ASC: Automatically Scalable Computation

Abstract

We present an architecture designed to transparently and automatically scale the performance of sequential programs as a function of the hardware resources available. The architecture is predicated on a model of computation that views program execution as a walk through the enormous state space composed of the memory and registers of a single-threaded processor. Each instruction execution in this model moves the system from one state to a deterministically-determined subsequent state. We can parallelize such execution by predictively partitioning the complete path and speculatively executing each partition in parallel. Accurately partitioning the path is a challenging prediction problem. We have implemented our system in an architectural simulator of an x86 processor with a collection of state predictors and a mechanism for speculatively executing threads that explore potential states along the execution path. We demonstrate that we can achieve a factor of 256 speedup on 1024 cores while running an unmodified sequential program.

1. Introduction

The Automatically Scalable Computation (ASC) architecture is designed to meet two goals: it is straightforward to program and it automatically speeds up execution according to available physical resources. We define “straightforward to program” as requiring only that the programmer writes sequential code that compiles to a single-threaded binary program. The second goal requires that performance improves as a function of the number of cores and amount of memory available.

We begin with a computational model that views the data and hardware available to a program as comprising an exponentially large state space. This space is composed of all possible states of the registers and memory of a single-threaded processor. In this model, execution of a single instruction corresponds to a transition between two states in this space, and an entire program execution corresponds to a path or trajectory through this space. Given this model and a system with $N$ processors we would ideally be able to automatically reduce the time to execute a trajectory by a factor of $N$. In theory, this could be achieved by dividing the trajectory into $N$ equal partitions and executing each of them in parallel. Of course, we do not know the precise trajectory a program will follow, so we do not know, a priori, the precise points on the trajectory that will equally partition it. Nevertheless, if we attempt to predict $N - 1$ points on the trajectory and speculatively execute the trajectory segments starting at those points, we will produce a speedup if even a small subset of our predictions are accurate. From this vantage point, accurately predicting points on the future trajectory of the system suggests a methodology for automatically scaling sequential execution.

The primary design challenge in realizing this architecture is accurately predicting points that partition a trajectory. We break this challenge into two parts: (1) recognizing states from which accurate prediction is possible and will result in useful speedup, and (2) predicting future states of the system when the current state of execution is recognized as one from which prediction is possible.

Given solutions for these two challenges, a basic ASC architecture works as follows. While sequentially executing on one core, ASC allocates additional cores for predictive execution. Each predictive execution core begins executing at a different predicted state and explores a given length of time. We then store the results of predictive execution in a state cache: for example, as compressed pairs of start and end states. At appropriate times, the sequential execution core consults the state cache. If its current state matches a cached state on all relevant coordinates, it achieves speedup by “fast-forwarding” to the associated cached end state and then resumes execution. If the predicted states correctly and evenly partition the execution trajectory and the ASC components operate efficiently, we will achieve perfect and automatic linear speedup of the sequential execution. Thus, our architecture has the potential to produce arbitrary scalability, but its true efficacy will be a function of an implementation’s ability to (1) recognize good points from which to make predictions, (2) make accurate predictions, (3) efficiently query the cache, (4) efficiently allocate cores to the various parts of the architecture, and (5) execute trajectory-based computation efficiently.

The primary challenges to efficiently implementing ASC are (1) manipulating the potentially enormous state space and (2) managing the cache of state pairs. Although the entire state space of a program is sufficiently large (e.g., $10^7$ bits for one of our benchmark programs) that efficient manipulation seems intractable, we exploit the fact that predictable units of computation (think "functions or loops") often touch only a tiny fraction of that space (e.g., $10^3$ bits). Thus, we encode cache entries using a sparse representation that is itself compressible. In addition, a single cache entry can be used at multiple points in a program’s execution, effecting a generalized form of memoization.

We present an implementation of ASC, the Learning-based Automatically Scalable Computation (LASC) system. It is a trajectory-based x86 architectural simulator with a fast, adaptive algorithm for recognizing predictable states, a set of online learning algorithms that use observed states to learn predictors for future states, a resource allocator that selects optimal combinations of predictors in an online setting, and a cache that stores compressed representations of speculative executions.

We evaluate the performance of our system on a set of sequential benchmark programs with surprisingly good re-
We begin our description of the ASC architecture with the work, showing how conventional techniques can be cast into the main components of the architecture that support our trajectory-based model of computation and then walk through this work and avenues of future research.

The contributions of this paper are:

- a system architecture that automatically scales performance as a function of the number of cores and/or size of memory available,
- a fast and adaptive method for automatically identifying states from which predictions are tractable,
- a set of general-purpose predictors that learn from observed states,
- a theoretically sound method to adaptively combine predictions, and
- a prototype that demonstrates speedup on certain classes of unmodified x86 sequential binary programs on systems of up to 4096 cores.

The rest of this paper is organized as follows. We begin by presenting the ASC architecture in §2. We then discuss prior work, showing how conventional techniques can be cast into the architecture in §3. In §4 we describe LASC, our learning-based implementation of the ASC architecture. In §5, we present both theoretical and analytical results that demonstrate the potential of the architecture and the strengths and weaknesses of our prototype. In §6, we discuss the implications of this work and avenues of future research.

2. The ASC Architecture

We begin our description of the ASC architecture with the trajectory-based model of computation and then walk through the main components of the architecture that support our goals of being easy to program and yet amenable to automatic speedup.

2.1. Trajectory-based computation

We base our model of computation on the work of Waterland et al. [61]. Consider the memory and registers of a single-threaded processor as comprising a (very large) state space. Program execution then traces out a trajectory through this state space. Ignoring I/O, this execution is memoryless in that it produces a deterministic sequence of states, where each state depends only on the previous state.

Suppose we want to parallelize a program. Given \( N \) cores, we could achieve a speedup of \( N \) by partitioning its trajectory into \( N \) equal segments that we execute in parallel. Unfortunately, such partitioning requires that we are able to accurately predict \( N - 1 \) specific future states of the machine. If we were able to perfectly predict these states, we would simply predict the end state and ignore the computation entirely. If trajectories are random sequences of states through this state space then prediction would be intractable. However, trajectories are not random and for many programs are highly structured. Figure 1 gives an abstract depiction in a one-dimensional state space of the spectrum of predictability we might observe over program trajectories. Our experience to date suggests that few programs behave randomly as in the leftmost image, and that many real-world programs behave like the somewhat regular trajectory in the middle image.

