State Space Reconfigurability: An Implementation Architecture for Self Modifying Finite Automata

Ka-Ming Keung
Electrical and Computer Engineering
Iowa State University
Ames, IA 50011, U.S.A.
xskeung@iastate.edu

Akhilesh Tyagi
Electrical and Computer Engineering
Iowa State University
Ames, IA 50011, U.S.A.
tyagi@iastate.edu

ABSTRACT

Many embedded systems exhibit temporally and behaviorally disjoint behavior slices. When such behaviors are captured by state machines, the current design flow will capture it as a union of all the behavior slices, and map it using traditional state assignment followed by logic synthesis. Such implementations costs are proportional to the union of all the behavior slices (in area, energy and delay). We propose to use self-modifying finite automata (SMFA), that have been studied from complexity-theoretic perspective, for expressing and implementing such adaptive behaviors in embedded systems. Towards this end, we present an implementation architecture for SMFAs. We compare the area, time and energy costs of SMFA implementations with the classical logic space (FSM) implementations for four adaptive behaviors.

Categories and Subject Descriptors: I.6.5 [Modeling methodologies]: Model Development

General Terms: Design, Theory

Keywords: SMFA, FSM, Reconfigurability, Architecture

1. INTRODUCTION

Many embedded systems have components with intrinsic dynamically variable behavior (a planetary rover designed to respond to a variety of terrains and conditions) or are adaptive in order to optimize some aspect of the system performance (many wireless protocols). If these behaviors carry state, the need for a reconfigurable FSM arises. A reconfigurable FSM implementation however should have the following, desirable "continuity" property. The original behavior is specified within a state space, the space of the state transition diagram. The behavior level evolution (reconfigurability) occurs in this state space. For a ϵ change in the state space (addition of a new state), the δ change in the implementation space should also be bounded. The classical implementation space, logic space, does not satisfy this continuity property. We refer to the next state and output combinational logic function architecture as logic space. The design flow maps the state space to logic space through state assignment. This logic space implementation is then minimized through multi-level logic synthesis tools such as SIS [6]. We will denote by $\mathcal{L}(f)$ the logic space transformation (through state assignment and logic synthesis) of a state space behavior f. An incremental reconfiguration of f in state space may add or delete a transition; or add or delete a state. Let us denote such ϵ-neighborhoods of f by $f + \Delta$. The main problem with logic space implementations is that $\mathcal{L}(f + \Delta)$ may not be in any δ-neighborhood of $\mathcal{L}(f)$, a requirement for continuity. In other words, a small, incremental change in state space (a small reconfiguration) may result in a cataclysmic change in logic space implementation (a complete context switch on configuration).

The other extreme is to implement the reconfigurable FSM as a lookup table of all the transitions rules. This table however needs to be associative (content addressable memory, CAM). The cost (delay and energy specifically) of such an implementation is likely to be exorbitantly expensive. The self modifying finite automata (SMFA) offer an in-between solution. They offer the reconfiguration flexibility without abrupt configuration context switches, and yet have implementation costs closer to the logic space FSM implementations rather than state space implementations (as transition rules lookup table). SMFAs [4], [5] were proposed more as a complexity-theoretic abstraction. They do however provide an excellent specification framework for reconfigurable, adaptive finite state machines in embedded environments. In this paper, We propose an implementation architecture for SMFAs. We evaluate the implementation costs of four dynamically variable behaviors with respect to logic space implementations, state space implementations, and SMFA implementations for both ASIC and FPGA platforms.

The rest of this paper is organized as follows. Section 2 discusses related work on reconfigurable state machines. Section 3 presents the definition of SMFA with a few examples. The SMFA implementation architecture is described in Section 4. Section 5 presents all the experimental data on area, time and energy of both ASIC and FPGA implementations. Finally, Section 6 concludes the paper.

CASES'06, October 23–25, 2006, Seoul, Korea

2. BACKGROUND & RELATED WORK

Self-modifying code has been of interest to computer scientists from the beginning of computer programming era. It was especially relevant in the old days of limited resources when self-modifying code allowed a program to run in lim-

ited amount of memory through self-modification. It has also been used as a software obfuscation technique [1]. Over time, though operating systems have come to frown upon self-modifying code and have made it increasingly difficult by making the text memory space non-writeable.

The question of whether there were to be any advantages with reconfigurable automata was a natural extension of self-modifying code. Rubinstein & Shutt [4], [5] were one of the first ones to develop an automata-theoretic model of finite automata that can modify their own structure during computation. They show that the power of first order, one register SMFA is somewhere in between a push-down automata and context-sensitive languages. Neto [3] develops a similar notion for push-down automata (PDA).

There have not been many attempts at implementations of self modifying finite automata. Koster and Teich [7] describe a model for reconfigurable finite state machine. Their methodology is very much grounded in the existing state machines, however. They introduce additional states in the FSM for reconfiguration. There is no discussion of the behaviors that can be captured.

3. SELF MODIFYING FINITE AUTOMATA

3.1 SMFA

Self modifying finite automata (SMFA) [4], [5] was introduced to explore the complexity hierarchy between regular and context-free languages (corresponding to deterministic or nondeterministic finite automata, DFA or NFA, and push-down automata, PDA). To that end, it is an exercise in how to specify minimal modifications from a base DFA or NFA. Recall that a base DFA is more or less what digital design community labels as FSM. A DFA is defined as $M = (\Sigma, Q, S, F, \delta)$ where Σ is the input alphabet, Q is the state set, $S \in Q$ is the start state, $F \subseteq Q$ is the set of final states, and δ is a set of transitions from $(Q \times \Sigma)$ to Q. Note that the notion of output does not exist in a DFA. The acceptance (characteristic function of a set) is the main computation issue which is captured by the set of final states F.

Name Space for Dynamic State Addition and Deletion: An SMFA is allowed to self-modify its state set and transition rules. How are the new states to be created to be specified? This is not an issue of just a new name space. A more relevant aspect is the ability to specify the relationships between these new states themselves. *State name registers* are associated with a DFA towards this end. A single register gives the weakest variation of DFA (least self-modification). This state name register has two functions in its interface: *new* and *old*. $new[r]$ applied to Register r (just $new[0]$ for a single register SMFA, or just *new* in this default case) generates a new name (with some implicit enumeration scheme for the name space) and stores it in that register. The simplest enumeration scheme could be a cyclic counter of some k bits for a k-bit register r. These *state name space registers* are assumed to retain their last name as well which is returned with the control interface $old[r]$. Hence the set of actions that occur on the control command *new* applied to Register r are: $old[r] \leftarrow new[r]$; $new[r] \leftarrow new\_value$.

