Kelley, Wachs, et al. Published in Design, Automation & Test in Europe Conference & Exhibition, 2011.

Intermediate Representations for Controllers in Chip Generators

Kyle Kelley, Megan Wachs, Andrew Danowitz, Pete Stevenson, Stephen Richardon, Mark Horowitz
Department of Electrical Engineering
Stanford University
Email: {kkelley,wachs,danowitz,jpeter,steveri,horowitz}@stanford.edu

***Abstract*—Creating parameterized "chip generators" has been proposed as one way to decrease chip NRE costs. While many approaches are available for creating or generating flexible data path elements, the design of flexible controllers is more problematic. The most common approach is to create a microcoded engine as the controller, which offers flexibility through programmable table-based lookup functions. This paper shows that after "programming" the hardware for the desired application, or applications, these flexible controller designs can be easily converted to efficient fixed (or less programmable) solutions using partial evaluation capabilities that are already present in most synthesis tools.**

I. Introduction

Digital design has become an increasingly difficult task. Technology scaling continues to increase the number of transistors per chip, which increases design complexity and verification effort. The non-recurring engineering (NRE) costs associated with creating a modern application-specific integrated circuit (ASIC) are now around $40M[1], dominating the total design cost and severely reducing the economic viability of ASIC solutions for all but the highest volume chips. Moreover, post 90-nm technology scaling has seen the end of traditional voltage scaling, bringing energy efficiency, and not performance, to the forefront of design considerations. Experience demonstrates that specialized designs achieve the best energy efficiency, leading to a challenging impasse: we need to build ASICs but ASICs are too complicated and expensive to build.

Reconfigurable designs are a natural approach to tackling the NRE cost issue, since one design can amortize the high NRE cost over multiple applications. However, the configuration memories and overly generic logic in reconfigurable designs bring substantial area/energy/performance overheads. For example, FPGAs are known to incur about an order of magnitude more area and energy per gate than an ASIC. Even when the configuration is done at a higher level of abstraction, flexible hardware can still cause substantial overheads [2]. Therefore, it is interesting to consider whether we can get the benefits of lower NRE costs, without incurring large energy overheads.

If our goal is to reduce the design NRE costs, runtime configuration is more than we require. Fabrication costs, which include masks and first silicon, are generally less than 10% of total NRE cost [1]. Thus one could get a 10x reduction in NRE costs by minimizing non-fab design and verification costs, such as by creating a flexible design and customizing it pre-silicon. In other words, if we start with a highly flexible design abstraction, and then automate the process of using configuration information to produce an efficient implementation, we could produce chips with 10x lower NRE. This concept has been called a "Chip Generator" [3][4].

Creating a chip generator will almost certainly leverage many different techniques, ranging from using algorithmic hardware constructors, high-level compilers that process system parameters and produce low-level RTL, to writing ultra-generic RTL and relying on synthesis tools to optimize the design. This range of possibilities leads to a question about what kinds of optimizations can be done to RTL before and during synthesis, which might help us better understand the type of RTL that the generator needs to produce to yield competitive results. This paper focuses on a subset of these questions, examining flexible table-based controller structures, and how well synthesis tools can convert these flexible structures to efficient concrete implementations. We focus on table-driven controller structures, such as FSMs and microcode sequencers, since they are the most common method of creating flexible control [5] [6] and are even used to simplify the creation of non-configurable designs.

One technique for converting table-based control into an efficient logic implementation is partial evaluation, an established compiler trick that uses known information about program inputs at compile time to streamline generic code. In the ideal case, the generator's job is straightforward because it only needs to produce the table of bits. If synthesis tools can't handle this type of design, the generator's job becomes more difficult, and it will need to help perform this transformation. The remainder of this paper focuses on experimentally quantifying the ability of current tools and techniques to convert table-based controller information to efficient RTL.

The next section reviews table-based controller design and provides a quick overview of a table-based protocol controller in the Smart Memories chip. We then discuss the optimization techniques that are needed to perform partial evaluation of these table-based structures, and show that modern synthesis tools already possess many of these properties. We also explore current limitations of this flexible design approach and extensions to circumvent them.

