In Proceedings of the 28th International Symposium on Computer Architecture (ISCA), June 2001.

Automated Design of Finite State Machine Predictors for Customized Processors

Timothy Sherwood Brad Calder

Department of Computer Science and Engineering
University of California, San Diego
{sherwood,calder}@cs.ucsd.edu

Abstract

Customized processors use compiler analysis and design automation techniques to take a generalized architectural model and create a specific instance of it which is optimized to a given application or set of applications. These processors offer the promise of satisfying the high performance needs of the embedded community while simultaneously shrinking design times.

Finite State Machines (FSM) are a fundamental building block in computer architecture, and are used to control and optimize all types of prediction and speculation, now even in the embedded space. They are used for branch prediction, cache replacement policies, and confidence estimation and accuracy counters for a variety of optimizations.

In this paper, we present a framework for automated design of small FSM predictors for customized processors. Our approach can be used to automatically generate small FSM predictors to perform well over a suite of applications, tailored to a specific application, or even a specific instruction. We evaluate the use of these customized FSM predictors for branch prediction over a set of benchmarks.

1 Introduction

Customized Processors use compiler analysis and design automation techniques to take a generalized architectural model and create a specific instance of it which is optimized to a given application or set of applications. This process can range from whole processor synthesis to having a general processor core where a few specific architecture components are customized. Typically the design relies upon optimizing and combining well understood lower level primitives.

One such primitive is the finite state machine predictor, which is most commonly used as a branch prediction counter or as form of confidence estimation. The more established uses of these finite state machine predictors (e.g., branch prediction) are now finding their way into embedded processors and DSPs [14].

A *Finite State Machine* (FSM) consists of a set of states, a start state, an input alphabet, and a transition function that maps an input symbol and current state to next state. A Moore machine extends this with an output on each state. If one considers the case where the alphabet and output symbols are constrained to be only $\{0, 1\}$, a finite state machine can be used to generate yes/no predictions. The output at a given state is its prediction of the next input. Either intentionally or accidentally, many FSMs in computer architecture are of this form, the most famous of these probably being the two-bit saturating counter. In a branch predictor, these simple state machines generate predictions of taken after a series of takens, or not-taken after seeing a series of not-takens.

In this paper we present an automated approach for creating finite state machines for prediction. We use profiles to generate a language of predictions of a certain type, which is expressed as a regular expression. The regular expression is then converted into a FSM. The VHDL for synthesis is then generated from the FSM. Using this approach we can automatically generate FSM predictors that perform well over a suite of applications, tailored to a specific application, and even a specific instruction. We demonstrate the ability to generate accurate custom FSM predictors by automatically designing customized branch predictors. We examine customization of state machine predictors for individual branches.

In Section 2, we describe the area of custom processor design. Section 3 describes current FSM predictors used in computer architecture, and prior work into automatically generating predictors. Section 4 details our approach to automatically generate FSM predictors. Section 5 uses our automated approach to generate a custom branch prediction architecture, presents miss rates for our custom architecture, and examines in detail some of the FSM predictors generated by our approach. Section 6 describes other potential uses for FSM predictor customization and initial results for using our approach for automatic generation of confidence estimators for value prediction. Finally, section 7 provides a summary of our research.

2 Customized Processor Design

The rapidly growing embedded electronics industry demands high performance, low cost systems with increased pressure on design time. The gap between the two current methods for dealing with such systems, ASICs and off-the-shelf processors, leave many unsatisfied with the trade offs between long design cycles and lower performance.

The designer could choose to design a custom ASIC, which has the advantages of high performance at the cost of long design and verification times. An alternative choice is an off the shelf embedded processor. Embedded processors allow rapid development times and low development cost in terms of both time and money, but with performance typically lagging behind ASICs.

To address this problem there is an emerging technique which will add a new and important point in the spectrum of solutions. Automatically generated customized processors have the promise of delivering the needed performance with only slightly inflated design time.

A customized processor is a processor tailored to an individual application or a set of applications. The idea became a common research subject in the late eighties and early nineties with projects such as SCARCE [18] and The Architect's Workbench (AWB) [9]. SCARCE was a flexible VLSI core for building application specific processors. AWB was a system built to help the designers evaluate design tradeoffs for building embedded processors.

Currently there is a resurgence of interest customized processors with many research projects and companies working in this domain. A great deal of the interest comes from increased system level integration, where it is increasingly common to place different parts of a system all onto one chip. To support this it is now common for vendors to sell descriptions of processor cores rather than actual silicon itself. These processor core descriptions can then be customized for a given application.

There are two major approaches to customized processors, one approach working towards pre-silicon customization, and the other approach pressing for post-silicon reconfiguration or adaptability. The first approach, which is being adopted by such systems as Hewlett Packard's PICO project [20, 1, 23] and Tensilica's Xtensa [10, 7], is intended to produce low-cost high-speed fixed hardware for embedded systems. The other approach attempts to take advantage of post-silicon customization through reconfigurability. Examples of systems that support reconfigurability are Altera's NIOS [27] processor core and the CHIMEARA chip [32].

There is also another range of freedom for these chips, which is the design level. Some systems, such as CHIMEARA [32], and PRISC [21], concentrate their application tailoring at the level internal to a functional unit. These systems work by tailoring the processor's functional units to the application running on it. For example, they might merge commonly executed expressions into a single instruction. Other options are to perform the customization at the architectural level of registers and number of functional units such as Xtensa [10], SCARCE [18] and Lx [8]. This allows the system designer to make high level architectural tradeoffs for the application such as relieving register pressure or removing unused functional units. Still other designs have co-processors for a given application or type of application. Xtensa [7] supports the tight integration of co-processors into an architecture, while PICO [20, 1, 23] automatically generates co-processors in the form of a custom designed systolic array.

Even though there is a long history and many different projects, all of the projects have a similar high level overview. Begin with a configurable or parameterizable architectural template and customize it to fit the application at hand. Because all of these systems target a very specific application or suite of applications, and these applications are under the designers control, any one of these approaches could benefit from the techniques we present.

3 Prior Work

In this section we summarize current FSM predictors, and prior work into automatically finding or generating branch predictors.

