Automatically Scalable Computation That Is More Scalable And Automatic

A THESIS PRESENTED
BY
PETER KRAFT
TO
THE DEPARTMENT OF COMPUTER SCIENCE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
BACHELOR OF ARTS
IN THE SUBJECT OF
COMPUTER SCIENCE

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
MARCH 31 2017

CONTENTS

ABSTRACT

We present significant new development of the Automatically Scalable Computation (ASC) project, a new method of automatic parallelization that transforms parallelization into a machine learning problem. ASC speeds up computation by predicting future states of a program's memory and registers and speculatively executing from those predicted states. Prior work on ASC has demonstrated the ability to speed up a limited class of programs using a single additional core. It has also suggested that ASC has the potential to scale that speedup to many cores for a larger class of programs. We realize that potential by developing a new version of ASC. Our version incorporates a new architecture that can scale to many cores as well as a notion of data dependency that expands the class of programs ASC can speed up by making the prediction problem much easier. Using this new version of ASC, we demonstrate automatic speedup on a large class of programs, including programs essential to scientific computing and programs resistant to other methods of automatic parallelization. This speedup increases near-linearly with the number of cores supplied to ASC.

ACKNOWLEDGEMENTS

I would like to extend my gratitude to my thesis advisor and mentor, Margo Seltzer, for the enormous amount of support she has given me and time she has spent guiding me through this thesis. I would also like to thank Amos Waterland for introducing me to ASC. I am grateful to the PRISE fellowship for letting me work on this thesis during the summer and to my classmates in CS261 for their suggestions and advice. Special thanks go to Sophia Feng, Tomas Reimers, and Wendy Woodin for their support and proofreading, as well as to the entire Pforzheimer House community for helping me stay sane while I worked on this thesis. Finally, I would like to thank my family for their unending support (and editing).

1 INTRODUCTION

1.1 PARALLELIZATION AND ASC

One of the most important problems in computer science is parallelization. Almost all modern computers contain several execution engines, called cores. In theory, most computationally expensive programs can be run on multiple cores simultaneously and will run faster as they use more cores. However, in reality, writing programs that use multiple cores efficiently is much more challenging than writing traditional programs that use only a single core. This problem is called the problem of parallelization, and programs that are able to use multiple cores simultaneously are called parallel programs (as they must perform computations *in parallel* on multiple cores).

Because parallel programming is difficult, most programmers write programs that are sequential, meaning they can use only a single core. This is extremely inefficient as it means auxiliary cores are wasted, their processing power completely unused. Given the prevalence of multicore computers, it is crucial to develop ways to make it as easy as possible for programs to fully exploit multiple cores. This will allow programs to make use of future developments in computer architecture, since these developments will likely lead to more cores, not faster cores.

The holy grail of parallelization research is *automatic parallelization*. Automatic parallelization is the automatic transformation of a sequential program into a parallel one. Automatic parallelization allows programmers to have their cake and eat it too—they can write code that is sequential, and therefore easy to write, but still have it run as fast as parallel code. This thesis presents developments to an automatic parallelization project, Automatically Scalable Computation (ASC). ASC uses prediction and speculative execution to allow an unmodified sequential binary program to exploit multiple cores and run faster than it would on a single core (Waterland et al., 2014). We improve ASC significantly, allowing it to speed up more types of programs and also produce more speedup in programs.

The central idea of ASC is the program state, defined as the values of the individual bits in a program's memory and registers. ASC monitors running programs and extracts their states. It then builds a machine learning model to make predictions about potential future states of the program. Next, ASC has auxiliary cores speculatively execute from the predicted future states. ASC then creates a cache that maps between these predicted states and later states (called speculated states) resulting from execution of the predicted states. If the main thread of the program ever realizes that its current state is identical to one of the predicted states in the cache, it can instantly update its current state to the speculated state corresponding to the end state of speculative execution from the predicted state, fast-forwarding itself into the future and taking advantage of computation done on an auxiliary core. By continuously repeating this cycle of prediction, speculative execution, caching, and fast-forwarding, ASC can allow a single-threaded program to take full advantage of multiple cores (Waterland et al., 2014). We diagram this in Figure 1.

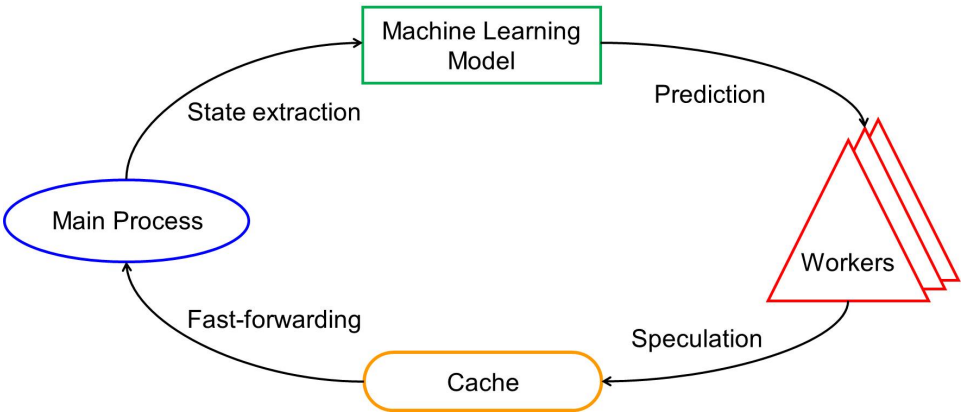

Figure 1: The main cycle of ASC. ASC extracts states from the main process. The machine learning model then predicts future states of the main process from the extracted states. The workers speculatively execute from the predicted states on auxiliary cores, storing pairs of predicted and speculated states in the cache. If the main process ever enters a cached predicted state, it fast-forwards to the corresponding speculated state, speeding itself up.

1.2 Versions of ASC

Before we began our work, two implementations of ASC existed—a simulated version and a bare-metal version. Both implement the predict-speculate-cache-fast-forward cycle described above. However, their execution engines differ. The simulated version is described in Waterland et al. (2014), where it is named the Learning-based Automatically Scalable Computation (LASC) system. LASC is a simulator that implements 79 opcodes of the 32-bit x86 instruction set. It executes free-standing binary programs compiled with `gcc` restricted to those opcodes. As a simulator, it executes binaries orders of magnitude more slowly than native execution. As a result, while LASC was an extremely useful proof-of-concept for demonstrating that the core ASC concepts could work, it could not produce true speedup and could run only a small number of programs.