Self-modifying transition: A self-modifying transition has the form $p \overset{x/a}{\rightarrow} q$ specifying a transition from state p on input x to state q. The a is a modification action. In general, these modification actions can be *add* or *delete* on either a state or a transition. Let us focus on the action of adding a transition. Such a transition would have form $p \overset{x/add\ s \overset{x'}{\rightarrow} t}{\rightarrow} q$. A zero-*th* order transition does not have a modification action. A *first order* transition adds a transition that is zero-*th* order. In general, a transition is nth order if it adds a $(n-1)$st order transition. The added transition is not limited to using states in the fixed state set Q_0. It can refer to newly created states through one of the state name space registers such as $new[0]$ or $new[3]$ or $old[0]$. For instance, the following are valid first-order transition rules: $p \overset{x/add\ s \overset{x'}{\rightarrow} new[0]}{\rightarrow} q$, $p \overset{x/add\ old[0] \overset{x'}{\rightarrow} new[0]}{\rightarrow} q$. When it is clear that the SMFA under discussion has only one state name register, we will often abbreviate $new[0]$ and $old[0]$ into *new* and *old* respectively.

Self Modification Actions: A self modifier in a transition can either *add* a transition, or delete a transition, or delete a state. The delete state action can be specified as *delete* $old[r]$. These state deletion actions are useful from implementation perspective since they reduce the number of bits required for a state name register r.

DEFINITION 1 (**SMFA**). *An SMFA is a quintuple* $M = (\Sigma, Q_0, S, F, \delta_0, r)$ *where* Q_0 *and* δ_0 *specify the base set of states and transitions and* r *is the number of state name registers. The transitions in* δ_0 *are self-modifying kind. An SMFA is said to be* n*th order if there exists an* n*th order transition in* δ_0 *and there is no* $(n+1)$*st or higher order transition in* δ_0*. Note that a zero-th order, zero-register SMFA is a base finite state automaton.*

A first-order, 1-register SMFA is the least variation from the fixed FSM to incorporate self-modification. In this paper, by default, we will refer to a first-order, 1-register SMFA by the term SMFA.

3.2 Four Example Behaviors

We describe the four adaptive behaviors used to evaluate different implementation architectures. The first two are taken from the original paper on SMFAs [5].

The first SMFA to be illustrated formally is for a language known not to be regular (the canonical a^ib^i language used in pumping lemma). The language is $\{a^ib^i | i > 0\}$. The SMFA for this language is shown in Figure 1. The start state is S. The final accepting state is q_f. When the state machine is in S, it transitions on empty string λ to State A. This transition adds a new rule $new \overset{\lambda}{\rightarrow} q_f$. Figure 2 shows the state machine at this stage. Let us trace the evolution of this machine on input *aabb*. The first a creates a link between a newly created state *new* and the *old* state on a b (Figure 3). It is preparing to enforce a matching b for this a at a later stage. When the second a comes along, yet another new state linking into the old *orphaned* piece on b is created (Figure 4). At this point, a new link from q_x into the last *new* state on λ is also added due to λ following aa (adding this rule due to the transition between A and q_x). Now, it is ensured that the only way to get from q_x to q_f is with string bb. Note that this language has an adaptive behavior wherein new states are added in stack order (LIFO).

The second adaptive behavior we describe is for a language that is known not to be context-free (even a push-down automata cannot accept it). The language is $\{ww | w \in$

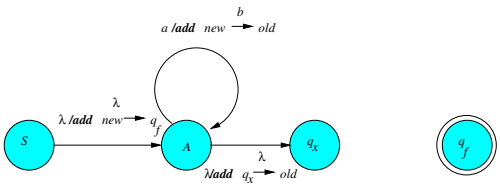

Figure 1: SMFA for Language $\{a^i b^i | i > 0\}$

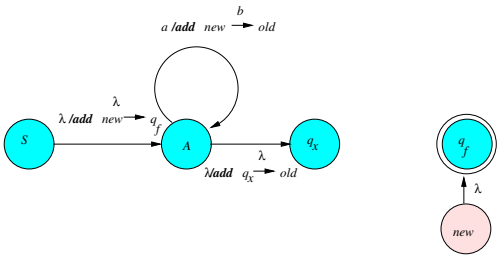

Figure 2: SMFA for Language $\{a^i b^i | i > 0\}$ in State A

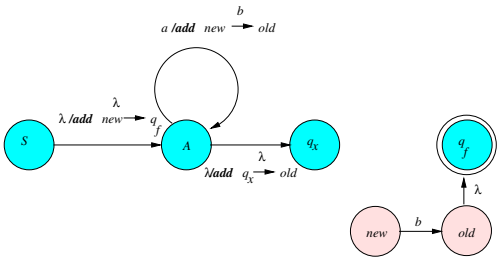

Figure 3: SMFA for Language $\{a^i b^i | i > 0\}$ Right After Reading a

$\{a, b\}^*\}$. The trick is remembering enough information about w so that the second w can be matched against it. Once again, the SMFA for this language is given in Figure 5. The start state is S'. The main activities (of adding on adaptive FSM) occur in State S. We will illustrate this SMFA with the example of input string $abab$ where $w = ab$. Consider this SMFA when input a has been read. First on λ preceding a, the transition from S' to S occurs adding a *new* state with a transition from q_x on λ. The first a on the self-transition within S adds a new transition to a *new* state from the old one on a (from the identical subsequent copy of w). Figure 6 shows the SMFA at this point. The next b (having read ab) also self-transitions within S and adds another new state that requires a b to transition. The λ following ab leads to the transition between S and q_x which also completes the SMFA by adding a transition from *new* (now *old*) to q_f on λ. This final SMFA is shown in Figure 7. Note that the new states are created in FIFO order for this language.