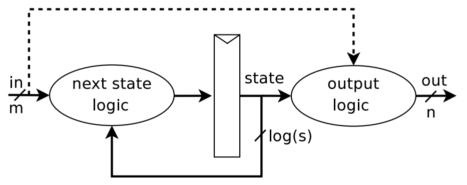

Fig. 1. A generic finite state machine. Output logic may or may not depend on the input according to style. Note required storage element.

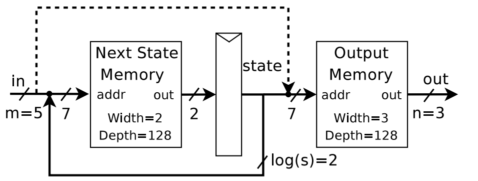

Fig. 2. A 5-input, 4-state, and 3-output FSM implemented with asynchronously readable memories.

II. Reconfigurable Controller Design

We begin by quickly reviewing configurable combinational logic because (as detailed in sections II-A and II-B) it is the fundamental building block of reconfigurable controllers. An arbitrary boolean function can be implemented by storing the function's truth table in a programmable memory, and addressing the memory using the function's inputs. In this setup, an arbitrary function with m inputs and n outputs can be implemented in a memory of width n and depth 2^m. We note that such structures are common and can be found in designs under a variety of different names, such as programmable decoders, ROMs, programmable logic arrays (PLAs), and lookup tables (LUTs) in FPGAs [7].

A. Finite State Machines

Finite state machines (FSMs) are a convenient abstraction that aid in the design of simple controllers. FSMs are sequential control circuits characterized by a finite number of internal states, state transitions, and outputs. They are typically represented as finite state diagrams, which depict the various states and transitions among them. Fig. 1 shows a generic S-state FSM hardware implementation, in which state transitions depend on the current state as well as current inputs, and outputs depend on the current state and (depending on style) inputs.

The ability to design flexible FSMs is particularly relevant for chip generators because FSMs are the brains behind hardware operation, so flexible FSMs enable different operational modes within one larger framework. A reconfigurable FSM can be realized by using programmable tables to implement its combinational logic bubbles (both next-state and output). For example, Fig. 2 shows how a 4-state FSM with 5 inputs and 3 outputs can be implemented with two memory elements: a 2-bit-wide next-state memory with 2+5=7 address bits (128 entries), and a 3-bit wide output memory also with 2+5=7 address bits (128 entries).

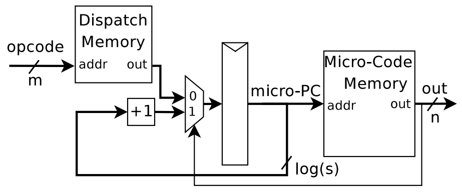

Fig. 3. A generic microcode sequencer.

B. Microcode Sequencers

Microcode sequencers are FSMs whose conceptual operation is described by microprograms instead of finite state diagrams. Microprograms are a series of simple microinstructions: low-level operations that assert particular control signals on a given cycle. We refer to the bit-level representation of microinstructions as microcode. Due to their sequential nature (as well as their resemblance to assembly programming), many designers find microprograms to be more convenient than finite state diagrams for describing controllers, particularly as the design complexity grows. In practice, microcode format varies from being inefficiently encoded but more readable (known as horizontal microcode) or efficiently encoded but difficult to read (vertical). Many microprogramming systems employ horizontal formats to simplify the paths between the controllers and the datapath units [8], using separate subfields to control different units in the design.

Despite their different controller abstractions, the operation of programmable FSMs and programmable microcode sequencers turns out to be similar. Fig. 3 shows the hardware implementation of a typical microcode sequencer, which resembles the FSM implementation in Fig. 2. Note the microcode memory performs similarly to the output logic of FSMs, and the primary difference is the next-state logic. In FSMs, the next-state logic is fully general, allowing direct transition from any state to any other state. In microcode sequencers, on the other hand, the expected transition is a trivial increment to the next sequential microprogram counter. Other state transitions (jumps) are flagged and handled by dedicated dispatch tables, which tend to be small for many practical designs. For these reasons, microcode sequencers are often the more efficient way to implement runtime reconfigurable controllers. For purposes of pre-silicon (design-time) reconfigurability, however, we do not need to make significant distinctions between FSMs and microcode sequencers, because they both share the same underlying table-driven logical descriptions. For these reasons we will use the terms “microcode sequencer” and “table-based controller” synonymously.