To improve on LASC, the "bare-metal version" of ASC was created. The bare-metal version of ASC does not contain a simulator. Instead, it runs target programs directly on bare metal and controls their execution through the `ptrace` API and through direct memory access. As a result, unlike LASC, it can speed up real programs. Full details of how the bare-metal version operates are given in Section 3. In the remainder of this paper, we discuss the bare-metal version and our new version of ASC, which is based on it, unless otherwise noted.

1.3 ASC Predictions

The most important requirement of ASC is the ability to make accurate predictions about future states of its target program. If ASC can make accurate predictions,

it can use them to run simultaneous speculative executions on an arbitrarily large number of auxiliary cores. ASC can then use the results of these speculations to fast-forward the target program far into the future, thus automatically speeding it up by an amount scaling near-linearly with the number of cores. Producing this speedup is the primary goal of ASC. However, it requires accurate predictions. Therefore, the most important problem confronting ASC is the problem of making more accurate predictions. If this can be achieved, everything else follows naturally.

There are two ways to improve ASC's prediction capabilities. The most obvious way is to improve the machine learning model. This can be effective in some cases. However, because time spent predicting is time not spent speculating, prediction speed is a harsh constraint on the model. Moreover, improving the model is for many programs futile because their states contain bits that are fundamentally unpredictable, often because they are set by complex calculations.

The other, and typically better, way to improve ASC's prediction capabilities is to reduce the number of bits that ASC needs to predict correctly. This is possible because ASC needs only correctly predict bits whose values affect the speculation. These bits are often a small subset of the overall program state. For example, as we show in Section 5, speeding up a matrix multiplication program requires only correctly predicting 150 bits (mostly in loop counters or the flags registers) out of many billions. In an ideal world, ASC would know the exact subset of the program state that a speculation depends on, predict only those bits, and ignore the rest. This is not always possible, of course, but any progress towards that ideal will both improve ASC's effectiveness and enlarge the class of programs it can speed up.

As an example of how the bits on which a calculation depends can be a small and easy-to-predict subset of the program state, consider a simple program that squares every element of an array (Listing 1). The overall program state contains a large number of bits that are relatively hard to predict, such as the results of the squaring operations. It also contains some bits that are relatively easy to predict, such as the loop counter. The squaring operations depend only on the latter; the value of the square of the second number in an array does not change based on the value of the square of the first number. Therefore, the only bits ASC needs to predict correctly are those in the loop counter, which are trivial to predict and form a tiny subset of the overall state. The challenge lies in identifying those bits and communicating them to ASC.

```
void square_array(int* A, int* B, int array_len) {
  for(int i = 0; i < array_len, i++)
    B[i] = A[i] * A[i];
}
```

Listing 1: A simple squaring function that fills array B with the squares of the integers in array A. Each loop iteration changes two values: the loop counter i and an entry $B[i]$ in B. The latter is much harder to predict than the former, but ASC only needs to accurately predict the former to usefully speculate.

Waterland et al. (2014) were fully aware of the need to minimize the number of bits that ASC needs to predict correctly. They also recognized that the only bits that could affect a computation were the bits read during that computation. As a result, they used the LASC simulator to keep track of every memory location a program reads from during computation and convey that information to ASC so it knows only those bits need be predicted correctly. This idea, which massively reduces the number of bits ASC needs to predict correctly, is called data dependency tracking. However, this tracking is only possible because of the simulator; as a result, it does not exist in the bare-metal version of ASC.

1.4 Goals and Contributions

Given the importance of improving ASC's prediction capabilities by reducing the number of bits ASC needs to predict, our primary contribution is to develop a new version of ASC based on the bare-metal version that performs data dependency tracking. This is an extremely important step because the data-dependency tracking in LASC was only possible because of the simulator, and was thus fundamentally incapable of being used in a real program to achieve real speedup. We demonstrate that data-dependency tracking on real programs running on real computers is realistic. Our new version of ASC has the ability to determine which bits are read from during a computation and recognize those are the only bits that need to be predicted correctly for valid speculation. By doing this, we greatly expand the range of programs ASC can speed up and achieve substantial speedup on several programs ASC could never speed up before.

Specifically, to implement data dependency tracking, we build a tool that runs on top of Pin, Intel's dynamic instrumentation package for x86 (Luk et al., 2005). This tool monitors a program while it is running and executes instrumentation code that keeps track of every memory location the program reads from and writes to during a computation. It then conveys that information to ASC. This allows ASC to know which bits are read from during a computation and therefore which bits it needs to predict correctly.

In addition to developing data dependency tracking, we implement a variety of systems within ASC that improve its function and ability to speed up programs in various ways. The first of these is a new, more flexible machine learning model. Data dependency tracking requires adapting which bits are predicted based on which bits need to be predicted. The machine learning model built into the bare-metal version of ASC, a neural net, made doing this difficult. Moreover, its prediction speed scales extremely poorly with the number of bits. Therefore, we developed a new machine learning model based on decision trees that did not have these disadvantages.

We also develop a new cache and scheduler for ASC that allow it to simultaneously manage a large number of worker processes and therefore scale speedup to a large number of cores. The bare-metal version of ASC only supported a single worker process. This means that it could only exploit a single auxiliary core. If

given a machine with more cores, it could not use them. Given that ASC's primary goal is producing near-linear speedup of programs by exploiting large number of cores, this was extremely problematic. Therefore, we developed a system to manage many worker processes, scheduling them so all workers were always performing useful speculation and caching their results so speculations from accurate predictions could be exploited as necessary.

Our new system is able to achieve substantial speedup on a variety of programs, including several that are essential to scientific computing. We also show that ASC is capable of automatically memoizing programs that are amenable to memoization, thus achieving superlinear scaling where other automatic parallelization systems could not. Our predictors are also capable of finding patterns that allow parallelization of certain types of programs, such as linked list traversal, that other automatic parallelization systems struggle with.

