GPU Acceleration of a Genetic Algorithm for the Synthesis of FSM-based Bimodal Predictors

Martin Burtscher[1] **and Hassan Rabeti**[2]
[1]Department of Computer Science, Texas State University, San Marcos, TX, USA
[2]Department of Mathematics, Texas State University, San Marcos, TX, USA

Abstract - *This paper presents a fast GPU implementation of a genetic algorithm for synthesizing bimodal predictor FSMs of a given size. Bimodal predictors, i.e., predictors that make binary yes/no predictions, are ubiquitous in microprocessors. Many of these predictors are based on finite-state machines (FSMs). However, there are countless possible FSMs and even heuristic searches for finding good FSMs can be slow when billions of predictions need to be assessed. We designed such a search heuristic that maps well onto GPU hardware. It is based on a multi-start genetic algorithm. On our six traces, the resulting FSMs are 1% to 29% more accurate than saturating up/down counters. On a Kepler-based GTX 680, the CUDA implementation evaluates 18 to 73 billion predictions per second, which is 14 to 18 times faster than a multicore version running on a hex-core Xeon X5690 with hyper-threading.*

Keywords: GPGPU, genetic algorithm, automated design, finite-state machines, bimodal predictors

1. Introduction

Modern processors contain large numbers of finite-state machines (FSMs), many of which are used as bimodal predictors. Such FSMs can be found in branch predictors [13, 15, 19], memory-disambiguation hardware [20], cache way predictors [2], confidence estimators [9], and selectors in hybrid predictors [14]. Their purpose is to improve performance and/or reduce power consumption [17]. We use FSMs to compress program execution traces in real time [16]. In nearly all of these applications, the FSM has to repeatedly make a 1-bit prediction, *i.e.*, a bimodal prediction, and is then updated with the true 1-bit outcome. *E.g.*, for every branch instruction, an FSM might predict whether it will be taken or not. After the branch has executed, the FSM is updated with the true direction the branch took. The goal is to make as many correct predictions as possible. However, there are countless choices of FSMs and it is generally unknown which FSM is the best for a given task.

An n-bit FSM holds n bits of internal state, which serves as its 'memory'. The 1-bit prediction is a function of the current state, such as choosing one of the n bits. During an update, the FSM transitions from the current state to a new state based on the input (true outcome) bit. Conceptually, a bimodal n-bit FSM implements a transition table like the one shown in Figure 1, where the n bits of current state are concatenated with the input bit to form an address (index) to select a row in the table, which holds the next n-bit state.

As the boxed-in letters in Figure 1 illustrate, the transition table consists of $n \times 2^{n+1}$ independent bits, yielding 2 ^ ($n \times 2^{n+1}$) possible n-bit FSMs. Whereas not all bit assignments result in meaningful FSMs (*e.g.*, there are redundancies and not every FSM can reach all states), the number of possibilities grows super-exponentially with n. There are 16 possible 1-bit FSMs but 65,536 possible 2-bit and 281.5 trillion possible 3-bit bimodal FSMs. Hence, using an exhaustive search to determine the best n-bit FSM is not computationally tractable on current workstations for $n > 2$.

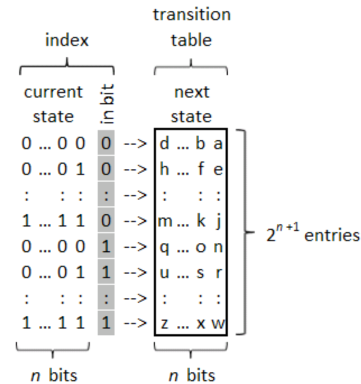

Figure 1. State transition table of an *n*-bit bimodal FSM

A saturating up/down counter is a specific bimodal FSM that works as follows. Its n-bit state is interpreted as an n-bit value. When updated, the value is incremented if the input bit is 1 and decremented otherwise. However, the value is never incremented above 2^n-1 and never decremented below 0, *i.e.*, it saturates at the minimum and maximum. The prediction is the most significant bit (MSB). The saturating up/down counter is so called because it counts the number of 0 and 1 outcomes that were encountered in the recent past. If there were many zeros, the count is low and the MSB a '0'. Conversely, if there were many ones, the count is high and the MSB a '1'. Hence, this FSM essentially makes a majority prediction over the recently seen events. The saturating up/down counter works well in practice, which is why it is widely used. However, it has known weaknesses. For example, it performs poorly on sequences of alternating zeroes and ones. Also, it tends to make the same prediction after a '1 1 0 1' sequence as it does after a '1 0 1 1' sequence.

Whereas there is generally only one piece of logic that im-

plements the FSM in hardware, the *n*-bit state itself is often replicated, resulting in an array of states, to improve the prediction accuracy by retaining separate state for different instructions, cache lines, *etc*. Some of the lower bits of the program counter (PC) of the executing instruction are typically used to select an entry in the state array.

Since performing an exhaustive search for finding the best FSM is computationally intractable for all but the smallest problem sizes, heuristic approaches for finding near-optimal solutions need to be used. Examples include simulated annealing [1], genetic algorithms [8], ant colony optimization [3], and multi-start search algorithms [5]. We use a combination of a genetic and a multi-start algorithm because it maps particularly well to current GPUs.

Our algorithm generates multiple sets of random transition tables (*i.e.*, FSMs) and then attempts to improve each set independently using a genetic algorithm (GA) until a locally optimal solution is reached. In each GA step, the FSMs of the current 'population' are evaluated to determine how many correct predictions they make on a given input. (The input is a trace of 1-bit events and their corresponding PC values to index the state array.) Then, the next generation of FSMs is created using mutation and crossover operations. A quarter of the new population is generated by mutating random bits of the best-performing FSM from the previous generation, that is, each bit in the state-transition table is randomly flipped with 25% probability. The remaining three quarters of the new population is generated by combining the best FSM with a randomly selected FSM from the previous generation (we chose these values because they result in a simple implementation and good performance). Each of these crossovers uses a different random bit mask to select which bits should be taken from the best FSM. Each bit has a 75% chance of coming from the better 'parent' FSMs. The best FSM is copied over into the new generation to ensure that the performance never drops.