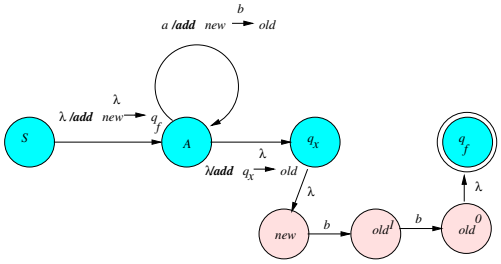

Figure 4: SMFA for Language $\{a^i b^i | i > 0\}$ Right After Reading aa

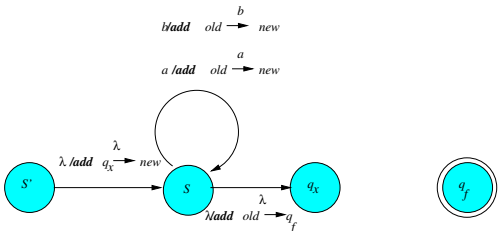

Figure 5: SMFA for Language $\{ww | w \in \{a, b\}^*\}$

The third behavior that we implemented deals with an adaptive network protocol for wireless sensor networks called SPIN [2]. It is actually a family of protocols. We developed the following SPIN energy driven adaptivity protocol.

Consider a sensor that has a choice of two protocols to process its data, $A1$ and $A2$. Each of these protocols/algorithms is implemented along six different energy consumption points. The sensor keeps track of its current availability (in its energy source, a battery). We have quantized the battery energy level into 6 levels $E1$, $E2$, ... $E6$. We will use a simplified version of energy management system with respect to only one activity A associated with a sensor subsystem S_A. In general, though, S_A could be an image processing system for vision. In such a stereo vision system, one action $A1$ could be to construct a stereo image from left and right black and white cameras, and the other action $A2$ could be

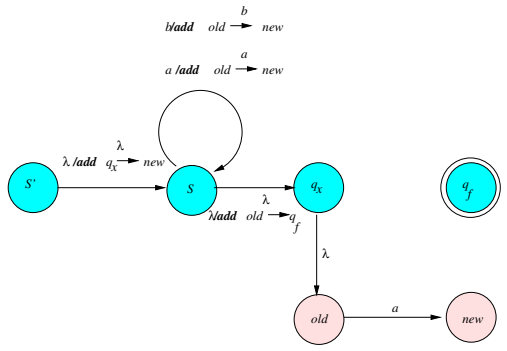

Figure 6: SMFA for ww with $w = ab$ Right after Reading a

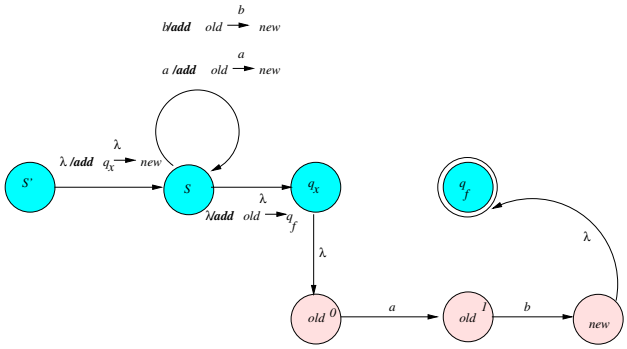

Figure 7: SMFA for ww with $w = ab$ Right after Reading ab

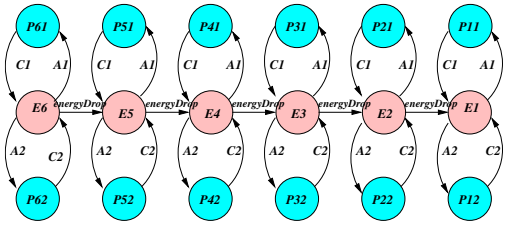

Figure 8: Transition Diagram for Six Energy Level SPIN Protocol

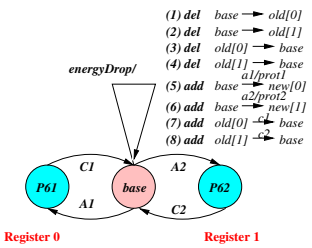

Figure 9: Initial Version of SMFA Implemented SPIN Protocol

to construct the stereo image from left color camera and right black and white camera. A request for one algorithm is denoted by A_1 and for the other algorithm by A_2. All the possible energy management states with respect to system S_A are shown in Figure 8. On energy sensing signal *EnergyDrop*, the energy management system transitions from E_6 full state to E_5, from E_5 to E_4 and so on. When a request for service A_1 comes along, depending on the current energy state E_i, P_{i1} version of the algorithm is instantiated. C_1 and C_2 indicate completion signals for the corresponding algorithms.

Note, however, that the system S_A needs to exhibit behavior corresponding to exactly one of the energy states E_1, E_2, E_3, E_4, E_5, and E_6. The system S_A designed along the lines of Figure 8 incorporates the union of behaviors over all energy states: $P_{11} \cup P_{12} \cup P_{21} \cup P_{22} \cup P_{31} \cup P_{32} \cup P_{41} \cup P_{42} \cup P_{51} \cup P_{52} \cup P_{61} \cup P_{62}$. In reality, though only one of the behaviors $P_{i1} \cup P_{i2}$ is instantiated. A self-modifying version of full behavior from Figure 8 is shown in Figure 9. This is represented as a self modifying finite automata (SMFA). We note here that the state *base* corresponds to any of the E_i states. It self-modifies itself to morph into a different behavior on a transition on *energyDrop*. This transition includes many self-modification steps to create new states and transitions, and to delete some old states and transitions.

In the beginning, the state machine processes data through $P61$ and $P62$. When the energy level drops, the sensor deletes states $P61$ and $P62$ and adds states $P51$ and $P52$ into the state machine. Note from Figure 9, that this behavior needs two state name registers: *Register 0* to keep track of Algorithm 1 state names P_{61}, P_{51}, P_{41}, P_{31}, P_{21}, P_{11} and *Register 1* to keep track of Algorithm 2 state names. After the drop down in energy state to level $E5$, the state machine is transformed in to the one shown in Figure 20. The size of this self-modifying behavior is closer to a single behavior slice instead of the full behavior proportional to the number of energy states.