3.1 FSM Predictor Implementations

The majority of FSM predictors used in prior research are *Saturating Up and Down* (SUD) counters. Four values define a SUD counter – (saturation threshold, correct increment, wrong decrement, and a prediction threshold). A SUD counter can have a value between 0 and the saturation threshold. If the prediction is correct, the counter is incremented by the correct increment value. If it is incorrect, the counter is decreased by the wrong decrement value.

The majority of current processors use a branch predictor that indexes into a table of 2-bit SUD counters [26, 31] The counter is incremented when the branch is taken, and decremented with not-taken, with a saturating threshold of 3. When the counter has a value less than or equal to 1, the branch is predicted as not-taken. If the counter has a value greater than 1 then it is predicted as taken. Some of these branch prediction architectures have several prediction tables, and a *Meta* table of SUD counters are used to pick which predictor to use.

Jacobsen et. al. [13] proposed the idea of confidence estimation and examined its relationship to branch prediction. In their study, they used saturating up and down (SUD) counters, and *Resetting Counters* to provide the confidence estimation. A resetting counter resets the counter back to 0 when there is a misprediction. Lick et. al. [30] searched using profiling for branch history patterns that provided cor-

rect predictions and that also had a high degree of confidence. They then examined a confidence estimator that predicted high confidence when these patterns were seen, and low confidence when the other patterns occurred. Grunwald et. al. [11] presented several new metrics for evaluating confidence estimators and provided a detail comparison of using SUD counters, resetting counters, and static confidence estimation.

Our focus in this research is an automated approach for generating small FSMs. The outcome of each state of our FSM counter is a binary decision of yes or no. This captures most of the implementations of FSM, but not all. Confidence estimators can return a number representing a probability instead of a binary decision, and several different actions could be performed based upon the degree of confidence returned. Even so, most confidence estimation implementations have only one prediction threshold for the SUD counter used, and therefore fit into the space of FSM counters our approach addresses.

3.2 Automatically Generating Predictors

Burtscher and Zorn [2] examined using profiles to find N-bit value prediction histories that were highly confident. For each possible history, their profile gave what the prediction probability would be if values were predicted using that history. They then used this profile, along with a desired accuracy they wanted to achieve, to select which histories would be used to generate value predictions, and which histories would predict low confidence. This was then used to guide confidence estimation for their value prediction architecture.

Chen et. al. [5] examined using techniques from data compression to improve the performance of branch prediction. They looked at using *Prediction by Partial Matching* (PPM), where there are M tables from size 2 to 2^M. Each PPM entry contains a frequency for the number of times the next bit was 0 (not-taken) and the number of times it was (1) taken. All of the PPM tables are then searched in parallel for each history length. The PPM table entry that had the highest probability was then used for the prediction.

Emer and Gloy [6] have the closest prior work to ours where they examine using genetic programming with feedback to search the predictor design space. They developed a language to describe valid branch prediction architectures, which consists of a variety of predictor primitives (e.g., counters, tables, values), their sizes, and functions to combine these primitives. Using genetic programming techniques, they search for new predictors by performing crossovers and mutating recent candidates, and they evaluate each predictors potential by examining how well it performs for a given set of benchmarks. In contrast, our approach automatically builds FSM predictors from behavioral traces, without searching. Our approach can generate FSM predictors that cannot be represented easily or naturally by the branch prediction language defined by Emer and Gloy. Conversely, our approach does not generate or examine the design space of table based prediction architectures. Therefore, our approach is better for finding efficient predictors for small design areas, whereas their approach can potentially find better solutions for designs with larger areas, and it may be beneficial to combine the two approaches.

4 Automated Design of a FSM Predictor

To design a FSM predictor we go through a fairly complex design flow starting with profile information and finishing with synthesizable VHDL code. While the techniques described here can be used for any sort of predictor, we describe building FSM predictors for branch prediction throughout this section to help explain our approach. For clarity use a more general notation of 1 and 0 instead of taken and not taken respectively. We start off with a high level overview of the design chain we have developed and then discuss each step in detail with an example.

4.1 Overview

Regular expressions provide us with a way to specify patterns and then have them converted into finite state machines because they are both related by formal language. A formal language is a set of strings, either finite or infinite, made up of a sequence of elements from an alphabet, in our case the set of zeros and ones.

A regular expression provides a recursive way of testing an input string to see if it belongs to a given language. The alternative is to define an iterative way of testing a string, which is what a finite state machine does. The two are related because they both perform the same function, recognizing strings, and a mapping from one to the other can always be found.

We now show a way of exploiting this relationship for the purpose of creating a predictor. Let us say that $s_i = \{b_1, b_2, ..., b_i\}$, which means s_i is the string formed by the concatenation of all the input sequences from the start until i. Now let $s = \{s_1, s_2, ..., s_i\}$. This is the set of all possible inputs the predictor could see over time. If we pick a subset L, of s, where L is the set of input strings that satisfy some metric we have chosen, we can say that L is the language of predict 1.

In this way, the problem of creating a FSM predictor reduces to that of finding a regular expression that describes L in a compact manner. Once we have the regular expression that recognizes L we can use established methods to convert it into a finite state machine that recognizes L. If we do this translation correctly the finite state machine will indicate that it recognizes a member from L when it sees an input string in L. Because we picked L to be only the subset of s that we

want to predict 1 on, we know that when we recognize this input string sequence that we should predict 1.

To find the language L we use profile information from the application. From the profile a model of the data is built, and this model can then be analyzed to find histories that are biased toward predicting 1. This set of histories is then compressed to a usable form and converted into a regular expression.

4.2 Modeling

To design a FSM predictor we start by tracing the target application suite, creating a representative sequence of predictions for the applications. Since the intended use of our approach is for customized processor design in the embedded space, we believe that collecting accurate traces is possible.

Another issue is determining what to trace, which depends on the intended use of the predictor. For branch prediction, we have two states 0 and 1, and the trace consists of a series of branch outcomes. We will use the following trace for the rest of the examples in this section:

$$t = 0000\ 1000\ 1011\ 1101\ 1110\ 1111^{1}$$

The next step after trace generation, is to build up a statistical model for the data. For this we use an Nth order Markov Model. An Nth order Markov Model is a table of size 2^N which contains $P[1|last\ N\ inputs]$ for each of the possible 2^N last N inputs in the trace. N serves as a limit to the amount of history that we may use in making our predictions.