This paper makes the following contributions.

- It presents the first GPUGA for optimizing predictor FSMs.
- It describes how to efficiently map this algorithm to GPUs and compares its performance to multicore CPU code.
- It provides results for Fermi- and Kepler-based GPUs.
- It analyzes, visualizes, and discusses the best FSMs.
- The CUDA source code is publicly available at http://cs.txstate.edu/~burtscher/research/FSM_GA/.

The rest of this paper is organized as follows. Section 2 explains the CUDA implementation in detail. Section 3 summarizes related work. Section 4 presents the evaluation methodology. Section 5 evaluates the parameter space and discusses the performance results. Section 6 concludes the paper with a summary.

2. CUDA implementation

The combination of a multi-start search with a genetic algorithm for determining well-performing FSMs was chosen because it is particularly well suited for GPU acceleration. It avoids potential performance hurdles such as uncoalesced memory accesses, thread divergence, and inter-block dependencies. Moreover, it naturally maps to the GPU's block and thread hierarchy and takes advantage of the block scheduler for load balancing.

Each population of FSMs is evaluated in its own block. This makes the blocks independent except for a single atomicMax operation to determine the globally best FSM. Each GA-based search terminates when the performance of the best FSM has not improved over the previous generation. This means that some blocks have to evaluate more generations than other blocks do, resulting in load imbalance. However, the GPU's block scheduler automatically launches another block as soon as one block has finished executing, thus keeping all SMs busy until the scheduler runs out of new blocks towards the end.

For all but very short inputs, the innermost loop that evaluates the prediction accuracy is the most time consuming code section. It iterates over the trace entries, contains no control transfers in its body and is therefore thread divergence-free, reads the trace data in a fully coalesced manner from global memory and also performs fully coalesced reads and writes of the state arrays in local memory. The code exclusively uses integer data and operations.

Users can parameterize the implementation along four dimensions: (1) the population count, which determines the number of blocks, (2) the population size, which determines the number of threads per block, (3) the number of entries per state array, and (4) the size of the FSM. For clarity, we only focus on 3-bit FSMs in this paper.

Given the above assignments and current GPU specifications, the population count has to be between 1 and 65,535 on Fermi and between 1 and 2^{31}-1 on Kepler, the population size needs to be between 1 and 1024, and the number of entries in the state arrays has to be a power of two (for efficiency) between 1 and 32,768 due to local-memory size limitations. All FSM state arrays are initialized to zero. The LSB of the FSM's state is used for making predictions.

To maximally exploit the GPU hardware, it is advisable to select a population count that is substantially larger than the number of blocks the SMs can execute concurrently (to fully load the GPU and to allow the scheduler to balance the load). The population size should be a multiple of 32 (to fill warps entirely) and at least 192 on Fermi (because it can run up to 8 blocks per SM) and 128 on Kepler (because it can run up to 16 blocks per SM) to reach 1536 and 2048 threads per SM, respectively. Larger population counts and sizes result in longer runtimes but potentially also better results. The number of entries in the state arrays is likely problem dependent, but shorter arrays result in better data-cache performance and therefore better overall throughput.

The input trace consists of a sequence of 2-byte values, one value per event, where the least significant bit is the true outcome and the remaining 15 bits represent the bottom 15 bits of the PC (that are not always zero). The only constraint is that the trace has to fit into the GPU's main mem-

ory. For example, a GPU with 2 GB of DRAM can process traces with up to one billion events.

Even though the GA is orders of magnitude faster for large FSMs than an exhaustive search, it still needs to evaluate

state transitions. Assuming a trace with one million events, 128 populations, a population size of 512, and an average of 5 generations, this amounts to 328 billion state transitions to be evaluated. At 30 billion state transitions per second on a fast GPU, this takes about 11 seconds to execute. The same parameters but with a one-billion-event trace result in a runtime of 3 hours, highlighting the importance of accelerating even genetic algorithms. Note that many and/or long traces are necessary to improve the generality of the FSM. Large population sizes and large population counts in particular are needed to improve the prediction accuracy by allowing the GA to diversify, *i.e.*, not get stuck in a local maximum.

The code uses random numbers to initialize the transition tables of the first generation of FSM, to determine the mask values for the crossover operations, and to select bits to flip for the mutation operations. We use the XORWOW pseudo-random number generator from the cuRAND library that is included with CUDA 5.0.

For comparison purposes, we also wrote a multicore CPU version of our code. It is largely the same as the CUDA implementation. In particular, the most time-consuming loop that iterates over the trace entries is identical. The CPU code parallelizes the loops that iterate over the FSMs of a population using OpenMP *parallel for* directives with a dynamic schedule. Since the code uses the *rand_r* function from the standard C library to generate the random numbers, the results between the C and the CUDA implementations are not directly comparable, which is why we only compare the throughputs.

3. Related work

Fogel *et al.* first developed evolutionary programming [6] and considered using it to evolve FSMs for time-series predictions [7]. Similar to their approach, we evolve FSMs using mutations and crossovers of state transition tables to find better machines. Holland furthered the application of evolutionary techniques by creating Genetic Algorithms (GAs), *i.e.*, a framework of genetic operations on populations of individuals [10].

Since the introduction of CUDA, many genetic algorithms have been accelerated using GPUs, in particular the fitness evaluation, which generally represents the overwhelming majority of the computation (also indicated by our results) [12]. However, to the best of our knowledge, there is no prior work on GPU acceleration of a genetic algorithm for determining good FSMs. The following three projects are the most similar to our work in that their goal is also to automatically generate well-performing FSMs.