All of the three behaviors described until now have deterministic adaptation: all the adaptive transitions are known in advance. The fourth behavior is chosen for the *unexpected* nature of its adaptation. It is derived from a paper by Wickramsinghe and Alahakoon [8] on adaptive agent architecture. They propose four types of states (behavior), inherited – which captures the known fixed behavior module, training – these states are designed for the agent to be trained to learn the new behaviors, experience – to capture the trained data, and unexpected – to capture the unexpected part of the behavior. A traffic light example that illustrates these learning concepts from the viewpoint of a pedestrian is also presented. We developed our own example of a traffic light controller that adapts itself to either ambient responses such as weather and/or pedestrian and automobile behavior.

Figure 10 shows such a traffic controller state machine. The machine contains a counter. The controller has four modes of operation, *Sunny Mode*: Red light stays for 5 periods and then Green light stays for 5 periods, *Busy Mode*: Red light stays for 2 periods and then Green light stays for 8 periods, *Rainy Mode*: Red light stays for 5 periods and then Green light stays for 3 periods and then yellow light stays for 2 periods, and *Snow Blinking Mode*: Red light stays for 1 period and then no light for 1 period. The controller adapts itself between these modes.

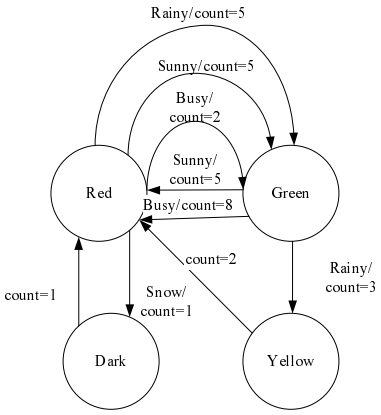

Figure 10: FSM for Adaptive Traffic Controller

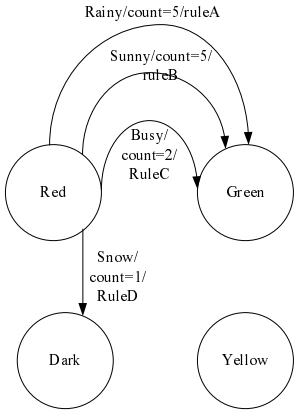

Figure 11: SMFA for Adaptive Traffic Controller

In the fixed FSM in Figure 10, which is the union of all these modes, all transitions are explicitly expressed. In the SMFA version in Figure 11, only four transitions are expressed. When it's sunny day, when the counter counts to five, the machine goes from Red to Green. At the same time, Rule C applies which adds a transition from Green to Red and the condition is 5 periods count. Moreover, a self-delete rule is set for this transition. It means that the transition will be be executed once.

In the rainy day mode when the counter counts to five, the machine goes from Red to Green. At the same time, Rule A applies which adds 1. a transition from Green to Red and the condition is 5 periods count. 2. a transition from Yellow to Red and the condition is 2 periods count. Self-delete rules are set for these transitions. It means that each transition will be be executed once. These rules are as follows: Rule A: Add Green ('(count=3)) Yellow + Rule: Self-delete; Add Yellow ('(count=2)) Red + Rule: Self-delete. Rule B: Add Green ('(count=8)) Red + Rule: Self-delete. Rule C: Add Green ('(count=5)) Red + Rule: Self-delete. Rule D: Add Dark ('(count=1)) Red + Rule: Self-delete.

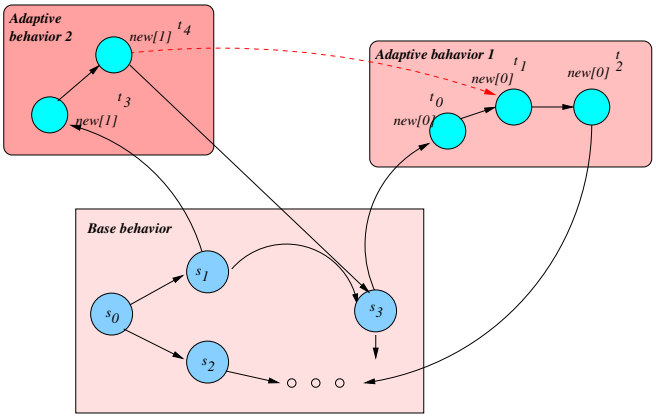

Figure 12: Example SMFA Behavior Specification

3.3 Behavior Specification with SMFA

In this section, we hone our intuition about what types of adaptive behaviors are specifiable with an SMFA. Let us imagine the scenarios and supporting mechanisms leading to reconfiguration. For a given SMFA $M = (\Sigma, Q_0, S, F, \delta_0)$ we refer to the FSM induced by the base state set Q_0 and the transitions in δ_0 with their modification actions removed as the base FA (finite automata), $base(M)$.

The base behavior for M is to stay within $base(M)$. Figure 12 illustrates an example SMFA. The base behavior is captured in states s_0, s_1, s_2, and s_3. When a window for adaptive behavior appears, and note that this window would be with respect to existing transitions entirely within $base(M)$ or Q_0, some new states and transitions can be added. The block labeled "*adaptive behavior 1*" shows one such instantiation. Note that the added transition to serve as the first transition to take M out of the base behavior would have the form $s_i \rightarrow new[l]$ or $s_i \rightarrow old[l]$ where $s_i \in Q_0$. The transition to add such an added transition is its *trigger transition*. The trigger transition will capture system conditions under which reconfiguration needs to start.

These triggers can be either state dependent, or input dependent, or (s_i, x) dependent. A consistent trigger transition for *adaptive behavior 1* could be $s_1 \overset{x_{01}/add\ s_1 \overset{x_{11}}{\rightarrow} new[0]}{\rightarrow} s_2$. If only one state name register is available, the only possible way to stay within an execution path completely contained within *adaptive behavior 1* is to create added transitions that link to each other through common states accessible through $old[0]$ and $new[0]$. The example state transition graph of Figure 12 can be created with the first-order transition $s_1 \overset{x_{02}/add\ old[0] \overset{x_{12}}{\rightarrow} new[0]}{\rightarrow} s_2$. Note that in Figure 12 we have annotated $new[0]^{t_i}$ to show the time at which the value of $new[0]$ was bound to the Register 0. We have assumed that $t_0 < t_1 < t_2$. Then finally the SMFA can exit the adaptive behavior through a transition $s_2 \overset{x_{03}/add\ old[0] \overset{x_{13}}{\rightarrow} s_5}{\rightarrow} s_5$.