Figure 1: Abstract depiction of the prediction difficulty spectrum. Within each box, a program’s trajectory traces out a path through state space as it executes. The program’s location in state space is abstractly represented by a 1-dimensional projection on the vertical axis as a function of time on the horizontal axis. The spectrum ranges from extremely hard to predict pseudo-random programs on the far left all the way to trivial loop-based programs on the far right. Real-world programs lie in between.
we discuss the various components, we illustrate some possible values produced by instructions tend to repeat from a small set of values [18], and dependencies [41]. It has also been shown that values produced by instructions tend to repeat from a small set of values [34] and that instructions often have the same input and output values [55].

These repetitive or predictable aspects of execution are particularly amenable to the modeling approach of trajectory-based computation, a conceptual framework that illuminates unexpected geometric and probabilistic structure in systems by focusing attention on their state vectors, which contain all information necessary to deterministically transition to the next state of the system. The ASC architecture is designed to exploit such predictability or repetition.

2.2. Architectural Components

We present the architecture by walking through Figure 2. As we discuss the various components, we illustrate some possible design decisions that map existing work into this architecture.

A single thread, called the main thread, begins program execution (A). While the program runs, recognizers (B) strategically identify states along a program’s trajectory that are useful for predicting future states. Intuitively, states are easier to predict when they follow a recognizable pattern or are drawn from a recognizable distribution. For example, a recognizer could use static analysis to determine a condition that, when satisfied by a state, indicates that the program is at the top of a loop or is entering a function that is called repeatedly. A different recognizer could use semantic-free compressive sensing techniques to dynamically identify states that are close to previously seen states under some distance metric. As we will explain in §4.3, the default recognizer in our prototype identifies regular, widely-spaced points that form a sequence of ‘supersteps’ for which the predictors demonstrate the ability to make accurate predictions.

A set of online learning algorithms train predictors (C), which build models for strategic points along the trajectory. Individual models may predict the whole state, single bits or larger features such as bytes, words, or combinations of words. Different predictors may produce competing models of any flavor: deterministic, stochastic, probabilistic, heuristic, etc. Regardless of the type of model, each predictor must accept an input state and generate a prediction for a future state. It is reasonable, but optional, for the recognizer to use feedback from the predictors to identify characteristics of ‘predictable’ states, and our recognizer uses such feedback.

The allocator (D) is responsible for allocating and scheduling hardware resources for the predictors, recognizers and speculative execution threads (E). It determines how many threads to schedule for each component and how long to spend on each task. Using information from the recognizer, the allocator decides when to ask the predictors for their estimates for future states along with their uncertainty. The allocator then attempts to maximize its expected utility as it decides which predicted states to dispatch to speculative execution threads and how long each thread should execute for each prediction.

The speculative execution threads then enter the results of their executions into the trajectory cache (F), which maintains pairs of start/end states that correspond to execution of a specific number of instructions. The main thread can query the cache, and if it ever matches a start state in the cache, it fast-forwards to the farthest corresponding end state, thus speeding up execution by jumping forward in state space and execution time. Like the allocator, the main thread uses information from the recognizer to decide when to query the cache.

Any implementation of the ASC architecture must maintain correctness of program execution. Speeding up execution only when there is a strict match on the entire state would lead to a conceptually straightforward but suboptimal cache design in which cached trajectories are simply represented as key-value pairs that map complete start states to complete end states. A much better cache design is possible, as we explain in §4.2, by keeping track of the bits read from and written to each state vector during execution, which allows for speeding up execution any time there is a match on the subset of the state relevant to that execution.

2.3. Discussion

There are many different ways to think about the ASC architecture. For example, when the strategic points picked by the
recognizer correspond to function boundaries and the speculative execution threads cache the results of function call execution, ASC is “speculatively memoizing” function calls. ASC memoization is more general than conventional memoization [40], because it can memoize any repeated section of computation, not just function calls.

ASC exploits the same patterns as a parallelizing compiler when it identifies states corresponding to the top of a loop, speculatively executes many iterations of the loop in parallel, then stores the resulting state pairs in its cache. ASC parallelization of loop execution is more general than conventional compiler loop parallelization, because it can speculatively execute in parallel the iterations of any loop whose dependencies have a learnable pattern, including loops with significant data and control dependencies, rather than just loops that static analysis can prove to have no dependencies.

The ASC architecture is a general model that speeds up unmodified sequential programs compiled with a standard toolchain. It can scale in two ways: (1) by adding more memory so that more cache entries can be stored, and (2) by adding more cores for prediction, speculation and cache lookup. In §4, we present details of our prototype implementation of the ASC architecture.

3. Related Work

As mentioned above, ASC employs mechanisms that sometimes behave like conventional techniques for speeding up execution. It is therefore instructive to examine the breadth of conventional techniques to see how they might map to an ASC architecture, thus demonstrating that ASC is complementary to rather than competitive with conventional techniques. We find the fact that the ASC architecture can express existing state-of-the-art techniques as well as enable new approaches to parallelization (e.g., learning patterns that have not yet been coded into parallelization systems) one of its most appealing attributes.

There are three broad categories of work that share our goal of automatically scaling program execution: parallelizing compilers, software systems that parallelize binary programs, and hardware parallelization. Although each category of work started out with conceptual models rather different from ours, notions of statistical prediction and speculative execution have independently arisen in all three.

3.1. Compiler Parallelization

Traditional compiler parallelization based on static analysis [1] has produced sophisticated research compilers [3, 8]. Although compilers using these techniques can automatically parallelize most loops that have regular, well-defined data access patterns [28], the limitations of static analysis have become apparent [23]. When dealing with less regular loops, parallelizing compilers either give up or generate both sequential and parallel code that must use runtime failsafe checking [51]. The ASC architecture is able to speed up irregular loops by using online probabilistic inference to predict likely future states, as we show in §5. However, it can also import the sophisticated static analyses of traditional parallelizing compilers in the form of probability priors on loops that the compiler was almost but not quite able to prove independent.