C. Motivating Example: Smart Memories Protocol Controller

Smart Memories is a chip multiprocessor with a memory system flexible enough to support traditional shared memory, streaming, and transactional memory programming models on the same hardware substrate[9][10]. The system was designed to be a multiprocessor whose user could program not only the processors, but the memory system as well. To implement this memory system the designers added table-based control

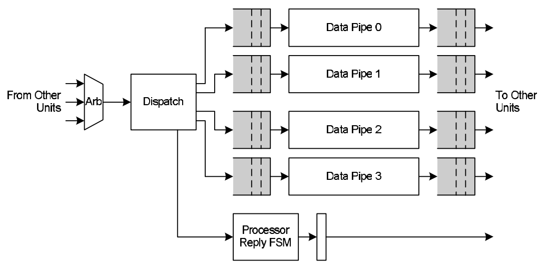

Fig. 4. A block diagram of an internal unit of the PCtrl.

systems to the processor, local memory, and cache/protocol controller (PCtrl). Our discussion focuses on the PCtrl since it is the most complex: shared among four two-processor tiles, it moves data in and out of local memory blocks and implements different memory protocols (such as multiprocessor cache coherence) based on the execution mode. The PCtrl consumes 14% of the chip's area.

Fig. 4 shows internals of one of the functional units of the PCtrl, giving an example of how table-based controllers can be used. When implementing cache protocols, the PCtrl performs transfers between different processors' caches. The precise timing of each transfer depends on user-settable cache line size, as well as the access width to the caches (which can be single or double words). The Dispatch block issues line read and line write commands to four data pipes (leading to local memory in each two-processor tile). These commands, along with appropriate timing, are stored as microcode in a configuration memory inside the Dispatch unit as a table that can be altered to program various cache configurations.

The microcode representation for controllers has a number of documented advantages. It facilitates patches late in the design cycle. Sorin et al. argue that a single table-driven approach can be used in many design phases, including specifying, documenting, and verifying cache coherence protocols[11]. Firoozshahian et al. go a step further and describe how programmable, table-driven controllers can allow a memory controller to support different memory models and protocols within a CMP system[9]. However, these table-driven implementations come with significant area and cycle-time costs from the added memories and address decoding logic. Our desire to leverage many of the advantages of microcode-based controllers, coupled with a desire to achieve implementation efficiency with chip generators, naturally leads to our main question of whether we can produce efficient controller implementations from these microprograms alone, or whether we need to explore other representations. The optimization methodology that will help us achieve our goal is broadly known as partial evaluation, discussed in the next section.

III. Partial Evaluation of Microcode

Partial evaluation, a way to specialize generic programs, has been an effective software technique for years. It uses known information about program inputs at compile-time to reveal new optimizations that were previously unavailable, allowing the compiler to produce better code. This methodology lets programmers write broad general-purpose programs that then compile into specific optimized code instances. The C++ Standard Template Library (STL) is a common software implementation that relies on partial evaluation.

Despite its prevalence in software, partial evaluation (PE) methodologies in hardware design have been primarily limited to data-path optimization in domain-specific frameworks. McKay et. al. apply PE to FPGA synthesis of generic data-path elements for DSP chips [12]. Leonard and Mangione-Smith apply PE to a DES algorithm where the secret key is known and fixed [13]. Mukherjee and Vemuri use PE to optimize DSP data-path elements at the transistor level [14]. This paper extends this strategy to include control-path elements as well as data-path elements. Not only do we want efficient functional (data-path) units, but we want to efficiently control them in different ways, and by doing so we enhance our ability to build useful chip generators.

In general, for partial evaluation of reconfigurable controllers to be effective, we desire the optimized controller to approach the area and timing efficiency of a directly implemented (non-programmable) controller. Our hand-tuned results in section III-C explore this. In our experience, a synthesis compiler needs a few key optimization techniques before it can properly perform partial evaluation of table-based structures. Beyond standard logic reduction methods, these techniques include the ability to identify any known restrictions that might simplify a signal state (thus, a non-optimally encoded signal), propagate these restrictions downstream, and perform typical logic optimizations using this state information. We note that it is not uncommon in large designs to find signals that are not encoded optimally, either intentionally, for instance to reduce the decoding logic need through storing fully decoded fields in horizontal microcode, or unintentionally, such as occurs when reusing generic modules. We will refer to the these optimization properties as *state propagation* and *state folding*.

