Compilation for Scalable, Paged Virtual Hardware

Ph.D Thesis Proposal

Eylon Caspi

February 22, 2001

Abstract

Reconfigurable computing devices such as field programmable gate arrays (FPGAs) have demonstrated 10x-100x gains in performance and functional density over microprocessors for a variety of applications [13], yet their commercial use is limited primarily to serving as single-task ASIC replacements, which largely ignores their programmability and severely limits their applicability. SCORE (Stream Computations Organized for Reconfigurable Execution) [8] [9] addresses this underutilization of reconfigurable technology by introducing a compute model rooted in paged virtual hardware, analogous to virtual memory. The paged model provides a framework for device size abstraction, automatic dynamic reconfiguration, binary compatibility among page-compatible devices, and automatic performance scaling on larger devices, without recompilation.

A key problem in compiling for SCORE is the partitioning of programs into communicating, fixed-size hardware pages. The partitioning must be sensitive to inter-page communication, which in a virtualized model has unknown delay and may require run-time buffering memory. Existing heuristics for circuit partitioning (wire min-cut, FM, spectral, *etc.*) are not sufficient because they do not fully address the impact of communication on run-time performance and buffering. In this paper, we propose performance-oriented techniques for automatically synthesizing and partitioning SCORE computations into pages. The problem is formulated as a transformation on streaming state machines and utilizes a variety of high-level, functional information from the unpartitioned program. We propose a methodology for evaluating partitioning techniques in terms of overhead on circuit area and performance, and we show preliminary results for parts of the partitioning methodology. Development and experimentation is done within the existing SCORE software infrastructure, which is under continuing support and development by the Berkeley BRASS group.

1 Overview

Reconfigurable computing devices such as field programmable gate arrays (FPGAs) have demonstrated 10x-100x gain over conventional microprocessors in performance and functional density (operations per area-time) for a variety of applications [13]. The strength of reconfigurable computing comes from its combining of spatial execution with programmability— the former allows computational data paths to be highly parallel, while the latter allows data paths to be highly specialized to the application at hand. The commercial marketplace has relegated reconfigurable devices to be used primarily as ASIC replacements, executing only a single static configuration. This usage ignores many of the key performance benefits of reconfigurable technology, for instance dynamic reconfiguration and run-time specialization.

The Berkeley SCORE project contends that the present underuse of reconfigurable technology is due in great part to a lack of any unifying compute model to support its key technology benefits, to ease programming effort, and to enable software to survive and automatically scale to ever-improving, next-generation hardware. To this end, SCORE introduces a parallel computation model based on streaming data-flow graphs of communicating state-machines. A key element of the SCORE execution model is hardware virtualization, wherein a computation is partitioned into fixed-size hardware pages (analogous to virtual memory pages) that are automatically loaded and executed in available physical pages. Paging enables resource hiding in the programming model, liberating the programmer's algorithmic decisions from device size constraints. Paging also enables binary compatibility between page-compatible devices, as well as performance scaling, in that application performance will automatically improve on a larger device with more physical pages, without recompilation. The efficiency of paged execution in SCORE relies heavily on the streaming data-flow aspect of the compute model, which exposes a program's communication patterns and enables the construction of good partitions and good schedules for run-time page loading.

Efficient partitioning of a SCORE program into communicating, fixed-size pages must be highly sensitive to inter-page communication cost. It is important to minimize the performance impact of potentially very long (and in fact unknown) inter-page communication latency. Also, it is important to minimize the run-time storage requirements for buffering inter-page communication under virtualization. The full dynamics of communication cost are not captured by existing techniques for structural circuit partitioning (*e.g.* wire minimization by min-cut [23] [29] [34] and Fiduccia-Mattheyses [15]; critical-path delay minimization by cone covering [28] [35]; wire length minimization by spectral partitioning [19]) nor by existing techniques for finite state machine decomposition (*e.g.* decomposition for logic minimization [1], wire minimization [24], power minimization [2]). New techniques are needed to efficiently partition SCORE programs and their streaming state machine primitives.

The purpose of this Ph.D project is to explore and develop the mapping of SCORE programs from the programming model to the paged execution model. My

focus will be on partitioning to satisfy the page area, IO, and timing constraints of a parameterized hardware model (note, this focus is different from and does not replace traditional compiler optimizations, which are meant to improve code parallelism and reduce work, independent of a hardware model). I will develop clustering techniques to partition and regroup state machines with data-paths under area and IO constraints. I will implement the proposed partitioning techniques within the existing SCORE compiler framework. I will evaluate the "efficiency" of the proposed partitioning techniques in terms of their effect on circuit area (control overhead, page fragmentation) and performance (delay, total run-time) for a variety of applications and hardware parameter settings, using the existing SCORE simulation infrastructure. The goal of this project as a supporting component of SCORE is to demonstrate that automatic page generation can be efficient in the sense that it does not lead to substantial degradation in circuit area and performance.

The remainder of this paper is organized as follows. Section 2 (Reconfigurable Computing) discusses some benefits and shortcomings of reconfigurable technology that are addressed by SCORE. Section 3 ("The SCORE Compute Model") describes the SCORE model, and in particular, the features of its programming model that make it amenable to analysis. Section 4 ("Synthesis and Partitioning Methodology") presents an approach for compiling the programming model, focusing primarily on page partitioning. Section 5 ("Evaluation Methodology") proposes studies for evaluating the partitioning methodology within the existing SCORE software framework. Section 6 ("Proposed Schedule") recounts the particular tasks to be done for this thesis work and proposes an 18-month schedule for completing those tasks. Related work is sprinkled throughout the paper, as needed.

2 Reconfigurable Logic

A *reconfigurable* computing device is a field-programmable processor that can implement arbitrary custom data-paths using a sea of programmable logic/arithmetic elements and a programmable interconnect. Examples include complex programmable logic devices (CPLDs), field-programmable gate arrays (FPGAs), and more coarse-grained sea-of-ALU and sea-of-processor devices such as Chameleon RCP [10] and Pact XPU. Reconfigurable devices have been used successfully for a large variety of application domains, including for instance, image compression [32], DNA sequence matching [4], boolean satisfiability [36], and encryption [14]. In many cases, reconfigurable devices realize 10-100x gains in performance and in functional density (operations per area-time) over microprocessors [13]. This computational power comes from the combining of spatial execution with programmability. The spatial computing style can exploit the fine-grained parallelism of an application by executing it directly as a circuit or network of parallel components. Programmability enables customizing and specializing the device's data paths to the problem at hand without the costly and lengthy process of creating a new, custom chip.

