

DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU



HARVARD LIBRARY
Office for Scholarly Communication

ASC: Automatically Scalable Computation

Citation

Waterland, Amos, Elaine Angelino, Ryan P. Adams, Jonathan Appavoo, and Margo Seltzer. 2014. "ASC: automatically scalable computation." In Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, March 1-5, 2014, Salt Lake City, UT: 575-590.

Published Version

doi:10.1145/2541940.2541985

Permanent link

http://nrs.harvard.edu/urn-3:HUL.InstRepos:34309064

Terms of Use

This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Open Access Policy Articles, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#OAP

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story .

Accessibility

ASC: Automatically Scalable Computation

Amos Waterland
Elaine Angelino
Ryan P. Adams
Harvard University
apw@seas.harvard.edu
elaine@eecs.harvard.edu
rpa@seas.harvard.edu

Jonathan Appavoo
Boston University
jappavoo@bu.edu

Margo Seltzer
Harvard University
margo@eecs.harvard.edu

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 its current point in state space to a deterministic subsequent point. 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 using a functional simulator that emulates the *x*86 instruction set, including a collection of state predictors and a mechanism for speculatively executing threads that explore potential states along the execution path. While the overhead of our simulation makes it impractical to measure speedup relative to native *x*86 execution, experiments on three benchmarks show scalability of up to a factor of 256 on a 1024 core machine when executing unmodified sequential programs.

Categories and Subject Descriptors I.2.6 [*Artificial Intelligence*]: Learning—Connectionism and neural nets; C.5.1 [*Large and Medium ("Mainframe") Computers*]: Super (very large) computers

Keywords Machine learning, Automatic parallelization

1. Introduction

The Automatically Scalable Computation (ASC) architecture is designed to meet two goals: it is straightforward to program and it automatically scales up execution according to available physical resources. For the first goal, we define "straightforward to program" as requiring only that the programmer write sequential code that compiles into 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 continues executing for 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 start 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 per-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '14, March 01 - 05 2014, Salt Lake City, UT, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2305-5/14/03. . . $15.00.
http://dx.doi.org/10.1145/2541940.2541985

fect 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 functional simulator (TBFS) that emulates the *x*86 instruction set with a fast, adaptive algorithm for recognizing predictable states, a set of on-line learning algorithms that use observed states to learn predictors for future states, a resource allocator that selects optimal combinations of predictors in an on-line 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 results. LASC achieves near-linear scaling up to a few hundred cores and continues non-negative scaling up to a few thousand cores. Our benchmark programs are purely computational (i.e., they do not perform I/O after loading their initial input), and they have obvious parallel structure that makes them amenable to manual parallelization. However, they differ in the kinds of parallelism they exhibit and how amenable they are to traditional automatic parallelization approaches. Our goal is to demonstrate the potential of the ASC architecture to automatically identify and exploit statistical patterns that arise during program execution.

The contributions of this paper are: (1) a system architecture that automatically scales performance as a function of the number of cores or size of memory available, (2) a fast and adaptive method for automatically identifying states from which predictions are tractable, (3) a set of general-purpose predictors that learn from observed states, (4) a theoretically sound method to adaptively combine predictions, and (5) a prototype that demonstrates significant scaling on certain classes of unmodified *x*86 sequential binary programs on systems of up to 4096 cores.

The rest of this paper is organized as follows. We begin with a discussion of prior work, showing how conventional techniques can be cast into the ASC architecture in §2. We then present the ASC architecture in §3 and our learning-based implementation of the ASC architecture (LASC) in §4. 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 for future research.

2. Related work

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 arises from conceptual models rather different from ours, notions of statistical prediction and speculative execution have independently arisen in all three.

2.1 Compiler parallelization

Traditional compiler parallelization based on static analysis [1] has produced sophisticated research compilers [3, 9]. Although these compilers can automatically parallelize most loops that have regular, well-defined data access patterns [32], the limitations of static analysis have become apparent [27]. When dealing with less regular loops, parallelizing compilers either give up or generate both sequential and parallel code that must use runtime failsafe checking [55]. The ASC architecture is able to speed up irregular loops by using on-line 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) [14, 60, 61] arose in response to the limits of compile-time static analysis. TLS hardware applies speculative execution to code that was not fully parallelized by the compiler. This hardware backstop allows automatic speculative parallelization by TLS compilers [31, 40, 41, 48, 51, 62, 70] without fully proving 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 [26, 34] 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 demonstrating scaling on up to hundreds of nodes for loops without loop-carried dependencies [33]. Our LASC prototype implementation currently runs on clusters and would benefit from importing hints produced by these compilers in the form of probability distributions.

2.2 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 inducing some runtime overhead.

Binary rewriter parallelization systems [35, 64, 68] take as input a sequential binary executable program and produce as output a parallel binary executable. Dynamic code generating

binary parallelization systems [13, 27] 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 [67], inspired by dynamic binary optimization systems [6], 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 of future state vectors. Most instruction control flow analysis and optimization techniques can be transformed into probabilistic predictors by using appropriately rich feature representations. While engineering such representations may be challenging, "kernelized" machine learning algorithms can efficiently perform predictive computation in potentially large or infinite feature spaces with minimal overhead. Such savings is possible because many important machine learning tools interact with the data via the inner products of feature vectors. By replacing these inner products with positive definite Mercer kernels (the so-called "kernel trick" [7, 19]), we can give the algorithm implicit access to large and rich feature spaces. This means that ASC can import existing work in binary parallelization as "run ahead" approximators [71] that take a completely consistent probabilistic form.

2.3 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 [49] execute individual instructions in parallel, speculatively executing around dependencies using powerful branch prediction [30, 57, 58] and load value prediction [38] strategies and can use trace caches [54] to implicitly enable multiple simultaneous branch predictions. Other approaches make multiple cores appear as one fast out-of-order core [10, 28], shorten execution time by speculatively removing nonessential instructions in parallel [47], and speculatively parallelize programs at the thread [2, 14, 18, 24, 60, 69] or SIMD [16] level, many of which rely on some degree of compiler or runtime [17] assistance.

ASC makes the same contract to the programmer as a superscalar or dynamic multithreading processor: transparent, automatic scaling 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 bits can give rise to a low-entropy joint distribution over points in state space.

Although LASC presently exists only as a software implementation of the ASC architecture, our goal is that every component be amenable to an efficient hardware implementation. For example, probabilistic computing hardware [22, 63] can greatly accelerate our learning algorithms, trace caches could be extended to store our results from executing a trace of instructions, and circuitry similar to transactional memory implementations can expedite dependency tracking.

