multiplexor, and the only combinational path going through two units is the one formed by BR going to the upper multiplexor and then to the adder. If this path became critical, multiplexor retiming could also be applied to the upper multiplexor.

The performance gain of Figure 17(d) comes at a cost of duplicating the ID stage, with the resulting area overhead. At this point, speculation comes into play. The two ID nodes can be merged into a single one shared by the two inputs of the multiplexor. Hence, each clock cycle the scheduler of the shared ID module must perform a branch prediction and decide which of the tokens arriving to the ID stage should be granted access to the unit so that the throughput is maximized. The scheduler can implement any state-of-the-art branch prediction algorithm, enhanced to understand the elastic protocol.

Misprediction and correction are handled naturally by the handshake between the shared module and the multiplexor. If the multiplexor requires a channel, and the scheduler predicted the other one, the scheduler will see back-pressure coming from the predicted channel and will be able to correct the misprediction.

This speculation framework can also be used to efficiently integrate into elastic systems, telescopic units and error correction and detection protocols [Galceran-Oms et al. 2009]. Using speculation and antitoken insertion, precomputation [Hassoun and Ebeling 1996] can also be added to the set of possible transformations.

6.3. Multithreading

Elastic systems implement in-order single-threaded pipelines. This section outlines some possible research directions to extend elastic systems in order to support multithreading.

There are few studies on automatic pipelining with multithreading. In Dimond et al. [2006], a multithreaded pipeline can be generated from a parameterized in-order pipelining template. In Nurvitadhi et al. [2010], a more general multithreaded pipelining approach is proposed, starting from an initial transactional specification.

The advantage of elastic systems is that they can be modeled using marked graphs amenable to formal performance analysis and optimization. The natural extension in order to support multithreading is to use colored Petri nets [Jensen 1997], where each token is assigned a color (i.e., a thread). Token order must be preserved only for the tokens that belong to the same color. Current analysis and optimization methods of elastic circuits should be extended to support multiple colors.

In the simplest implementation of a multithreaded elastic system, each thread would own a copy of an automatically generated pipeline. Next, the independent copies can be merged using the sharing transformation. Each clock cycle, the scheduler of a shared block would decide which thread can use it. Figure 18 shows a simple pipeline with two threads. Each thread has its own register file, and the functional units $F1$ and $F2$ are shared by the threads.

This approach is simple and allows total thread scheduling freedom. However, it also has a large area overhead, since each EB and the whole elastic controller is replicated once per thread. New transformations such as sharing of EBs may alleviate this overhead. A more efficient implementation can be obtained by adding a color identification to the handshake protocol of elastic controllers. All the control primitives should be extended and verified accordingly. In particular, different versions of EBs are possible depending on whether they allow reordering of tokens with different colors (better performance), or they behave strictly as FIFOs (simpler and smaller in area).

7. CONCLUSIONS

Elasticity offers a new set of transformations that can be systematically applied to improve the performance of systems. These transformations are capable of modeling well-known techniques for pipelining. Thus, the data and control logic structures associated with high-throughput pipelines can be automatically generated and design frameworks for microarchitectural exploration can be conceived.

The use of early evaluation and antitokens is a novel paradigm in pipelining, which can contribute to new design methods based on the optimization of the most frequent operations. We believe that automating the design of pipelines is an essential step to confront the complexity of high-speed circuits.

REFERENCES

BENINI, L., MICHELI, G. D., LIOY, A., MACII, E., ODASSO, G., AND PONCINO, M. 1999. Automatic synthesis of large telescopic units based on near-minimum timed supersetting. *IEEE Trans. Comput. 48*, 8, 769–779.

BLOCH, E. 1959. The engineering design of the stretch computer. In *Proceedings of the IRE/AIEE/ACM Eastern Joint Computer Conference.* 48–58.

BUFISTOV, D., CORTADELLA, J., GALCERAN-OMS, M., JÚLVEZ, J., AND KISHINEVSKY, M. 2009. Retiming and recycling for elastic systems with early evaluation. In *Proceedings of the ACM/IEEE Design Automation Conference.* 288–291.

BUFISTOV, D., JÚLVEZ, J., AND CORTADELLA, J. 2008. Performance optimization of elastic systems using buffer resizing and buffer insertion. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. 442–448.

CARLONI, L. 2006. The role of back-pressure in implementing latency-insensitive systems. *Electron. Notes Theoret. Comput. Sci. 146*, 2.

CARLONI, L. AND SANGIOVANNI-VINCENTELLI, A. 2000. Performance analysis and optimization of latency insensitive systems. In *Proceedings of the ACM/IEEE Design Automation Conference.* 361–367.

CARLONI, L. AND SANGIOVANNI-VINCENTELLI, A. 2003. Combining retiming and recycling to optimize the performance of Synchronous circuits. In *Proceedings of the 16th Symposium on Integrated Circuits and System Design (SBCCI)*. 47–52.

CARLONI, L., MCMILLAN, K., SALDANHA, A., AND SANGIOVANNI-VINCENTELLI, A. 1999. A methodology for correct-by-construction latency insensitive design. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. IEEE Press, 309–315.

CARMONA, J., CORTADELLA, J., KISHINEVSKY, M., AND TAUBIN, A. 2009. Elastic circuits. *IEEE Trans. Comput.-Aid. Design 28*, 10, 1437–1455.

CASU, M. 2011. Half-buffer retiming and token cages for synchronous elastic circuits. *IET Comput. Digital Techn. 5*, 318–330.

CASU, M. AND MACCHIARULO, L. 2007. Adaptive latency-insensitive protocols. *IEEE Des. Test Comput. 24*, 5, 442–452.