More formally, an n-bit signal y has $k = 2^n$ possible states in a physical design: $y \in \{0, 1, 2, 3, ..., 2^n - 1\}$. If we know of any restrictions on y, then $k < 2^n$. For example, if we know that y is one-hot encoded, then we know $y \in \{1, 2, 4, 8, ..., 2^{n-1}\}$ and $k = n$. If y is used in a downstream ones-counter circuit, the compiler can evaluate all n values of the circuit and infer that the output is a constant 1, allowing the ones-counter logic to be removed altogether. We note that the most prevalent form of this technique is a familiar subset known as constant propagation and folding, where $k = 1$.

We now turn to the practicality of design by partial evaluation; that is, we explore the efficacy of modern synthesis tools to produce optimized controller implementations from generic microcode specifications. We first compare optimized table-based implementations with fixed non-programmable implementations to confirm expected logic optimizations and the practicality of using microprogram specifications (or, more generally, tables) with chip generators. We then highlight some limitations with this approach that affect both non-optimally encoded wide microinstruction formats and specialized con-

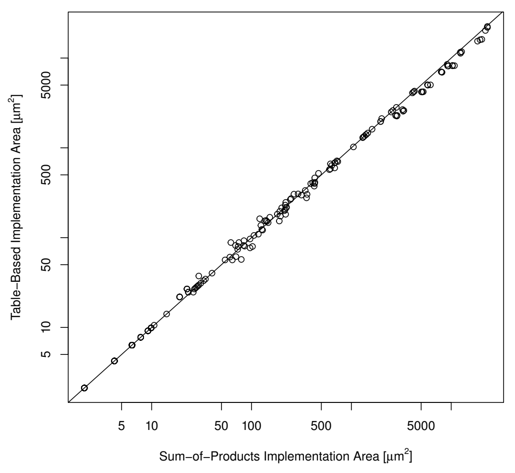

Fig. 5. An area comparison of combinational logic synthesis results for various random designs. Note the equal-area line (intercept 0, slope 1).

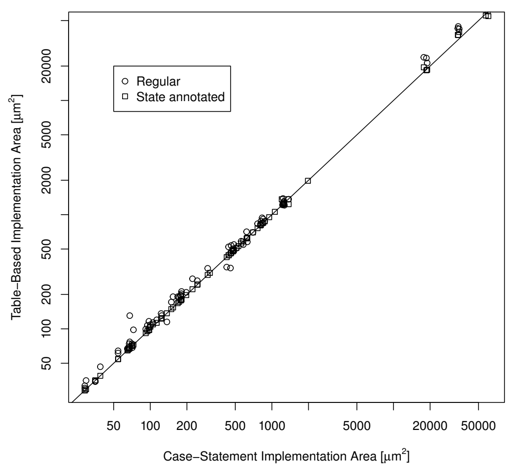

Fig. 6. An area comparison of FSM synthesis results for various random controller designs. Note the equal-area line (intercept 0, slope 1).

trollers with unreachable states. We conclude by evaluating these techniques on the Smart Memories PCtrl.

We chose to use Synopsys Design Compiler D-2010.03 to synthesize our designs as it is an industry standard tool, but we have observed similar results with other tools. The designs were coded in SystemVerilog and the synthesis library was TSMC 90nm.

A. Constant Propagation and Folding

We start with the reconfigurable structures described in Section II and demonstrate how closely they synthesize to their ideal directly-implemented counterparts when relying on simple constant propagation and folding. We wrote reconfigurable versions of each component using SystemVerilog. Python scripts then generated random configuration parameters for these reconfigurable designs, as well as the corresponding direct Verilog implementation for each. We then synthesized these pairs of designs over a sweep of achievable timing targets to generate synthesis results for a wide variety of design sizes and topologies. Note that we only compare designs that synthesized to identical timing targets.

1) Table-Based Combinational Logic: Fig. 5 compares the area synthesis results for many different combinational logic functions (tables of depth $d \in \{2, 8, 16, 32, 64, 256, 1024\}$ and width $w \in \{2, 4, 16, 32, 64\}$). The "direct" implementations were written using sum-of-product assignments for each output bit. In the ideal case all points would lie on the solid line because there would be no difference between the partial evaluation of tables and the direct implementations. However, the discrete nature of the standard cell library coupled with the "bumpy" nature of the tool's optimization surface leads to various local minima, causing the tool to find similar (but not identical) designs when starting from widely different (albeit logically equivalent) RTL descriptions. In fact, we sometimes observe slightly better results for table-based representations, especially for larger functions, suggesting sum-of-product representations are not always ideal for the tool. These observations confirm our expectation that the synthesis tool is effective at partial evaluation of combinational logic tables via constant propagation and folding.