Emer and Gloy introduced an algebraic-style notation to express state identification and feedback processes [4]. In their genetic programming search, they represent individuals by a tree that consists of predictor, function, and terminal nodes. The predictors contain dedicated memory (used in dynamic predictions), size and index information as well as conditions for updating the state of the predictor (feedback process). Functions are internal relation operations such as XOR or SATUR (saturating add). Terminals handle the input and updates for each prediction problem. These nodes can be modified in the genetic programming process to evolve more sophisticated predictors. *E.g.*, by performing a crossover they might combine one predictor's function with another predictor (with some constraints) or modify the size of memory allotted for that predictor. The result of the genetic programming search is the most successful predictors with the smallest misprediction ratio (fitness measure) as well as their configurations. Note that Emer and Gloy employ genetic programming to search for (arbitrarily complex) candidate predictors whereas we explore candidate transition tables of fixed-size bimodal FSMs.

Sherwood and Calder introduced an approach that automatically builds FSM predictors designed to find efficient n^{th}-order Markov model FSMs for small design areas by analyzing profile information [18]. They do not use a genetic algorithm. Rather, they express sets of compact strings in form of regular expressions. By mapping these regular expressions to FSMs, the FSMs can identify the input strings of their corresponding language. A key difference between their work and ours is the use of an nth-order Markov model compared to our genetic search. This results in the cost of having to maintain a Markov table for the history of probabilities. Moreover, much of their work is not directed towards performance, which is one of our key objectives.

Jackson and one of us proposed a pure hardware implementation of a genetically evolving set of bimodal FSMs for confidence estimation that does not require intervention from the user or profiling [11]. Confining the method to hardware allows for dynamic adaptation but restricts the population count and size to very small values compared to the software solution presented here.

4. Experimental methodology

4.1 Systems and compilers

We evaluate the CUDA code on two GPUs, a Fermi-based GeForce GTX 480 and a Kepler-based GeForce GTX 680. The GTX 480 has 15 SMs with 480 CUDA cores in total, 1.5 GB of global memory, is clocked at 1.4 GHz, and supports compute capability 2.0. The GTX 680 has 8 SMXs with 1536 CUDA cores in total, 2 GB of global memory, is clocked at 1.05 GHz, and supports compute capability 3.0. The compiler is nvcc version 5.0. The CUDA source code is the same for both GPUs, but the compiler flags are '-O3 -arch=sm_20' for the Fermi and '-O3 -arch=sm_30' for the Kepler. The code uses 48 kB of L1 data cache and 16 kB of shared memory per SM.

The CPU code is written in C, parallelized with OpenMP,

and run on two hex-core Xeon X5690 CPUs with hyper-threading, *i.e.*, 24 threads in total. The two processors are clocked at 3.47 GHz, have a 12 MB L3 cache each, and share 24 GB of main memory. Each CPU core has dual 32 kB L1 caches and a 256 kB L2 cache. We use gcc version 4.4.6 with the '-O3 -msse4.2 -fopenmp' switches. The operating system is 64-bit CentOS version 6.3.

To maximize the performance, we hardcode the user selectable parameters, *i.e.*, the population count, the population size, the number of elements in the state arrays, and the FSM size in both the C and CUDA codes. This requires a recompilation after every parameter change but results in faster program execution. Since each of our experiments takes several minutes or longer to run, the approximately one second of compilation time is easily amortized.

4.2 Measurements

All timing and throughput measurements are performed by instrumenting the source code, *i.e.*, by adding code to count the number of generations and to read a timer before and after the measured code section. We measure the wall time of the CUDA kernel or the C function that evaluates the FSMs and performs the genetic algorithms – which, on our traces, represents essentially all of the total runtime. Each experiment is conducted once because tests showed the runtimes to be quite stable between multiple runs with identical parameters.

4.3 Trace datasets

We use six datasets for our evaluation. They were extracted from two SPEC programs running on a 64-bit RISC machine. One program is gcc compiling a 638-line C program that implements the Barnes-Hut *n*-body simulation algorithm. The other program is mcf, a combinatorial optimization code running the provided train input. We extracted three traces from the user and library code of both programs (*i.e.*, we did not capture the operating system code, which is negligible in SPEC programs). The first trace records, for all executed branch instructions, whether they were taken or not. The second trace records, for all executed load instructions, whether their effective addresses are stride prefetchable. The third trace records, for all executed load and store instructions that hit in a 2-way associative data cache, whether the first or the second set holds the accessed data.

Table 1. Trace information

Program	Trace type	Length [entries]	Length [MB]	Ones [%]	Unique PCs	$2^{H(PC)}$
gcc	branch outcome	60,666,667	115.7	27.0	14,881	2754.7
gcc	prefetchability	97,155,132	185.3	48.6	22,631	4476.8
gcc	way selection	144,637,560	275.9	50.4	26,420	4900.5
mcf	branch outcome	29,474,825	56.2	45.1	943	89.8
mcf	prefetchability	38,047,003	72.6	40.0	1,698	142.3
mcf	way selection	61,234,883	116.8	51.7	2,562	119.3

Table 1 summarizes pertinent information about each dataset. The 'ones' column indicates the percentage of the trace entries with a true outcome of '1', that is, how biased the entries are. The unique PCs reflect how many of the 32,768 possible PC values occur in the trace. This determines the maximum number of state-array entries that will be used. However, some PCs occur rarely whereas others are very frequent. To account for this variability, we also computed the entropy of the PCs: *H(PC)*. Raising 2 to the power of this entropy yields a 'weighted' number of PCs and therefore state-array entries, *i.e.*, a measure of the working-set size below which significant aliasing is likely to occur.