Adaptive behavior 2 is triggered similarly. However, if two adaptive behaviors were to interact, the only way a cross-transition between the two of them can be specified (with respect to the states that have not yet been created) is to use at least two registers. That is the role of multiple state name registers: to cross-fertilize adaptive behaviors. Otherwise each adaptive "connected component"(such as *Adaptive behavior 1* in Figure 12) is limited to source its *context* from the base behavior and must exit into the base behavior as well. With two name registers though, we can create a cross-connected-component shown in dashed line in Figure 12. The trigger transition for *adaptive behavior 2* could be $s_0 \overset{x_{20}/add\ s_1 \overset{x_{30}}{\rightarrow} new[1]}{\rightarrow} s_1$. The sustaining transition could be captured as $s_0 \overset{x_{21}/add\ s_1 \overset{x_{30}}{\rightarrow} new[1]}{\rightarrow} s_3$. Finally, the cross-fertilization transition between *adaptive behavior 2* and *adaptive behavior 1* is given by $s_3 \overset{x_{22}/add\ old[1] \overset{x_{31}}{\rightarrow} old[0]}{\rightarrow} s_5$.

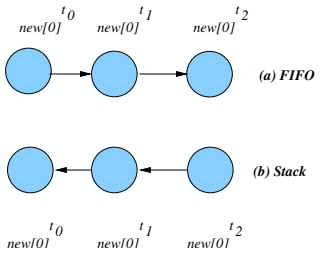

Figure 13: State Creation versus Firing Order

A few other points worth noting here have to do with the relative ordering of transition rule addition and firing (instantiation).

1. For a single state name register SMFA, the adaptive behavior can only grow by creating transitions that point from *new* to *old* or from *old* to *new*. Consider the example of Figure 13 (a). The time order of state creation is left to right $t_0 < t_1 < t_2$. The modification rule here has the form $old[0] \rightarrow new[0]$. State $new[0]^{t_0}$ would have been created with a transition from one of the base states in Q_0. Hence the only entry point for this adaptive behavior is $new[0]^{t_0}$. Once $new[0]^{t_0}$ is entered, the entire behavior $new[0]^{t_0}$, $new[0]^{t_1}$, $new[0]^{t_2}$ has to be traversed (like each instruction in a basic block has to be executed once the control flow drops into it). The order of instantiation is the same as order of generation (creation) or these transitions can be stored in a first-in first-out (FIFO) queue.

2. Transitions in Figure 13 (b) are also added in left to right order as $new[0]^{t_0}$, $new[0]^{t_1}$, $new[0]^{t_2}$. However, they are linked with a modification rule of the form $new[0] \rightarrow old[0]$. Hence the entry point into this behavior is $new[0]^{t_2}$. The states are traversed in a reverse order of their creation. These transitions can be stored in a stack.

3. Cases where there is another entry point into a single adaptive behavior (such as another modification $adds_1 \rightarrow new[0]^{t_1}$. At the time of creation though, $new[0]^{t_1}$ will be referred to by $old[0]$) can also be handled as follows. The base state machine can also generate a value for the FIFO head pointer to account for the multiple entry points. For stack behavior, a stack pointer could be initialized by the trigger transition.

4. Note that these are temporal locality characteristics of the adaptive behavior (stack or FIFO). Branching with multiple registers makes these locality sets more complicated (diverse) necessitating the need for stack and FIFO pointer output.

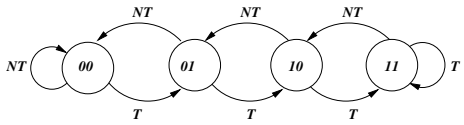

Figure 14: Saturating Counter per Branch in History Table

3.4 Example SMFA

We illustrate an example SMFA with some of the properties from the previous discussion. Consider dynamic branch prediction in computer architecture. A branch instruction's behavior (in terms of *taken* (T) or *not taken* (NT)) is captured in a history table. Each entry of the table is a small saturating counter state machine as shown in Figure 14 which is updated with respect to the input T/NT. The example counter is a 2-bit saturating counter. Sometimes, it may be desirable to maintain only 1-bit counter (when it is a forward going branch not inside a loop), and sometimes even a 3-bit counter might be needed when the branch has significant hysteresis. This choice of the counter size could be one of the adaptive behaviors. The other key parameter in dynamic branch prediction scheme is the set of bits used to hash into the branch prediction/history table. Invariably, the history table size is significantly smaller (say 1024 entries) than the address space (say 32-bits). An easy hash function is to take least significant 10 bits (for 1024 entry table) from the program counter (PC). However, there are collisions which can pollute the prediction table and reduce its accuracy. Such pollutions on context switch are more difficult to handle. A simple adaptive strategy may be to concatenate some (k) of the most significant bits with $10-k$ least significant bits. A simplistic adaptive strategy may be

to associate the number of most significant bits that are concatenated with the nesting level of a branch. The intuition behind such a strategy is that a deeply nested branch would have been instantiated many times, and hence aliasing would be minimal. Let a branch at top level have nesting level 0, a branch inside one nested loop level 1, and so on. One adaptive function may be to concatenate $3 - l$ most significant bits of PC for a branch at nesting level l. This is the second adaptive behavior. The trigger for the second behavior is branch nesting level (which we assume can be estimated within the microarchitecture dynamically). The trigger for the first behavior is the branch offset. If it is negative signifying a loop, the counter is 2 or 3 bit counter based on nesting level. A nesting level higher than 3 could lead to 2-bit counter, else a 3-bit counter is used. Note that branch prediction table is looked up in the first pipeline stage along with the instruction fetch. We do know the nesting level without looking at the instruction and hence the second behavior hash function can be adapted appropriately. For the first behavior, the adaptation (the counter size) only needs to occur after looking up the current prediction (state of he counter). By that time, the instruction would have been fetched and the offset sign can be determined. The two adaptive behaviors also interact, since the primary trigger for the first behavior is offset sign, but a secondary trigger is the trigger for the second behavior.