Dynamic reconfiguration, *i.e.* changing device function during program execution, is a powerful but underused consequence of the programmability in reconfigurable devices. Conceptually, this feature enables delayed binding of the device's function, so that the function can be more specialized to the problem being solved. One use of late binding is data-driven mode selection, for example in MPEG video encoding, loading a separate configuration for each frame depending on whether its type is I, P, or B. Another use of late binding is hardware virtualization, *i.e.* the phasing of a computation larger than the device as a sequence of configurations, loaded as needed. Yet another use of late binding is specialization around constants, as in the SAT solver of [36] that generates a custom configuration for each boolean formula to be analyzed.

In the commercial market, reconfigurable logic devices are used primarily as ASIC replacements, executing only a single, static configuration. This usage ignores dynamic reconfiguration and thus neglects some of the most powerful performance benefits of reconfigurable technology such as dynamic specialization. Part of the problem is that commercial EDA tools for reconfigurable devices presently mirror ASIC tool flows, with high optimization for single, static configurations, and no systematic support for dynamic reconfiguration. More generally, there is no accepted model for conveniently expressing programs that rely on dynamic reconfiguration in a high-level, device-independent manner, without sacrificing substantial device performance. SCORE [8] [9] provides one such model.

3 The SCORE Compute Model

SCORE (Stream Computations Organized for Reconfigurable Execution) is a compute model for scalable reconfigurable computing. As a compute model, SCORE provides a framework for specifying program behavior, including language semantics for a programming model and execution semantics for an execution model. The model supports scalability in the sense that an application written and compiled for SCORE will automatically run on any architecturally compatible device and will automatically run faster on a larger device. This scalability is rooted in a hardware virtualization model based on paging, where a program is sliced into fixed-size, virtual hardware pages (analogous to virtual memory pages) that are automatically loaded and executed on available physical pages. Efficient virtualized execution depends critically on good program analysis and page synthesis. This section describes the key elements of the SCORE model, and in particular of the programming model, that lend well to analysis for synthesis. For a more complete treatment of SCORE, see [9].

The SCORE model was designed to address several properties and shortcomings of reconfigurable computing technology. The use of paging enables software to survive and scale to next-generation hardware without recoding and recompilation (this is a property enjoyed by microprocessors that was key to the success

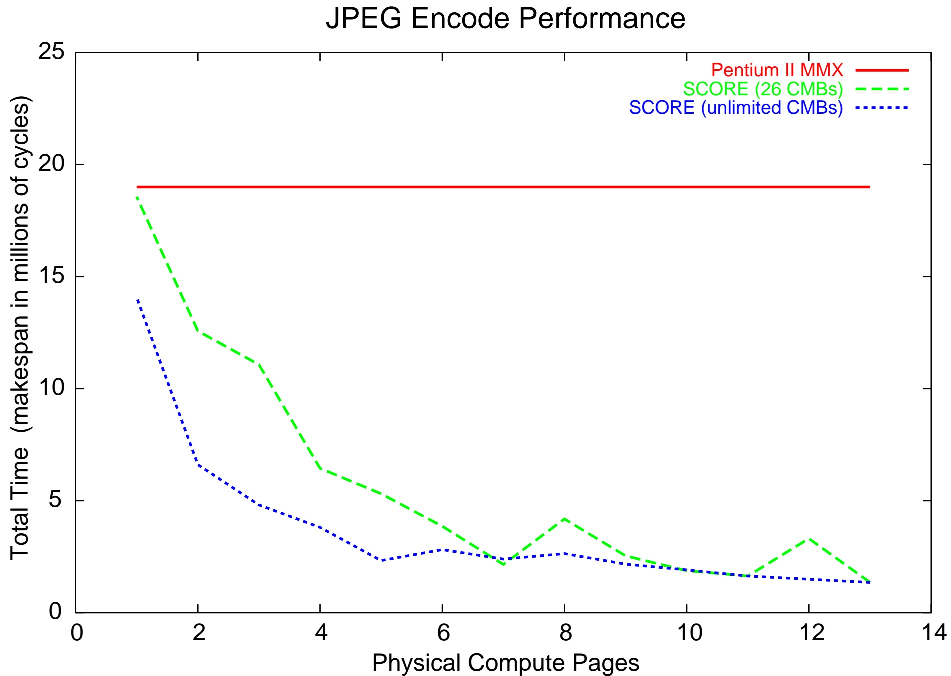

Figure 1: Run-times for JPEG encoding of Lena's face on different size devices (copied from [9]; CMB refers to on-chip memory blocks)

of their commercial software market). Most existing programming tools for reconfigurable systems expose the capacity of hardware resources (logic, IO, memory) to the programmer. This forces the programmer to do manual resource management and to make device-dependent algorithmic decisions so that the circuit will fit into hardware. Paging enables resource hiding in the programming model, helping to separate algorithmic decisions from device implementation. Paging also enables binary compatibility of applications on any page-compatible device, as well as automatic performance improvement on larger devices. Paging enables an automatic area-time tradeoff for choosing a device size, as demonstrated in Figure 3 for JPEG encoding on a collection of (simulated) devices with different page counts. Furthermore, paging can increase the effective functional density of a device (operations per area-time) by allowing less active components of the computation to be swapped out rather than consume valuable device area. This is evidenced in Figure 3 by the fact JPEG encoding can run in a device half the size of the application with negligible performance penalty.

SCORE's use of streaming is motivated by the high cost of device reconfiguration (typically microseconds to milliseconds) and the need to hide that cost for efficient virtualization and dynamic reconfiguration. Existing approaches to reducing or hiding reconfiguration time have limitations. For example, the use of a configuration cache to reduce reconfiguration time to the order of cycles (*e.g.* DPGA [12],

Xilinx TMFPGA [31]) incurs cost in device area (for the cache) and power (for frequent reconfiguration). The use of configuration stripes to allow pipelined reconfiguration (*e.g.* PipeRench [17]) effectively limits allowable circuit topologies to be feed-forward. Streaming can reduce reconfiguration frequency by allowing a loaded configuration to batch-process large amounts of data. This effectively amortizes the cost of each reconfiguration over a large data set. Streaming data flow models (*e.g.* SDF [26]) support such batched execution since their token-flow semantics allow but do not require a process to act whenever it has inputs.