For our example trace, we build up the following probabilities for a second order Markov table (N=2):

$$P[1|00] = 2/5 \quad P[1|01] = 3/5$$
$$P[1|10] = 3/4 \quad P[1|11] = 6/8$$

$P[1|00]$ is generated by finding all times that 00 is followed by a 1, in this case there are 5 cases of the pattern 00, and 2 of them are followed by a 1.

The corresponding predict 0 probabilities are calculated by subtracting the predict 1 probability from 1 because we only have 2 symbols in our alphabet. Note that while this scales exponentially with the size of N, it is still very reasonable for even the largest values of N we have examined. Having more knowledge of history after a certain point does not improve accuracy and we found that point was well within the reach of the techniques described. For the predictors we are generating we did not see the need to go beyond $N = 9$.

4.3 Pattern Definition

Now that we have the probabilities that we need, we can continue with the next step which is picking the histories that we will eventually build the language L from. In the case of a branch predictor, where we simply wish to minimize the number of mispredictions, the task is straight forward. We simply pick all the histories that have a probability of preceding a 1 which is greater than or equal to $1/2$ to form the language "predict 1". In our example above, that would be the set of histories $\{01, 10, 11\}$. We would like to predict a 1 whenever one of these histories appears in the input based upon our profile, since they had a probability greater than $1/2$. Of course histories with probability equal to $1/2$ can go either way.

We can make further design trade offs at this point in the form of don't care patterns. Some patterns only occur very rarely, and their inclusion into the set of histories will have almost no effect on the performance of the predictor but will make the job of minimizing more complicated. These history patterns can be placed into a third set, separate from the "predict 1" and "predict 0" sets, called the "don't care" set. We have found that by placing only the 1% least seen histories in the "don't care" set we can reduce the size of the predictor by a factor of two with negligible impact on prediction accuracy.

Once we have passed this stage of the design flow, the function of the predictor is set and will not change significantly. The finite state machine that will be designed will return a 1 when histories in the "predict 1" set are seen, and 0 when histories in the "predict 0" set are seen. The output of the "don't care" set is still undecided at this point, and the next stage will take advantage of this to compress the description of the sets.

4.4 Pattern Compression

Now that we have our three sets, "predict 1", "predict 0" and "don't care", the next step is to compress our description of the sets into a usable form. This is done by a standard logic minimization tool, where the input is a truth table. The input side of the truth table is the history patterns captured by our Markov Model, and the output side is the set that the history pattern belongs to. If we continue with our example, we have the out set partitioned up as follows:

$$\begin{aligned} \text{"predict 1"} &= \{01, 10, 11\} \\ \text{"predict 0"} &= \{00\} \\ \text{"don't care"} &= \emptyset \end{aligned}$$

From this we generate a truth table, where all the histories that are contained in the "predict 1" set have there output defined as one, and likewise for the "predict 0" set.

$$\{00 \rightarrow 0,\ 01 \rightarrow 1,\ 10 \rightarrow 1,\ 11 \rightarrow 1\}$$

We then use the logic minimization tool Espresso [22], to minimize this truth table down to a set of logic functions that

[1] the trace is divided up into groups of four only for readability, it has nothing to do with the trace itself

describe the set of inputs that produce a 1. The logic function that is output is our compact description of the "predict 1" set, and it recognizes those histories that we wish to predict 1 or 0 on. This step also merges the "don't care" set into the "predict 1" and "predict 0" sets in such a way as to minimize the number of unique terms. The logic function that is generated is a sum of products representation. The "predict 1" set will satisfy this function and the "predict 0" set will not.

$$((x\ 1) \vee (1\ x))$$

Here we see the sum of products representation of our truth table, where an x represents an input that is unimportant to the outcome of the function. From this description we can now build our regular expression that will capture the language L.

4.5 Regular Expression Building

Once we have the minimized representation of the set of histories we need to build a regular expression which will match these history patterns whenever they come up in the sample input. More specifically, whenever we see an input string, whose trailing N bits match the minimized patterns for the "predict 1" set we return true.

We can build a regular expression as follows. Each term is a clause, each 0 is a 0, each 1 is a 1, and each don't care, denoted as x, matches either a 0 or a 1. Let use start with a single term, from above, $(1\ x)$. This translates to the regular expression $1\{0|1\}$ because it is a 1 followed by either a 0 or 1. Similarly $(x\ 1)$ translates to $\{0|1\}1$. Since a match can be caused by either one, we write the whole expression as $\{1\{0|1\}\} \mid \{\{0|1\}1\}$. However this is not quite complete.

Remember that the language L must describe all possible inputs for which it needs to return 1, not just the last two bits of the input. We overcome this problem by specifying the language to be any string that ends in the desired histories, which can be done by concatenating any number of 1s or 0s in the front of the histories. Thus our final regular expression for the above history is:

$$\{0|1\}^*\{\ 1\{0|1\} \mid \{0|1\}1\ \}$$

Once we have the desired regular expression there is still the matter of converting it to an efficient FSM.

4.6 FSM Creation

The first step in building a FSM from a regular expression is the construction of a non-deterministic finite state machine, which is a state machine that can be in more than one state at a time. Building a non-deterministic FSM is a fairly straight forward process of enumerating paths. Once the non-deterministic FSM is completed it is converted to a deterministic state machine using subset construction [12]. Subset construction can sometimes lead to an exponential blow up in the number of states, and there are better algorithms known which take advantage of reduced alphabets, but we have found subset construction to be more than sufficient for the predictors we examine in this paper.