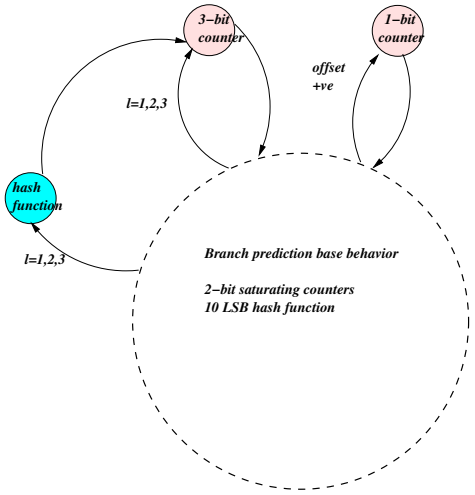

Figure 15: Adaptive Behaviors in Branch Prediction

Figure 15 demonstrates these adaptive behaviors for dynamic branch prediction informally. The space of adaptation is the number of counter bits and PC derived hash function. The base behavior with respect to these parameters is 2-bit saturating counters and a hash function extracting 10 least significant bits of PC. The second adaptive behavior is the left-most transition going out of base behavior envelope. The trigger event is the value of nesting level ($l = 1, 2, 3$). In the state *hash function*, the $(3 - l + 1)$ most significant bits of PC are concatenated with 7 least significant bits to give the 10-bit hashed value. The second trigger transition going out of base behavior is also based on the event set $l = 1, 2, 3$. This leads to the adaptive state *3-bit counter* which modifies the saturating counter to count with 3-bits, and also outputs the T/NT token by interpreting the 3-bits suitably. Note

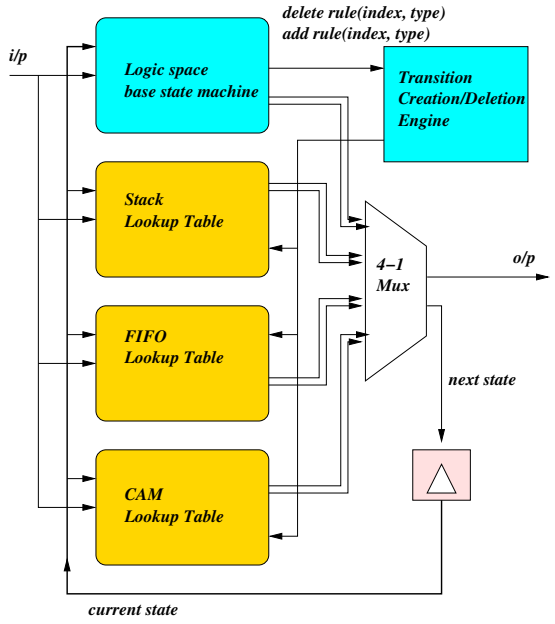

Figure 16: SMFA implementation Architecture

that *adaptive behavior 2* could also lead to *adaptive behavior 1* as in the transition from *hash function* to *3-bit counter*. Another component of *adaptive behavior 1* is triggered by a positive branch offset leading to the adaptive state *1-bit counter*.

4. SMFA IMPLEMENTATION ARCHITECTURE

4.1 Implementation Architecture

Recall that logic space implementations are fast and energy efficient for a given behavior. The lookup table state space implementations, on the other hand, are easily and continuously reconfigurable. An ideal implementation architecture for SMFA should retain the best of both worlds. The base state machine may be implemented as a logic space state machine. The adaptive behavior could be mapped to a lookup table of transitions. Furthermore, the access time and energy for the lookup tables could be reduced significantly if a CAM (content addressable) lookup can be avoided. As we discussed in Section 3, many applications create and/delete state sets in a regular pattern based on the locality characteristics of the SMFA. The two most commonly occurring locality sets in a single-register SMFA (one under consideration) are stack and FIFO. This argues that state lookup tables be supported for all three disciplines: CAM, stack, and FIFO.

Figure 16 shows an implementation architecture for SMFA driven by these considerations. The base state machine is implemented in logic space. The adaptive behaviors are classified into either stack, FIFO, or general CAM locality sets, and are implemented in a lookup table with stack, FIFO, or general CAM organization respectively. The *Transition Creation/Deletion Engine* (TCDE) is the block responsible for adding and deleting a transition to the appropriate lookup table (stack, FIFO or CAM). Recall that since we have as-

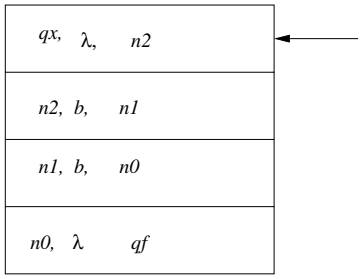

Figure 26: Stack Lookup Table for Language $\{a^i b^i | i > 0\}$

sumed a first-order SMFA, the added rules themselves cannot add any new transition rules. Hence all the additions will be activated from the base transitions in δ_0 which are implemented in the logic space base FSM. This base FSM is enhanced to generate a control signal to the TCDE *add rule*. This tells TCDE to copy a rule from its own memory into the corresponding lookup table. The *type* parameter of *add rule* indicates the specific lookup table (stack, FIFO or CAM) to which a new rule needs to be added. The rule itself is specified as an *index* (a parameter of *add rule*) into a predetermined ordering of all the new rules. Note that TCDE writes the corresponding rule into a stack at the stack pointer location (head). Similarly, a FIFO advances its tail pointer to reflect the new rule addition.

It is worth noting that both the rule and state *deletion* capabilities are crucial for an efficient SMFA implementation. If the behavior schedule always works with some k states in its working set, whereas it does need $n > k$ states for the behavior specification, by deleting the states not in the working set on the basis of some trigger event, we can keep the CAM size of rename space size proportional to k or $\log k$.