The SCORE model combines the above ideas in representing a computation as a streaming data-flow graph of compute operators, where operators are autonomous tasks that may have internal state. The abstract model is a special case of Kahn process networks [20] in that streams act as unbounded-capacity FIFOs; stream reads are blocking; stream writes are non-blocking; and operators are deterministic and continuous (in the Kahn sense of prefix order). The model is in fact a special case of data-flow process networks [25] in that each operator has a set of firing rules that describe the inputs required at each step of execution. The firing rules are sequential and will be sequenced by a finite state machine. SCORE operators using these firing rules are referred to as streaming finite state machines (SFSM). The model supports segmented memory in that a memory block (segment) can be encapsulated in a data-flow operator and given a sequential or random-access stream interface to the rest of the graph (single reader, single writer). The SCORE model also allows for meta-operators, referred to as streaming Turing machines (STM), that have the added ability to allocate new computational graphs (operators, streams, and segments). The analysis of a SCORE program for synthesis, scheduling, and execution is phased by such allocations and is done separately for each intermediate computational graph.

A salient feature of the SCORE model is that its primitives are used consistently in the programming model (*i.e.* concrete language) and in the execution model (*i.e.* technology-specific page configurations). In both cases, a computation is a streaming data-flow graph of state machine operators. The difference is in the binding of resource constraints. Operators in the execution model are hardware pages with fixed area, fixed number of IOs, and technology-specific timing. Operators in the programming model have no such constraints on size, IO, timing, or other complexity. Similarly, streams in the execution model have finite physical buffering, whereas streams in the programming model have unbounded capacity. Thus, compilation and page synthesis for SCORE can be formulated as a mapping from one SCORE graph to another (*i.e.* a transformation on streaming state machines) such that the resulting operators satisfy hardware constraints.

The SCORE model is expressive enough to be undecidable in the sense that a program is not guaranteed to terminate nor to execute with bounded stream depth. This is potentially a problem, since SCORE streams have semantically unbounded capacity but must execute with physically finite buffers on any real device (in effect,

```
select (input  boolean    s,
        input  signed[16] t,
        input  signed[16] f,
        output signed[16] o)
{
  state S(s):  if (s) goto T; else goto F;
  state T(t):  o=t;   goto S;
  state F(f):  o=f;   goto S;
}
```

Figure 2: Canonical select actor in TDF

stream writes to a full stream become blocking in the execution model). This problem cannot be resolved by page synthesis and is in fact resolved by supporting transparent buffer expansion in the execution model. To allow a program with unbounded memory requirements to operate after filling a bounded stream buffer, the SCORE run-time page scheduler suspends the program, allocates a new buffer segment, reroutes the offending stream to buffer through that segment, and resumes the program.

The SCORE programming model provides a concrete language, the Task Description Format (TDF) [7], for describing SCORE's streaming state machine primitives and their composition in data flow graphs (TDF describes SFSMs but not STMs). An operator in TDF is an extended finite state machine (EFSM) with special semantics for streaming IO and firing rules. Each state has an associated input signature to specify the inputs required for that state to fire, plus a firing action. The execution semantics are that, upon entering a state, the operator will issue blocking reads to all streams in that state's input signature. Once the input tokens are available, the operator will fire, *i.e.* consume the tokens and execute the associated action. The firing action consists of statements in a subset of C syntax that has no looping constructs (for, while) and uses goto to specify a state transition. An operator may have state in the state machine as well as in register variables that are visible in all firing actions.

Figure 3 shows an example TDF operator that implements a token selector (this is the canonical *select* actor from Lee's SDF [26] and Buck's BDF [3]). This operator selectively consumes and copies a token from one of its two inputs (`t`, `f`) to its output (`o`) depending on the value of a boolean select input (`s`). Because the firing rules match on input presence but not on input value, the state machine must first consume `s` then branch to consume either `t` or `f`. This is not necessarily the most efficient execution sequence, and this program can in fact be pipelined to consume `s` simultaneously with a previous `t` or `f`. Although the input signature of

a state is static, the output signature can be dynamic, since the TDF syntax allows stream write operations (assignment to a named output stream) to be conditioned inside an if-statement. Nevertheless, it is possible to transform a program to have static output signatures by decomposing if-statements into state branchings, such that output operations are exposed unconditioned in another state.

It is interesting to note that TDF implements a special case of data-flow process networks [25]. An actor in a process network is controlled by a set of firing rules and can fire on any rule whose pattern of input values/presences matches available inputs. An actor is functional but can represent state using a feedback arc. Hence, a TDF operator is equivalent to a process net actor, where the state is a feedback arc, and where each state's firing rule is extended in the actor to match the state's number on the state feedback arc. Because TDF firing rules match input presence but not value, the firing rules of the actor do not value-match any input except the state feedback arc. This style of firing rules is guaranteed to be sequential and deterministic.

4 Synthesis and Partitioning Methodology

The problem of partitioning a SCORE computation into pages is unique for a number of reasons. Partitioning must be sensitive to the cost model of paged execution, in particular the fact that inter-page communication delay is not known and may be quite long. Inter-page delay depends on page placement and on details of the interconnect fabric, which may differ for different devices of a page-compatible family. Under virtualization, inter-page delay also depends on the page loading schedule and may include page swapping delay. It is critical to cluster computational feedback paths within pages whenever possible, since feedback loops cannot be pipelined and must incur the full loop latency.

Traditional approaches for structural circuit partitioning (*e.g.* wire min-cut [23] [29] [34], delay-optimal DAG covering [28] [35]) are not appropriate for SCORE because their objective functions do not address communication delay, in particular delay around feedback paths. The same is true for most existing approaches to finite state machine decomposition (*e.g.* minimize logic [1], minimize wires [24], minimize power [2]). The approaches of [1] and [24] are particularly bad for delay because their decompositions rely on tight feedback loops between state machine partitions (sub-machines).[1] The approach of [2] bears some relevance because it minimizes communication frequency between sub-machines. However, none of these approaches directly treat the decomposition of data-paths attached to the FSMs. Ignoring the delay and communication structure of those data-paths can lead to

[1] [1] describes a methodology for FSM decomposition with arbitrary topology for sub-machine communication. In this framework, the only topology that avoids tight feedback between sub-machines, and hence the only topology that might be useful for page partitioning, is the purely feed-forward (cascade) decomposition.