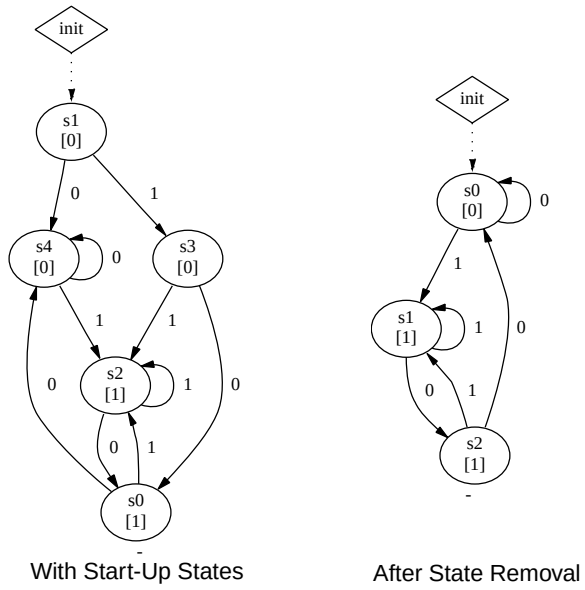

Figure 1: The state machine on the left was generated from the sequence t shown in section 4.2 and optimized with Hopcroft's partitioning algorithm. The state machine on the right is what results from removing the start-up states and renumbering. This is the final state machine for t. Note that the patterns ending in 01, 10, and 11 are still captured correctly. The number in brackets shows the prediction produced by each state in the state machine.

At this point we now have a fully implementable finite state machine, however there are two more steps we take to reduce the number of states used by the state machine. We start by applying Hopcroft's partitioning algorithm [12]. This algorithm removes both unreachable and redundant states, although there are still unneeded states in the state machine.

4.7 Start State Reduction

Since the state machine must recognize all strings in the language, there are unnecessary start up states that are only used at the beginning of the execution where history bits are undefined. Since we are only interested in the steady state operation of the state machine, i.e. those strings with a length greater than N, we can cut out those nodes that are only used in parsing strings less than N in length. This goes against what was said earlier about not changing the semantics of

the state machine, but this optimization only effects the behavior of the state machine on a small constant number of strings. There can be up to 2^N start up states, and they typically account for around one half of all states in the machine.

If we look to figure 1 the problem can be clearly seen. The figure shows the state machines generated from our original sequence t. In the state machine on the left, the states S1 and S3 are not need needed after just the first couple of accesses to the state machine. We would like to remove these to save both state and transition logic complexity.

This can be done by removing all nodes unreachable from the steady state operation of the state machine. All start–up states are unreachable from the steady state operation of machine because you can never have undefined history past the start–up phase. We exploit this fact by simply removing nodes unreachable from steady state which we then know are start up nodes.

Figure 1 shows that the startup states S1 and S3 have been removed and the remaining states have been renumbered. Note that the behavior of the finite state machine is still unchanged past those start up states.

4.8 Synthesis

At this point we have almost reached our final destination. We have a finite state machine which produces prediction based on the input. The predictions are governed by the last N bits of the input string, and that information is efficiently encoded into a finite state machine, as can be seen in figure 1. Starting from any node, and traversing the edges in the FSM following patterns (01, 10, 11) in our "predict 1" set will end at a node with a prediction of 1. Similarly, we will predict 0 for the pattern (00) in our "predict 0" set, starting from any node.

The final step is to actually do synthesis. Since the finite state machine is a well understood and commonly used primitive, every synthesis tool has some form of finite state machine input format. The job of synthesis is to find a efficient hardware implementation for the state machine. This includes finding a good encoding for the states and their transitions. We translate our description of the finite state machine to VHDL, which is then read and analyzed by the Synposys design tool.

5 Customized Branch Predictors

In this section we examine applying our automated approach to designing FSM predictors to the customization of branch predictors. We first describe our customized branch prediction architecture, and then the training approach we use for building the customized FSM branch predictors. We then evaluate the performance and area of the customized predictor and compare it to a range of prior branch predictors.

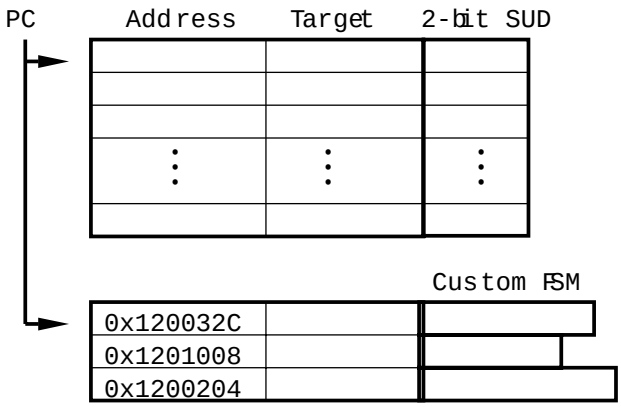

Figure 2: Customized branch prediction architecture. The architecture contains a traditional coupled BTB with 2-bit saturating up-down counters for conditional branch prediction. In addition, customized branch predictors are added for individual branches. A customized branch entry contains a tag (address), target and a custom FSM predictor for predicting the direction of the branch. The tag is associated with the branch that the FSM predictor was built for and is locked down by the system software.

5.1 Customized Branch Prediction Architecture

It is increasingly common for current embedded processors to have branch prediction. For example, Intel's XScale (StrongARM-2) processor [14] has a 128 entry Branch Target Buffer (BTB), and each entry in the BTB has a 2-bit saturating counter which is used for branch prediction. Our goal is to build a branch predictor for a given application that will have the performance of a large general purpose predictor, with about the same area that is already being used by embedded branch predictors.

Our approach is to take a baseline predictor such as the local 2-bit counters from the XScale architecture, and extend this with custom designed FSM predictors for the hard to predict branches. We use the standard two bit counters for most branches and use the custom FSM predictors for branches that do not work well with the default predictor. In doing this, we limit both the amount of additional area we have to add to get good performance and the amount of code we have to hard-wire into the processor.

The custom branch architecture we propose is seen in figure 2. We extend XScale's coupled BTB branch prediction architecture with a set of custom predictors that are hard-wired to particular branches. These custom predictors have the addresses that are associated with them locked down by the system software. While the state machines are fixed in hardware and are targeting specific branches, we wish to allow some software configurability should a re-compile of the

software be needed after fabrication. This will allow the branches to move around in the address space but still use their custom state machines as long as the branch prediction patterns do not change.

The address of the branch is used to index into the BTB as well as the custom predictors. The custom branch entries perform a fully associative tag lookup to search for a match. If there is a match in the standard table then the two-bit saturating up-down counter is used to predict the bias of the branch. If instead there is a match in the custom table then the output on the current state of the corresponding state machine is used for prediction. In the next section we describe how we generated the FSM predictors hard-wired into this architecture.