Thread-Level Speculation (TLS) [13, 56, 57] arose in response to the limits of compile-time static analysis. TLS hardware aggressively applies speculative execution to code that cannot be fully parallelized by the compiler. The virtue of this hardware is that it enables automatic speculative parallelization by TLS compilers [27, 36, 39, 45, 48, 58, 65], which do not need to fully prove the absence of dependences across the threaded code they generate. However, TLS performance sensitively depends upon the compiler making good choices for speculative thread code generation and spawning. The ASC architecture can exploit TLS hardware or Transactional Memory [22, 30] if it is available and makes it easy to experiment with decompositions of execution flow. Compiler choices that yield good decompositions for TLS are also likely to produce recognizable and predictable patterns for ASC, and vice versa.

Clusters of computers have been targets of recent speculative parallelizing compilers that have demonstrated scaling on up to hundreds of nodes for loops without loop-carried dependencies [29]. Our LASC prototype implementation currently runs on clusters and would benefit from importing hints produced by these compilers in the form of probability distributions.

3.2. Binary Parallelization

Software systems that automatically parallelize binary programs also have a long research history. They share with ASC the properties of not requiring the program source code, of having perfect visibility into patterns that arise at runtime, and of operating with the cost of some constant runtime overhead.

Binary rewriter parallelization systems [31, 60, 63] take as input a sequential binary executable program and produce as output a parallel binary executable. Dynamic code generating binary parallelization systems [12, 23] assume the existence of TLS hardware and apply the same control flow graph analyses used by conventional TLS compilers to sequential binary programs not originally compiled for TLS. Dynamic binary parallelization systems [62], inspired by dynamic binary optimization systems [5], transparently parallelize sequential binary programs by identifying hot traces of instructions that can sometimes be replaced with a shorter, speculatively executed instruction sequence whose semantics are equivalent on the current input dependencies.

The ASC architecture contrasts with these systems in that it does not itself attempt any analysis of instruction control flow semantics or basic block optimizations. It simply models execution as the time evolution of a stochastic process through a state space. This means that it will exploit any method that can produce probabilistic predictions for the relevant bits of future state vectors. Most instruction control flow analysis and optimization techniques can be transformed into probabilistic predictors via a technique from machine learning called “the kernel trick” [6, 17]. This means that ASC can import
existing work in binary parallelization as “run ahead” approximators [66] that take a completely consistent probabilistic form as predictors for future states whose uncertainty can be reasoned about and managed with decision theory [37].

3.3. Hardware Parallelization

Hardware that transparently speeds up sequential binary programs as a function of transistor count is the subject of intense research and has resulted in many successful commercial implementations. Superscalar processors [46] execute individual instructions in parallel, speculatively executing around dependencies using powerful branch prediction [26, 53, 54] and load value prediction [34] strategies and can use trace caches [50] to implicitly effect multiple simultaneous branch predictions. Other approaches make multiple cores appear as one fast out-of-order core [9, 24], shorten execution time by speculatively removing nonessential instructions in parallel [44], and speculatively parallelize programs at the thread [2, 13, 16, 20, 56, 64] or SIMD [14] level, many of which rely on some degree of compiler or runtime [15] assistance.

ASC makes the same contract to the programmer as a superscalar or dynamic multithreading processor: transparent, automatic speedup of sequential binary programs as a function of transistor count. ASC prediction is more general than branch, load value, and return value prediction because it models the time evolution of the complete state vector and exploits the fact that correlations between arbitrary bits can give rise to a low-entropy joint distribution over points in state space.

Although LASC presently exists only as an architectural simulator, our long term goal is that every component is amenable to an efficient implementation in hardware. For example, a trace cache could be extended to store the result of executing a trace of instructions, using ternary content-addressable memory to match on the subset of relevant coordinates. We have designed LASC to take advantage of recent advances in neuromorphic [38] and probabilistic computing [19, 59] to efficiently implement our learning algorithms in hardware.

4. Implementation

LASC, the learning-based ASC, is an implementation of the ASC architecture that transforms the problem of maximizing speedup into a machine learning problem. We begin with an overview of our implementation and then discuss the inner workings of each component in the following subsections.

In LASC, the cache, recognizer, predictors, and allocator are all built into a trajectory-based architectural simulator (TBAS) that we discuss in §4.1. The core of the TBAS is a transition function that interprets its input as an x86 state vector, simulates one instruction, and outputs a new state vector. The main and speculative threads execute by repeated calls to this transition function.

Each time the TBAS executes an instruction on the main thread, it invokes the recognizer to rapidly decide whether the resultant state matches a pattern that the recognizer has identified. If the current state matches the recognizer’s pattern, it sends the current state to the predictors and queries the distributed cache: fast-forwarding on a cache hit. Meanwhile, the predictors update their models and predict future states based on the current state. The allocator then takes all the predictions, combines them, and selects a set of states on which to launch speculative threads.

By factoring the problem of predicting the entire state into the problem of predicting smaller, conditionally independent portions of the state, we parallelize the work of the predictors across cores. Specifically, this parallelizes both training, the task of learning a predictive model for future states, and prediction itself, the task of using a learned model to materialize a predicted state.

4.1. Trajectory-Based Architectural Simulator

Our simulator’s key data structure is the state vector, representing the full state of computation. Execution of an instruction occurs via a transition function:

```
transition(uint8_t *x, void *g, uint32_t n)
```

It may seem a poor choice to represent the state as a bit vector, but this representation allows us to learn and make predictions with massively bit-parallel binary classifiers.

The transition function accumulates dependency information at byte (not bit) granularity. For each byte in the state vector, the dependency structure $g$ maintains one of four statuses: read, written_after_read, written or null. When speculation begins, all statuses are set to null. During speculation, when we read a byte whose status is null, we update the status to read. If we later write that same byte, we update its status to written_after_read. If we write a byte that has never been read, we update its status to written.