Compiler Optimizations
Pipeline Extraction
Data Path Mapping
Partition Large States
Schedule DF into State Sequences
Cluster States
Pack Pages
Synthesize Page FSM

Figure 3: Proposed passes for page synthesis and partitioning

partitions with disastrous inter-page communication delay and bandwidth.

The multi-rate and potentially dynamic-rate nature of SCORE suggests that it may be profitable to cluster for the "common case," *i.e.* to cluster high-activity feedback paths at the expense of cutting lower-activity ones. Reducing the incidence of inter-page communication in this manner is useful for reducing the total sequential delay of inter-page communication as well as reducing the run-time storage required to buffer virtualized streams. The observations of this and the previous paragraph suggest that efficient partitioning for SCORE cannot be structural, *i.e.* at the circuit level. It must rely on functional information from the program structure and perhaps on profiling information from previous executions.

Partitioning for SCORE must manage a number of optimization goals. First and foremost, it must avoid inter-page communication delay. It must also strive to minimize inter-page bandwidth, to minimize total area (*i.e.* minimize overhead of control logic and page fragmentation), and to maximize page utilization (avoid having pages or parts of pages sit idle). Given the complexity of the partitioning task, my thesis work will focus first on managing inter-page delay. To this end, we must consider not only clustering but also program transformations to directly reduce the size of feedback loops.

Figure 4 shows the proposed flow for page synthesis and partitioning. Partitioning using high-level, functional information is done not in a single pass but in a collection of passes distributed throughout the synthesis flow. The remainder of this section describes components of the proposed flow, in three major categories: optimizations, scheduling, and partitioning. We concentrate on the latter.

4.1 Optimizations

The synthesis flow begins with a place-holder for a variety of possible, traditional compiler optimizations. Classic work-reducing optimizations include constant folding, copy propagation, subexpression elimination, hoisting loop invariants out of loops, and strength reduction. A variety of transformations to enhance ILP (instruc-

tion level parallelism) are also possible, ranging from arithmetic tree rebalancing to loop unrolling and software pipelining (a relevant hardware adaptation of software pipelining is described in [5]). TDF is amenable to much of the existing lore of compiler optimizations because its state flow structure can be analyzed in much the same way as traditional basic block flow. In fact, it should be possible to adapt an existing imperative language compiler such as SUIF [18] [33] to optimize TDF. Note, however, that many of the more powerful loop transformations of parallelizing compilers (*e.g.* loop splitting, loop fusion) are not applicable to the streaming style of SCORE, since operators do not have random access to their inputs. The net benefit of any given transformation depends on the hardware cost model and may in fact be negative. Unfortunately, determining that benefit may require a complete synthesis pass (*e.g.* loop unrolling expands circuit size and may actually reduce performance due to inter-page delay incurred in partitioning this larger circuit). My thesis work will implement some of the low-hanging fruit among optimizations (*e.g.* constant folding), but aggressive compiler optimization must be considered beyond the scope of this thesis.

4.2 Scheduling

The "Data Flow Mapping" phase of Figure 4 assigns device-dependent area and delay costs to each data-path operation of a TDF program. For FSM-based execution on fixed-frequency hardware, this step actually requires a timing explosion, whereby each data-path operation is decomposed into a sequence of single-cycle steps (*e.g.* partial products of a multiplier). This can be naturally handled by a module-based synthesis approach, where the sequence of single-cycle steps is either read from a data-path library or is synthesized using a tree grammar of allowable single-cycle operations. For the latter approach, we could leverage the BURG approach of the Garp C compiler [6] (developed within the Berkeley BRASS group).

Time-exploded operations should subsequently be scheduled within the state machine to enhance pipeline parallelism and to reduce register usage. Some parallelism can be recovered by traditional DAG scheduling of the exploded state sequences (levelizing the circuit, scheduling ASAP/ALAP/*etc.*, and treating each level as a state). More parallelism could be recovered by software pipelining a loop after the timing explosion. For greatest generality we can apply traditional code motion techniques [27] [22], whereby an operation is pulled back as early as possible, across states and state branchings.

4.3 Partitioning

Page partitioning for SCORE can be treated as two conceptually separate tasks: decompose large operators to fit in a page, and cluster small operators to pack a page. The former (decomposition) requires non-trivial transformations to partition

streaming state machines and must be done in a manner sensitive to their tight, internal feedback structure. The latter (page packing) is aimed at reducing page fragmentation but should also be sensitive to inter-operator feedback. In the methodology proposed here, these two tasks are handled in a sequence of separate passes. In theory, a sequenced approach may be less effective than a well-constructed, single-pass, globally-optimizing approach. However, a single-pass approach is restricted to optimizing a single metric. A sequenced, multi-pass approach has the advantage of being able to optimize different metrics at different levels of granularity, and in doing so, using more kinds of functional information from the program's intrinsic structure and hierarchy. Phases of the proposed multi-pass approach are described below.

Pipeline Extraction

Finite state machines naturally contain feedback paths involving registers and logic that must be traversed for each execution cycle. Data-path feedback may in fact stretch over multiple execution cycles, since deep data flow can be sequenced over multiple states before a registered result reconverges or is used for conditional state branching. Such feedback paths can be problematic for page partitioning if they must be cut and placed across multiple pages. *Pipeline extraction* is a mechanism for shrinking those feedback paths by hoisting parts of the data flow out of the state machine. In particular, data flow that is applied unconditionally to a stream input or output is not truly controlled and need not be sequenced by the state machine. That data-flow can be converted into a feed-forward stream transformation, *i.e.* an input or output pipeline, that resides in its own operator(s) outside the state machine. A simple TDF example is shown in Figure 4.3, where input `x` is always compared to zero, regardless of when it is read, and can therefore be hoisted out to an input pipeline.

The benefit of pipeline extraction is primarily to simplify the cyclic core of state machine operators, so as to reduce delay and avoid large feedback paths. Extracted pipelines have an additional benefit for partitioning, since they are static-rate, feed-forward structures. Hence, pipelines can be analyzed as DAGs rather than FSMs, enjoying more freedom for scheduling, clustering (*e.g.* serial composition), and cutting. This property is useful in the later phase of page packing. Note that Figure 4.3 shows the extracted pipeline in TDF state machine syntax, but this is primarily for clarity, and the operator is in fact stateless (single-state, no registers).