2) Table-Based Controllers: Fig. 6 compares the synthesis results for many different FSMs (inputs $m \in \{2, 8\}$, outputs $n \in \{2, 8, 16\}$, and states $s \in \{2, 3, 8, 16, 17\}$). Note that these results generalize to microcode sequencers as well due to their aforementioned implementation similarities. The direct implementation was written using a series of case statements, the style recommended by the tool vendor for automatic detection and optimization of the FSM states. The flexible implementation used combinational tables as in Section II to describe next-state and output logic. This change in coding style prohibited the synthesis tool from automatically detecting the FSM state encodings, leading to some variance in the synthesized areas as compared to the preferred implementations (especially for $s \in \{3, 17\}$ cases, which aren't efficiently coded in binary). In a second experiment we used the Design-Compiler options *set_fsm_state_vector* and *set_fsm_encoding* to manually annotate the state signal of the controller for the generic designs [15]. The plot demonstrates that providing the tool with this extra information resulted in nearly identical synthesis results between the annotated and direct implementations. It is fairly straightforward to automatically determine these state annotations from the FSM tables (or, equivalently, microcode), and so we do not see this as a real issue for a chip generator. Hence, we can use a flexible table-driven controller style but still achieve the synthesis benefits of a direct implementation.

B. State Propagation and Folding

Although we have demonstrated that we can achieve good implementation efficiencies for isolated controllers, we must

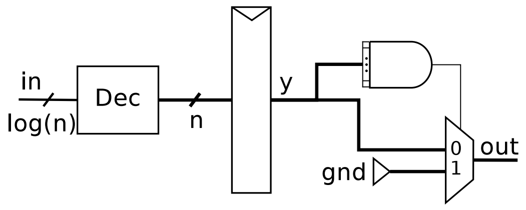

Fig. 7. An example design to investigate state propagation and folding optimizations. Note the mux before the output is unnecessary if the signal y is one-hot encoded.

also consider logic optimizations downstream of the controller outputs when the outputs are not fully encoded (e.g., horizontal microcode). This section explores the optimization of designs with k states, $1 < k < 2^n$, by examining the synthesis results of the small example design in Fig. 7. The one-hot decoder *Dec* allows us to specifically focus on cases where $k = n$, but we expect these results to generalize to other values of *k*. Note that when the signal *y* is one-hot, the mux on the output becomes redundant because the bitwise-AND gate should always evaluate to 0. This is the key optimization that we expect the synthesis compiler to make in this example. Although this is a relatively simple design, its synthesis properties demonstrate a number of interesting features that are consistent with our experiences on more complex designs.

We synthesized this design for a variety of different bus widths $n \in \{2, 4, 8, 16, 32, 64, 128\}$ with easily achievable timing constraints and also varied the flip-flop element to use different reset modes: no reset, synchronous reset, and asynchronous reset. Fig. 8 plots the comparative synthesis results of the generic and direct versions. The purely combinational examples (no flops) always synthesized to the ideal case, suggesting the tool correctly infers state propagation and folding in purely combinational logic. However, in the presence of flops (without retiming), all of the synthesized designs failed to achieve ideal areas. With retiming enabled, optimal designs were achieved in some circumstances but overall the effect was inconsistent. Furthermore, we note the type of flop also inconsistently influenced the outcome.

These observations suggest the synthesis compiler does not perform state propagation over flop boundaries [1]. Note that we already encountered a similar situation with the states of table-driven controllers because the tool is unable to automatically recognize FSM states from tables alone. Using a similar work-around, we manually annotated the states of the signal y after the flop boundary, and plotted these results with filled markers in Fig. 8. It is clear that manual state annotation allows synthesis to perform the necessary optimizations in cases where $n \leq 32$. Although horizontal microcode can be hundreds of bits, its independent subfields that drive different units tend to each be smaller than 32 bits, and so manual annotation of each subfield will be effective. Again, it is straightforward for a generator to produce these annotations if it has the

[1]There are published algorithms that address this [16] but we don't know of any commercial tool that incorporates them yet.