3. The ASC architecture

We begin our description of the ASC architecture by discussing the trajectory-based model of computation. We then discuss each main component of our architecture.

3.1 Trajectory-based computation

We base our model of computation on the *trajectory-based computation* framework of Waterland et al. [65, 66]. Consider the memory and registers of a single-threaded processor as comprising an exponentially large state space. Program execution then traces out a *trajectory* through this state space. If all input data is loaded up front, this execution is memoryless in that it produces a deterministic sequence of states, where each state depends only on the previous state.

In order to parallelize a program using N cores, we can in principle 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 states, we would simply predict the end state and ignore the computation entirely. If trajectories were random sequences of states through this state space then prediction would be intractable. However, trajectories are not random, and many programs have regularly structured trajectories. Our experience is that many real-world programs exhibit regular structure, suggesting that prediction may often be feasible.

Three features of real-world program execution make the prediction problem significantly easier in practice: (1) each transition depends on and modifies only a small part of the state, (2) large portions of the state are constant, such as the program's machine instructions, or nearly constant, such as data that is written once, and (3) real programs tend to be built from modular and repeating structures, such as functions and loops. Many aspects of execution have been empirically shown to have repetitive or predictable patterns [56]. These include branch directions [30, 58], branch targets [37], memory data addresses [5], memory data values [21], and dependencies [43], while values produced by instructions tend to repeat from a small set [38], and instructions often have the same input and output values [59].

These repetitive or predictable aspects of execution are particularly amenable to the modeling approach of trajectory-based computation [65], which illuminates unexpected geometric and probabilistic structure in computer systems by focusing attention on their *state vectors*, each of which contain all information necessary to deterministically transition to the next state vector. The ASC architecture is designed to amplify predictable patterns in execution into automatic scaling.

3.2 Architecture components

We present our architecture by walking through Figure 1, along the way illustrating some possible design decisions that map existing work into this architecture.

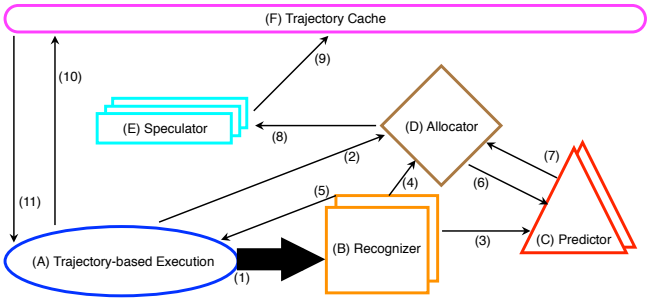


Figure 1. ASC Architecture. The trajectory-based execution engine sends its current state to the recognizers (1) and to the allocator (2). The collection of recognizers selects particular states that appear to be good input to the predictors and sends them to the predictors (3), the allocator (4), and back to the trajectory-based execution engine (5). Using the filtered stream of states obtained from the recognizers, the predictors build models of execution from which they will predict future states. The allocator's job is to assign predictors, recognizers and speculative threads to cores. Based on the states obtained from the recognizer, the allocator sends the current state to the predictors (6), requesting that they use that state to generate a predicted future state (7). Given those predictions, the allocator dispatches speculative threads, each with a different predicted state and number of instructions to execute (8). Each speculative thread executes its specified number of instructions and inserts its start-state/end-state pair into the trajectory cache (9). Finally, at intervals suggested by the recognizer (5), the trajectory-based execution engine consults the state cache to determine if its current state matches any start-states in the cache (10). If there are any matches, the cache returns the end-state farthest in the future (11).

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 amenable to prediction. 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 metric learning to dynamically identify states that are similar to previously seen states. 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 our predictors make accurate predictions.

A set of on-line 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 or heuristic. 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.

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 their results 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 upon which that execution depends.

3.3 Discussion

There are several 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 [42], 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 scales 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.

4. Implementation

LASC, the learning-based ASC, is an implementation of our architecture that turns the problem of automatically scaling sequential computation into a set of machine learning problems. We begin with an overview of our implementation and then discuss the details of each component.

In LASC, the cache, recognizer, predictors, and allocator are all built into the trajectory-based functional simulator (TBFS) that we discuss in §4.1. At the heart of the TBFS is a transition function that interprets its input as an *x*86 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 TBFS 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 combines the predictions and selects a set of predicted states on which to launch speculative threads.

By factoring the problem of predicting a complete state vector into the problems of predicting smaller, conditionally independent portions of a state vector, we parallelize 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 functional simulator

Our simulator's key data structure is the *state vector*, which represents the complete state of computation. Our system executes an instruction by calling the *transition function*: `transition(uint8_t *x, uint8_t *g, int n)`, where `x` is the state vector of length n bits passed as a byte array of dimension $\frac{n}{8}$, and `g` is the *dependency vector* also of dimension $\frac{n}{8}$. The transition function has no hidden state and refers to no global variables. The transition function simply fetches the 32 bits that contain the instruction pointer of the location in state space represented by `x`, fetches the referenced instruction, simulates the instruction according to the *x*86 architecture, and writes the resultant state changes back to the appropriate bits in the state vector. It may seem a poor choice to represent state as a bit array, but this gives us the mathematical structure of a vector space, and allows us to learn and make predictions using massively bit-parallel binary classifiers.

The transition function accumulates dependency information in `g` at the byte—rather than bit—granularity. For each byte in the state vector `x`, the corresponding byte in `g` maintains one of four statuses: `read`, `written_after_read`, `written` or `null`, with the remaining $256 - 4$ codes reserved. When a speculative execution worker starts, it first sets all bytes in `g` to `null`, then calls the transition function in a loop. The transition function automatically uses a simple finite state machine to update the dependency vector on every call. When the transition function reads a byte whose status is `null`, it updates the corresponding status to `read`. If it later writes to that same byte, it updates the corresponding status to `written_after_read`. If it writes to a byte that has never been read, it updates the corresponding 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 stop a speculative thread, the set of bytes with dependency state `read` or `written_after_read` (but *not* `written` or `null`) identifies precisely those bytes on which the speculative computation depends. Therefore, we can match a cache entry when the current state matches the start state of the cache entry merely on bytes having statuses of `read` or `written_after_read`. Not only does this improve the cache hit rate, 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`; this has an interpretation as a translation symmetry in state space.

4.2 Cache

We exploit dependencies to efficiently represent cache entries. Each cache entry is a representation of a start state and an 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 the TBFS. 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 tradeoff. Each node responds with an integer value indicating the length of its longest matching trajectory – zero for 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 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 2. 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 [45].