Pipeline extraction has already been implemented in the TDF compiler. Figure 4.3 shows the effect of pipeline extraction on 47 operators from a variety of applications coded in TDF (JPEG, MPEG, Wavelet, IIR). For each operator, the figure shows the operator's data-path area (estimated from a module-based data-path cost library) broken down into extracted and unextracted components. Many operators benefit from partial extraction. Some operators can be extracted in their entirety—this is the case for stateless operators, where the state machine syntax serves only

```
foo (input unsigned[16] x, ...)
{
  state bar (x):
    if (x==0) ...
}
```

(a)

```
foo (input boolean xz, ...)
{
  state bar (xz):
    if (xz) ...
}

pipe (input unsigned[16] x,
      output boolean xz) {
  state only(x):  xz=(x==0);
}
```

(b)

Figure 4: Pipeline extraction example, (a) before, (b) after

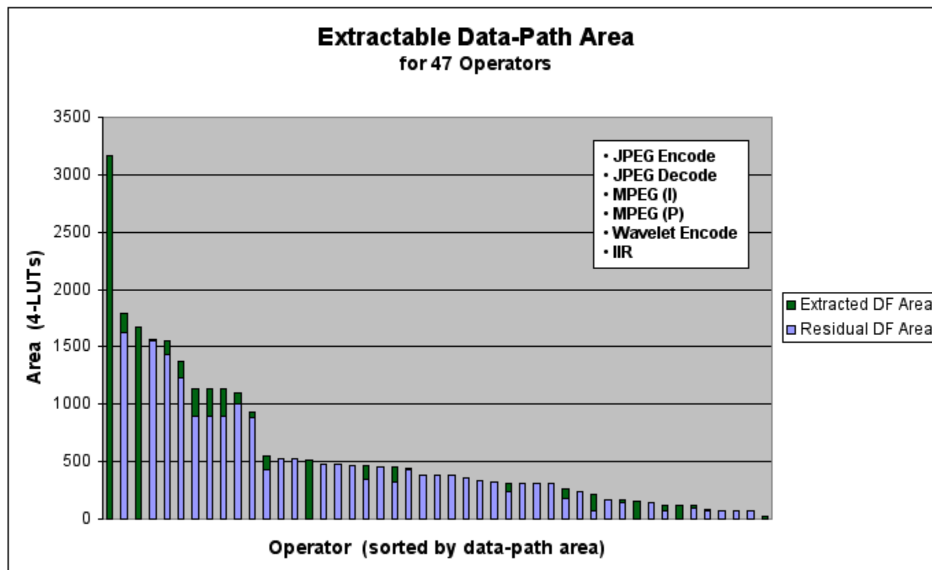

Figure 5: Pipeline extraction results

to specify an input signature (more generally, it should be possible to convert into a pipeline any FSM that represents a cyclo-static data-flow operator with no internal, register-based feedback).

State Clustering

The state clustering transformation is the primary method by which SFSMs are decomposed into page-size chunks. Our proposed approach for partitioning SFSMs is targeted at minimizing the sequential delay of inter-page communication, with a secondary objective of reducing inter-page bandwidth. A key observation is that the sequential delay of an FSM is captured directly in its state flow. We cluster the logic of communicating states into pages so as to avoid the latency of transferring to a next state on a different page. In this decomposition model, control is active in only one page of a partitioned state machine at a time, and control transfer to another page requires the transmission of an *activation token*. We can optimize "for the common case" by striving to cluster common state sequences (loops) within pages, so that an SFSM can maintain control within a page for extended periods and only infrequently incur the sequential delay of transferring control to states on another page. This approach is similar to trace scheduling for VLIW processors [16], where "traces" or common paths of control flow through an instruction sequence are scheduled and optimized as units. Under virtualization, pages that contain particularly infrequent states (*e.g.* special case handling) may be swapped out of hardware altogether. This is an example where virtualization yields higher functional density by avoiding the allocation of inactive circuits in hardware.

We formulate delay-optimal SFSM decomposition as a partitioning of the state transition graph to minimize the cut of state transition probabilities (as edge weights), under area and IO constraints. Note that min-cutting the state transition probabilities directly captures the notion of minimizing inter-page state transfer. Each state node represents the complete logic associated with that state (input signature, data-path from firing action, next-state logic) and is assigned an appropriate area for its node weight.

Multi-way min-cut partitioning under area constraints can be done using the balanced min-cut approach of Yang and Wong [34]. Theirs is an iterative approach capable of rejecting partitions that do not fall within a specified area bound. The approach is flow-based, using the Ford-Fulkerson augmenting paths method, to repeatedly cut between randomly-chosen source and sink nodes. Although their algorithm requires multiple min-cut computations, it achieves the same asymptotic cost as a single min-cut by retaining intermediate residual flows across iterations. The basic algorithm for selecting an area-constrained partition is shown in Figure 4.3.

Yang and Wong's approach is also able to constrain the IO wires of accepted partitions, but this rests in the fact that their min-cuts are performed directly on structural wire graphs (*i.e.* netlists). This is not directly applicable in our formulation of cutting state transitions, where wires are not represented. One way to

```
randomly select source and sink nodes.
repeat:
  min-cut to produce source partition X, sink partition X'.
  if (A_min <=area(X)<= A_max)
    accept partition X
  else if (area(X)< A_min)
    collapse X into source,
    collapse additional node from X' into source.
  else if (area(X)> A_max)
    collapse X' into sink,
    collapse additional node from X into sink.
```

Figure 6: Area-constrained partition selection in the Yang+Wong balanced min-cut partitioning approach

add IO constraints to our formulation is to augment the graph with inter-state data flow edges representing wires, with each such edge weighted by its bit width (an additional wire is created for each state transition edge to represent its associated activation stream). Since the weights of bit-widths and state transition probabilities are not comparable, they must be scaled and mixed for summing in the min-cut algorithm. A linear mix can be used whereby wire bit widths are scaled by c and transition probabilities are scaled by $1-c$. Parameter c ($0 \leq c \leq 1$) becomes a dial for choosing the objective of the min-cut: $c=0$ considers only transition cuts, while $c=1$ considers only wire cuts. To choose a partition that is transition-rate-optimal, area-constrained, and IO-constrained, the partition selection algorithm of Figure 4.3 should be altered to choose the area-feasible cut with minimum c. This can be done by wrapping the algorithm in a linear or binary search that varies c. The increase in algorithmic complexity can be made a constant factor by limiting the number of c values searched for each partition (*e.g.* binary search with a depth of 4).