In the remainder of this paper, we further motivate and present these systems, then report their performance. In Section 2, we discuss related work that motivated and inspired ASC. In Section 3, we go into further detail about how the bare-metal version of ASC operates and how its component systems act and interact. In Section 4, we describe the operation and implementation of our new version of ASC. In Section 5, we report on the results of these new systems with particular focus on speedup and scaling on a variety of target programs. In Section 6, we conclude the paper and investigate the implications of these results on the future of ASC.

2 Related Work

The goal of ASC is automatic parallelization of single-threaded code. As a result, ASC shares history with and has drawn ideas from a large body of previous work. This work can be divided into two broad categories. The first of these is compiler parallelization, which compiles single-threaded source code to a multi-threaded binary. The other is binary parallelization, which automatically parallelizes a single-threaded binary. Both categories can be further subdivided into static and dynamic parallelization. Static parallelization transforms a single-threaded source or binary into a multi-threaded binary. Dynamic parallelization speeds up a single-threaded binary (sometimes unmodified, sometimes specially compiled or transformed) at runtime by using additional cores. Out of all these categories, ASC is best classified as a dynamic binary parallelization system.

2.1 Compiler Parallelization

The most traditional method of automatic parallelization is static compiler parallelization. This can take different forms. One approach is to not automatically parallelize at all, but to make it as easy as possible to parallelize manually. One example for C/C++ is OpenMP, which uses pragmas to provide an API to the compiler to indicate opportunity for parallelization. If one's program contains a paral-

lelizable loop, one can simply indicate this with a pragma and OpenMP will do the difficult work of parallelizing it (Dagum and Menon, 1998). This is not automatic, but it is effective for code written in the right languages with clearly parallelizable computations. A related example is Cilk, a superset of C/C++ that introduces keywords for forking and joining threads and implements a scheduler to use them (Blumofe et al., 1995). Like OpenMP, Cilk will not parallelize automatically, but it enables extremely scalable parallelization of certain constructs with only a few lines of code.

Beyond parallelization libraries, researchers have also developed truly automatic static compiler parallelization. In its most basic form, automatic static compiler parallelization involves recognizing and multithreading loops whose iterations do not depend on each other or depend on each other in simple, parallelizable ways. This has proven highly successful at automatically parallelizing programs that exhibit regular data access patterns that can easily be statically inferred from the source code (Blume et al., 1994). However, that restriction is extremely limiting and leads to extremely conservative parallelization (Mehrara, 2011). Too many parallelizable programs lack enough regular structure to easily parallelize, and partitioning most programs into concurrent threads with no pragmas or directives is extremely difficult (Hertzberg, 2009).

To overcome these limitations, thread-level speculation (TLS) was developed. TLS involves parallelizing code with ambiguous dependencies by making certain assumptions about the dependencies, then using hardware to check at runtime whether those assumptions were correct. If they are, the speculations are used, if not, they are not (Steffan et al., 2000). Similar to TLS is hybrid analysis, proposed by Rus et al. (2003), which performs relatively liberal static analysis then does runtime failsafe checking. Both these techniques are similar to ASC in that they involve making speculations and using them only if they are correct, but retain a limiting dependence on static analysis and, in the case of TLS, on specialized hardware.

A variety of other hybrid approaches have been created that use both compilers and dynamic analysis. One approach is to pipeline loops into separate threads that do not have cyclic dependencies, sometimes with the help of software transactional memory (Raman et al., 2010). This is unpredictable due to the complexity of the pipelining transformation, though, and can just as easily slow a program down as speed it up. Another technique, and the one most similar to ours, is to have the compiler parallelize a loop while dynamically transmitting information about data dependency between cores to guarantee synchronization and correctness. This was implemented by Campanoni et al. (2012) for the HELIX project. Like our approach, HELIX can parallelize a variety of programs. However, it remains dependent on static analysis, and thus regular data access patterns, and it can easily be crippled by communication overhead.

2.2 Binary Parallelization

The limitations of compiler parallelization have inspired a variety of alternatives that operate directly on binaries, whether statically or dynamically. Binary parallelizers lose access to sometimes-valuable semantic information contained within source code. In exchange, they also lose their dependency on having the source code and, if they are dynamic, gain access to runtime information.

The variety of binary parallelization techniques is, if anything, even greater than the variety of compiler parallelization techniques. One approach is direct binary translation, which transforms the sequential binary into a paralellized binary. This has many of the same advantages and disadvantages as static compiler parallelization—it is extremely powerful when it works (even more so than compiler parallelization, as it does not require the source code and has access to the library binaries), but does not work on many programs due to the limits of static analysis (Kotha et al., 2010). A closely related technique is *slicing*, which uses static analysis of data flow to divide a program into parallel slices while using speculation to work around rare dependencies (Wang et al., 2009).

Alternatively, dynamic binary parallelization techniques abandon static analysis altogether. One of the first examples of this work is Dynamo (Bala et al., 2000). Dynamo did not actually do parallelization at all; it was an optimizer that identified critical sections in code as it ran and made them faster. However, it inspired work such as Yang et al. (2011), that monitors program exection for frequently-executed *hot traces* which can be parallelized as they reappear. Such approaches share ASC's predict-speculate-fast-forward paradigm, but are limited by their reliance on hot traces alone, unlike ASC's predictions.

2.3 Dynamic Instrumentation

Our data dependency tracking for ASC determines which bits a program reads from or writes to during a computation. Analyzing a program's operation to obtain information like this is an important problem that has been approached in many different ways. One possibility is full transformation, where the code is translated to an intermediate representation before being run on a virtual machine or simulator. The most prominent example of this sort of binary instrumentation is Valgrind (Nethercote and Seward, 2007). Using a virtual machine makes instrumentation much easier, as one can simply have the virtual machine record whatever it is simulating. For that reason, it was the approach taken by LASC (Waterland et al., 2014). However, the tradeoff for this ease of instrumentation is poor performance. Valgrind is in the best case four or five times slower than native code and is much worse with extensive instrumentation, while LASC is multiple orders of magnitude slower than native. The bare-metal version of ASC was developed in large part to avoid these performance compromises.