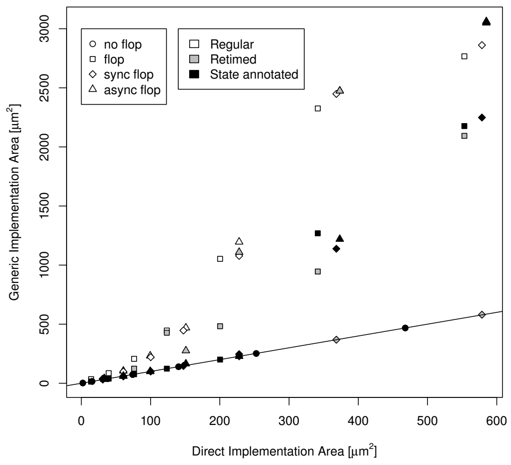

Fig. 8. A comparison of synthesis results for the design shown in Fig. 8. The equality line (intercept 0, slope 1) is shown.

controller microcode, and so we can achieve downstream logic optimizations with the outputs of inefficiently coded controllers.

C. Optimizing Smart Memories PCtrl

We now examine these synthesis techniques on the PCtrl, an example of a realistic table-driven controller design. Storing all the microcode for this controller takes area, as do the associated multiplexers/decoders. To understand this overhead, we compare the original flexible design ("Full") to a partially evaluated design ("Auto") for two different memory configurations: "Cached" and "Uncached". We further compare these with hand-optimized controller instances ("Manual") to understand the optimizations missed by automatic synthesis. Fig. 9 summarizes the area consumption of each design (separated into combinational and sequential logic). All designs were synthesized using TSMC 90nm technology with a 5ns clock.

The automatically optimized (via partial evaluation) controller instances halved the non-combinational area of the full design by removing all configuration memories, and halved the combinational area by simplifying access logic and propagating constants, representing a 7% reduction in overall chip area. Moreover, a similar optimization strategy could be applied to flexible logic elsewhere in the processor and local memories, further increasing the gains.

The manually-tuned versions include optimizations that would occur if the tool properly supported state-propagation across flop boundaries. Primarily, these optimizations involve identifying and removing unnecessary (i.e., unreachable) states for specific memory modes. Since almost all of the controller states are required to support caches, the gains from manual optimization in cached modes were minimal. In contrast, supporting uncached memory requires far fewer control states, leading to an additional 16% in area and power savings in the controller. This additional 16% only represents 1.1% of the

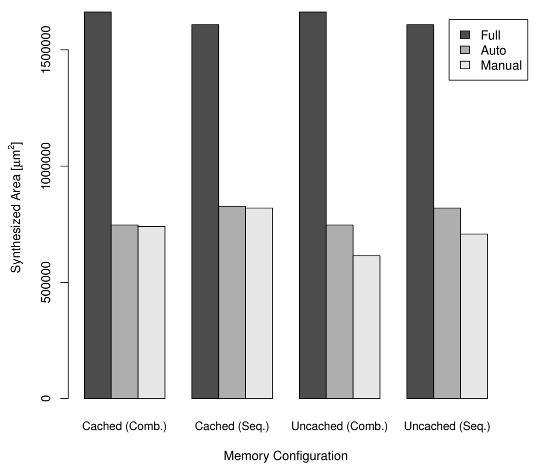

Fig. 9. Combinational and sequential area usage for PCtrl instances.

overall chip area in our example so it is not a particularly significant gain. However, as designers continue pushing the limits of flexible hardware, we expect these state-propagation optimizations to become more important, and so we will eventually require a design methodology that incorporates them.

IV. Conclusion

Chip generators are a promising design approach to mitigate increasing NRE cost issues by amortizing the costs over multiple target applications, but several challenges must be met. This paper addressed the challenge of how to design and model flexible-but-efficient controllers in a generator framework. We showed that it is possible to use extant partial evaluation features in modern synthesis tools to optimize table-based controllers, leading to efficient implementations. This implies that a simplified generator design only needs to produce different microcode for different target designs, allowing design flows to continue using existing microprogramming tools for controller design.