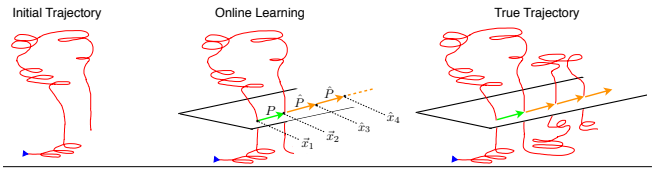


Figure 2. From the initial trajectory, we recognize a hyperplane that cuts the true trajectory at regular, widely-spaced points $\{\vec{x}_1, \vec{x}_2, \dots\}$, all sharing the same IP value. Our online learning problem is then to learn an approximation $\hat{P}$ to the true function P that maps $\vec{x}_i$ to $\vec{x}_{i+1}$. Once we have learned $\hat{P}$, we make predictions as $\hat{x}_{i+1} = \hat{P}(\vec{x}_i)$.

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 *on-line learning* problem [8], in which we wish to learn from the sequence of examples a probability distribution $p(\hat{\mathbf{x}}' \mid \mathbf{x})$ that represents our belief that $\hat{\mathbf{x}}'$ will be the next state vector that will have the same IP as the current state vector $\mathbf{x}$.

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 $\mathbf{x}$ to model a particular bit, a 32-bit quantity, or other feature of the next observation $\mathbf{x}'$. The case of bits is particularly interesting, because it reduces to the binary classification problem, one of the best developed areas of machine learning [52].

We exploit the bit-level conditional independence decomposition to formulate our prediction problem probabilistically, in light of this viewpoint's practical [7, 52] and theoretical [29] advantages. When the current state $\mathbf{x}$ has the RIP value, $p(\hat{\mathbf{x}}' \mid \mathbf{x}, \boldsymbol{\theta})$ expresses our belief that $\hat{\mathbf{x}}'$ is the next state with that IP value according to our model with parameters $\boldsymbol{\theta}$. Since each state is simply a binary vector of n bits, we factor the joint predictive distribution for $\hat{\mathbf{x}}' \mid \mathbf{x}$ into a product of per-bit predictive distributions

$$p(\hat{\mathbf{x}}' \mid \mathbf{x}, \boldsymbol{\theta}) = \prod_{j=1}^{n} p(\hat{x}'_j \mid \mathbf{x}, \boldsymbol{\theta}) \tag{1}$$

$$= \prod_{j=1}^{n} \text{Bernoulli}(\hat{x}'_j \mid \theta_j(\mathbf{x})) \tag{2}$$

where $\hat{x}'_j$ is the j-th bit of the predicted state $\hat{\mathbf{x}}'$. 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 $\mathbf{x}$ as input. Under this model, the probability of a predicted state $\hat{\mathbf{x}}'$ 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. One of the allocator's jobs is to generate a pool of predictions and then decide which ones to send to speculative threads. Given a state $\mathbf{x}$ and model $\boldsymbol{\theta}$, there are two straightforward methods for generating predictions from Eq. 2. The most probable prediction, $\hat{\mathbf{x}}'_{\text{ML}}$, is produced by

maximizing each term, i.e., setting each bit to its most probable value. Alternate predictions for $\hat{\mathbf{x}}'$ can be generated, for example, by strategically flipping the most uncertain bits of $\hat{\mathbf{x}}'_{\text{ML}}$ to give the second and third most likely predictions, and so on. These predictions can be used as input to Eq. 2 to recursively generate predictions farther along the trajectory. 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 which predictions to use for speculation.

In practice, we learn binary classifiers only for the *excitations* of program execution, defined as those bits that have been observed to change more than some threshold number of times (once, by default) from one instance of the RIP value to the next. One of the recognizer's key responsibilities is to find RIP values that exploit our observation that many temporary and even some global variables change but then reset to their starting values during execution between state vectors sharing certain RIP values. The program's excitations induce a strong form of sparsity, as we need learn only those bits that change across states with the same IP value rather than all bits that ever change. Some of the programs we evaluate in §5 have a state space dimensionality of $n > 10^7$, of which $> 10^5$ bits change over the lifetime of the program, but of which < 300 change between observations of a certain RIP value.

4.4.1 Interfaces

Each predictor must at minimum implement three interfaces: `update(`$\mathbf{x}, j$`)`, `predict(`$\mathbf{x}, j$`)` and `reset()`. For each bit j known to be non-constant between occurrences of the RIP, the main thread calls `update(`$\mathbf{x}, j$`)`. Each predictor then uses the difference between its previous prediction $\hat{x}_j$ for the j-th bit of $\mathbf{x}$ and the newly observed actual value x_j of that bit to update its internal model parameters [25].

After the predictors have updated their models, the main thread asks for predictions by calling `predict(`$\mathbf{x}, j$`)` 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 $\mathbf{x}$ 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 invokes predictors in multiple contexts, so each must supply a `reset()` routine that causes them to discard their current models and start building new ones. For example, the recognizer calls `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 that implement the interfaces described in §4.4.1. The results in this paper use only four discrete learning algorithms – two trivial ones, *mean* and *weatherman*, and two interesting ones, *logistic regression* and *linear regression*. The mean predictor simply learns the mean value of each bit and issues predictions by rounding. The weatherman predictor predicts that the next value of each bit will be its current value.

Logistic regression is a widely-used learning algorithm for binary classification [7], in which one is given a vector $\mathbf{x}$ that one must classify as either 1 or 0. It assumes that one has a stream of labeled examples $\{(\mathbf{x}, y), (\mathbf{x}', y'), \dots\}$, where each label y is a 1 or 0. The goal is to correctly predict a new vector $\mathbf{x}''$ 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 $\mathbf{x}'$ given the current state vector $\mathbf{x}$. Logistic regression defines a family of functions, each parameterized by a weight vector $\mathbf{w}$ of dimension $n+1$. We do on-line learning for logistic regression via one fast stochastic gradient descent weight vector update per new observation, where $\mathbf{w}$ is updated to $\mathbf{w}'$ based on the difference between the true label x'_j and the label predicted by $\mathbf{w}$ when evaluated as $\hat{x}'_j = (1+e^{-\mathbf{w}\cdot\mathbf{x}})^{-1}$, where $\mathbf{w}\cdot\mathbf{x} = w_0 + w_1 x_1 + \cdots + w_n x_n$. Although strictly speaking logistic regression can be thought of as treating each input bit as independent, it is not by any means naive or inappropriate, as many seemingly difficult learning problems boil down to linear separability in a high dimensional space [15], which is precisely what its weight vector $\mathbf{w}$ models.