Another approach to instrumentation, and the one we adopt, is to use an extremely lightweight just-in-time compiler (JIT) to run the binary code and insert

instrumentation as appropriate. Prominent examples of this are DynamoRio, developed by Bruening et al. (2003) (and based off of Dynamo) and Pin, developed by Luk et al. (2005). Both tools have extremely low base overhead (on the order of 10% for DynamoRio and 30% for Pin) because of the optimization of their JITs and both allow the insertion of a wide range of instrumentation and analysis code into target binaries. We chose to use Pin due to its more powerful API and instrumentation capabilities. We further describe Pin and our Pintool in Section 3.5.

3 BACKGROUND

In this section we describe the fundamental components of ASC, describe the implementation of the bare-metal version of ASC, and provide an overview of Pin. We first discuss some of the core ideas and terminology in ASC. Then, we detail its components and operations. Next, we describe how these systems work together to produce speedup. Finally, we discuss the Pin instrumentation system that we use to track data dependencies.

3.1 INITIALIZATION OF ASC

The bare-metal version of ASC simultaneously manages a main process and several worker processes. The main process is the target program that ASC speeds up. The workers are alternative instances of that program running on different cores starting from predicted states. ASC begins operation by launching them all normally as children with `fork` and `exec`. Next, ASC calls `ptrace` on each process. `ptrace` is a Linux system call that allows one process (ASC itself) to control another (the main process or a worker). It is most commonly used in debuggers. ASC uses `ptrace` to set breakpoints in the processes it controls and read and replace the contents of those processes' registers.

After instrumenting the main process and workers with `ptrace`, ASC sets the `ptrace` breakpoints. ASC sets the breakpoints to a specific presupplied target instruction pointer. The target instruction pointer breakpoint serves as a reference from which predictions will be made and cache lookups will be conducted. Every time the main process's execution reaches the breakpoint, ASC checks if the main process's state corresponds to a cache entry and fast-forwards the main process to the corresponding speculated state if it does. Then, ASC predicts the main process's state at future instances of the breakpoint and has worker processes speculatively execute from those predictions when they reach their breakpoints. This cycle is discussed in more detail in Section 3.3. Because the breakpoint is so important to ASC's operation, it must be on an instruction that the main process executes regularly during computation (but not too regularly, or overhead will be excessive). Optimal breakpoints are found manually for simpler programs and with the aid of Gaussian processes for more complex ones.

3.2 COMPONENTS OF ASC

Once all processes are initialized and instrumented, ASC is ready to begin operating. To better explain ASC's operation, we first describe its three key components: the machine learning model, the cache, and the scheduler:

- The machine learning model makes predictions about future states of the main process. The model takes as input the current state of the main process and returns as output a prediction of the state of the main process the next time it reaches the breakpoint.
- The cache caches correspondences between predicted and speculated states. A predicted state is a state predicted by the machine learning model, while a speculated state is a state generated by speculative execution from a predicted state on a worker process. The main process can look up its current state in the cache and a hit returns the (speculated) state of the next breakpoint.
- The scheduler determines from which (predicted) state the machine learning model should speculate. It uses the machine learning model to make predictions arbitarily far into the future to ensure that all workers are executing from different predicted states so they can all add new and useful entries to the cache.

Maintaining these data structures requires ASC to perform two key operations, gathering and scattering. Gathering and scattering are equivalent to reading states from processes and writing them to processes, respectively. They make use of the `ptrace` API as well as direct memory access. Direct memory access is a system implemented in bare metal and supported by Linux that allows a process to access main memory independent of the CPU. ASC uses it to rapidly read from or write to a child process's memory while allowing the CPU to do other useful work. To be more explicit:

- Gathering is the copying of a program's memory and register values into a buffer, called a state vector. The registers are copied into the buffer using the `ptrace` API, while the memory values are copied using direct memory access.
- Scattering is the copying of memory and register values from a buffer, called a state vector, into a program. Scattering replaces memory and register values of the program with corresponding values from the state vector. The registers are copied into the program using the `ptrace` API, while the memory values are copied using direct memory access.

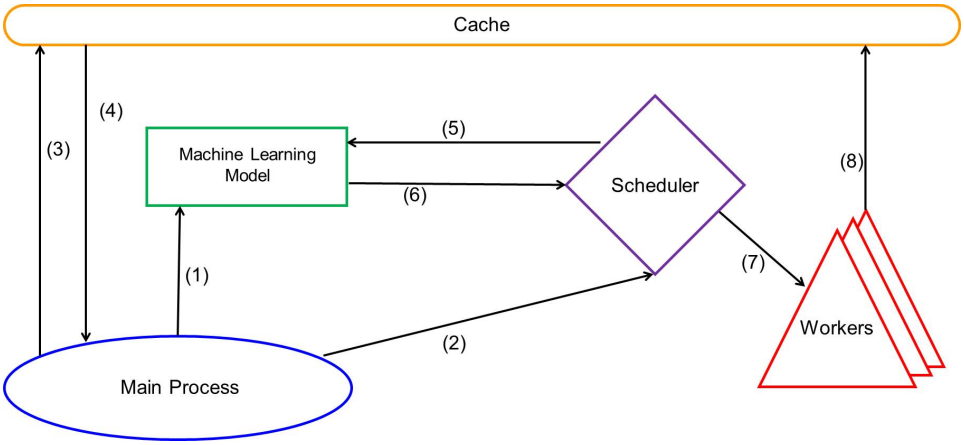

Figure 2: A diagram of ASC's design. Arrows represent flow of state. The main process uses its state to train the machine learning model (1) and provide a basis for prediction to the scheduler (2). It also queries the cache for predicted states identical to its current state (3). If the cache contains such a predicted state, it returns the corresponding speculated state to fast-forward the main process (4). The scheduler sends the main process's most recent state to the machine learning model (5) and receives predictions for future states of the main process (6). It then scatters these into the workers for speculation (7). When the workers are done speculating, they add the predicted and speculated states to the cache (8) for the main process to look up later.

3.3 Operation of ASC

With these key data structures and operations defined, we can explain how ASC operates. We diagram ASC's operation in Figure 2. As described earlier, ASC begins operation by initializing the main process and workers. It launches all of them normally as children, using `fork` and `exec`, then takes control of all of them using the `ptrace` API. It then uses `ptrace` to set the target instruction pointer as a breakpoint into the main process and into each worker, then sleeps and waits for processes to reach their breakpoints.