Multi-way partitioning using area-constrained balanced min-cut has already been implemented in the SCORE TDF compiler. IO constraints and c-searching have yet to be added.

Breaking Large States

The state clustering algorithm presented above requires that each state node itself satisfy the area and IO constraints of a page. States that do not satisfy these constraints must themselves be partitioned. The logic of any given state is a DAG, so it can be partitioned using existing DAG partitioning and covering techniques [11] [21] [35]. Covering may decompose a state into several parallel pages. This nominally breaks the assumption of the state clustering algorithm that only a single page of a partitioned SFSM is active at a time. That assumption is key for equating

a min-cut on state transition probabilities with minimization of sequential inter-page delay. To restore this assumption, the pages of each partitioned state should be pre-clustered into a single state node for state clustering (such a node will violate area and IO constraints and requires special casing in the algorithm; no external nodes will be merged with it).

Page Packing

To reduce page fragmentation, the extracted pipelines and partitioned SFSMs from previous passes should be packed together into pages. This can be done using the balanced min-cut algorithm of Yang and Wong, this time cutting a conventional netlist to minimize wire cuts. Because SFSMs are indivisible at this point, they are represented in the graph as individual nodes. Pipelines, on the other hand, can and should be cut, so their components are exposed in the graph as separate nodes. The balanced min-cut approach will naturally merge and re-cut serially-composed pipelines.

The balanced min-cut algorithm must be modified to reject certain partition topologies. For instance, it is not possible to cascade two dynamic data-flow operators in one page, since the link between them (a stream) may require unbounded buffering, and that is not viable in page logic. Since SFSMs may have dynamic rates, the conservative approach is to allow only a single SFSM per page, possibly clustered with input and output pipelines. In addition, parallel composition of SFSMs is restricted by the page hardware model, which may limit the number of independent SFSMs supported per page (a product SFSM composed of several independent SFSMs would have multiple firing rules enabled in each product state, which must be supported by the page's firing logic, if any).

Note that page packing by balanced min-cut clustering does not directly address feedback loops between user-specified operators. Such loops can be clustered into a page only if their constituent SFSMs are small enough to simultaneously fit in a page. A more general methodology is needed to allow the clustering of communicating states from two separate operators without clustering the entire operators. A possible approach is to generalize the area- and IO-constrained state clustering algorithm to operate on a graph representing the states of all operators simultaneously.

5 Evaluation Methodology

The effectiveness of the proposed partitioning techniques will be evaluated by measuring their overhead on circuit area and performance. Partitioning transformations are expected to introduce area overhead in control logic (as additional states in decomposed SFSMs) and in page fragmentation, as well as delay overhead in inter-page communication. This overhead will be measured for a variety of applications over

a variety of different page sizes. The basis of comparison will be an idealized, unpartitioned implementation of the given application, synthesized using the same area/time cost model, but assuming that hardware pages are always precisely the right size for the operator mapped into them. Overhead can be taken as the ratio of the desired metric between the partitioned and unpartitioned implementations. Note that this approach does not compare the absolute performance of our synthesis and partitioning methodology to any other system. Rather, it evaluates the effectiveness of particular transformations strictly within the framework of our synthesis methodology.

In this framework, a number of interesting studies can be formulated regarding overhead for different page parameters. Pure area overhead can be measured as a circuit property, without executing an application. The characteristic graph of area overhead over different page sizes is shown in Figure 5(a) (nominally with unconstrained page IO). We expect that too small a page size will lead to control overhead, whereas too large a page size will lead to fragmentation overhead. A minimum overhead point should exist for some intermediate page size. Although that area-optimal page size is application-dependent, it would be a strong result to find that it is similar across a set of applications. An analogous characteristic graph can be found for area overhead over different page IO capacities (nominally with unconstrained page area), as shown in Figure 5(b). Because the page area overhead for each IO port is small, this graph is not likely to suffer from fragmentation (*i.e.* excessive overhead from unused wires) for large page IO capacity. But this analysis does not model the device area overhead for using a correspondingly richer interconnect network— a characteristic graph accounting for network area will probably have a local minimum in the middle. More generally, we can generate a combined, 3D graph of area overhead versus page area and page IO capacity. The minimum curve and isoclines of the surface should reveal an interesting tradeoff between page area and IO.

Pure performance overhead can be measured separately from area cost by examining the total run-time (makespan) of an application executed fully spatially, *i.e.* executed on a device large enough to hold the partitioned application without virtualization. The characteristic graphs of interest are makespan overhead for different page sizes (Figure 5(a)) and for different page IO capacities (Figure 5(b)). In the absence of virtualization, makespan overhead is due primarily to communication delay, which always benefits from fewer partitions and cuts. Hence, the characteristic graphs should be monotonically decreasing for larger page sizes and page IO capacities.

A more realistic formulation of performance overhead is to compare the execution of partitioned and unpartitioned circuits on identically-sized devices. Because a partitioned circuit has area overhead, it will naturally incur an associated performance overhead for virtual execution of that larger circuit (due to page swapping and stream buffer management). Figure 5(a) shows the characteristic graph of

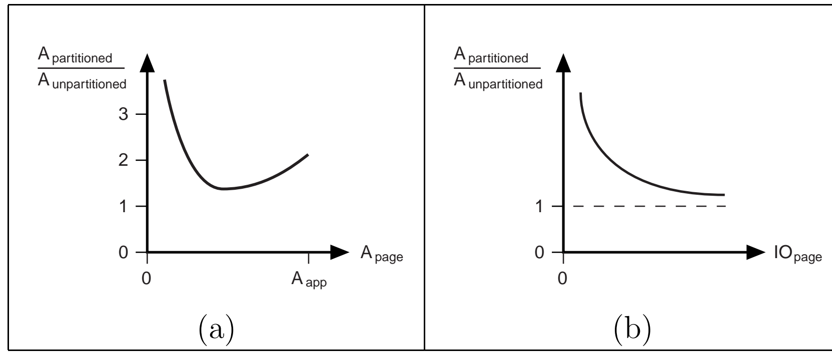

Figure 7: Hypothetical characteristic graphs of pure area overhead from partitioning (a) over different page sizes, (b) over different page IO capacities