TCDE: The transition addition and deletion engine (TCDE) is the heart such an implementation. We describe TCDE along with the stack lookup table blocks within the context of the language $\{a^i b^i | i > 0\}$ described in Section 3. The user specifies the base state machine, and a template for the new rules to be added in a format. The following BLIF like format description will be adequate for $\{a^i b^i | i > 0\}$.

```
lambda s  A  add, type 0
a      A  A  add, type 1
lambda A  qx add, type 2

.type 0: new lambda qf
.type 1: new b      old
.type 2: qx  lambda old
```

The top three lines describe the base automata of Figure 1 which will be implemented in the "Logic space base state machine" block of Figure 16. The last two columns in these lines indicate whether an add or delete self-modification action is associated with this transition. If the last two columns are empty, then the transition is taken to be a classical transition. The last column specifies a template for the modification in terms of enumerated types. These templates are expanded in a later type section. For instance, the line with `.type 0` specifies to TCDE what kind of rule needs to be added. The keyword `new` indicates that TCDE should use its internal name space counter to generate a new state name. The fact that all instances of `new` occur in the first column of the type template, and all instances of `old` occur in the last column of this template, suffices to conclude that this SMFA has stack locality. TCDE adds these rules to the stack lookup table as shown in Figure 26. When the SMFA is in state shown in Figure 2, only the first rule at the bottom would have been added. All the four rules shown would have been added after reading aa as in Figure 4. At this point, the base automata is in State q_x. On input λ and current state q_x, the base automata does not fire. The stack lookup table however has a match. Note that the stack lookup table only matches against the rule at the top pointed to by the stack pointer. After this match, this rule is popped, and stack pointer points to the rule ($n2$, b, $n1$). The current state now is $n2$, and on input b, once again the stack lookup table fires and pops this rule, and so on until the rule at the bottom of the stack forces the SMFA into the state q_f. Note that the state name registers/counters are embedded inside the TCDE as well.

All the four blocks (logic space base FSM, stack, FIFO and CAM lookup tables) interpret the current state cs along with the input x to see if one of the rules owned by them fires. If it does, they drive the next state ns and the output y. A multiplexer then selects the output and next state of the block with a firing rule. Note that the time complexity of lookup in the stack and FIFO lookup tables is constant with respect to the number of entries in these tables. This is not the case for the CAM lookup table. The time required to search n entries to determine a firing rule will be a function of n. It will be proportional to n for a one-comparator implementation.

5. EXPERIMENTAL RESULTS

In this section, we present the experimental setup and results. The four adaptive behaviors described in the previous section, $\{a^i b^i | i > 0\}$, $\{ww | w \in \{a, b\}^*\}$, SPIN energy driven FSM, and adaptive traffic controller were implemented in fixed FSM, SMFA, and lookup table styles both as an ASIC and FPGA based system. These implementation styles are compared with respect to area, time and energy/power. Due to the coding complexity, fixed FSM of $ww5$ is not available.

The designs were written in Verilog Code. They were simulated by ModelSim *6.0d*. After the simulation, the designs were imported into Cadence BuildGates Physically Knowledgeable Synthesis(PKS) to generate the schematic and draft layout using Cadence GSCLib Library. This is a 0.18μ technology. The frequency, energy and area data are reported from PKS. For the FPGA implementation data, these designs were imported into Xilinx *ISE 6.1i* environment. The frequency and LUT data are based on Virtex II Pro device xc2vp2 with speed grade -7.

Note that $\{a^i b^i | i > 0\}$ and $\{ww | w \in \{a, b\}^*\}$ are not regular languages. Hence, a comparable FSM does not exist. However, if they were to be accepted by an FSM, that fixed FSM would have to be a union of all possible dynamic (adaptive) behaviors. For instance, for $\{a^i b^i | i > 0\}$ such a fixed FSM, if restricted to strings of length 4 would form the regular expression $\{ab, aabb\}$ which can be accepted by an FSM. This has been our strategy. Specifically, we have implemented input length limited versions of these languages as follows. The language $\{a^i b^i | i > 0\}$ was implemented for

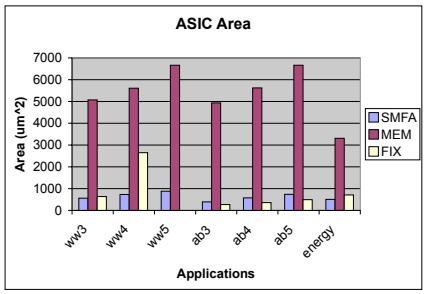

Figure 17: Area Comparison of ASIC Implementations of Three Behaviors

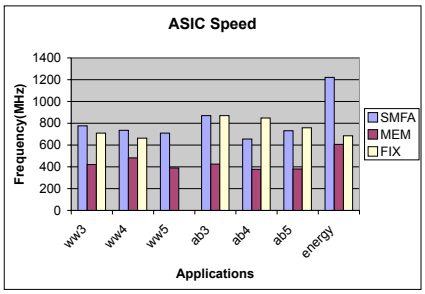

Figure 18: Delay Comparison of ASIC Implementations of Three Behaviors

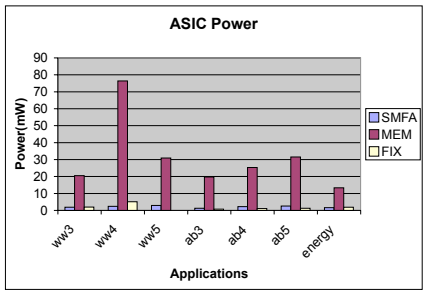

Figure 19: Power Comparison of ASIC Implementations of Three Behaviors

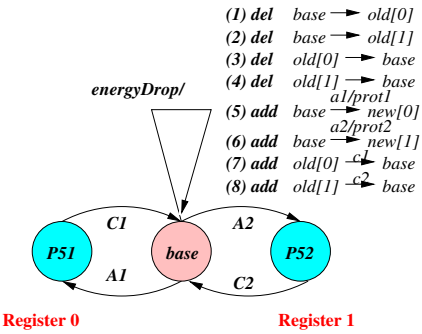

Figure 20: SMFA for SPIN Protocol with Energy Level 5

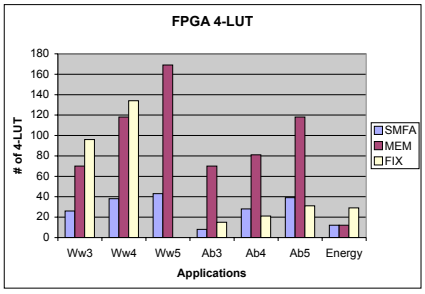

Figure 21: 4-LUT Count Comparison of FPGA Implementations of Three Behaviors

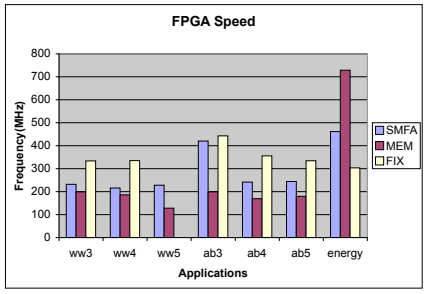

Figure 22: Delay Comparison of FPGA Implementations of Three Behaviors

Figure 23: Area Comparison of SMFA, Memory and Fixed Implementations for Adaptive Traffic Controller