5.2 Generating Traces to Train FSM Branch Predictors

There are many different types of branch traces one could concentrate on to generate FSM predictors, and we only focus on one in this paper. For the custom branch prediction architecture in Figure 2, traces are generated on a global basis, and then used to generate a FSM predictor for individual branches. To generate a FSM for a specific branch, the traces used to train the FSM generator could include the local history of the branch, global history, or a combination of the two. We examined all of these and found that it is better to concentrate on capturing global correlation rather than a local history pattern because of the global correlation's tendency to be more repeatable across different inputs, and our approach is efficient at finding and taking advantage of global correlations between branches.

The first step we perform in building our custom branch prediction architecture is to profile the application with our base predictor, in this case the local 2-bit counters. This identifies those branches that are causing the greatest amount of mispredictions. For each of these branches we generate a Markov Model as discussed in section 4. To generate the Markov Models that we need for the branches we are concentrating on, we keep track of a single global history register of length N. When a branch is encountered in the trace, we update that branch's Markov Model with the outcome of the branch, given the history in the global history register. The Markov table is then fed into the FSM generator, and a customize FSM predictor is generated for that branch. Since the number of global histories that a given branch might see before it is being predicted is small compared to the 2^N possible histories, the Markov Models can be compressed down significantly by only storing non-zero entries. For all the custom branch prediction results in this paper we use a history of length 9.

For this design only the branch that the custom predictor is generated for uses the FSM for prediction, based on a tag match as described earlier. We update the custom finite state machines in a different manner than the standard local 2-bit counters. We update all of the custom predictors in parallel on every branch, rather than only matching branches. On an update, the branch predictor moves each FSM predictor from one state to the next based upon the prediction of the branch.

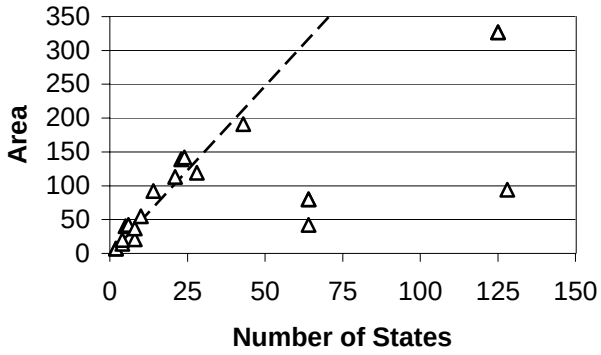

Figure 3: Area of a random sample of the custom FSM predictors taken from all of the benchmarks we examined. The area was determined by Synopsys and plotted against the number of states in that machine. The strong linear bound allows us to estimate area for design tradeoffs and evaluation

5.3 Methodology

We used ATOM [28] to profile the applications and generate the Markov Models over the full execution of the benchmarks. All applications were compiled on a DEC Alpha AXP-21264 architecture using the DEC C compiler under OSF/1 V4 operating system using full optimizations (`-O4`). We chose a set of six benchmarks which could conceivably be used in an embedded environment and have interesting branch behavior. Three of the applications `compress`, `ijpeg`, and `vortex` are from the SPEC95 benchmark suite. We also include three programs from the MediaBench application suite. `Gsm decode` does full-rate speech transcoding, `g721 decode` does voice decompression, and `gs` is an implementation of a postscript interpreter.

The Markov models for the top mispredicted branches in each program were generated using ATOM. After the Markov models were generated, generating all of the FSM predictors for each program using our automated approach took from 20 seconds to 2 minutes on a 500 MHZ Alpha 21264. To evaluate the performance of the FSM predictors, we modeled the custom branch prediction architecture in Figure 2 in ATOM, and gathered their misprediction rate and compared that against general purpose predictors.

5.4 Estimating Area of FSM Predictors

In order to enable us to make high level design choices before synthesis, it is important to be able to estimate the area that our automatically generated state machines will take up. To establish a relationship between the state machine descriptions and their area we took a random sample of custom FSM predictors generated across all of the benchmarks we examined. These account for 10% of all of the FSM predictors generated. We then synthesized these state machines with Synopsys.

Figure 3 shows the number of states in the state machine versus the area of the implementation. Each triangle represents one FSM predictor. The x-axis is the number of states in a given state machine, while the y-axis is the area as reported by Synopsys.

The results show, for most state machines, that the area is linearly proportional to the number of states in the machine. This trend line is drawn in with a dashed line. However this is not the case for all of the FSM predictors. For the state machines with a large number of states, the area is much less than would be predicted by this approximation. For these FSM predictors, there is a large number of states, but the machine is highly regular. Because of this the state machine can be optimized down to a size much smaller than even some of the more chaotic state machines with less state.

We use the linear line in Figure 3 to estimate the area for the rest of the FSM predictors in the results to follow. Even though the approximation does not hold for all of the predictors, it does bound the area of the predictors by the number of states in the state machine. We can use this approximation to make conservative estimates of area. For the rest of this paper we use this approximation to quantify area rather than performing synthesis on each state we wish to examine.

5.5 Branch Prediction Results

We compare out customized predictor against three other branch predictors. The first predictor we compare against is the the gshare predictor predictor of McFarling [17]. The second predictor is a meta chooser predictor that contains a two-level local history branch prediction table, a global history table, and a meta chooser table that determines whether to use the local or global prediction for predicting the current branch. We call this the *Local Global Chooser* (LGC) predictor, and it is similar to the predictor found in the Alpha 21264. The final predictor we compare against is a set of per-branch two bit counters, as is found in the XScale processor. The XScale processor has a 2-bit counter associated with every branch target buffer entry, and not-taken is predicted on a BTB miss.

Figure 4 contains the results for the six programs comparing our custom FSM predictors to the gshare, LGC, and XScale predictors. The x-axis is the total area of the predictor, including the BTB structure, while the y-axis is the misprediction rate. The results for the LGC and gshare were gathered over a range of sizes. We present results for custom predictors trained on inputs, which are different than the ones used for measuring performance. These results are denoted custom-diff. The custom-same results are when we use the same input for training and comparison, and it provides a limit to how well the FSM predictors may perform for that input.

The curve for the custom FSM predictors is generated by increasing the number of branches to be customized. The top-left most point in the curve is the XScale architecture with custom branch predictor. As we add more FSM predictors, the number of mispredictions is reduced and at the same time the area to implement the predictor grows. The results show that for all programs the misprediction rate decreases as we devote more and more chip area to the prediction of branches.