CIMATTI, A., CLARKE, E., GIUNCHIGLIA, E., GIUNCHIGLIA, F., PISTORE, M., ROVERI, M., SEBASTIANI, R., AND TACCHELLA, A. 2002. NuSMV Version 2: An OpenSource tool for symbolic model checking. In *Proceedings of the International Workshop on Computer Aided Verification*. Lecture Notes in Computer Science Series, vol. 2404, Springer.

CORTADELLA, J. AND KISHINEVSKY, M. 2007. Synchronous elastic circuits with early evaluation and token counterflow. In *Proceedings of the ACM/IEEE Design Automation Conference.* 416–419.

CORTADELLA, J., KISHINEVSKY, M., AND GRUNDMANN, B. 2006a. Synthesis of synchronous elastic architectures. In *Proceedings of the ACM/IEEE Design Automation Conference.* 657–662.

CORTADELLA, J., KONDRATYEV, A., LAVAGNO, L., AND SOTIRIOU, C. 2006b. Desynchronization: Synthesis of asynchronous circuits from synchronous specifications. *IEEE Trans. Comput.-Aid. Design 25*, 10, 1904–1921.

DIMOND, R., MENCER, O., AND LUK, W. 2006. Application-specific customisation of multi-threaded soft processors. *IEE Proc. Comput. Digit. Techniques 153*, 173–180.

GALCERAN-OMS, M., CORTADELLA, J., AND KISHINEVSKY, M. 2009. Speculation in elastic systems. In *Proceedings of the ACM/IEEE Design Automation Conference.* 292–295.

GALCERAN-OMS, M., CORTADELLA, J., KISHINEVSKY, M., AND BUFISTOV, D. 2010. Automatic microarchitectural pipelining. In *Proceedings of the Design, Automation and Test in Europe (DATE)*. 961–964.

INTERNATIONAL TELECOMMUNICATION UNION. 2003. Recommendation ITU-T H.264: Advanced Video Coding for Generic Audiovisual Services. ITUT.

HASSOUN, S. AND EBELING, C. 1996. Architectural retiming: Pipelining latency-constrained circuits. In *Proceedings of the ACM/IEEE Design Automation Conference.* 708–713.

HENNESSY, J. AND PATTERSON, D. 1990. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publisher Inc.

JACOBSON, H. M., KUDVA, P. N., BOSE, P., COOK, P. W., SCHUSTER, S. E., MERCER, E. G., AND MYERS, C. J. 2002. Synchronous interlocked pipelines. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems.* 3–12.

JENSEN, K. 1997. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use.* Springer-Verlag.

JÚLVEZ, J., CORTADELLA, J., AND KISHINEVSKY, M. 2010. On the performance evaluation of multi-guarded marked graphs with single-server semantics. *Discrete Event Dynamic Syst. 20,* 3, 377–407.

KAM, T., KISHINEVSKY, M., CORTADELLA, J., AND GALCERAN-OMS, M. 2008. Correct-by-construction microarchitectural pipelining. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD).* 434–441.

KISHINEVSKY, M., KONDRATYEV, A., TAUBIN, A., AND VARSHAVSKY, V. 1994. *Concurrent Hardware: The Theory and Practice of Self-Timed Design.* John Wiley & Sons.

KOGGE, P. M. 1981. *The Architecture of Pipelined Computers.* McGraw-Hill.

KRSTIĆ, S., CORTADELLA, J., KISHINEVSKY, M., AND O'LEARY, J. 2006. Synchronous elastic networks. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD).*

LEISERSON, C. E. AND SAXE, J. B. 1991. Retiming synchronous circuitry. *Algorithmica 6,* 1, 5–35.

LU, R. AND KOH, C.-K. 2003. Performance optimization of latency insensitive systems through buffer queue sizing of communication channels. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD).* 227–231.

MANOHAR, R. AND MARTIN, A. 1998. Slack elasticity in concurrent computing. In *Mathematics of Program Construction*. Springer, 272–285.

MARPE, D., SCHWARZ, H., AND WIEGAND, T. 2003. Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard. IEEE *Trans. Circuits Syst. Video Technol. 13,* 7, 620–636.

NURVITADHI, E., HOE, J., LU, S., AND KAM, T. 2010. Automatic multithreaded pipeline synthesis from transactional datapath specifications. In *Proceedings of the ACM/IEEE Design Automation Conference.* ACM, 314–319.

REESE, R., THORNTON, M., TRAVER, C., AND HEMMENDINGER, D. 2005. Early evaluation for performance enhancement in phased logic. *IEEE Trans. Comput.-Aid. Design 24*, 4, 532–550.

SOVIANI, C., TARDIEU, O., AND EDWARDS, S. 2006. Optimizing sequential cycles through Shannon decomposition and retiming. In *Proceedings of the Design, Automation and Test in Europe (DATE)*. European Design and Automation Association, 1085–1090.

SU, Y.-S., WANG, D.-C., CHANG, S.-C., AND MAREK-SADOWSKA, M. 2007. An efficient mechanism for performance Optimization of variable-latency designs. In *Proceedings of the ACM/IEEE Design Automation Conference*. 976–981.

SUTHERLAND, I. E. 1989. Micropipelines. *Comm. ACM 32*, 6, 720–738.

VIJAYARAGHAVAN, M. AND ARVIND. 2009. Bounded dataflow networks and latency-insensitive circuits. In *Proceedings of the 7th International Conference on Formal Methods and Models for Codesign (MEMOCODE)*.

WILLIAMS, T. 1994. Performance of iterative computation in self-timed rings. *J. VLSI Signal Proc. 7,* 1, 17–31.

Received July 2011; revised August 2011; accepted September 2011