Linear regression is a widely-used learning algorithm used to fit a curve to real-valued data [7]. The word "linear" refers to the fact that the predicted curve is a *linear combination* of the input data, and does not imply that the predicted curve will be a straight line. It takes as input a stream of examples $\{(\mathbf{x}, y), (\mathbf{x}', y'), \dots\}$, where each label y is a real number. The goal is to correctly predict the y'' associated with each new vector $\mathbf{x}''$ before seeing the true answer. In our setup, the labels $y = \phi_i(\mathbf{x}')$ are produced by interpreting the i-th 32-bit word of the state vector $\mathbf{x}'$ as an integer. Like logistic regression, linear regression defines a family of functions, each parameterized by a weight vector $\mathbf{w}$ of dimension $K+1$, and on-line learning involves updating the current weights $\mathbf{w}$ to $\mathbf{w}'$ based on the difference between the true label $\phi_i(\mathbf{x}')$ and the label predicted by $\mathbf{w}$ when evaluated as the polynomial $\widehat{\phi_i(\mathbf{x}')} = w_0 + \sum_{k=1}^{K} w_k \phi_i(\mathbf{x})^k$.

Linear regression is most useful when our system needs to predict integer-valued 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 with different learning rates, and then unify their predictions using the Randomized Weighted Majority Algorithm discussed in the next section. Learning, like speculative execution, occurs in parallel and out of band with execution on the main thread. We use the fast, on-line forms of the gradient descent learning algorithms for both linear and logistic regression.

4.5 Allocator

The allocator is responsible for unifying multiple predictions into state vectors from which it schedules speculative execution.

4.5.1 Combining multiple predictions

We use an approach known as "prediction from expert advice" [8]. 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 internally operate at a higher semantic level. Prediction from expert advice gives us a theoretically sound way to combine wildly-different predictors. In 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 [8, 11, 12, 39].

The allocator uses the Randomized Weighted Majority Algorithm (RWMA) [39] because it comes with strong theoretical guarantees that bound regret. These regret 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 error incurred by the weighted majority votes for that bit will be nearly as low as if we had clairvoyantly only used that best predictor from the start. 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 weights it assigns to each predictor.

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 main thread.

By combining per-bit predictions, the allocator produces a single distribution with the form of Eq. 2. Thus, the allocator combines the results of multiple on-line learning algorithms using a regret minimization framework to form a single unified 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, predicted states. It allows us to both generate predictions and assign them probabilities that represent our belief that they will be correct. The allocator then uses this equation to 'roll out' predictions for k supersteps in the future by using predicted states as input to recursively 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 scaling 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 binary programs. Second, we want to demonstrate that our prototype system achieves some of these benefits in practice, limited only by implementation details that require further 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 software implementation of the ASC architecture and present micro-benchmark 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. While each of the three benchmarks is an unmodified *x*86 binary program, their opcode use is fairly standard and simple. Our fine-grained dependency-tracking simulator is undergoing active improvement, but it does not yet fully support executing benchmark programs with the complexity of SPECINT.

Each of our three benchmark programs can be manually 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 manually 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 requires essentially rewriting their parallel implementation from scratch. To escape this treadmill, our work demonstrates the exciting possibility of simply running *one* sequential binary program on a laptop, a commodity multicore system, and a massively parallel supercomputer, obtaining attractive scalability on all three.

Our first benchmark is the `Ising` kernel, a pointer-based condensed matter physics program. It came to our attention because our colleagues in applied physics found that existing parallelizing compilers were unable to parallelize it. 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 that use dynamic data structures are notoriously difficult to automatically parallelize because of the difficulties of alias analysis in pointer-based code [50]. We demonstrate that by predicting the addresses of linked list elements, LASC parallelizes this kernel.

The second benchmark is the `2mm` multiple matrix multiply kernel in Polybench/C, the Polyhedral Benchmark suite [46]. It computes $D = \alpha ABC + \beta D$, where A, B, C, D are square integer matrices and α, β are integers. This benchmark is, in principle, highly amenable to conventional parallelizing compiler techniques, at least on a shared memory multiprocessor if not on a supercomputer. We tried every setting of loop parallelization that we could find for GCC 4.7.3, but in the best case the resultant OPENMP parallel binary still ran *slower*

than the sequential binary. In any case, we use this benchmark to demonstrate that the on-line learning algorithms in LASC automatically identify the same structure that a static compiler identifies, but without requiring language-level semantic analysis. By identifying the dependencies of repeating patterns in execution—e.g., dot products between rows of A and columns of B—neither LASC nor conventional compilers do value prediction for the entries of D.

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 [36]. The property being tested 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 the latter parallelization opportunity in the outer loop, but importantly also automatically memoizes parts of the inner loop, since in practice the sequence does converge for all integers being tested.

5.2 Predictor accuracy

As our prediction accuracy relies in part on the recognizer's ability to find IP values that induce a predictable sequence of hyperplane intersections, we first examine our superstep statistics. Then, we examine the accuracy of our individual predictors, the regret-minimized error rate of the predictor ensemble, and the resulting trajectory cache hit rates.

One of the recognizer's responsibilities is to find a value of the recognized instruction pointer (RIP) that occurs often enough to be useful but not so often as to make the speculative execution stored in each resultant cache entry too short to be worth the lookup cost. In Table 1 we show the recognizer's performance on our three benchmarks in terms of the number of instructions in the full program execution (total time), how long it took to identify a useful RIP (converge time), and the number of speculatively executed instructions encapsulated in a typical cache entry (average jump).

	Ising	2mm	Collatz
Total time (cycles)	2.3×10^{10}	7.5×10^{9}	2.0×10^{11}
Converge time (cycles)	2.3×10^{7}	2.5×10^{7}	1.0×10^{5}
Average jump (cycles)	1.2×10^{7}	1.3×10^{7}	3.8×10^{6}
State vector size (bits)	2.0×10^{5}	5×10^{7}	3×10^{3}
Cache query size (bits)	640	808	160
Lines of C code	75	154	15
Unique IP values	206	162	40

Table 1. Recognizer statistics for each benchmark.

Since our cache is distributed over parallel machines, queries to and responses from the cache are compressed using the Meyers binary differencing algorithm [44]. Table 1 shows that the average cache query message size for our three benchmark programs is 3200 parts per million (ppm), 16 ppm, and 53331 ppm, respectively, of the full state vector size (cache response messages are even smaller).