The first thing that is very noticeable is that there is little to no difference between custom-diff and custom-same. This implies that our training output has done a good job capturing the behavior that is inherent to the program. One could certainly use more than one input for training, and indeed this would be a good idea for verifying that the models generated correctly capture the behavior of the program.

For all of the programs, the custom predictors achieve the lowest misprediction rate of any predictor for their size. To beat the performance of these small predictors you would need a general purpose predictor that is 2 to 5 times larger. For some of the programs, even these large table sizes cannot perform better than our custom predictors.

For the program `compress` all of the benefit comes from the state machine for one branch. The misprediction rate is reduced from 23% to 16.5% by adding one custom FSM predictor. Adding more FSM predictors simply increases the area with little to no improvement in misprediction rate. Moderate table sizes of a LGC can outperform our customized predictors because there the branch causing the most mispredictions benefits from having local history. This branch would benefit from having a loop count instruction in a embedded processor, or could easily be captured via customizing the branch predictor to perform loop termination prediction to predict the branch [24].

For `g721`, we can see that the XScale does a very good job of capturing the behavior of most of the branches in the program. However with little extra area we can reduce the misprediction rate from 8% to just over 7%. For `vortex` and `gs` the misprediction rate is reduced significantly from XScale's default local 2-bit counters. For `gs` it is reduced from just under 5% to just over 4%, and for vortex the improvement is a reduction in miss rate from 13% to 3%. For these two programs, a local-global chooser of over two times the size of the custom predictor is needed to get a lower misprediction rate.

The best results are seen for `ijpeg` and `gsm`. For both

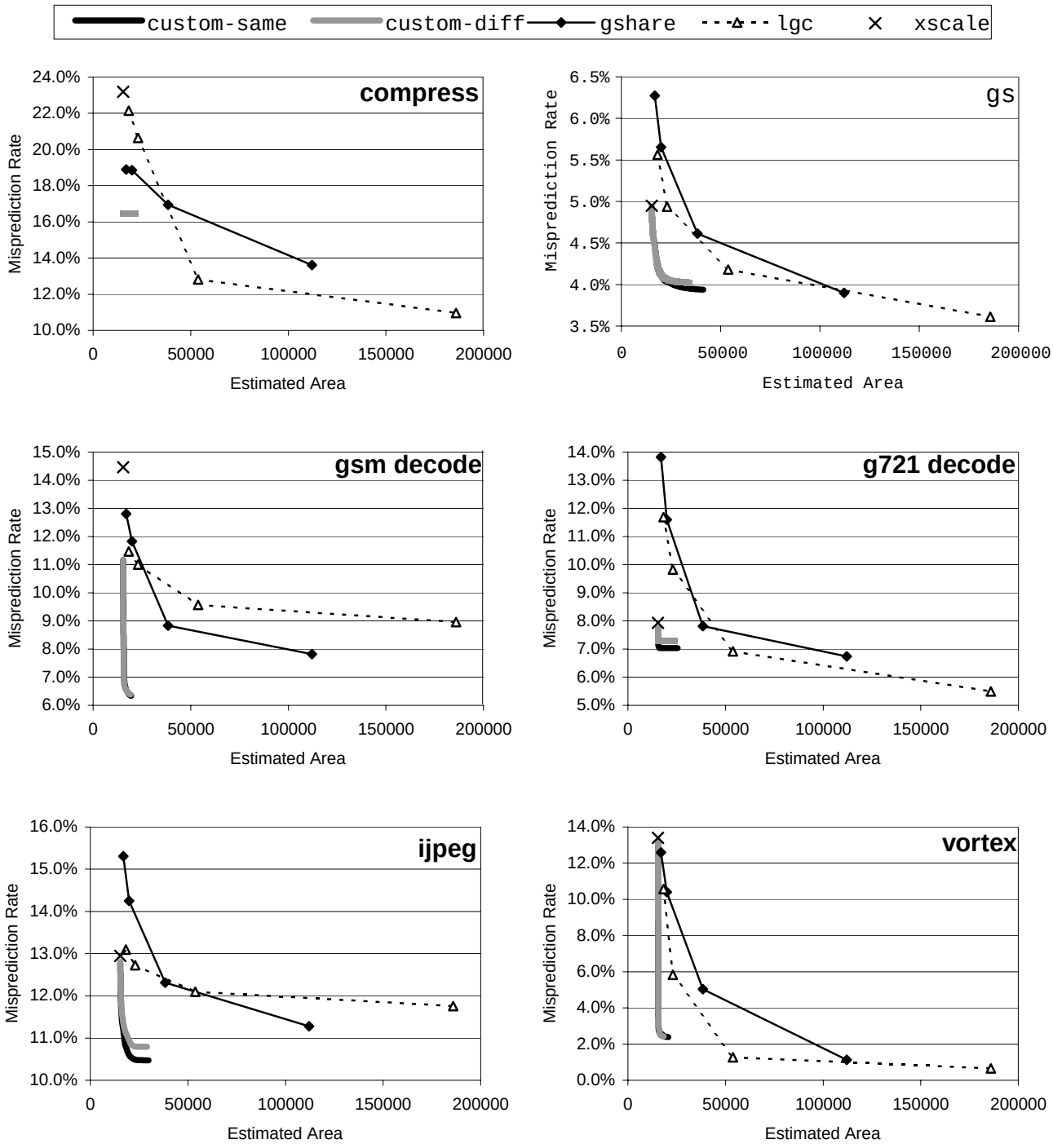

Figure 4: Misprediction rate versus estimated area for the six benchmarks examined. Results are shown for the baseline XScale predictor, gshare, a meta predictor with a chooser between local and global history (LGC), and the customized branch predictor. The customized branch predictor results are for training on a the same input used for testing, custom-same, and training on a different input, custom-diff.

of these programs the misprediction rate is far below that of even the largest table we examined. These programs do not benefit from local history. This can be seen by comparing how LGC performs against gshare for these programs. Since our scheme captures the global correlation so efficiently we can outperform the largest gshare by half a percent for `ijpeg` and over a full percent for `gsm` while only using a fifth of the area.