When a worker process reaches its breakpoint, ASC first gathers the process's state into a state vector. Then, ASC uses that state vector to update the cache. To do this, ASC checks what state the worker began computation from, then caches the correspondence between that previous (predicted) state vector and the current (speculated) state vector. Next, the scheduler uses the machine learning model to predict a future state of the main process from its last observed state. Then, ASC scatters that predicted state into the worker, using the `ptrace` API and direct memory access to copy the contents of the state vector into the appropriate registers and memory locations of the worker. Finally, ASC restarts the worker, beginning speculative execution from the new predicted state.

When the main process reaches its breakpoint, ASC first gathers the main process's state vector. Then, if the machine learning model is still being trained, ASC adds the correspondence between the recently gathered state vector and the

last gathered state vector from the main process to the model's training set. Next, ASC checks the cache to see if any of the predicted state vectors in the cache match the gathered state vector. If one does, ASC scatters the speculated state corresponding to that predicted state into the main process. This fast-forwards the main process into the speculated state. Then, ASC restarts the main process.

3.4 Implementation of ASC

Now that we have described how ASC is meant to operate, we will describe the actual implementations of the core data structures of the bare-metal version of ASC and the implications of those implementations on our new version of ASC.

- The machine learning model implemented in the bare-metal version of ASC is a shallow but fully-connected neural net. This net takes every bit in a process's current state as input and returns as output every bit of its predicted state the next time it reaches the breakpoint.
- The cache contains only a single slot. Every time a worker process reaches the breakpoint, it places its own predicted and speculated states into that slot. The main process checks only against that single slot.
- The scheduler is capable of scheduling only a single worker process. It simply has that process speculate from the most recent prediction made. This is the reason why the cache can operate with only a single slot.

As a result of these implementations, the bare-metal version of ASC is extremely limited in its functionality. Because it can only support a single worker process (while the abstract ASC design allows for arbitrarily many), it cannot exploit more than a single additional core in speeding up a target program. Moreover, because its machine learning model is a fully-connected neural net, it scales poorly with the size of the state vector and cannot handle programs with large state vectors. Therefore, to make our new data dependency system for ASC truly functional, we need to not just implement it but also improve the core systems of ASC—the machine learning model, the cache, and the scheduler—to fully exploit ASC's potential power.

3.5 Overview of Pin

The data-dependency tracking in our new version of ASC makes extensive use of Pin, a dynamic instrumentation tool developed by Intel (Luk et al., 2005). Pin uses a just-in-time (JIT) compiler to insert and optimize machine code. The JIT runs directly on the instruction set architecture (ISA) itself, compiling the original ISA to an instrumented ISA without any intermediate or any need to see the source code. The JIT maintains a code cache that stores the compiled ISA so that instrumentation only has to be done once. The target program's original code is never executed, only the instrumented code from the cache.

To instrument a program, Pin makes use of a Pintool, a package of routines that is compiled as a shared library and loaded by Pin. A Pintool contains two types of routines: analysis routines and instrumentation routines. Analysis routines do the actual useful work of the Pintool while instrumentation routines determine when analysis routines should be run. For example, in a Pintool that counts memory reads, the analysis routine might increment a counter, while the instrumentation routine would tell Pin to insert a call to the analysis routine after every unique memory read.

When an application is launched under Pin, the Pin binary first starts the target application in its own address space, then uses `ptrace` to take control of the application's execution, then loads the Pintool. The Pin binary then begins JITting the target application. Whenever Pin reaches an instruction it has not seen before, it calls the Pintool's instrumentation routines to determine which analysis routines should be run after that instruction. It then inserts calls to those analysis routines around the instruction, adding both the instruction and the analysis routine calls to the code cache. In the example of a Pintool that counts memory reads, if Pin reached a move instruction (which contains a single memory read), it would add after that instruction an increment instruction for the counter. The code cache entry for the move instruction would then become the move instruction followed by the increment instruction. After instrumentation is complete, the instruction and analysis routines are executed from the code cache. If the instruction is ever encountered again, Pin simply runs the instrumented code cache entry without any need to rerun the instrumentation routine.

4 IMPLEMENTATION

In this section, we discuss our improvements to ASC. First, we explain how we reworked ASC's three core systems: the machine learning model, the cache, and the scheduler. Next, we discuss the theory behind data dependency tracking and the details of how it is used in ASC. Then, we describe the implementation of the Pintool, which actually performs data dependency tracking. We also describe additional systems we have to implement within ASC to ensure consistency between the uninstrumented main process and instrumented worker processes. Finally, we describe how all these updated systems work together during ASC's operation.

4.1 MACHINE LEARNING MODEL

The bare-metal version of ASC uses a neural net as its machine learning model. Specifically, it uses a fully-connected net with a single hidden layer. The major advantage of using neural nets is that they can be trained online so that ASC can learn as it executes without the need to pause for batch training. However, nets have several disadvantages. They are opaque; one cannot reason about what causes nets to generate the output that they generate. They are also inflexible; the sizes of their input and output cannot readily be changed without complete retraining. Worse,

because the particular net the bare-metal version used is fully connected, the time complexity of prediction and training scale quadratically with the number of bits predicted. This makes the net too slow to be useful for large numbers of bits.

To correct these problems, we choose to use simple decision trees as our learners. We choose decision trees because the training set consists entirely of binary values (bits) with no noise. This avoids the two largest disadvantages of decision trees: their difficulty in appropriately splitting many-valued input and their tendency to overfit to noise. However, it does not affect their advantages, the most important of which is that because we train a separate tree for each output bit, the number of trees and thus time complexity of prediction and training increases linearly with the number of bits predicted (as our decision trees have a set maximum depth and therefore effectively constant size). This allows trees to massively outscale the quadratic neural nets. Beyond that, trees are fully transparent, allowing us to easily reason about a tree's behavior simply by examining it. This makes it much easier to determine whether ASC will work on a particular program and to find an optimal breakpoint instruction.