Without this dependency tracking, we could exploit speculative trajectories only when the current state of execution matched a cached start state in its entirety. However, the dependency state lets us match significantly more frequently. When we begin a speculative thread, we initialize the dependency state to all null values. When we stop a speculative thread, the set of non-null bytes in the dependency state identifies precisely those bytes on which the speculative computation depends. Therefore, a cached entry produces a hit when the current execution state matches the start state on only those bytes with statuses of read or written_after_read. Not only does this improve the hit rate of the cache, but it makes the predictors’ jobs easier too: they need to correctly predict only those same bytes. When we find a cache hit, the main thread fast-forwards to the end state of the cache entry by updating only those bytes with statuses written or written_after_read. Intuitively, this uses “translation” to match on trajectories that are “parallel” in state space.
The recognizer identifies strategic points corresponding to function-level semantics. ASC cache uses dependency information to support such approximate matching. We implement the necessary dependency tracking as an integral part of program compilation and architecture as well as state-of-the-art machine learning algorithms. Our implementation of the ASC architecture must maintain appropriate, as we show schematically in the rightmost panel of Figure 4.5.

We use the following parallel algorithm to find a good IP value. The allocator dispatches all the available cores to particular IP values that have been observed but not yet rejected. Each core initializes a private copy of our learning algorithms and executes the program from its current state. When it encounters a state with its assigned IP value, it sends that state vector to the predictors. The (local) allocator integrates all the predictions to produce a predicted state, which it caches (locally). As the core continues execution, it checks for matches against its growing local cache of predictions. When it finds a match, it records the number of instructions between the state from which a prediction was made and the predicted state, which is a proxy for the utility of the speculative execution that would result if the prediction had been used.

Some IP values are easily predicted but are so closely spaced on the trajectory that speculative execution from them is not worth the communication and lookup costs. Other IP values are widely spaced on the trajectory but very hard to predict. The recognizers select the set of IPs whose resulting states were the best predicted relative to the other IP values considered. We call these the set of recognized IPs (RIP) and the lengths of the corresponding trajectories supersteps. After selecting one or more RIPs, workers are allocated to the various RIPs, and their predictors begin making predictions for future states with that same RIP. For the purposes of exposition, we will assume only a single RIP, but everything applies when we have multiple RIPs on which we are speculating.

We investigated many heuristics for finding good IP values before settling on this method. In retrospect, this method should have been obvious as it is biased towards the precise criterion of interest: how well the predictors can predict future states. In our prototype, speculative executions need to be at least $10^4$ instructions for their benefit to outweigh their cost, so we ignore predictions, regardless of accuracy, if they are not sufficiently far in the future.

### 4.2. Cache

We further use dependency tracking to represent cached state pairs efficiently. Each cache entry contains a representation of its start state and end state. The start state represents only those bytes with read or written_after_read statuses and the end state represents only those bytes with write or written_after_read statuses. We store a sparse representation of the relevant byte indices and their corresponding values.

A portion of the cache exists on each core participating in a computation, because we implement the cache directly in our simulator. Each core that generates a speculative execution stores that execution in its portion of the cache. The main thread queries this parallel distributed cache, at intervals indicated by the recognizer, by broadcasting its current state vector, either as a binary delta against its last query or as the full state vector, depending on the computation/communication trade-off. Each node responds with an integer value indicating the length of the longest matching trajectory it holds or 0 in the case of a cache miss. Finding the largest integer, and thus the most useful matching trajectory in the whole cache, is a reduction, so we use MPI’s parallel binary tree max operator to limit bandwidth consumption. On our Blue Gene/P system, each pairwise max comparison is implemented in an ASIC, further speeding up the reduction. The main thread then does a point-to-point communication with the node that sent the largest integer to obtain the corresponding end state to which it will fast-forward, while all other nodes go back to running learning algorithms and doing speculative execution.

### 4.3. Recognizer

The recognizer’s job is to identify states in the trajectory from which prediction is both tractable and useful. This requires finding states for which the speculative execution from predicted states will produce few non-null bytes in the dependency vector and the values of the corresponding bytes in the state vector are predictable. In other words, states for which resultant speculative computation depends on only a small number of predictable values.

We find these states by exploiting the geometric and probabilistic structure of our state space. In particular, we find a hyperplane that cuts the execution trajectory at regular, widely-spaced intervals, as depicted in Figure 3. Our default recognizer induces such a hyperplane by picking only states that share a particular instruction pointer (IP) value. Thus, given a sequence of state vectors corresponding to the same IP, the predictors try to learn the relevant parts of the state vector that will occur at future instances of this IP value. Fortunately, long-running programs tend to be composed of loop structures and functions that correspond to repeated sequences of instructions.

We use the following parallel algorithm to find a good IP value. The allocator dispatches all the available cores to particular IP values that have been observed but not yet rejected. Each core initializes a private copy of our learning algorithms and executes the program from its current state. When it encounters a state with its assigned IP value, it sends that state vector to the predictors. The (local) allocator integrates all the predictions to produce a predicted state, which it caches (locally). As the core continues execution, it checks for matches against its growing local cache of predictions. When it finds a match, it records the number of instructions between the state from which a prediction was made and the predicted state, which is a proxy for the utility of the speculative execution that would result if the prediction had been used.

Some IP values are easily predicted but are so closely spaced on the trajectory that speculative execution from them is not worth the communication and lookup costs. Other IP values are widely spaced on the trajectory but very hard to predict. The recognizers select the set of IPs whose resulting states were the best predicted relative to the other IP values considered. We call these the set of recognized IPs (RIP) and the lengths of the corresponding trajectories supersteps. After selecting one or more RIPs, workers are allocated to the various RIPs, and their predictors begin making predictions for future states with that same RIP. For the purposes of exposition, we will assume only a single RIP, but everything applies when we have multiple RIPs on which we are speculating.

We investigated many heuristics for finding good IP values before settling on this method. In retrospect, this method should have been obvious as it is biased towards the precise criterion of interest: how well the predictors can predict future states. In our prototype, speculative executions need to be at least $10^4$ instructions for their benefit to outweigh their cost, so we ignore predictions, regardless of accuracy, if they are not sufficiently far in the future.
4.4. Predictors

Our default implementation invokes the predictors only when the current state has the recognized IP. Thus, given an example sequence of state vectors all having the same IP, our predictors try to predict future state vectors that will also have that IP. This is an online learning problem [7], in which we wish to learn from the sequence of examples a probability distribution \( p(\hat{x}' \mid x, \theta) \) that represents our belief that \( \hat{x}' \) will be the next state vector that will have the same IP as the current state vector \( x \).