5.6 Custom Finite State Machine Examples

Figures 5 and 6 are two examples of simple finite state machines that were generated using the techniques presented. Figure 5 serves as a good starting point for understanding how the state machines are used to generate predictions.

The state machine in figure 5 was automatically generated to target a particular branch in `ijpeg`. Each state is annotated with a prediction, shown in brackets. The state machine is updated by traversing an edge. An update with taken will traverse the edge labeled 1, while not-taken will traverse the edge labeled 0. This particular state machine was generated to capture a branch that is highly correlated with the branch that is two branches back in the history. For example, we would like to predict a 1 if the history patterns is "10" or "11" and predict 0 in all other cases. As can be seen in the figure, this is successfully captured by the finite state machine.

If you start in any state of the machine and you follow two transitions, first a 1 and then either a 0 or a 1, you will end up in a state that is labeled a 1. The converse is also true. If you follow an edge labeled 0, followed by either a 1 or a 0, the state you end up in will predict 0. This property is is maintained into even the most complicated predictor.

Figure 6 is a slightly more complex state machine, this time generated for one of the branches in `gs`. This state machine captures two different patterns, $0x1x$ and $0xx1x$ where x is a "don't care". If, as above, you traverse any set of edges that match either of these patterns you will end up at a state predicting a 1. If the edges you traverse do not match these patterns you will end up in a state predicting 0.

For this branch there are four global history patterns that are seen the majority of the time, 001001010 which is biased taken, 010011010 which is biased not-taken, 010101010 which is biased taken, and 110010010 which is biased taken. There are other patterns which contribute but these represent over 90% of the patterns seen by this state machine. If you trace through these patterns, or just the last five digits of them, starting from any state you will end up in a state that predicts correctly. Because of this fact, it does not matter that we are updating the branch predictors for branches that may have never been trained on. No matter what state the machine has gotten itself into, as long as the final sequence of branches leading up to the branch is captured by one of the patterns the state machine will make the correct prediction.

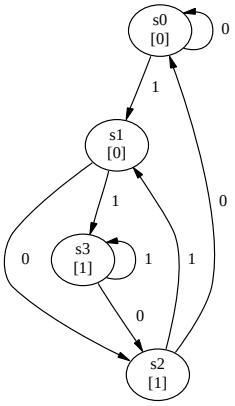

Figure 5: Finite state machine generated for a branch in `ijpeg`. This simple state machine captures the history pattern $1x$.

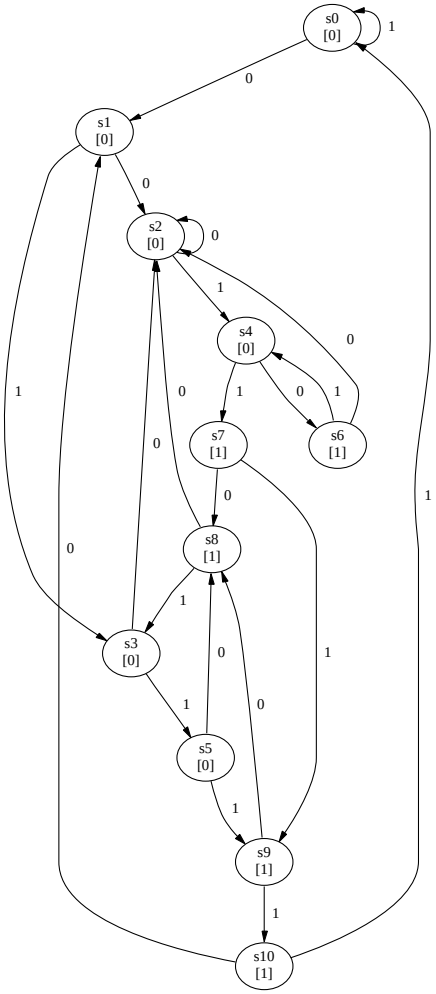

Figure 6: A slightly more complex state machine generated from `gs` that captures two patterns. Patterns that match $0x1x$ or $0xx1x$ predict taken while all others predict not-taken.

These results show that our automated steps can be used to accurately identify branches that are highly correlated, and the nature of the correlation. This information could then be used to potentially guide additional compiler and hardware optimizations.

6 Other Uses of Finite State Machine Predictors

In this section we describe how FSM predictors are used in a few areas of computer architecture, and summarize initial results for using automated FSM predictors to guide confidence estimation used to guide value prediction.

6.1 Cache Management

Cache management schemes have been proposed that perform intelligent replacement [16], cache exclusion [29], and they use a small FSM counter to determine when the optimization should be applied. In addition, prefetching architectures have used FSM predictors to determine when to initiate prefetching for a load and to guide stream buffer allocation [25].

6.2 Power Control

Manne et. al. [15] examined using confidence estimation to find branches that had a high miss rate, and then for those branches, stall the fetch unit until the branch direction is resolved. This can save a significant amount of power for branches that have a high miss rate, and is beneficial for low power processors. Musoll [19] proposed using hardware predictors to predict whether an access to certain hardware blocks can be avoided in order to save power.

6.3 Value, Address, and Load Speculation

Confidence counters have been used to guide many speculative execution architectures. These include value prediction, address prediction, instruction reuse, memoization, load speculation, and memory renaming [3]. In all of these architectures, speculation occurs to hide latency caused by a real or potential dependency. When a misprediction occurs, the penalty can be costly. Therefore, confidence estimation counters are used to guide when to use the speculative optimization.

Calder et. al. [4, 3] and Burtscher and Zorn [2] both noted the need for improved confidence estimation for value prediction. In [4, 3], we examined *Saturating Up and Down* (SUD) counters with several different saturation thresholds and wrong decrement values to vary the accuracy and coverage, and this was done by trial and error. We showed that no one SUD counter worked best for all programs. A very accurate SUD counter was needed for mispredicted values when using squash recovery to obtain increases in performance, but this resulted in low coverage of potential value predictions. In contrast, when value prediction used re-execution recovery, it did not have to be as accurate, since the miss penalty is small, and the SUD counter could instead concentrate on achieving a high coverage.

We examined using our automatically generated FSM predictors for value prediction confidence estimation. The level of accuracy that can be achieved from a value prediction architecture can be easily configured by increasing or decreasing the prediction threshold of the confidence estimation counter. We examined generating custom FSM confidence estimators to achieve 50%, 60%, 70%, 80%, and 90%, and 95% accuracy, and examined the corresponding coverage that resulted. We compared this result to the best confidence estimators that have been proposed for value prediction in prior research [4, 2].