Since the recognizer is effecting an optimization over the set of IP values, Table 1 also shows the number of lines of C code and the number of instruction addresses for each benchmark. Since speculative execution begins and ends at RIPs, the average jump is identical to the average interval between RIP occurrences. The ratio between total time and average jump approximates the program's available scalability (in our system), while the converge time is a lower bound on its sequential portion (in our system). As Table 1 shows, our system converges to a useful RIP and begins speculative execution in less than 10^8 instructions, and fast-forwards execution by about 10^7 instructions per jump, so we can in principle automatically scale these benchmarks to thousands of cores.

In Table 2 we examine the error rates of our learning algorithm ensemble in terms of the percentage of incorrect state vector predictions. The data demonstrate that we derive benefit from combining multiple predictors and that it is important to weight the various predictors correctly. The first row of Table 2 gives the default error rate that occurs when we weight each predictor on each bit equally. The second row gives the optimal achievable error rate for our set of predictors if we were able to clairvoyantly use the best predictor for each bit. The third row shows that our on-line regret minimization is able to mix and match the best predictor per bit to achieve an actual error rate within 0.3% of optimal.

	Ising	2mm	Collatz
Equal-weight error rate (1 core)	99.1%	92.6%	99.9%
Hindsight-optimal error rate (1 core)	1.1%	10.2%	1.7%
Actual error rate (1 core)	1.2%	3.2%	1.9%
Total predictions (1 core)	2003	599	25000
Incorrect predictions (1 core)	25	19	475
Cache miss rate (32 cores)	0.5%	2.9%	0.3%

Table 2. Prediction error rates and cache miss rates.

The error rates reported in Table 2 are lower than one might expect for high-dimensional spaces. Since state vectors need only match cache entries on the latter's dependencies, our predictor ensemble need only correctly predict a subset of all bits in order to get credit for a correct overall prediction. The cache miss rates on 32 cores reported in Table 2 are even lower than the error rates in Table 2, since with more available parallelism a wider variety of learning algorithm hyperparameters are explored, resulting in more than one prediction per future state of interest. For example, although `Ising` shows an ensemble error rate of 1.2%, we observe that with 32 cores its cache miss rate is just 0.5%.

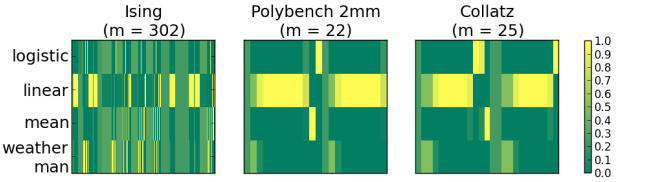


Figure 3. Weight matrices for the benchmarks; rows are the four predictors; columns are the m excited bits (bits that have changed from one instance of the RIP to the next).

To evaluate our system's regret minimization, Figure 3 shows the final weight matrices for each benchmark. The columns are the program excitations; i.e., 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 `Collatz` and `2mm`

show a strong preference for the linear regressor, although there are several bits in the flags register for which the logistic regressor is absolutely crucial. We almost removed the mean and weatherman predictors, assuming they would be too simple to provide additional value, but the `Ising` weight matrix clearly shows that all four algorithms contribute significantly.

5.3 Software implementation of the ASC architecture

The instruction execution component of TBFS implements 79 opcodes of the 32-bit *x*86 instruction set. It executes free-standing static binary programs compiled with GCC. This produces a simple implementation about which we can easily reason and manually verify, while permitting us to run C programs. The restricted functionality is due to the simplified nature of our prototype and is not fundamental to LASC.

We used three experimental testbeds: an *x*86 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 has ASIC hardware acceleration for reduces.

Our software implementation of the ASC architecture has a baseline instruction simulation rate of 2.6 million instructions per second (MIPS) and a dependency tracking instruction simulation rate of 2.3 MIPS. The baseline instruction rate is the number of instructions per second executed when dependency tracking and cache lookups are disabled. The dependency instruction rate shows that dependency tracking has an overhead of approximately 13% above pure architecture simulation. At 2.3 MIPS, our execution overhead is less than that of cycle-accurate simulators, which are usually at around 100 KIPS [20], but more than that of conventional functional simulators, which are usually at around 500 MIPS [20]. There is nothing inherent in either ASC or LASC that necessitates our current execution overhead. Rather, the goal of this paper is to demonstrate the potential of ASC, so we use a testbed that prioritizes ease of experimentation over performance.

Our simple implementation of speculative threads also incurs a significant overhead. Workers make predictions for some k-th future instance of the RIP by making recursive predictions. Currently, each worker does this independently to avoid communication costs. The worker with rank k predicts k supersteps in the future, which means that prediction time is currently a linear function of rank, with a prediction wall-clock time of about $103 \cdot k$ μs on Blue Gene/P.

5.4 Scaling results

In this section we show *scaling* results, measured by dividing the single-threaded wall clock time of each benchmark by its parallel execution wall clock time. This ratio normalizes for fixed cost overheads in our system, almost all of which arise from our relatively slow functional simulator, but we do *not* subtract the cost of learning, lookup or any other cost from the wall clock times.

Figure 4 shows scaling results on the `Ising` benchmark for both the 32-core server and the Blue Gene/P supercomputer.

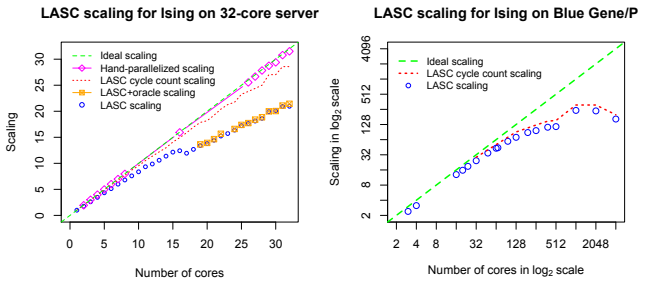


Figure 4. Scaling results for `Ising` benchmark.

We show several parallelization results to tease apart the inherent scalability of the program, the limits of our predictors, and the bottlenecks in our implementation. On the 32-core server, the hand-parallelized results show that it is possible to achieve perfect scaling by first iterating over the list, partitioning it into up to 32 separate lists and then computing on each list in parallel. In LASC, potential scaling is limited by the cache hit rate. With infinitely fast cache lookups, we would obtain the performance illustrated by the "cycle count scaling" 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 scaling lines overlap demonstrates that we are limited only by inefficiencies in our prototype, rather than by prediction accuracy.