By definition, each state vector contains all the information necessary to compute future states. We decompose the prediction problem into a set of conditionally independent predictors, where each predictor conditions on the state vector \( x \) to model a particular bit, a 32-bit quantity, or other feature of the next observation \( \hat{x}' \). The case of bits is particularly interesting, because it reduces to the binary classification problem, one of the best developed areas of machine learning [49]. Decomposition in general also allows us to parallelize both learning (model training) and prediction (materializing predicted states).

We exploit the bit-level conditional independence decomposition to formulate our prediction problem probabilistically, in light of this viewpoint’s practical [6, 49] and theoretical [25] advantages. When the current state \( x \) has the RIP value, \( p(\hat{x}' \mid x, \theta) \) expresses our belief that \( \hat{x}' \) is the next state with that IP value according to our model with parameters \( \theta \). Since each state is simply a binary vector of \( n \) bits, we factor the joint predictive distribution for \( \hat{x}' \mid x \) into a product of per-bit predictive distributions

\[
p(\hat{x}' \mid x, \theta) = \prod_{j=1}^{n} p(\hat{x}'_j \mid x, \theta) \quad (1)
\]

\[
= \prod_{j=1}^{n} \text{Bernoulli}(\hat{x}'_j \mid \theta_j(x)) \quad (2)
\]

where \( \hat{x}'_j \) is the \( j \)-th bit of the predicted state \( \hat{x}' \). This factoring is a convenient computational choice that means we learn \( n \) separate binary classifiers \( \theta_j(\cdot) \), which is straightforward to parallelize. The \( j \)-th term in Eq. 2 is the probability that \( \hat{x}'_j = 1 \), conditioned on a model \( \theta_j(\cdot) \) that takes a state \( x \) as input. Under this model, the probability of a predicted state \( \hat{x}' \) is the product of these \( n \) terms.

We use Eq. 2 to make allocation decisions, as at any moment it encapsulates everything our learning algorithms have currently discovered, even when the learners themselves are not probabilistic. The job of the allocator is to generate a pool of predictions and then decide which ones to send to speculative threads. Given a state \( x \) and model \( \theta \), there are two straightforward methods for generating predictions from Eq. 2. The most probable prediction, \( \hat{x}_{\text{ML}} \), is produced by maximizing each term, i.e., setting each bit to its most probable value. Alternate predictions for \( \hat{x}' \) can be generated, for example, by strategically flipping the most uncertain bits of \( \hat{x}_{\text{ML}} \) to give the second and third most likely predictions, and so on. Further, these predictions can be used as input to Eq. 2 to generate predictions farther along the trajectory; this process can be iterated. Eq. 2 gives the probability of each of these predictions under the model, providing a direct way to compare them. As we describe in §4.5, this allows us to use expected utility maximization to decide what predictions to use for speculation.

In practice, we learn binary classifiers only for the bits that have ever changed from one instance of the RIP to the next; we call these bits excited, and they typically correspond to only a tiny fraction of the entire vector. This provides a strong form of sparsity, as we need learn only those bits that change across states with the same IP value, not all bits that ever change. Some of the programs we evaluate in §5 have a state space dimensionality of \( n > 10^5 \), of which \( > 10^5 \) bits change over the lifetime of the program, but of which \( < 300 \) change between observations of the RIP.

4.4.1. Interfaces Each predictor implements three functions: \texttt{update} \((x, j)\), \texttt{predict} \((x, j)\) and \texttt{reset}(). For each bit \( j \) known to be non-constant between occurrences of the RIP, the main thread calls \texttt{update} \((x, j)\). Each predictor then uses the difference between its previous prediction, \( \hat{x}_j \), for the \( j \)-th bit of \( x \) and the newly observed actual value \( x_j \) of that bit to update its internal model parameters [21].

After the predictors have updated their models, the main thread asks for predictions by calling \texttt{predict} \((x, j)\) for each non-constant bit \( j \). The predictors then issue predictions for bit \( j \) at the next instance of the recognized IP value. These per-bit predictions are mixed and matched, as will be described in §4.5.1, weighted by each predictor’s past performance on each bit, into the single distribution in Eq. 2. Predictors are free to extract whatever features from \( x \) they wish, but must express predictions at the bit level. Predictors at the feature level share state between related bits to make this efficient.

Our system uses the predictors in several different contexts, using \texttt{reset()} when it needs them to discard their current models and start building new ones. For example, the recognizer calls \texttt{reset} when searching for an initial RIP or when a change in program behavior renders the current RIP useless.

4.4.2. Prediction Algorithms LASC is extensible, so it can support any number of predictors, as long as the predictors implement the interfaces described above. The results in this paper use only four discrete learning algorithms, two trivial ones, mean and weatherman, and two interesting ones, online logistic regression and online linear regression. The mean algorithm just learns the mean value of each bit, and the weatherman predictor predicts that the new value of the bit will be the same as the current value.

Logistic regression is a widely-used learning algorithm for binary classification [21], in which one is given a vector \( x \) that one must classify as either 1 or 0. It assumes that one has a stream of labeled examples \( \{(x, y), (x', y'), \ldots\} \), where each label \( y \) is a 1 or 0. The goal is to correctly predict a new vector \( x'' \) as 1 or 0 before seeing its true label \( y'' \). In our setup, the labels \( y \) are the \( j \)-th bit of the next state vector \( x' \) given the current state vector \( x \). Logistic regression defines a family of functions, each parameterized by a weight vector \( w \) of dimension \( n + 1 \). Online learning for logistic regression involves
updating the current weights each time a new observation is introduced. Our implementation of online logistic regression is the standard gradient descent approach, where we use only those bits that have been observed to change.

Linear regression is a widely-used learning algorithm used to fit a curve to real-valued data [21]. It takes as input a stream of examples \( \{(x, y), (x', y'), \ldots\} \), where each label \( y \) is a real number. The goal is to correctly predict the \( y' \) associated with each new vector \( x' \) before seeing the true answer. In our setup, the labels \( y = \phi_i(x) \) are produced by interpreting the \( i \)-th 32-bit word of the state vector \( x \) as an integer. Like logistic regression, linear regression defines a family of functions, each parameterized by a weight vector \( w \) of dimension \( K + 1 \), and online learning involves updating the current weights with each new observation. Linear regression tends to be useful when we need to predict features such as loop induction variables, while logistic regression is more general and attempts to predict any bit whatsoever. We run multiple instances of each predictor, using different weight parameters and then combine the predictions using a randomized weighted majority algorithm, discussed in the next section.