5. Results

Unless otherwise stated, the default parameters for our genetic algorithm are a population count of 128, a population size of 512, and 1024 entries in the state arrays. These population counts and sizes result in good 3-bit FSMs and in high throughputs on the GPUs, as they map well to the given architectures. We picked 1024-entry state arrays because that is a reasonable size for hardware tables.

5.1 FSM quality

We first evaluate the quality of the best 3-bit bimodal FSMs that the genetic algorithm finds by comparing them to the 3-bit saturating up/down counter as well as to the optimal bimodal 1-bit and 2-bit FSMs, which were determined with an exhaustive search. Figure 2 plots the misprediction ratio in percent against the state-array size for the four types of FSMs. The left panels refer to gcc and the right panels to mcf. The top pair of panels shows the results for the branch outcome traces, the middle pair for the stride prefetchability traces, and the bottom pair for the cache way traces. Note that the y-axes are different for each panel and are not zero based to improve readability.

The optimal 1-bit FSM performs relatively poorly, especially on the two branch outcome traces, because it retains the least amount of state. Nevertheless, it occasionally outperforms the 3-bit saturating up/down counter on the non-branch traces, particularly with large state arrays. On mcf's cache way trace, the optimal 1-bit FSM is consistently and significantly better than the 3-bit counter, which is the worst FSM on that trace. This highlights that saturating counters are not always good choices, particularly when predicting non-branch events.

On both branch traces, the optimal 2-bit FSM is, in fact, the 2-bit saturating up/down counter (with large state arrays). Interestingly the 3-bit saturating up/down counter always outperforms the 2-bit counter on gcc, but on mcf the 3-bit counter is sometimes worse than the 2-bit counter. The optimal 2-bit FSM always outperforms the optimal 1-bit FSM because the 2-bit FSMs are a superset of all possible 1-bit FSMs. The optimal 2-bit FSM often beats the 3-bit counter except on the gcc branch trace. Yet, the optimal 2-bit FSM never outperforms the best 3-bit FSM produced by the GA, indicating that the genetic algorithm works well.

In fact, on our traces, the GA always yields the best FSM for all state-array sizes tested. These FSMs perform 1% to 90% better than the optimal 1-bit FSM, 1% to 29% better than the optimal 2-bit FSM, and 1% to 41% better than the

3-bit saturating counter. Importantly, on all six traces, the best 3-bit FSM often outperforms (by up to 26%) the optimal 2-bit FSM with twice the state-array entries, making the 3-bit FSM the more state-efficient solution. Similarly, the best 3-bit FSM often outperforms (by up to 52%) the optimal 1-bit FSM with four times as many state-array entries, again making the 3-bit FSM more size efficient.

Interestingly, on five of the six traces, the best FSMs sometimes perform worse with larger state arrays. This generally happens at the low end, where the aliasing in the state array is high. Apparently, increased aliasing does not always hurt the prediction accuracy. In fact, the mcf cache-way trace is best predicted by all four FSM types when they are only given one entry in the state array. Clearly, there is a substantial amount of correlation between the selected cache way in this trace, which, overall, is the most difficult-to-predict of our six traces.

Notwithstanding the constructive aliasing in very small state arrays with 16 or fewer entries, we find that the entropy-based minimal number of needed entries (*cf.* the last column in Table 1) accurately indicate the state-array size above which the performance improvement flattens out in all six panels of Figure 2.