4.2 CACHE

The bare-metal version of ASC did not implement a true cache. This was because it was built to only support a single worker. Instead, it simply remembered the last speculated state the worker had returned along with the predicted state it had speculated from. When the main process reached the breakpoint, ASC checked the main process's state against that predicted state and, if they matched, fast-forwarded using the speculated state. This can also be thought of as a single-entry cache. Needless to say, this design cannot scale to multiple workers and cores. Moreover, it caused problems even with a single worker as potentially useful older predictions could be overwritten by newer ones.

To scale our system to many cores, we designed a true cache that allows for arbitrarily many entries. Our cache implements two functions, `cache_add` and `cache_lookup`. `cache_add` takes in a predicted state z_p, a speculated state z_s, and masks m_r and m_w of all bits that were read from and written to during the computation of the speculated state from the predicted state and adds the tuple (z_p, z_s, m_r, m_w) to the cache. `cache_lookup` takes in the main process state z_m and checks if any cache entry (z_p, z_s, m_r, m_w) contains a predicted state z_p which is identical to z_m in all bits read during the computation of z_s; that is, if:

$$z_m \wedge m_r = z_p \wedge m_r$$

If one does, it returns that entry's speculated state, else, it returns null. The reasoning behind this use of the read-mask is discussed in Section 4.4.

We decided to implement the cache as a hash table. Originally, the key for a tuple (z_p, z_s, m_r, m_w) would be a hash of $z_p \wedge m_r$, the bitwise AND of the predicted state and the read mask. We would look up a main process state z_m by hashing

$z_m \wedge m_r$, looking it up in the table, and if there was a match checking if $z_m \wedge m_r$ is fully identical to the returned $z_p \wedge m_r$. The problem with this, however, is that m_r is not part of z_m, it is instead a part of the hash table entries. Each entry has its own unique read mask. This means we could not do a single hash and lookup; instead, we would have to separately compute and look up the hash of $z_m \wedge m_r$ for every unique m_r in any entry of the table. This would make the speed of hash table lookup linear on the size of the table, which is less than ideal for a hash table.

To avoid this problem, we needed a fast hash function that did not depend on the read-mask and did a good job of distinguishing states. We realized that the way to develop such a hash function was to hash the values of the live registers. While the set of memory values that were read from might change between iterations of a program, the set of registers that are live at the breakpoint never changes. It is part of every m_r. Moreover, the values of the registers do a good job of discriminating states: for all programs we tested on, if the states of the live registers were identical, the states of the areas of memory under the read mask were probably also identical. Given the centrality of registers to all computations, this is likely true in general for nonadversarial programs. This allows us to implement the cache as a hash table where the "hash" of each tuple (z_p, z_s, m_r, m_w) is a simple hash function run over the values of the live registers of z_p. When we lookup a main process state z_m we simply hash its live registers and look up the hash in the table. This is equivalent to asking if there is an entry (z_p, z_s, m_r, m_w) where the register values of z_p and z_m are identical. If there is such an entry (z_p, z_s, m_r, m_w) we check if $z_m \wedge m_r = z_p \wedge m_r$. This is equivalent to asking if the memory values are also identical under the read-mask. If the equality is true, we return z_s (subject to modification using the procedure described in Section 4.4).

The astute reader may have recognized that it is sometimes necessary to do multiple sequential cache lookups. This occurs when the final modified state retrieved from the cache matches some other cache entry. Depending on the scheduler, this can happen an arbitrary number of times in succession. Originally, we solved this problem by having `cache_lookup` iterate. If it found a speculated state, instead of returning it would look up that speculated state as a predicted state and repeat until it had reached a speculated state that had no match in the cache. However, this solution became extremely expensive when state vectors became large as it required synchronous performance of a large number of whole-state-vector Boolean operations in the main process. To optimize this problem of iterated lookups, we have asynchronous threads optimize the cache. We search for pairs of entries (z_p, z_s, m_r, m_w) and (z'_p, z'_s, m'_r, m'_w) such that $z_s \wedge m'_r = z'_p \wedge m'_r$; that is, pairs of entries where the speculated state of the first matches the predicted state of the second under the read-mask of the second. If we find such a pair, we combine it into a single entry $(z_p, z_t, m_r \vee m'_r, m_w \vee m'_w)$ where z_t is the state constructed from z_p and z'_s using the procedure described in Section 4.4 and the read- and write-masks are combined to reflect the set of all bits read from and written to during the computation of z'_s from z_p. This means that `cache_lookup` does not have to iterate, whatever speculated state it finds the first time is the state it returns.

4.3 SCHEDULER

In the same way the bare-metal version of ASC did not implement a true cache, it did not implement a true scheduler. Again, this was because it was designed to only support a single worker. Every single time this worker reached the breakpoint, its predicted and speculated state would be added to the single-entry cache. Then the worker would be restarted with a new predicted state generated by the machine learning model from the most recent state of the main process. As we have stated, this system does not scale to multiple cores and workers and needed to be replaced.

To get scalability, we implemented a new scheduler for ASC. The goal of the scheduler is to assign different predictions to different workers so that every worker is usefully speculating about the future of the main process. We developed the idea of timesteps in ASC to implement a simple, effective, low-overhead scheduler. We define a timestep as the period of the main process reaching the breakpoint. The first time it reaches the breakpoint is the first timestep, the second is the second timestep, and so on. We can use timesteps to simplify the scheduler by recognizing that the predictions created by our machine learning model are not only general predictions for the future but also predictions for a specific timestep. The scheduler maintains a large table with an entry for each timestep (modulo a large number). An entry in the table is marked if a worker is currently speculating or has speculated about a prediction for that timestep. Every time a worker process is ready for scheduling, the scheduler assigns it to the Nth unmarked entry in the table, where N is a small number chosen to ensure the speculation completes before the main thread reaches the timestep. Assuming the main process was last observed in timestep t and the chosen entry is for timestep t', the scheduler then has the machine learning model iteratively predict from the main process state $t' - t$ times to create a prediction for timestep t'. Next, the scheduler scatters that predicted state into the worker process and restarts it. This ensures that no two workers are ever working on the same prediction and that every speculation from a predicted state at some timestep t finishes before the main process reaches timestep t.