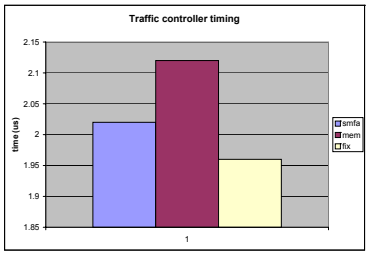

Figure 24: Time Comparison of SMFA, Memory and Fixed Implementations for Adaptive Traffic Controller

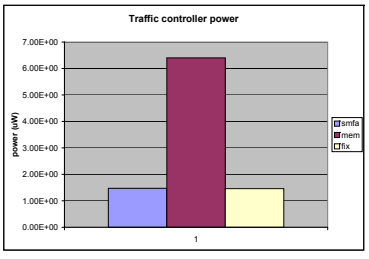

Figure 25: Power Comparison of SMFA, Memory and Fixed Implementations for Adaptive Traffic Controller

three different upper bounds on the string lengths: *ab3* implies strings of length upto 6, *ab4* implies strings of length upto 8, *ab5* implies strings of length upto 10. Similarly, $ww3$ means that $|w| \leq 3$, $ww4$ means that $|w| \leq 4$, and $ww5$ means that $|w| \leq 5$. The implementation architecture follows the schema presented in Figure 16. However, some language specific optimizations might have also been applied. *Energy* in these results stands for the energy-adaptive SPIN protocol implementation.

Figure 17 presents the data on the area of ASIC implementations of these behaviors. ASIC SMFA has 30% lower area than the fixed FSM on average, and 728% lower area than the lookup table implementation on average (over all the behaviors). Figure 18 reports the data on time/speed of ASIC implementations. The SMFA can be clocked at 16.47% higher frequency than a fixed FSM implementation, and 43% higher clock frequency than a lookup table implementation on average. Finally, the energy/power consumption of these behaviors is compiled in Figure 19. The power advantage of SMFA over lookup table implementation is 1313% on average! It however has 16% higher power than a fixed FSM implementation.

Figure 21 presents the data on the 4-LUT counts of FPGA implementations of these behaviors. The 4-LUT count for SMFA is 97% lower than that of a fixed FSM implementation and 230% lower than a lookup table implementation on average. Figure 22 reports the data on delay of these FPGA implementations. The clock frequency of SMFA is 4.6% higher than a fixed FSM implementation, and 22.3% higher than a lookup table implementation. Also note here that some frequencies exceed 500MHz in these graphs, which is the minimum frequency obtained during synthesis. If it is higher than the board speed, the design can only run at board speed.

The adaptive traffic controller was only implemented in ASIC. Figures 23, 24, and 25 show the area, delay, and power comparison of SMFA, memory and fixed FSM implementations respectively. The memory version is slow, power consuming and occupies more area. Fixed FSM and SMFA have similar speed and power consumption. SMFA's area is smaller than the fixed FSM area.

Note that as the sizes of these behaviors grows (such as $|w|$ in ww), the fixed FSM size grows non-linearly (super-linearly). Hence, for larger instances the area, speed, and power advantage of SMFA with respect to fixed FSM is likely to be even higher. The lookup table implementation is clearly inferior to an SMFA implementation with respect to all the attributes.

6. CONCLUSIONS

We took a theoretical automata, SMFA, proposed elsewhere, and developed an implementation architecture for it. The main goal behind adopting SMFA is its ability to specify adaptive finite state behaviors very nicely. With reconfigurable state machine behavior, a concise yet useful specification language is one of the biggest issues. Such an automata implementation is very useful in the design of reconfigurable embedded systems. We implemented three adaptive behaviors in SMFA, fixed FSM, and lookup table (memory) schemas both in ASIC and on an FPGA. We observe an area advantage for SMFA over fixed FSM of over 97% in an FPGA and about 30% in an ASIC. The speed advantage of SMFA over a fixed FSM is about 5% in FPGA and 16% in ASIC. SMFA implementations dominate the lookup table implementations by significantly larger margins. The SMFA advantage over fixed FSM implementations is likely to be even larger for larger instances of adaptive behaviors.

7. REFERENCES

[1] David Aucsmith. Tamper resistant software: An implementation. In Ross J. Anderson, editor, *Information Hiding 1996*, number LNCS:1174, pages 317–333. Springer-Verlag, Berlin Germany, 1996.

[2] W. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proc. MOBICOM, 1999, Seattle*, pages 174–185., 1999.

[3] João José Neto. Solving complex problems efficiently with adaptive automata. *Lecture Notes in Computer Science*, 2088:340–349, 2001.

[4] Roy S. Rubinstein and John N. Shutt. Self-modifying finite automata. In *IFIP Congress, Vol. 1*, pages 493–498, 1994.

[5] Roy S. Rubinstein and John N. Shutt. Self-modifying finite automata: An introduction. *Information Processing Letters*, 56(4):185–190, 1995.

[6] E. M. Sentovich, K. J. Singh, L. Lavango, C. Moon, R. Muragi, A. Saldhana, H. Savoj, P. Stephen, R. Brayton, and A. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report Memorandum Number UCB/ERL M92/41, Electronics Research Laboratory, Dept. of EECS, University of California, Berkeley, 1992.

[7] J. Teich and M. Koster. (self-)reconfigurable finite state machines: Theory and implementation. In *Proceedings of 2002 Design, Automation and Test in Europe Conference and Exhibition (DATE'02)*, pages 559–566, 2002.

[8] L. K. Wickramasinghe and L. D. Alahakoon. Adaptive Agent Architecture Inspired by Human Behavior. In *Proceedings of IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'04)*, pages 450-453, IEEE Computer Society Press, 2004.