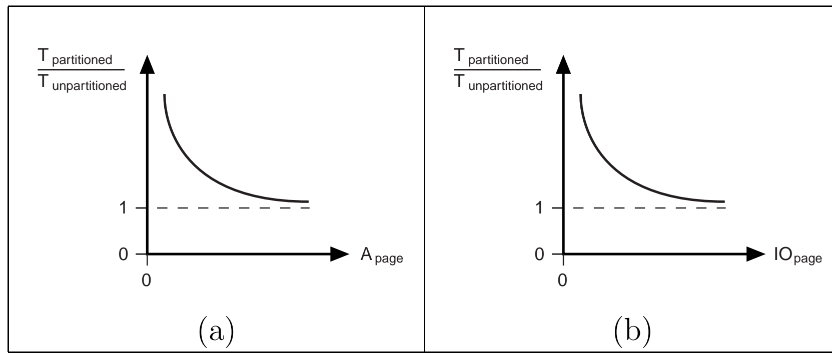

Figure 8: Hypothetical characteristic graphs of pure performance overhead from partitioning (a) over different page sizes, (b) over different page IO capacities

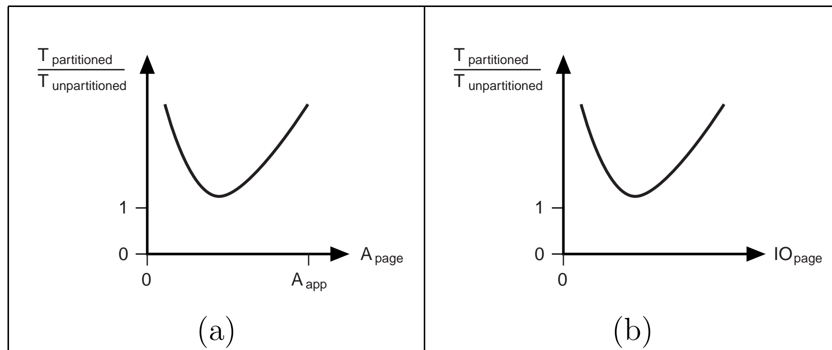

Figure 9: Hypothetical characteristic graphs of performance overhead from partitioning, executing in a fixed device size, (a) over different page sizes, (b) over different page IO capacities

makespan overhead over different page sizes, for a fixed device size. We expect to find a minimum point for some performance-optimal page size, by the same rationale as in Figure 5(a). As before, an analogous characteristic graph exists for makespan overhead over different page IO capacities. And as before, a combined, 3D graph can be generated for makespan overhead over different page size and page IO capacity. In each of these graphs, different curves/surfaces can be generated for different total device size. It would be interesting to rotate these graphs to show makespan overhead versus total device size, so as to see whether partitioning has different overhead for different degrees of virtualization.

6 Proposed Schedule

The design and experimental work for this thesis will be done within the existing SCORE software infrastructure, where substantial prior work can be leveraged. Synthesis and partitioning will be implemented as passes in the existing SCORE compiler, and performance evaluation will be done using the existing SCORE simulator. The TDF language compiler (`tdfc`) is presently capable of translating a TDF program into working, compilable, page simulation code in C++ that runs in tandem with an existing device simulator and run-time page scheduler. The translation treats SFSMs as structurally sound and generates simulation code to execute them verbatim, whether they satisfy hardware constraints or not (page area, IO timing). In addition, the compiler contains some initial work on synthesis and partitioning, including data-path cost estimation using an ad-hoc cost library, a pipeline extraction algorithm, an area-constrained state clustering algorithm, and a back-end to emit state machine logic to SIS [30] for FSM synthesis. A number of media-processing applications (*e.g.* JPEG, MPEG) have been written in TDF and successfully compiled, simulated, and analyzed in this framework.

Remaining work for this thesis can be categorized into several phases. I will first complete the initial implementation of synthesis and partitioning using the methodology of Section 4. Evaluating the effectiveness of this methodology as described in Section 5 will help pinpoint deficiencies in the methodology. I will address some of those deficiencies in a secondary implementation, including some optimizations and alternative partitioning formulations (ideas for which are sprinkled in this thesis proposal). I also plan to implement at least one additional application from the domain of the outside committee member, namely data analysis for a particle detector. Work will be presented in a written thesis and in conference/journal papers along the way. Some details for each phase of labor are listed below, and a proposed 18-month work schedule is shown in Figure 6. The schedule was designed with a factor of 2 slowdown to account for technical difficulties, occasional paper writing, and family life.

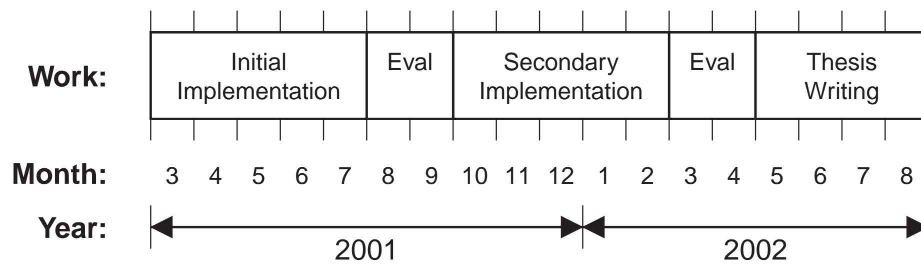

Figure 10: Proposed work schedule

Complete initial implementation

- Complete state clustering
- Decompose large states
- Page packing
- Bind data-path timing and reschedule into states
- Synthesize page SFSMs

Secondary implementation (possibilities)

- Optimizations: code motion, software pipelining
- State clustering with state logic duplication?
- Unified state clustering on all operators simultaneously?
- Recast TDF as BDF and cluster to min-cut stream activity?

Evaluation

- Review/modify existing applications for good style
- Manually decompose large states
  (enables partitioning/evaluation before coding automatic state decomposition)
- overhead studies

7 Acknowledgements

Thanks to all the members of the SCORE team who contributed to the software infrastructure on which this work is based. Michael Chu, Randy Huang, and Yury Markovskiy wrote the SCORE simulator and page scheduler. André DeHon and Laura Pozzi wrote the `tdfc` compiler with me and contributed greatly to the work in this paper. Joe Yeh wrote the SCORE application suite.