4.5. Allocator

The allocator is responsible for combining multiple predictions into a set of predictions to be speculatively executed.

4.5.1. Combining multiple predictions We use a technique known as ‘predicting from expert advice’ [7]. This approach makes no assumptions about the quality or independence of individual predictors or ‘experts’. Some predictors are better at predicting certain bits of our state vectors than others, so we want to mix and match predictors at the bit level, even if some of the predictors operate at a higher semantic level. Predicting from expert advice gives us a theoretically sound way to combine wildly-different predictors. Using this approach, the goal is to minimize regret: the amount by which the predictive accuracy of an ensemble of predictors falls below that of the single best predictor in hindsight [7, 10, 11, 35]. Minimizing a loss metric (e.g., regret) is equivalent to maximizing a utility.

We use the Randomized Weighted Majority Algorithm (RWMA) [35] to minimize regret, because it comes with strong polynomial-time theoretical guarantees that bound regret. These error bounds say, intuitively, that for each bit of the current program, if there exists in our ensemble a good predictor for that particular bit, then after a short time the weighted majority votes for that bit will be nearly as good as if we had used that best predictor. We get this guarantee at the cost of having to keep track of the per-bit error rate of each learning algorithm, since the RWMA algorithm uses these error rates to adjust the weight of each learning algorithm and to calculate its weighted majority prediction.

4.5.2. Scheduling speculative threads The allocator is responsible for combining predictions and then scheduling speculative executions. It picks the states from which to speculate and for how long to run each speculative computation by balancing the payoff of each state against its uncertainty about that state. For each potential speculative execution, the allocator uses Eq. 2 to calculate the expected utility: the length of the cached trajectory times the probability that it will be used by the master. Then it selects and schedules executions to maximize expected speedup.

By combining the per-bit predictions, the allocator produces a single distribution with the form of Eq. 2. Thus, the allocator combines the results of multiple online learning algorithms using a regret minimization framework to form a conditional probability distribution. This distribution is general in the sense that it can take any state as input. This includes unobserved states, and in particular, predictions for states. It allows us to both generate predictions and assign them probabilities that represent the belief they will be correct. The allocator uses this equation to ‘roll out’ predictions for many steps in the future by using predicted states as input to iteratively generate later predictions. Out of the total set of generated predictions, the allocator selects the subset that maximizes the expected utility of speculating from these predictions.

5. Evaluation

ASC is a new architecture strongly motivated by current trends in hardware and, in the case of LASC, demonstrates a way to leverage machine learning techniques for transparent speedup of execution. As such, we have no expectation for our implementation to immediately outperform decades of research on parallelizing compilers and hardware-based speculation. ASC is a promising long-term approach for which we have two goals in our evaluation. First, we want to demonstrate the potential of ASC by showing that we are able to make accurate predictions and that it is possible to use these predictions to produce significant scalability of sequential program binaries. Second, we want to demonstrate that our prototype system achieves some of these benefits in practice, limited only by implementation details that require a combination of engineering and tuning.

We first introduce the three benchmark programs we use and then present data that demonstrates the efficacy of our predictors and method for combining their predictions. Next, we describe the hardware platforms on which we evaluate our architectural simulator and present microbenchmark results to quantify critical aspects of its implementation. Then we present scaling results for both idealized and actual realizations of our implementation.

5.1. Benchmarks

We evaluate three benchmark programs to illustrate different weaknesses and strengths in our system. Each of these programs can be hand-parallelized; this choice allows us to compare ASC’s performance to manual parallelization and is also motivated by the widespread existence of diverse programs that could be hand-parallelized but are not. In our collaborations with computational scientists, we repeatedly encounter situations where the scientific team either lacks the expertise to manually parallelize their programs or have invested significant time in parallelizing an application for one piece of hardware only to discover that porting to a new machine re-
quires essentially rewriting their parallel implementation from scratch. Our experiments demonstrate that we can run the identical binary on a commodity multicores system as well as a massively parallel Blue Gene and obtain attractive speedups on both.

Our first benchmark is the Ising kernel, a pointer-based, computationally-intense condensed matter physics program. It came to our attention because our colleagues in applied physics found that existing parallelizing compilers were unable to produce the kinds of speedups they needed [29]. The program walks a linked list of spin configurations, looking for the element in the list producing the lowest energy state. Computing the energy for each configuration is computationally intensive. Programs such as this one that use dynamic data structures are notoriously difficult to automatically parallelize because of the difficulties of alias analysis in pointer-based code [47]. We demonstrate that by predicting the addresses of linked list elements, LASC effectively parallelizes this kernel.

The second benchmark is the 2mm multiple matrix multiply kernel in Polybench/C, the Polyhedral Benchmark suite [43]. It computes $D = \alpha ABC + \beta D$, where $A, B, C, D$ are square integer matrices and $\alpha, \beta$ are scalars. This is an example of a program amenable to conventional parallelizing compiler techniques. We use it to demonstrate that LASC uses its online learning algorithms to automatically identify the same structure that a compiler would identify, but without requiring language-level semantic analysis.

The third benchmark is the collatz kernel. This program iterates over the positive integers in its outer loop, and in its inner loop performs a notoriously chaotic property test [32]. This property is the conjecture that, starting from a positive integer $n$, then repeatedly dividing by 2 if $n$ is even and multiplying by 3 and adding 1 if $n$ is odd, this sequence will eventually converge to 1. This program is easily parallelized by spawning separate threads to test different values of $n$. LASC identifies that parallelization opportunity in the outer loop, but importantly also automatically memoizes parts of the inner loop that correspond to computing the conjectured convergence property.

5.2. Predictor Accuracy

As our predictions rely on the ability to find recognizable IPs, we first examine the behavior of our system’s superstep recognizer, responsible for finding points from which predictions are tractable. Then, we examine the accuracy of our individual predictors, the overall error rate of the ensemble, and the resulting trajectory cache hit rates.