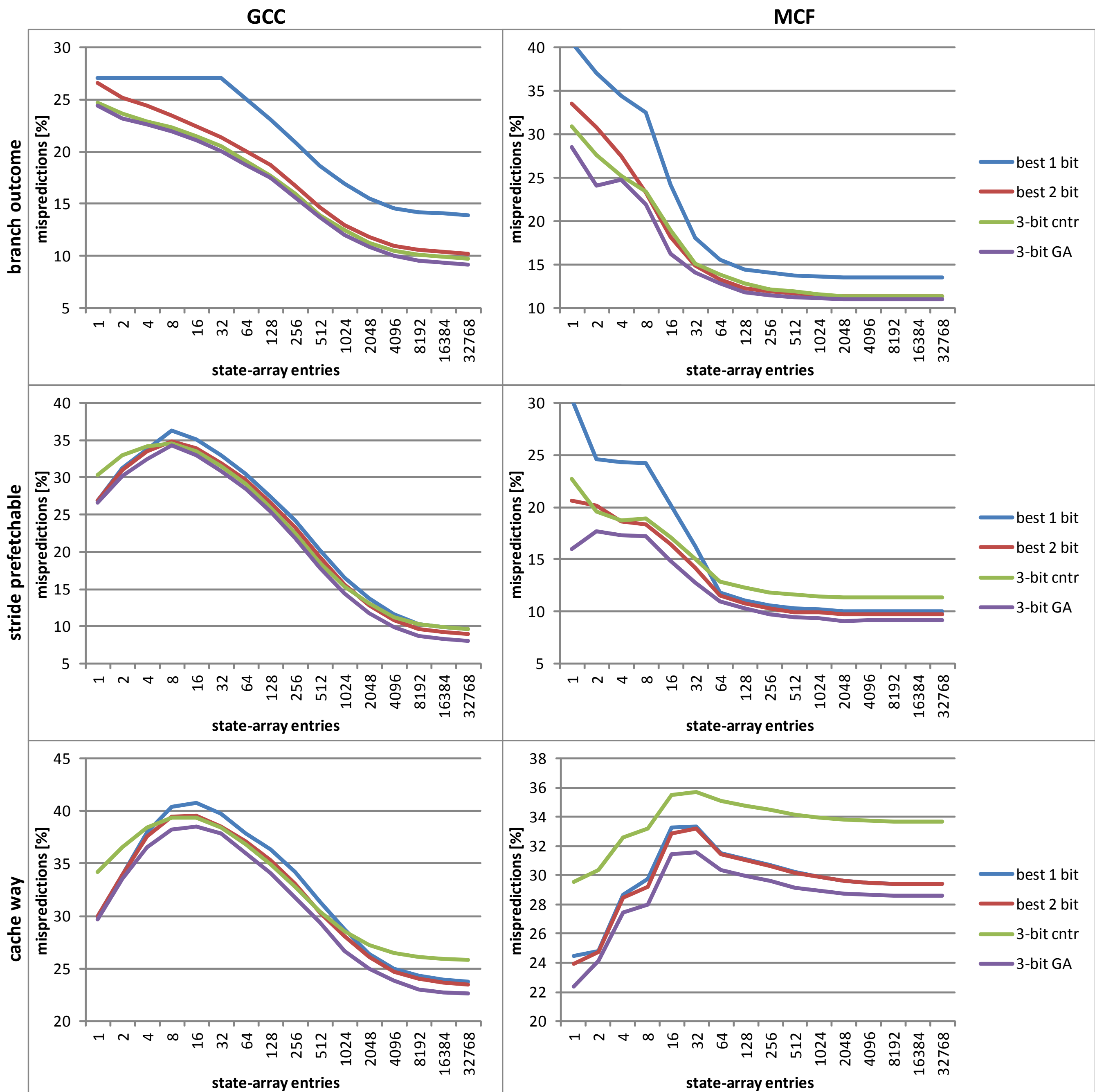

Figure 2. Percent misses (y-axes) for different state-array sizes (x-axes) of four bimodal FSMs on the six traces

5.2 Throughput comparison

This subsection compares the throughput (in billion state transitions evaluated per second) of the CUDA code running on two different GPUs and the OpenMP code running on a system with dual hex-core X5690 CPUs and hyperthreading. For clarity, we only show results for the stride prefetchability trace from mcf.

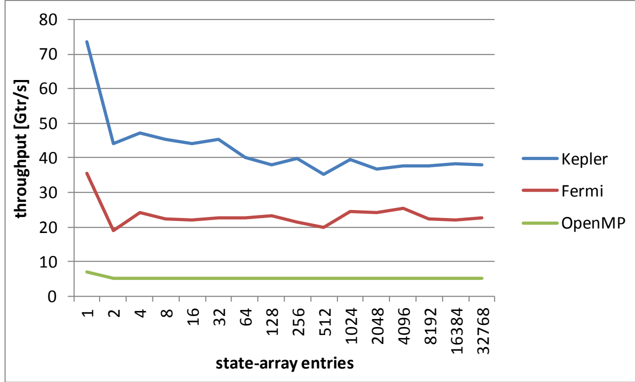

Figure 3. Throughput as a function of the number of state-array entries

Figure 3 shows the throughput for different state-array sizes. On all three processors, a single state yields the highest throughput because the compilers scalarize the 1-entry arrays. The Kepler evaluates 73.6 billion state transitions per second (Gtr/s) in this configuration, the Fermi reaches 35.5 billion, and the two CPUs together peak at 6.9 billion. All larger state-array sizes result in lower but relatively stable throughputs. The Kepler's throughput drops to under 40 Gtr/s for larger array sizes. The Fermi's throughput hovers around 23 Gtr/s. The CPUs' throughput is very stable at 5.3 Gtr/s. Thus, the Kepler outperforms the Fermi by about a factor of 1.5 to 2 and the CPUs by a factor of 7 to 9 or, in a chip-to-chip comparison, one CPU by a factor of 14 to 18.

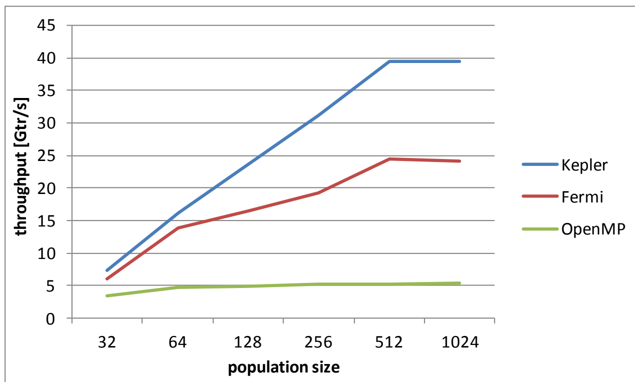

Figure 4. Throughput as a function of the population size

Figure 4 compares the throughputs for different population sizes. Beyond a population size of 32, the CPUs' throughput is almost constant, but the GPUs need a population size of at least 512 to reach their full potential. Since the population size equals the number of threads in a block, it appears that a block size under 512 threads results in inefficient utilization of the GPU hardware.

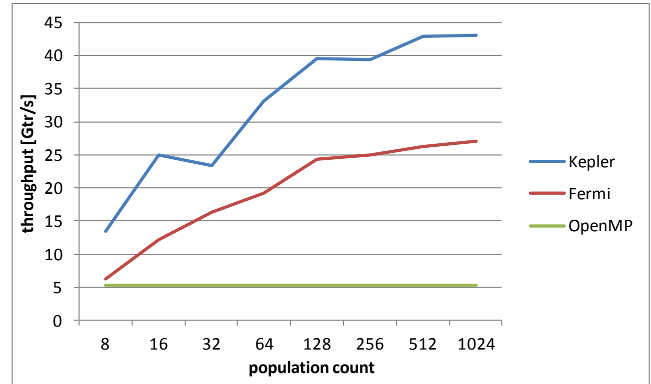

Figure 5. Throughput as a function of the population count

Figure 5 shows the throughputs for different population counts. Because the OpenMP code is parallelized over the FSMs within a population, there is no difference in its throughput when varying the number of populations. However, the CUDA code uses a hierarchical parallelization approach to match the GPU hardware. At least 128 populations (*i.e.*, thread blocks) are necessary to saturate the GPUs. Their performance keeps increasing beyond 128 blocks because larger numbers of blocks result in relatively less load imbalance towards the end when the scheduler runs out of blocks to allocate to the SMs. Note that the Fermi has 15 SMs, which means that a population count of 8 leaves almost half of the SMs with no work. Because SMs can run multiple blocks simultaneously, the Fermi needs at least 45 blocks with 512 threads each to fully load its SMs and the Kepler needs at least 32 blocks. However, at these numbers of blocks, no load balancing is possible as all blocks immediately start running. This is why the throughput only starts to flatten out at about 128 blocks.