This work is part of the Berkeley Reconfigurable Architectures, Software, and Systems (BRASS) project, supported by DARPA (contract number DABT63-C-0048), by the California MICRO program, and by grants from STMicroelectronics.

References

[1] P. Ashar, S. Devadas, and A. R. Newton. Optimum and heuristic algorithms for an approach to finite state machine decomposition. *IEEE Trans. on Computer-Aided Design*, 10(3):296–310, March 1991.

[2] L. Benini, G. De Micheli, and F. Vermeulen. Finite-state machine partitioning for low power. In *Proc. IEEE Int'l Symposium on Circuits and Systems (ISCAS '98)*, volume 2, pages 5–8, Monterey, California, May31–June3, 1998.

[3] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model.* PhD thesis, University of California, Berkeley, 1993. ERL Technical Report 93/69.

[4] Duncan Buell, Jeffrey Arnold, and Walter Kleinfelder. Searching genetic databases on splash 2. In *Proc. IEEE Workshop on FPGAs for Custom Computing Machines (FCCM '96)*, pages 97–109, Napa Valley, California, April 15–17, 1996.

[5] Timothy J. Callahan and John Wawrzynek. Adapting software pipelining for reconfigurable computing. In *Proc. Int'l. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2000*, San Jose, California, November 17–18, 2000. Available as: `http://brass.cs.berkeley.edu/documents/swp4rc.ps`.

[6] Timothy J. Callahan and John Wawrzynek. The garp architecture and c compiler. *IEEE Computer*, April 2000.

[7] Eylon Caspi. Programming SCORE. BRASS group internal technical report, April 2000.

[8] Eylon Caspi, Michael Chu, Randy Huang, Nicholas Weaver, Joseph Yeh, John Wawrzynek, and André DeHon. Stream computations organized for reconfigurable execution (score): Extended abstract. In *Conference on Field*

Programmable Logic and Applications (FPL '2000), LNCS, pages 605–614. Springer-Verlag, August 28–30 2000.

[9] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, Yury Markovskiy, John Wawrzynek, and André DeHon. Stream computations organized for reconfigurable execution (SCORE): Introduction and tutorial. http://brass.cs.berkeley.edu/documents/score_tutorial.pdf, August 2000.

[10] Chameleon systems, inc. http://www.chameleonsystems.com.

[11] Jason Cong and Yuzheng Ding. Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based fpga designs. *IEEE Transactions on Computer-Aided Design*, 13(1):1–12, January 1994.

[12] André DeHon. Dpga-coupled microprocessors: Commodity ics for the early 21st century. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1994.

[13] André DeHon. *Reconfigurable Architectures for General-Purpose Computing*. PhD thesis, MIT, 545 Technology Sq., Cambridge, MA 02139, September 1996.

[14] A. J. Elbirt and C. Paar. An fpga implementation and performance evaluation of the serpent block cipher. In *Proc. International Symposium on Field Programmable Gate Arrays (FPGA 2000)*, pages 33–40, Monterey, California, February 10-11, 2000.

[15] C. M. Fiduccia and R. M. Mattheyses. A linear time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, pages 175–181, 1982.

[16] Joseph Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[17] Seth C. Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. Piperench: a coprocessor for streaming multimedia acceleration. In *Proceedings of the 26th International Symposium on Computer Architecture (ISCA'99)*, pages 28–39, May 1999.

[18] Stanford SUIF Compiler Group. SUIF compiler system, http://suif.stanford.edu.

[19] L. Hagen and A. Kahng. New spectral methods for ratio-cut partitioning and clustering. *IEEE Trans. on Computer-Aided Design*, 11(9):1074–85, September 1992.

[20] G. Kahn. Semantics of a simple language for parallel programming. *Info. Proc.*, pages 471–475, August 1974.

[21] Kurt Keutzer. Dagon: Technology binding and local optimization by dag matching. In *Proceedings of the 24th ACM/IEEE Design Automation Conference (DAC)*, pages 341–347, 1987.

[22] Jens Knoop and Oliver Rüthing. Optimal code motion: Theory and practice. *ACM Trans. on Programming Languages and Systems*, 16(4):1117–1155, July 1994.

[23] B. Krishnamurthy. An improved min-cut algorithm for partitioning vlsi networks. *IEEE Trans. on Computers*, C-33(5):438–446, May 1984.

[24] Ming-Ter Kuo, Lung-Tien Liu, and Chung-Kuan Cheng. Finite state machine decomposition for i/o minimization. In *Proc. IEEE Int'l Symposium on Circuits and Systems (ISCAS '95)*, volume 2, pages 1061–4, Seattle, Washington, April28–May3, 1995.

[25] E. A. Lee and T. M. Parks. Dataflow process networks. *Proc. IEEE*, 83(5):773–801, May 1995.

[26] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[27] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[28] Rajmohan Rajaraman and D. F. Wong. Optimal clustering for delay minimization. In *Proc. 30th Int'l Conf. on Design Automation Conference (DAC '93)*, pages 309–314, Dallas, Texas, June 14–18, 1993.

[29] L. A. Sanchis. Multiple-way network partitioning. *IEEE Trans. on Computers*, 38(1):62–81, January 1989.

[30] Ellen M. Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Sis: A system for sequential circuit synthesis. UCB/ERL M92/41, University of California, Berkeley, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, May 1992.

[31] Steve Trimberger, Dean Carberry, Anders Johnson, and Jennifer Wong. A time-multiplexed fpga. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 22–28, April 1997.

[32] John Villasenor, Chris Jones, and Brian Schoner. Video communications using rapidly reconfigurable hardware. *IEEE Transactions on Circuits and Systems for Video Technology*, 5:565–567, December 1995.

[33] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer-Ann M. Anderson, Steven W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[34] H. Yang and D. F. Wong. Efficient network flow based min-cut balanced partitioning. In *Proc. IEEE Int'l Conf. on Computer Aided Design (ICCAD '94)*, pages 50–55, November 1994.

[35] H. Yang and D. F. Wong. Circuit clustering for delay minimization under area and pin constraints. *IEEE Trans. on Computer-Aided Design*, 16(9):976–986, September 1997.

[36] Peixin Zhong, Margaret Martonosi, Pranav Ashar, and Sharad Malik. Accelerating boolean satisfiability with configurable hardware. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, pages 186–195, Napa Valley, California, April 15–17, 1998.