4.4 DATA DEPENDENCY

The goal of the data-dependency tracking is to expand the set of programs on which ASC can achieve speedup by reducing the number of bits it must predict correctly. We first explain the model present in the bare-metal version of ASC, then explain our model and why it is guaranteed to be correct. In the bare-metal version of ASC, the cache consists of a single map $z_p \rightarrow z_s$, where z_p is a predicted state and z_s is the state speculatively executed from that predicted state. Because we assume programs are deterministic, this means that if the program is ever in state z_p, it must in the future enter state z_s. Therefore, if we observe that the main process is ever in state z_p, we can immediately fast-forward it to state z_s, allowing it to skip all the computation in between.

The data-dependency model extends this by introducing read and write masks.

Figure 3: A diagram of how cache queries use the read-mask. ASC is looking up main process state z_m in the cache. z_m is identical to cached predicted state z_p in their first byte (orange outline), but not in their second or third. However, the read-mask m_r tells us only the first byte of z_p was read during computation. Therefore, z_m and z_p match and the query is successful.

The cache consists of maps $z_p \rightarrow z_s$ along with, for each map, a read-mask m_r and a write-mask m_w. m_r and m_w are the sets of bits read from and written to, respectively, during the computation of z_s from z_p. Assume the main thread is in state z_m. We want to find state z_t, the state it will be in the next time it hits the breakpoint. If z_m were precisely equal to z_p, that would be easy, z_t would be z_s. That is the model described previously. However, the read-mask and write-mask allow us to do more than that. Assume that z_p and z_m are equivalent *under the read mask*; that is, ASC was able to successfully predict all bits that were read during the computation proceeding from z_p. In algebraic terms:

$$z_m \wedge m_r = z_p \wedge m_r$$

This is diagrammed in Figure 3. If the equation is true then the computations proceeding from z_p and from z_m are identical; bits that are not read from cannot affect the results of computations. This does not mean that z_t equals z_s, however, as there may be other bits not touched during the computation that differ between the two vectors. However, we can still derive z_t from z_s and z_m by using the write-mask. We recognize that the set of all bits that are changed between z_m and z_t is identical to the set of all bits changed between z_p and z_s, and that those bits are precisely the bits that are under the write-mask and take the values that they have in z_s. Therefore, to derive z_t, we simply set all the values in z_m that are under the write-mask to the appropriate values from z_s, while leaving the rest of the bits the same. To put this in algebraic form:

$$z_t = (z_s \wedge m_w) \vee ((z_m \vee m_w) \veebar m_w)$$

This is diagrammed in Figure 4. The equation is guaranteed to work, meaning that if we can dynamically generate read-masks and write-masks during the operation of worker processes, we need only correctly predict the bits that are under the read-

Figure 4: A diagram of how cache queries use the write-mask. ASC is looking up main process state z_m in the cache and has found a hit which has speculated state z_s and write-mask m_w. It now needs to construct true future state z_t. The write-mask tells us the only byte written to during the computation of z_s was the second one, so z_t is constructed with the first and third bytes of z_m (purple outlines) and the second byte of z_s (orange outline).

mask during a particular computation. For many important types of programs, this is a trivial fraction of the overall number of bits the program manipulates.

4.5 PINTOOL

Our system launches all workers using Pin and a custom Pintool. The Pintool contains only a single data structure, a large segment of memory that is shared with ASC, Pin's parent. This segment is large enough to contain copies of the registers, active areas of the program's memory, and a read-mask and write-mask each the same size as those active areas. We determine the active areas of memory and their size by parsing the main process's memory map in `/proc/pid/maps`. The Pintool also acts as a library that contains several instrumentation functions.

The Pintool contains instrumentation for instructions with certain properties. Pin automatically inserts into the worker process calls to the Pintool's instrumentation functions after any appearance of these instructions. Specifically, we instrument:

- Memory reads. Pin automatically instruments all instructions that read from memory, inserting multiple calls if the instruction reads from multiple locations. The instrumentation function for memory reads takes as an argument the address that was read from along with the size of the read (in bytes). It then updates the read-mask to mark those bytes as read from.

- Memory writes. Pin automatically instruments all instructions that write to memory, inserting multiple calls if the instruction writes to multiple locations. The instrumentation function for memory writes takes as an argument

the address that was written to along with the size of the write (in bytes). It then updates the write-mask to mark those bytes as written to.

- The breakpoint. We supply ASC's target instruction pointer to the Pintool and instrument the instruction it points to. The instrumentation function for the breakpoint takes as an argument a context data structure containing the values of all the worker process's registers (including flags registers) at the time the breakpoint is reached. The instrumentation function, when called, copies those register values as well as the contents of the program's active memory areas into the shared memory segment, then signals ASC and sleeps. While the instrumentation function sleeps, ASC gathers from the shared memory segment and then scatters into it, replacing the contents of the shared memory segment with new memory and register values from which it wishes to speculatively execute. ASC then signals the Pintool. The instrumentation function then wakes up and copies the new memory values back into their proper locations, clears the read- and write- masks, and restarts execution of the worker process using the new register values.

4.6 Memory and Stack Alignment

One assumption ASC makes is that the main process and workers are functionally identical. This allows ASC's notion of state to make sense. ASC relies on being able to copy the register and memory values of the main process into a worker process and have the worker process do exactly the same thing the main process would have done. This is actually quite restrictive; for example, ASC cannot be run with ASLR (address space layout randomization) because then the main and worker process stacks would be in different locations. If that occured, any attempt to copy the main process stack into the worker stack would either cause a write to invalid memory (if one tried to write the main process stack to the same addresses it had in the main process) or break every pointer into the stack (if one tried to write the main process stack over the worker process stack). This creates a potential risk for any attempt at instrumentation, as the instrumentation procedure must preserve the memory map of the worker in order for ASC to work.

Fortunately, the nature of Pin allows us to avoid this problem. Pin (and the Pintool) run in the same address space as the target program. When the target program is run under Pin, Pin is the first executable launched, but it quickly relocates itself (and the Pintool) elsewhere into the address space while allowing the target program's JITted instructions to run on the same stack and data segments as they would normally. This means that *any memory location that is valid in the main process is also valid in the worker (and belongs to the target executable, not to Pin or the Pintool)*. However, it also means that the arguments to Pin sit on the bottom of the worker's stack, so addresses in the stack are shifted down in the worker relative to the main process. This means the stacks of the main process and worker are unaligned and thus ASC cannot work.