In summary, the number of entries in the state arrays does not affect the throughput much, but the population count and size do. On both of our GPUs, the population size should be at least 512 and the population count 128 to fully exploit the hardware. At these sizes, the Kepler GPU is roughly nine times faster than our two high-end CPUs.

5.3 Parameter-space exploration

Figure 6 illustrates how the throughput on the Kepler and the misprediction ratio of the best 3-bit bimodal FSM depend on the population size, the population count, and the number of entries in the state arrays for the six traces.

Increasing the population size or count greatly improves the throughput but only minimally reduces the misprediction ratio. This is expected as genetic algorithms generally already produce a good solution on a single population. The purpose of the multiple populations (*i.e.*, the random restarts) is to provide variability to escape local maxima. For instance, going from 8 to 1024 populations improves the best FSM by 1.3% to 3.8%, and going from a population size of 32 to a population size of 1024 improves the best FSM by 1.7% to 3.7%.

Since the average number of generations is consistently between 4 and 6.5 in almost all of our experiments (not shown) and the runtime is proportional to the population size and count, the runtime can be drastically reduced by lowering the population count or the population size while only hurting the performance of the best FSM a little.

The throughput drops above 32 entries in the state arrays for the mcf branch outcome trace and especially for the three gcc traces. This is the result of the L1 data cache not being large enough to hold the active state-array elements. Mcf only has a few frequently executed load and store instructions, which is why its prefetchability and cache-way traces do not suffer from a similar drop in throughput.

5.4 Best FSMs

This section visualizes and discusses three of the FSMs that our genetic algorithm generated and compares them with the saturating up/down counter. We plot the states in square boxes and mark them with a ‘P’ in case of a positive (yes, true, ‘1’) prediction and an ‘N’ for a negative (no, false, ‘0’) prediction. The states are numbered for identification purposes only. The P states are completely interchangeable, as are the N states except for N0, which is the initial state.