We found that our custom FSM confidence estimators could consistently achieve higher coverage when confining both techniques to the same area and accuracy. For example, if we fix the accuracy to be achieved by the confidence estimation predictor to be 80%, then using our automatically generated FSM confidence estimators achieves 80% coverage, whereas the best existing confidence estimation approach achieved only a coverage of 63% of value predictions. Additional research into finding efficient FSM confidence predictors and evaluating their performance improvements is part of our future research.

7 Summary

Predictive finite state machines are a fundamental building block in computer architecture and are used to control and optimize all types of prediction and speculation. In this paper we present an automated approach to generating FSM predictors using profile information. This approach can be used to develop FSM predictors that perform well over a suite of applications, tailored to a specific application, or even a specific instruction.

The algorithm we present uses profiling to build a Markov model of the global correlation present in a target application. From this model we define a set of patterns to capture. These patterns are compressed into a usable form and then translated into a regular expression. From the regular expression we use subset construction to build a finite state machine. This state machine is then optimized for our uses using Hopcroft's Partitioning Algorithm and Start State Reduction.

We also present a customized branch prediction architecture that makes use of these custom built finite state machine predictors. The FSM predictors are generated for branches that are not easily captured by local two-bit counters. These custom state machines take up little area, and can efficiently and accurately capture the global correlation behavior of the target application. The global correlation is shown to be cap-

tured across input sets and results are presented for a variety of predictor sizes.

For all of the programs examined, our custom predictors achieve a misprediction rate less than a general purpose predictor of twice it's size or more. For two of the programs, the custom branch misprediction rates are lower than general purpose predictors of five times their size.

As customized processors become an increasingly attractive design option, the techniques presented in this paper will offer the embedded systems designers the ability to efficiently and effectively take advantage of application specific prediction structures to increase performance and reduce area.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by DARPA/ITO under contract number DABT63-98-C-0045, and a grant from Compaq Computer Corporation.

References

[1] S. G. Abraham and S. A. Mahlke. Automatic and efficient evaluation of memory hierarchies for embedded systems. In *32nd International Symposium on Microarchitecture*, 1999.

[2] M. Burtscher and B.G. Zorn. Prediction outcome history-based confidence estimation for load value prediction. *Journal of Instruction-Level Parallelism*, 1, 1999.

[3] B. Calder and G. Reinman. A comparative survery of load speculation architectures. *Journal of Instruction Level Parallelism*, 2, 2000.

[4] B. Calder, G. Reinman, and D. Tullsen. Selective value prediction. In *26th Annual International Symposium on Computer Architecture*, June 1999.

[5] I.-C. Chen, J.T. Coffey, and T.N. Mudge. Analysis of branch prediction via data compression. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[6] J. Emer and N. Gloy. A language for describing predictors and its application to automatic synthesis. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[7] G. Ezer. Xtensa with user defined dsp coprocessor microarchitectures. In *Proceedings of the International Conference on Computer Design, 2000 (ICCD2000)*, pages 335–342, September 2000.

[8] J. A. Fisher, P. Faraboschi, and G. Desoli. Custom-fit processors: Letting applications define architectures. In *29th International Symposium on Microarchitecture*, pages 324–335, December 1996.

[9] M. J. Flynn and R. I. Winner. Asic microprocessors. In *23th International Symposium on Microarchitecture*, pages 237–243, 1990.

[10] R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, March-April 2000.

[11] D. Grunwald, A. Klauser, S. Manne, and A. Pleskun. Confidence estimation for speculation control. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[12] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[13] E. Jacobsen, E. Rotenberg, and J.E. Smith. Assigning confidence to conditional branch predictions. In *29th International Symposium on Microarchitecture*, December 1996.

[14] S. Leibson. Xscale (strongarm-2) muscles in. *Microprocessor Report*, September 2000.

[15] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[16] S. McFarling. Program optimization for instruction caches. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 183–191, April 1989.

[17] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.

[18] H. Mulder and R. J. Portier. Cost-effective design of application specific vliw processors using the scarce framework. In *22th International Symposium on Microarchitecture*, 1989.

[19] E. Musoll. Predicting the usefulness of a block result: a micro-architectural technique for high-performance low-power processors. In *32nd International Symposium on Microarchitecture*, November 1999.

[20] B. Ramakrishna Rau and Michael S. Schlansker. Embedded computing: New directions in architecture and automation. In *7th International Conference on High-Performance Computing (HiPC2000)*, 2000.

[21] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *27th International Symposium on Microarchitecture*, pages 172–180, 1994.

[22] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for pla optimization. *IEEE Transactions on Computer Aided Design*, 6(5):727–750, 1987.

[23] R. Schreiber, S. Aditya, B.R. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider. High-level synthesis of nonprogrammable hardware accelerators. Technical report, Hewlett Packard Reseach Labs, 2000. HPL-2000-31.

[24] T. Sherwood and B. Calder. Loop termination prediction. In *3rd International Symposium on High Performance Computing*, October 2000.

[25] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *33rd International Symposium on Microarchitecture*, December 2000.

[26] J. E. Smith. A study of branch prediction strategies. In *8th Annual International Symposium of Computer Architecture*, pages 135–148. ACM, 1981.

[27] C.D. Snyder. Fpga processors cores get serious. *Microprocessor Report*, 14(9), September 2000.

[28] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[29] G. Tyson, M. Farrens, J. Mathews, and A. Pleszken. Managing data caches using selective cache line replacement. *International Journal of Parallel Programming*, 25(3), 1997.

[30] G. Tyson, K. Lick, and M. Farrens. Limited dual path execution. Technical Report CSE-TR 345-97, University of Michigan, 1997.

[31] T.Y. Yeh and Y.N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *20th Annual International Symposium on Computer Architecture*, pages 257–266, San Diego, CA, May 1993. ACM.

[32] A. Zhi, A. Moshovos, S. Hauck, and P. Banerjee. Chimaera: A high performance architecture with a tightly-coupled reconfigurable functional unit. In *27th Annual International Symposium on Computer Architecture*, June 2000.