The recognizer’s job is to find a point in the program, described by the value of the instruction pointer, that occurs repeatedly during execution with sufficiently long intervals between occurrences that we can amortize the cost of a cache lookup. In Table 1 we show its performance on our three benchmarks, by presenting the number of instructions in the full program execution (Total Length), the number of instructions executed before selecting the first RIP (First RIP), and the number of instructions in a typical cache entry, which corresponds to the interval between observed instances of the RIP (Mean Jump). The ratio between Total Length and Mean Jump approximates the inherent scalability of the program, and the First RIP is a lower bound on the sequential portion of a program’s execution. As Table 1 shows, we can often find a good RIP and begin speculating in less than $10^6$ instructions. With the observed ratios between the mean jump and total execution, we can, in theory, automatically scale their performance to hundreds or thousands of cores.

Table 1: Superstep statistics in number of instructions.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total Length</th>
<th>First RIP</th>
<th>Mean Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ising</td>
<td>$2.3 \times 10^{10}$</td>
<td>$1.1 \times 10^7$</td>
<td>$1.2 \times 10^7$</td>
</tr>
<tr>
<td>2mm</td>
<td>$7.5 \times 10^9$</td>
<td>$1.2 \times 10^6$</td>
<td>$1.3 \times 10^7$</td>
</tr>
<tr>
<td>collatz</td>
<td>$2.0 \times 10^{11}$</td>
<td>$1.0 \times 10^5$</td>
<td>$3.8 \times 10^6$</td>
</tr>
</tbody>
</table>

Table 2: Ensemble error rate statistics: lower numbers are better. Column 2 contains the error rates obtained by weighting all predictors equally. Column 3 represents the best possible error rate given the predictors we used, and Column 4 shows the actual error rates achieved by combining the predictors using regret minimization.

Next, in Table 2 we examine the error rate of our ensemble of learning algorithms, in terms of the percentage of full state vector predictions that were incorrect. The data demonstrate that we derive benefit from combining multiple predictors, that it is important to weight the various predictors correctly, and that the seemingly impossible task of predicting future execution states is actually tractable. The $2^{nd}$ column gives a reference error rate that would occur if we weighted each predictor on each bit equally. At the other extreme, the $3^{rd}$ column gives the ideal error rate for the particular set of predictors we used that could have been achieved if we were able to always use the best predictor for each bit. The $4^{th}$ column shows that the regret minimization approach does indeed mix and match the best predictor per bit to produce an overall error rate very close to the idealized results.

The error rates reported in Table 2 are for correctly predicting an entire state vector. In practice, we achieve a higher cache hit rate than this might suggest, because states need only match on dependent bits. As expected, we observe that the cache hit rate improves over time and as a function of the number of speculating threads. For example, although 2mm shows an ensemble prediction accuracy of 89.7%, we observe that with 32 cores the cache hit rate is 96%.
To evaluate the importance of the regret minimization algorithm, Figure 4 shows the final weight matrices for each benchmark. The columns are the bits that changed at least once between occurrences of the RIP. The rows are the four learning algorithms we described in §4.4.2, and the cells of the matrix are shaded by the magnitude of the weight assigned to each predictor for each bit. Both \texttt{collatz} and \texttt{2mm} show a strong preference for the linear regressor, although there are a few bits for which the logistic regressor is essential. We almost removed the mean and weatherman predictors, assuming they would be too simple to provide additional value, but the \texttt{Ising} weight matrix clearly shows that all four algorithms contribute significantly.

5.3. Architectural Simulator

The main goal of the implementation is to permit exploration of LASC. We developed an architectural simulator that implements 79 opcodes of the 32-bit x86 instruction set. It is a parallel MPI program that executes freestanding static binary programs compiled with \texttt{gcc} to use only the opcodes implemented. This produces a simple implementation about which we can easily reason and manually verify, while permitting us to run \texttt{c} programs. The simulator’s restricted functionality is due to the simplified nature of our prototype and is not fundamental to LASC.

We used three experimental testbeds: an x86 server with 32 cores clocked at 1.4 GHz with 6.4 GB of RAM total, an IBM Blue Gene/P supercomputer of which we used up to 16384 cores clocked at 850 MHz each with 512 MB of RAM, and a single-core laptop clocked at 2.4 GHz. The 32-core server runs Linux, while the Blue Gene/P runs the lightweight CNK operating system, and the laptop runs MacOS. All three systems provide an MPI communication infrastructure, but the Blue Gene/P provides ASIC hardware acceleration for MPI.

Table 3 provides the basic performance characteristics of our architectural simulator on one core of our x86 server. Prediction time is a function of how far in the future we are making predictions, so time here is expressed as a function of \( R \), the number of instances of the RIP into the future we are predicting.

We keep cache messages small using a compressed representation based on the Meyers binary differencing algorithm [42]. Messages range in size from about 20 bytes for \texttt{collatz}, whose state vector is only \( n = 3 \times 10^3 \) bits, to about 80 bytes for \texttt{Ising}, whose state vector is \( n = 2 \times 10^5 \) bits, to about 100 bytes for Polybench \texttt{2mm}, whose state vector is \( n = 5 \times 10^7 \) bits.

Having established the basic parameters of our implementation, we can now examine the scaling we achieve on our benchmark set.

5.4. Scaling Results

Figure 5 shows scaling results on the \texttt{Ising} benchmark for both the 32-core server and the Blue Gene. Speedup is the single-threaded wall clock time divided by the parallel time. We show several parallelization results to tease apart the inherent scalability of the program, the limits of our predictor, and the bottlenecks of our implementation. On the 32-core server, the hand-parallelized results show that its possible to achieve perfect scaling by simply iterating over the list, partitioning it into up to 32 separate lists and then computing on each list in parallel. In LASC, the potential speedup is limited by the cache hit rate. With infinitely fast cache lookups, we would obtain the performance illustrated by the “cycle count speedup” line. However, our cache lookup is not infinitely fast and produces the results shown in the LASC lines. The “oracle scaling” illustrates the performance our system could achieve with perfect predictions; this measurement holds everything else constant – including the recognizer and allocator policies as well as the times to compute predictions, speculative trajectories and cache queries – while ensuring that the prediction for any particular state is correct. The fact that the actual and oracle speedup lines overlap demonstrates that we are limited only by implementation artifacts, not predictive accuracy.