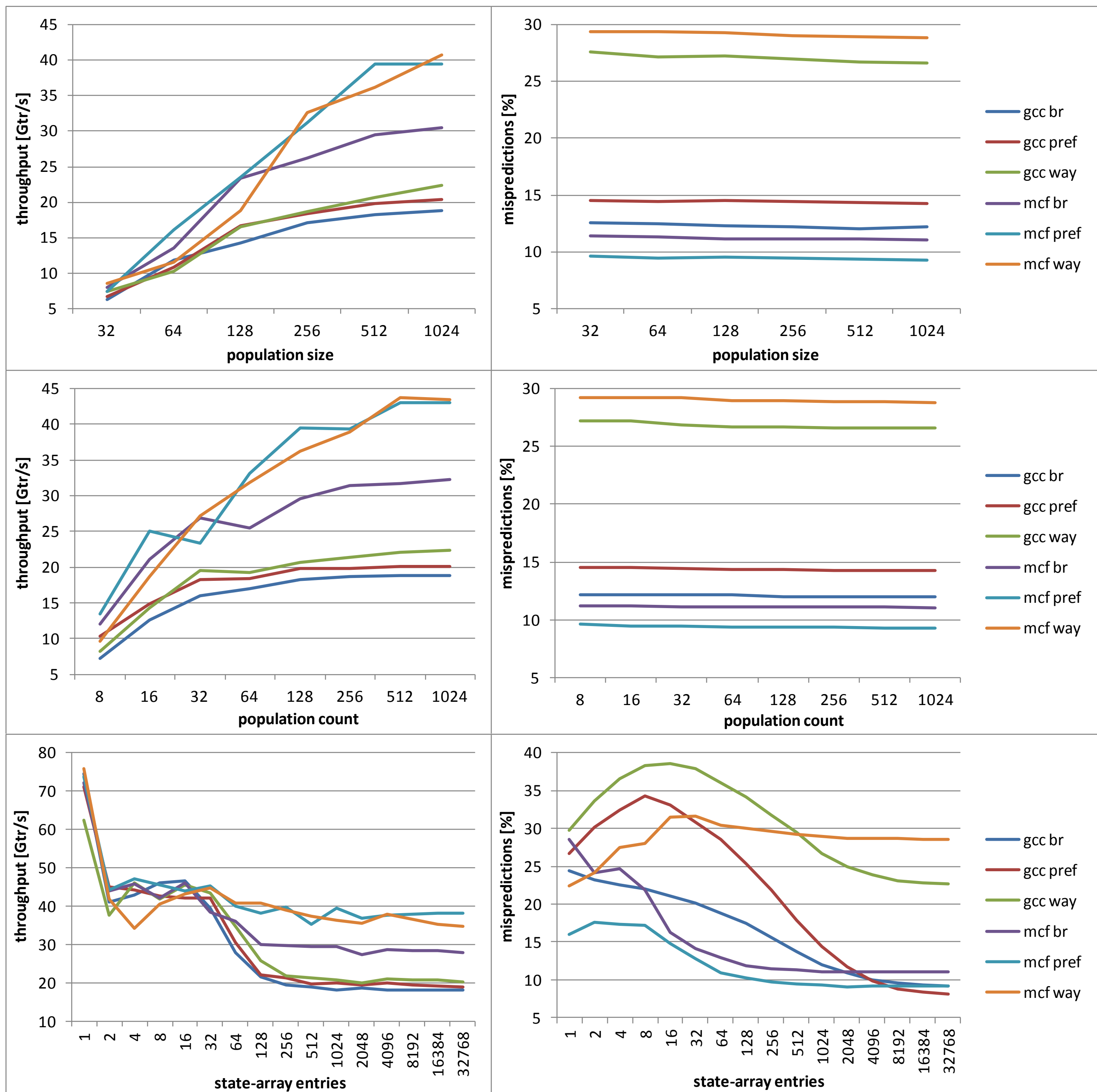

Figure 6. Throughput and misprediction ratio on the six traces as a function of different parameters

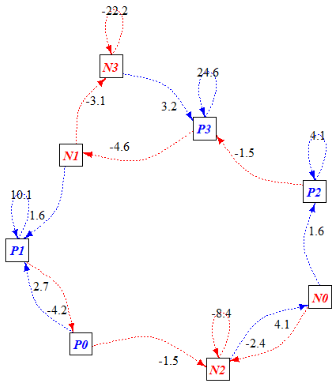

Figure 7. Best 3-bit FSM on the mcf cache-way trace

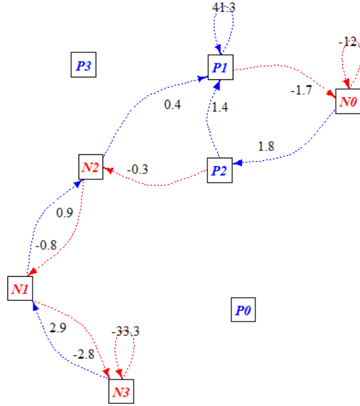

Figure 8. Best 3-bit FSM on the gcc prefetchability trace

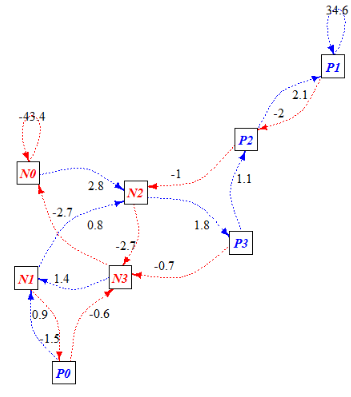

Figure 9. Best 3-bit FSM on the mcf branch-outcome trace

The state transitions are displayed by arrows labeled with the frequency of occurrence as a percentage of the total number of transitions. Negative labels indicate transitions on a '0' input and positive labels on a '1' input. Thus, all states have two outgoing edges.

The first FSM we want to discuss is the best 3-bit FSM for the mcf cache-way trace. This FSM is peculiar because it outperforms all other FSMs with just one entry in the state array. On every other trace, the largest state array yields the best results. Figure 7 shows how this FSM operates.

The most prominent feature of the FSM is its big cycle. This is a fundamental difference from a saturating up/down counter, which has a linear structure that looks like a doubly-linked list with loops (self edges) at the head and tail. The big cycle has several segments that are directional, *i.e.*, they have to be traversed in a specific direction (counterclockwise in the figure).

Quite a few states have loops that allow the FSM to stay in the same state. The two most frequently used states with loops are P3 and N3. The FSM stays in N3 as long as the input is '0' but switches to P3 upon encountering a '1'. However, the reverse is not true. After seeing a '0' in P3, the FSM first transitions to N1 before either going to N3 or P1. If it goes to P1, it has to traverse the entire cycle to get back to P3. N3, P3, and N1 form a small cycle, which is traversed about twice as frequently (3.1%) as the big cycle (1.5%). N1, P0, and N0 are transitional states that always force a transition to a different state.

The '0' transitions out of P1 and P2 are interesting in that they need to be followed by another '0' before the FSM starts predicting zero. Similarly, N2 needs to be followed by two '1' inputs before the FSM predicts one. The other five states provide no such hysteresis and immediately switch the prediction as soon as the opposite bit is seen. This may explain why saturating counters do not perform better on this trace. After all, n-bit counters with $n \geq 2$ always provide some hysteresis when leaving their looping states. However, providing no hysteresis, as the simple last-value predictor does, also does not perform well because the optimal 1-bit FSM (*cf.* Figure 2), which outperforms the last-value predictor, is noticeably worse than the best 3-bit FSM that the GA found. So this combination of some states with and some without hysteresis appears to be important for this hard-to-predict trace.

Moreover, there are several looping states that are followed by two forced transitions before reaching the next looping state, which also seems to be an important characteristic. In fact, the chain P1, P0, and N2 is essentially the inverse of the chain N2, N0, and P2, and even the traversal frequencies are similar. The exceptions are P2 and N3.

Since there is maximal aliasing in this FSM, nothing can be derived from it about the behavior of individual instructions other than that the correlation between instructions is apparently stronger than any self correlation.

The second FSM we studied is the best 3-bit FSM for the gcc stride-prefetchability trace with 32,768 state-array entries. This FSM is interesting because it has the highest accuracy of all the FSMs we tested. Since the optimal 1- and 2-bit FSMs and the 3-bit saturating counter also perform well, this trace appears to be easy to predict. Looking at Figure 8, we observe that the best 3-bit FSM only uses six of the eight possible states. P0 and P3 are never visited, indicating that this trace is, indeed, simple in structure. Also, folding P2 into P1 would essentially yield a saturating counter that only provides hysteresis after seeing many zeros in N3 but not after encountering many ones in P1 or many zeros in N0. Thus, this FSM has two looping zero states, one with (N3) and the other without (N0) hysteresis.