There are two different forces at work in the scaling curves of Figure 4. The first arises from the inherent parallelism of the `Ising` benchmark, and the second arises from the artifacts of our prototype. Although the potential energy calculation of `Ising` involves many deeply nested loops, the outermost pattern is just a linked list walk, which enables the recognizer 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 scaling we can expect is approximately 2000. Unsurprisingly, code inspection reveals that its internal linked list has exactly 2000 nodes, which explains the drop-off we see on Blue Gene/P at 2000 cores. However, our scaling peaks at roughly 1024 cores, due to the cost of our current implementation of recursive 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, workers make predictions from predictions for future instances of the RIP, but do not currently share this 'rollout' computation across cores, so prediction time grows linearly in the number of cores. Our next release will use a parallel transitive closure to greatly improve this.

Figure 5 shows scaling results for Polybench/C `2mm` on the 32-core server. Although `2mm` (matrix multiply) also has

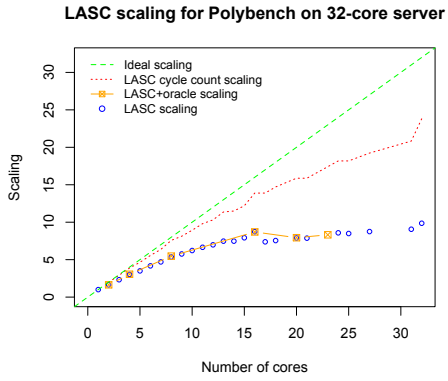


Figure 5. Scaling results for Polybench `2mm`.

somewhat regular structure arising from its dot product loops, its scaling is less impressive than that of `Ising` in that it asymptotes at about $10\times$, which makes it not worth benchmarking on Blue Gene/P. Again, there are two different forces at work. For `2mm`, the superstep accounts for slightly less than 0.2% of the total execution, which limits its potential scalability to 600. The cycle count line for `2mm` in Figure 5 shows encouraging potential scaling possible with our predictors. The oracle line in Figure 5 indicates that LASC is not limited by prediction accuracy for `2mm`, but, as with `Ising`, it is limited by the fact that it does not currently parallelize recursive prediction. 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. Given the relatively small message sizes our prototype achieved for transmitting predicted state vectors, we are optimistic that once we parallelize recursive prediction, our prototype will demonstrate greatly improved overall scalability.

Figure 6 shows scaling results for the `Collatz` benchmark on all three of our platforms. Tables 1 and 2 indicate that our system finds `Collatz` to be easily predicted (98.1% accuracy) with an available parallelism of about 25000 $\times$. However, the relatively small number of instructions between each of the 10^8 iterations of the outer loop forces our recognizer to adapt and consider only every 4000 instances of the RIP, which limits the overall parallelism. Our scaling results for `Collatz` on the 32-core and Blue Gene/P platforms are encouraging but somewhat expected, given that the outer loop of the benchmark is effecting an exhaustive search. What is more interesting is the structure that LASC automatically discovered in the inner loop of the benchmark. Recall that testing the Collatz conjecture is an iterative process of computing $n/2$ or $3n+1$. As the conjecture is never disproven, ultimately, every integer tested eventually converges to 1 through shared final subsequences that end in 1. For example, for all but very small integers, the final five elements must be 16, 8, 4, 2, 1. As the inner loop tests the conjecture, even though the process is chaotic, it produces patterns exploited by LASC. We demonstrate this by running `Collatz` on our single-core laptop, on which parallel speculation is not possible, but on which the recognizer still detects frequently occurring IP values and uses the intervening computation to effect generalized memoization. LASC scales up execution on a single core by using cache entries from the program's own past, which results in the scaling shown in the rightmost graph of Figure 6. As the outer loop tests increasingly large integers, memoized subsequences comprise smaller relative fractions of execution, so scaling eventually asymptotes. Note that we disabled this pure memoization capability for the 32-core server and Blue Gene/P experiments to highlight the scaling available exclusively from prediction and speculation.

5.5 Limitations

While we find these early results exciting, there are obvious limitations to our current implementation as well as limitations inherent in ASC. The results in §5.4 all showed relative scaling, rather than absolute speedup, because our prototype is several orders of magnitude slower than a native core at sequential execution. We are exploring the following directions for improving its performance. There are four components required to implement our architecture: (1) an execution engine used for both ground truth and speculative execution, (2) a mechanism for dependency tracking during execution, (3) a lookup mechanism for large, sparse bit vectors, and (4) a recursive prediction mechanism. For the first component, we are exploring dynamic compilation and process tracing. For the second component, we are exploring compiler static analysis that proves or probabilistically bounds which regions of state space are immutable, as well as transactional memory hardware that tracks dependencies and modifications in state space, reporting them in a representation efficient for transmitting state vector differences between processors. Current measurements of Intel's Haswell transactional memory report less than a factor of two slowdown for large read transactions and almost no slowdown for writes [53]. For the third component, we are exploring a binary decision trie that maximizes the expected utility of our cache by balancing the payoff of each cache entry against the probability of its use. For the fourth component, we are implementing a parallel transitive closure for recursive prediction.

A second limitation is that we have chosen benchmarks that programmers find trivial to parallelize. This was by design. While humans are capable of manually parallelizing programs, ensuring that the resultant parallel program runs well on a machine of any size remains challenging. Further, most computation is invoked by people whose interests lie in the results of the computation and not in the creation of the program. For such users with whom we've interacted, manual parallelization feels like wasted work that gets redone every time they change to a new machine. Our goal is to provide a mechanism that relieves programmers of this burden.

The class of programs for which our architecture is appropriate is, in principle, any program that has "information bottlenecks" in its state space; i.e., segments of execution whose results, when transmitted forward in time as dependencies of later segments of execution, can be significantly compressed. In practice, the limitations of our architecture arise from our system's ability to automatically identify and