There are two different forces at work here. The first is the inherent parallelism of the program, and the second is our implementation. Although the potential energy calculation of \texttt{Ising} involves many deeply nested loops, the outermost pattern is just a linked list walk, which enables the recognizer
Figure 5: Scaling results for Ising benchmark. The ideal scaling line represents linear scaling. The hand-parallelized line shows our results for the obvious manual parallelization. To provide a fair comparison, the hand-parallelized code is executed in our TBAS. The LASC cycle count speedup line represents the best speedup possible given actual cache hit rates, but an infinitely fast trajectory cache. The two remaining LASC lines show the speedup we obtain with oracle predictions and with our actual predictions.

to quickly find a good IP value for speculation; namely, one a few instructions into the prologue of the energy function.

As shown in Table 1, Ising’s superstep accounts for approximately 0.05% of its total instruction count, so the best speedup we can expect is approximately 2000. Unsurprisingly, code inspection reveals that the linked list has exactly 2000 elements. This explains the drop off we see on the Blue Gene/P at 2000 cores.

However, our scaling peaks at roughly 1024 cores, due to the implementation of long-range prediction. When LASC has many cores at its disposal, cores are dispatched to make predictions at increasingly distant instances of the RIP. As explained in §5.3, we iteratively make predictions over future instances of the RIP and do not share this computation across cores. This naive approach limits performance as the time it takes to make predictions grows linearly in the number of cores. Clearly, sharing predictions across the cores should induce a significant performance improvement.

Figure 6 shows the scaling results for 2mm on the 32-core server. Although 2mm also has a regular structure, its scaling is less impressive than for Ising. Once again, there are two different forces at work here. This time, the superstep accounts for slightly less then .2% of the total execution, limiting the potential scalability to 600.

The cycle count line shows the potential speedup possible with our predictors, and this is encouraging. As before, the oracle line suggests that the implementation is not limited by prediction accuracy, but again by simulator artifacts. As with Ising, we are limited by our failure to share state across cores. The effect is more pronounced here, because we have more bits to predict. As discussed in §5.2, we track two orders of magnitude more bits for 2mm than for Ising, so predictions take two orders of magnitude longer. The penalty for failing to reuse prior computations is simply too great in this case, suggesting that future implementations must be more efficient in their long-range predictions. Given the relatively small message size we use to transmit predicted bits, we are optimistic that we can find a better balance between communication and computation.

Lastly we turn to the collatz benchmark. Tables 1 and 2 suggest that collatz has an inherent parallelism of approximately 25,000 and excellent prediction accuracy (98.1%). However, if we look at the program, we see that we test the conjecture for $10^6$ different numbers. The number of instructions
We have presented an architecture and implementation that extracts parallelism automatically from structured programs. Although \texttt{collatz} and \texttt{2mm} have easily parallelized structure, \texttt{Ising} is more interesting, because it uses dynamically allocated structures, which are frequently not amenable to automatic parallelization. Our learning-based implementation is currently limited by the time it takes to train models, but is still able to scale to hundreds of cores. It is particularly encouraging that we are able to achieve high predictive accuracy, and even more exciting, that we can obtain cache hit rates significantly better than our predictive accuracy, by tracking dependencies during speculative execution, providing much more powerful matching capabilities.

There are a number of avenues for extending this work. A few straightforward improvements to our implementation will bring actual performance much closer to possible performance. Developing and evaluating different predictors and recognizers is an obvious next step. Hybrid approaches that use the compiler to identify structure have the potential to alleviate the bottleneck due to training time – we could begin speculative execution immediately based upon compiler input and simultaneously begin training models to identify additional opportunities for speculation. We have only just begun exploring reusing the trajectory cache across different invocations of the same program as well as slightly modified versions of the program. Incorporating persistent storage into this model is also a challenging avenue of future work. On one hand, persistent storage simply makes the state bigger; on the other, it makes the state sufficiently large that the approach could prove ineffective. However, in concurrent work, we find that programs exhibit strong regularity around I/O, and we are optimistic that we can leverage this regularity.

There are obvious parallels between our dependency tracking and transactional memory. It would be interesting to explore hybrid hardware/software approaches to ASC. While we have chosen a learning-based framework in which to realize ASC, there are radically different approaches one might take. We hope to explore such alternatives with other researchers.

6. Conclusion

We have presented an architecture and implementation that extracts parallelism automatically from structured programs. Although \texttt{collatz} and \texttt{2mm} have easily parallelized structure, \texttt{Ising} is more interesting, because it uses dynamically allocated structures, which are frequently not amenable to automatic parallelization. Our learning-based implementation is currently limited by the time it takes to train models, but is still able to scale to hundreds of cores. It is particularly encouraging that we are able to achieve high predictive accuracy, and even more exciting, that we can obtain cache hit rates significantly better than our predictive accuracy, by tracking dependencies during speculative execution, providing much more powerful matching capabilities.

There are a number of avenues for extending this work. A few straightforward improvements to our implementation will bring actual performance much closer to possible performance. Developing and evaluating different predictors and recognizers is an obvious next step. Hybrid approaches that use the compiler to identify structure have the potential to alleviate the bottleneck due to training time – we could begin speculative execution immediately based upon compiler input and simultaneously begin training models to identify additional opportunities for speculation. We have only just begun exploring reusing the trajectory cache across different invocations of the same program as well as slightly modified versions of the program. Incorporating persistent storage into this model is also a challenging avenue of future work. On one hand, persistent storage simply makes the state bigger; on the other, it makes the state sufficiently large that the approach could prove ineffective. However, in concurrent work, we find that programs exhibit strong regularity around I/O, and we are optimistic that we can leverage this regularity.

There are obvious parallels between our dependency tracking and transactional memory. It would be interesting to explore hybrid hardware/software approaches to ASC. While we have chosen a learning-based framework in which to realize ASC, there are radically different approaches one might take. We hope to explore such alternatives with other researchers.

7. Acknowledgements
References