The third FSM we investigated is the best 3-bit FSM for the mcf branch-outcome trace with 32,768 state-array entries. It is noteworthy in that it barely performs better than the 3-bit counter and the optimal 2-bit FSM, but much better than the optimal 1-bit FSM. Figure 9 displays its operation.

Aside from the two very frequent looping states N0 and P1, this FSM is quite strange. However, given that the odd parts are infrequently traversed and that the FSM does not perform much better than the best 2-bit FSM, we surmise that the oddities are not particularly important. Nevertheless, there are a few interesting observations. It takes four transitions to go from one to the other looping state, which

is one more than in a 2-bit saturating counter. Both looping states provide a hysteresis of one state just like the 2-bit counter does. However, N3 also provides a hysteresis on a one input and, very strangely, N1 provides a hysteresis on either input. In other words, when a '0' is seen in N1, the FSM predicts a one next and if a '1' is seen, it predicts a zero next. Clearly, the hystereses are important and explain why the optimal 1-bit FSM, which cannot provide any hysteresis, does not perform well.

6. Summary and conclusions

This paper describes a multi-start genetic algorithm for the synthesis of well-performing bimodal FSMs for designing hardware predictors. The implementation of this algorithm is GPU friendly in that it avoids potential performance bottlenecks and exploits the GPU's capabilities well.

It takes about a dozen cycles per GPU core to evaluate a state transition, *i.e.*, to make a prediction, check its correctness, and update the FSM's state based on the true outcome. On a GTX 680, our code assesses up to 73 billion state transitions per second. On our six traces with tens to hundreds of millions of entries, it takes just seconds to generate FSMs that outperform the saturating up/down counter, a widely-used FSM, in many cases by a large margin. Compared to OpenMP code running on a high-end hex-core Xeon X5690 with hyper-threading, the GPU code is 14 to 18 times faster.

We conclude that GPU acceleration is very useful in this domain and that our implementation exploits the GPU hardware well. Moreover, studying the resulting FSMs can provide insight into the structure of the traces, *i.e.*, the nature of the events being predicted, that explains why saturating up/down counters sometimes do not perform well.

7. Acknowledgments

This work was supported by NSF grants 1141022 and 1217231 and as well as donations from Nvidia Corporation.

8. References

[1] Aarts, E. and Korst, J. 1988. *Simulated annealing and boltzmann machines*. New York, NY; John Wiley and Sons.

[2] Bellas, N. *et al.* 1999. Using dynamic cache management techniques to reduce energy in a high-performance processor. *Low Power Electronics and Design, 1999. Proceedings. 1999 International Symposium on* (1999), 64–69.

[3] Dorigo, M. *et al.* 1996. Ant system: optimization by a colony of cooperating agents. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*. 26, 1 (1996), 29–41.

[4] Emer, J. and Gloy, N. 1997. A language for describing predictors and its application to automatic synthesis. *24th Annual International Symposium on Computer Architecture* (1997), 304–314.

[5] Feo, T.A. and Resende, M.G.C. 1995. Greedy randomized adaptive search procedures. *Journal of global optimization*. 6, 2 (1995), 109–133.

[6] Fogel, L. *et al.* 1966. *Artificial Intelligence through Simulated Evolution*. John Wiley.

[7] Fogel, L.J. *et al.* 1995. Approach to Self-Adaptation on Finite State Machines. *Evolutionary Programming IV: Proceedings of the Fourth Annual Conference on Evolutionary Programming* (1995), 355.

[8] Goldberg, D.E. 1989. Genetic algorithms in search, optimization, and machine learning. *Addison Wesley*. (1989).

[9] Grunwald, D. *et al.* 1998. Confidence estimation for speculation control. *25th Annual International Symposium on Computer Architecture* (1998), 122–131.

[10] Holland, J.H. 1975. Adaptation in natural and artificial systems, University of Michigan press. *Ann Arbor, MI*. 1, 97 (1975), 5.

[11] Jackson, S.J. and Burtscher, M. 2006. Self-optimizing Finite State Machines for Confidence Estimators. *Workshop on Introspective Architecture*. (2006).

[12] Langdon, W.B. 2011. Graphics processing units and genetic programming: An overview. *Soft Computing-A Fusion of Foundations, Methodologies and Applications*. 15, 8 (2011), 1657–1669.

[13] Lee, C.C. et al. 1997. The bi-mode branch predictor. *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on* (1997), 4–13.

[14] Loh, G.H. and Henry, D.S. 2002. Predicting conditional branches with fusion-based hybrid predictors. *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on* (2002), 165–176.

[15] McFarling, S. 1993. *Combining branch predictors*. Technical Report TN-36, Digital Western Research Laboratory.

[16] Milenkovic, A. *et al.* 2011. Caches and predictors for real-time, unobtrusive, and cost-effective program tracing in embedded systems. *Computers, IEEE Transactions on*. 60, 7 (2011), 992–1005.

[17] Peress, Y. *et al.* Re-Defining the Tournament Predictor for Embedded Systems. *Workshop on Optimizations for DSP and Embedded Systems* (2010), 53–61.

[18] Sherwood, T. and Calder, B. 2001. Automated design of finite state machine predictors for customized processors. *Computer Architecture. Proceedings. 28th Annual International Symposium on* (2001), 86–97.

[19] Yeh, T.-Y. and Patt, Y.N. 1993. A comparison of dynamic branch predictors that use two levels of branch history. *Proceedings of the 20th annual international symposium on computer architecture* (1993), 257–266.

[20] Yoaz, A. *et al.* 1999. Speculation techniques for improving load related instruction scheduling. *Computer Architecture, 1999. Proceedings of the 26th International Symposium on* (1999), 42–53.