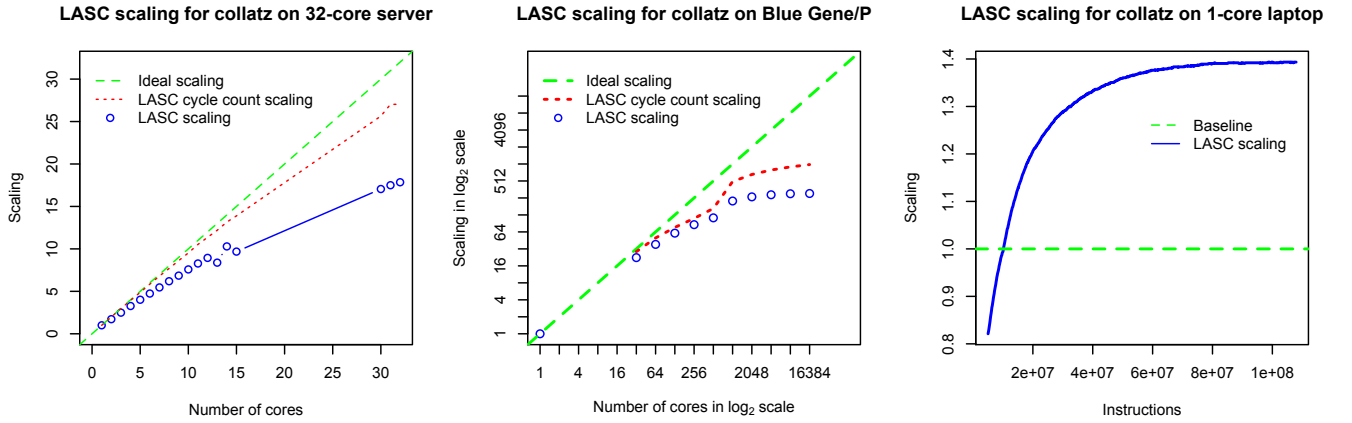


Figure 6. Scaling results for `Collatz` benchmark. The left and center plots are the same measurements as in Figure 4, illustrating scaling resulting from prediction and speculative trajectory evaluation. In contrast, the rightmost plot illustrates scaling by simply caching the program's past trajectory – a form of generalized memoization with no predictions involved.

approximate these information-flow structures in state space. Like other architectures that have tried to exploit parallelism over large windows of execution, we are also limited by our ability to store and look up large speculative state vectors. While these identification, prediction, and lookup problems are difficult in general, taken together they are a promising reduction from the overall automatic parallelization problem, as they are themselves inherently highly scalable. Since it is relatively straightforward for our architecture to incorporate the results of static analysis, the class of programs for which our architecture is appropriate has a large overlap with the set of programs targeted by conventional parallelizing compilers.

Lastly, the theory of P-completeness tells us that there are fundamental limits to parallelism [4, 23]. Some problems simply do not admit any parallel solution that runs significantly faster than their sequential solution, e.g., it is extremely unlikely that any system, human or automatic, will ever significantly parallelize an iterated cryptographic hash. Our goal is to design an architecture capable of pushing up against these information-theoretic limits of parallelism; our regret minimization framework (§4.5.1) is explicitly designed with this in mind, as it allows us to coherently incorporate predictive hints from a wide breadth of tools.

6. Conclusion

We have presented an architecture and implementation that extracts parallelism automatically from structured programs. Although `Collatz` and `2mm` have easily parallelized structure, `Ising` uses dynamically allocated structures, which are frequently not amenable to automatic parallelization. Our learning-based implementation is currently limited by its recursive prediction time, but is still able to scale to hundreds of cores. It is particularly encouraging that we are able to achieve high predictive accuracy, and that we obtain cache hit rates even higher than our predictive accuracy by tracking dependencies during speculative execution. 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. The ASC architecture can be extended naturally to encompass I/O. Like the contents of memory and registers, output data is computed from existing state. Input data will be handled by naturally extending LASC's prediction framework to modeling the input distribution, e.g., learning a probabilistic model consistent with the histogram of observed inputs. 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 realized ASC in a learning-based framework, there are radically different approaches one might take. We hope to explore such alternatives with other researchers.

Acknowledgments

The authors would like to thank Gerald Jay Sussman, Miguel Aljacen, Jeremy McEntire, Ekin Dogus Cubuk, Liz Bradley, Eddie Kohler, Benjamin Good and Scott Aaronson for their contributions. This work was supported by the National Science Foundation with a Graduate Research Fellowship under Fellow ID 2012116808 and a CAREER award under ID CNS-1254029 and CNS-1012798, and by the National Institutes of Health under Award Number 1R01LM010213-01. This research was awarded resources at the Argonne Leadership Computing Facility under the INCITE program, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] Vikram S. Adve, John Mellor-Crummey, Mark Anderson, Jhy-Chun Wang, Daniel A. Reed, and Ken Kennedy. An integrated compilation and performance analysis environment for data parallel programs. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[2] Haitham Akkary and Michael A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, MICRO 31, pages 226–236, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[3] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, pages 126–138, New York, NY, USA, 1993. ACM.

[4] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[5] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 176–186, New York, NY, USA, 1991. ACM.

[6] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[7] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[8] Avrim Blum. On-line algorithms in machine learning. In Amos Fiat and Gerhard J. Woeginger, editors, *Online Algorithms*, volume 1442 of *Lecture Notes in Computer Science*, pages 306–325. Springer, 1996.

[9] Bill Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: The next generation in parallelizing compilers. In *Proceedings Of The Workshop On Languages And Compilers For Parallel Computing*, pages 10–1. Springer-Verlag, Berlin/Heidelberg, 1994.

[10] Michael Boyer, David Tarjan, and Kevin Skadron. Federation: Boosting per-thread performance of throughput-oriented manycore architectures. *ACM Trans. Archit. Code Optim.*, 7(4):19:1–19:38, December 2010.

[11] Nicolò Cesa-Bianchi, Yoav Freund, David Haussler, David P. Helmbold, Robert E. Schapire, and Manfred K. Warmuth. How to use expert advice. *J. ACM*, 44(3):427–485, May 1997.

[12] Nicolo Cesa-Bianchi and Gabor Lugosi. *Prediction, Learning, and Games*. Cambridge University Press, New York, NY, USA, 2006.

[13] Michael K. Chen and Kunle Olukotun. The jrpm system for dynamically parallelizing java programs. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 434–446, New York, NY, USA, 2003. ACM.

[14] Marcelo Cintra, José F. Martínez, and Josep Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 13–24, New York, NY, USA, 2000. ACM.

[15] Adam Coates, Andrew Y Ng, and Honglak Lee. An analysis of single-layer networks in unsupervised feature learning. In *International Conference on Artificial Intelligence and Statistics*, pages 215–223, 2011.

[16] A Dasgupta. Vizer: A framework to analyze and vectorize intel x86 binaries. Master's thesis, Rice University, 2002.

[17] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 15–24, Washington, DC, USA, 2003. IEEE Computer Society.

[18] Pradeep K. Dubey, Kevin O'Brien, Kathryn M. O'Brien, and Charles Barton. Single-program speculative multithreading (spsm) architecture: compiler-assisted fine-grained multithreading. In *Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques*, PACT '95, pages 109–121, Manchester, UK, UK, 1995. IFIP Working Group on Algol.

[19] Maria florina Balcan, Manuel Blum, Yishay Mansour, Tom Mitchell, and Santosh Vempala. New theoretical frameworks for machine learning, 2008.

[20] Björn Franke. Fast cycle-approximate instruction set simulation. In *Proceedings of the 11th international workshop on Software & compilers for embedded systems*, pages 69–78. ACM, 2008.