Now that we have demonstrated microcode to be a suitable intermediate representation for generated controllers, an interesting question becomes what the input to the generator should be. For example, it may be possible to build a compiler that uses higher-level specifications to produce microcode for a given controller, so that users of the generator can obtain new design configurations even more quickly.

Lastly, we note that our results provide insight to chip generator architects seeking to compose flexible systems with generic modular RTL (beyond just flexible controller design). The current lack of automatic state propagation across flop boundaries in synthesis implies that ideal logic reductions may not always occur at a system-wide scope. To guarantee better results, modules will have to convey any specialized signal-encoding information to other modules, similar to how we proposed a generator annotate controller states and outputs. This motivates the need for a design scope beyond RTL, whereby designers not only create RTL modules, but also embed this extra knowledge about specific use cases alongside it.

Acknowledgement

The authors would like to thank Amin Firoozshahian and Alex Solomatnikov for their help and support. This work was supported by a Stanford Graduate Fellowship and with the support of the C2S2 Focus Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation subsidiary.

References

[1] M. Horowitz, "Keynote: Why design must change: Rethinking digital design," in *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, p. 267.

[2] A. Firoozshahian, "Smart memories: A reconfigurable memory system architecture," Ph.D. dissertation, Stanford University, 2008.

[3] O. Shacham, O. Azizi, M. Wachs, W.Qadeer, Z. Asgar, K. Kelley, J. Stevenson, A. Solomatnikov, A. Firoozshahian, B. Lee, S. Richardson, and M. Horowitz, "Why design must change: Rethinking digital design," *Micro, IEEE*, vol. PP, no. 99, 2010.

[4] A. Solomatnikov, A. Firoozshahian, W. Qadeer, O. Shacham, K. Kelley, Z. Asgar, M. Wachs, R. Hameed, and M. Horowitz, "Chip multi-processor generator," *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pp. 262–263, June 2007.

[5] D. W. Clark, "Pipelining and performance in the vax 8800 processor," *SIGARCH Comput. Archit. News*, vol. 15, pp. 173–177, October 1987.

[6] S. G. Tucker, "Microprogram control for system/360," *IBM Syst. J.*, vol. 6, pp. 222–241, December 1967.

[7] *Virtex-II Pro and Virtex-II Pro X FPGA User Guide*, Xilinx, November 2007.

[8] D. A. Patterson and J. L. Hennessy, *Computer organization and design (2nd ed.): the hardware/software interface*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.

[9] A. Firoozshahian, A. Solomatnikov, O. Shacham, Z. Asgar, S. Richardson, C. Kozyrakis, and M. Horowitz, "A Memory System Design Framework: Creating Smart Memories," in *ISCA '09: Proc. 36th Annual International Symposium on Computer Architecture*, 2009.

[10] O. Shacham, Z. Asgar, H. Chen, A. Firoozshahian, R. Hameed, C. Kozyrakis, W. Qadeer, S. Richardson, A. Solomatnikov, D. Stark, M. Wachs, and M. Horowitz, "Smart memories polymorphic chip multiprocessor," in *Proceedings of the Design Automation Conference*, 2009.

[11] D. J. Sorin, M. Plakal, A. E. Condon, M. D. Hill, M. M. K. Martin, and D. A. Wood, "Specifying and verifying a broadcast and a multicast snooping cache coherence protocol," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 6, pp. 556–578, 2002.

[12] N. McKay, T. Melham, K. W. Susanto, and S. Singh, "Dynamic specialization of xc6200 fpgas by partial evaluation," in *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1998, p. 308.

[13] J. Leonard and W. H. Mangione-Smith, "A case study of partially evaluated hardware circuits: Key-specific des," in *FPL '97: Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, 1997, pp. 151–160.

[14] M. Mukherjee and R. Vemuri, "A novel synthesis strategy driven by partial evaluation based circuit reduction for application specific dsp circuits," in *ICCD '03: Proceedings of the 21st International Conference on Computer Design*. Washington, DC, USA: IEEE Computer Society, 2003, p. 436.

[15] *Design Compiler Optimization Reference Manual*, Synopsys, March 2010.

[16] M. L. Case, V. N. Kravets, A. Mishchenko, and R. K. Brayton, "Merging nodes under sequential observability," in *Proceedings of the 45th annual Design Automation Conference*, ser. DAC '08, 2008, pp. 540–545.