[21] Freddy Gabbay and Freddy Gabbay. Speculative execution based on value prediction. Technical report, EE Department TR 1080, Technion - Israel Institue of Technology, 1996.

[22] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Daniel Tarlow. Church: a language for generative models. *CoRR*, abs/1206.3255, 2012.

[23] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to parallel computation: P-completeness theory*. Oxford University Press, Inc., New York, NY, USA, 1995.

[24] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS VIII, pages 58–69, New York, NY, USA, 1998. ACM.

[25] Milos Hauskrecht. Linear and logistic regression. Class lecture, 2005.

[26] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[27] Ben Hertzberg. *Runtime Automatic Speculative Parallelization of Sequential Programs*. PhD thesis, Stanford University, 2009.

[28] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 186–197, New York, NY, USA, 2007. ACM.

[29] E.T. Jaynes. *Probability Theory: The Logic of Science*. Cambridge University Press, 2003.

[30] Daniel A. Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, pages 197–, Washington, DC, USA, 2001. IEEE Computer Society.

[31] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Speculative thread decomposition through empirical optimization. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 205–214, New York, NY, USA, 2007. ACM.

[32] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[33] Hanjun Kim, Nick P. Johnson, Jae W. Lee, Scott A. Mahlke, and David I. August. Automatic speculative DOALL for clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 94–103, New York, NY, USA, 2012. ACM.

[34] Tom Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 105–112, New York, NY, USA, 1986. ACM.

[35] Aparna Kotha, Kapil Anand, Matthew Smithson, Greeshma Yellareddy, and Rajeev Barua. Automatic parallelization in a binary rewriter. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 547–557, Washington, DC, USA, 2010. IEEE Computer Society.

[36] Jeffrey C. Lagarias. The 3x+1 Problem: An Annotated Bibliography, II (2000-2009). *Arxiv*, August 2009.

[37] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *Computer*, 17(1):6–22, January 1984.

[38] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *ASPLOS*, pages 138–147, 1996.

[39] Nick Littlestone and Manfred K. Warmuth. The weighted majority algorithm. *Inf. Comput.*, 108(2):212–261, February 1994.

[40] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: A TLS compiler that exploits program structure. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 158–167, New York, NY, USA, 2006. ACM.

[41] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 166–176, New York, NY, USA, 2009. ACM.

[42] Donald Michie. "Memo" Functions and Machine Learning. *Nature*, 218(5136):19–22, April 1968.

[43] Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pages 181–193, New York, NY, USA, 1997. ACM.

[44] Eugene W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[45] Todd Mytkowicz, Amer Diwan, and Elizabeth Bradley. Computer systems are dynamical systems. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 19(3):033124, 2009.

[46] Louis-Noel Pouchet. Polybench/c: the polyhedral benchmark suite.

[47] Zach Purser, Karthik Sundaramoorthy, and Eric Rotenberg. A study of slipstream processors. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pages 269–280, New York, NY, USA, 2000. ACM.

[48] Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 269–279, New York, NY, USA, 2005. ACM.

[49] George Radin. The 801 minicomputer. In *Proceedings of the first international symposium on Architectural support for programming languages and operating systems*, ASPLOS I, pages 39–47, New York, NY, USA, 1982. ACM.

[50] Easwaran Raman, Neil Vachharajani, Ram Rangan, and David I. August. Spice: speculative parallel iteration chunk execution. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 175–184, New York, NY, USA, 2008. ACM.

[51] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 177–188, Washington, DC, USA, 2004. IEEE Computer Society.

[52] David Stork Richard Duda, Peter Hart. *Pattern Classification (Second Edition)*. John Wiley & Sons, Inc., 2001.

[53] C. G. Ritson and F. R. M. Barnes. Evaluating intel rtm for cpas. In P. H. Welch et al, editor, *Proceedings of Communicating Process Architectures 2013*. Open Channel Publishing Limited, 2013.

[54] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, pages 24–35, Washington, DC, USA, 1996. IEEE Computer Society.

[55] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. *Int. J. Parallel Program.*, 31(4):251–283, August 2003.

[56] Yiannakis Sazeides. Instruction-isomorphism in program execution. In *In Proceedings of the Value Prediction Workshop*, pages 47–54, 2003.

[57] Jeremy Singer, Gavin Brown, and Ian Watson. Deriving limits of branch prediction with the fano inequality, 2006.

[58] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, ISCA '81, pages 135–148, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[59] Avinash Sodani and Gurindar S. Sohi. An empirical analysis of instruction repetition. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS VIII, pages 35–45, New York, NY, USA, 1998. ACM.

[60] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, pages 414–425, New York, NY, USA, 1995. ACM.

[61] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 1–12, New York, NY, USA, 2000. ACM.

[62] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The stampede approach to thread-level speculation. *ACM Trans. Comput. Syst.*, 23(3):253–300, August 2005.

[63] Benjamin Vigoda. *Analog logic: Continuous-Time analog circuits for statistical signal processing*. PhD thesis, Massachusetts Institute of Technology, 2003.

[64] Cheng Wang, Youfeng Wu, Edson Borin, Shiliang Hu, Wei Liu, Dave Sager, Tin-fook Ngai, and Jesse Fang. Dynamic parallelization of single-threaded binary programs using speculative slicing. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 158–168, New York, NY, USA, 2009. ACM.

[65] Amos Waterland, Jonathan Appavoo, and Margo Seltzer. Parallelization by simulated tunneling. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, HotPar'12, pages 9–14, Berkeley, CA, USA, 2012. USENIX Association.

[66] Amos Waterland, Elaine Angelino, Ekin D. Cubuk, Efthimios Kaxiras, Ryan P. Adams, Jonathan Appavoo, and Margo Seltzer, *Computational caches*, Proceedings of the 6th International Systems and Storage Conference (New York, NY, USA), SYSTOR '13, ACM, 2013, pp. 8:1–8:7.

[67] J. Yang, K. Skadron, M. Soffa, and K. Whitehouse. Feasibility of dynamic binary parallelization. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, 2011.

[68] Efe Yardimci and Michael Franz. Dynamic parallelization and mapping of binary executables on hierarchical platforms. In *Proceedings of the 3rd conference on Computing frontiers*, CF '06, pages 127–138, New York, NY, USA, 2006. ACM.

[69] Jenn yuan Tsai and Pen-Chung Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Proceedings of the conference on Parallel architectures and compilation techniques*, PACT '96, pages 35–46, 1996.

[70] Hongtao Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 290–301, Feb.

[71] Craig Zilles and Gurindar Sohi. Master/slave speculative parallelization. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 35, pages 85–96, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.