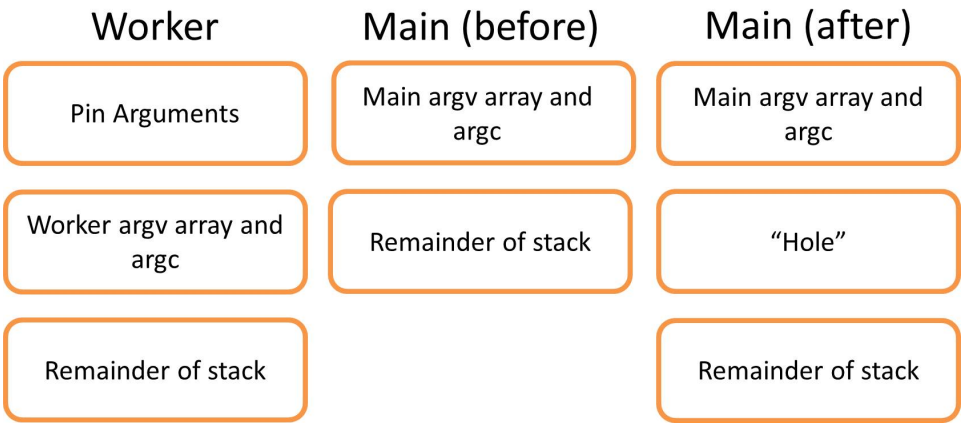

Figure 5: A diagram of the stack realignment described in section 4.6, with the convention that larger addresses are up and the stack grows down. The main process's stack is laid out like the middle column before the realignment and like the right column after the realignment. In each column, the stack pointer points to the top of the last box.

We solve this problem by aligning the stacks of the main and worker processes. We cannot do this by shifting the worker process's stack up, as then the pointers in the worker process's `argv` would not point to the correct strings (as they would have been overwritten in the upshift). Instead, we have to shift the stack of the main process down. To do this, we exploit an assumption made by ASC. This assumption is that programs run by ASC are compiled with a customized `crt0.o` that, among other things, stores the values of the `argc` and `argv` pointers, computed relative to the stack pointer `rsp`, in other registers before doing anything else. To exploit this fact, we start both the main and worker process in a suspended state, where no instructions have been executed and the only registers set are the instruction and stack pointers. We then allow the main process to run until it has set both the `argc` and `argv` pointers. Next, we set the main process's stack pointer to that of the worker. We do not actually modify the stack. This has the effect of creating a "hole" in the stack of the main process between `argc` and `argv` (whose locations the program still knows because those pointers were not modified) and the actual program stack. A diagram of this is shown in Figure 5. This ensures ASC's assumption of identical memory layouts holds, with the sole and unlikely exception of a program that directly manipulates the `argv` pointers in the loop containing the breakpoint. If this occurs, the worker will crash and be terminated and ASC will execute the main process as normal, but without producing speedup.

4.7 ASC Operation Summary

In the new system, ASC continues to operate much as described in the introduction and in Section 3. The three key data structures of ASC remain the machine learning model, the cache, and the scheduler. Each uses the new designs described

in sections 4.1, 4.2, and 4.3, respectively.

When ASC is run, it begins by launching the main process using `fork`, `exec`, and `ptrace` as normal. ASC then launches a number of workers using Pin and the Pintool described in Section 4.5. After that, ASC adjusts the stack of the main process using the alignment procedure described in Section 4.6. Then, ASC uses `ptrace` to set a breakpoint on the target instruction pointer in the main process, configures a signal handler to intercept signals from the workers, and goes to sleep waiting for processes to reach their breakpoints.

When a worker process hits the breakpoint, it sends a signal to wake ASC up, then goes to sleep. When woken, ASC first gathers the worker process's state, read-mask, and write-mask from the shared memory described in Section 4.5. Then, it uses those vectors to update the cache. Next, the scheduler uses the machine learning model to predict a future state of the main process from its last observed state following the timestep procedure described in Section 4.3. Then, it scatters that predicted state into shared memory for the worker and signals the worker to wake up, beginning speculative execution from the predicted state.

When the main process hits the breakpoint, ASC first gathers the main process's state. Then, if the machine learning model is still being trained, ASC adds the correspondence between the recently gathered state and the last gathered state to the model's training set. Next, ASC queries the cache to see if the gathered state matches any of the predicted states in the cache under their read-masks. If one does, ASC constructs the true future state using the procedure described in Section 4.4, then scatters it into the main process. Then, ASC restarts the main process.

5 EXPERIMENTS

We evaluate the performance of the new version of ASC. First, we describe the kernels we use for evaluation and the conditions in which we run them. Next, we prove that data-dependency tracking can massively reduce state space sizes for kernels that need it. Then, we explain the metrics we will use to evaluate ASC's performance. After that, we analyze the instrumentation overhead that comes from using Pin. Then, we evaluate the actual speedup that the new version of ASC can achieve. After that, we analyze the reasons why this speedup is not optimal. We demonstrate that for most kernels, the only barrier between our results and perfect 100%-per-core speedup (other than unavoidable factors such as Amdahl's Law) is Pin instrumentation overhead. Finally, we show that speedup increases linearly with the number of available cores.

5.1 KERNELS

Our experiments use a variety of kernels, small test programs that we try to speed up with ASC. We have six kernels total. The first three are adapations of the three originally used in Waterland et al. (2014):

- `collatz`: A kernel that iterates through a range of positive integers, testing

if each satisfies the Collatz conjecture. The Collatz conjecture states that if one starts with some positive integer n and sequentially divides it by two if it is even and multiplies by three and adds one if it is odd, the sequence will eventually converge to one. ASC parallelizes it by having different workers test the Collatz conjecture on different numbers. It is interesting because ASC automatically memoizes the results of the collatz conjecture, reusing the same speculation multiple times and thus outperforming conventional parallelization techniques (Waterland et al., 2014).

- `ising`: A pointer-based condensed matter physics program. It iterates through a linked list of spin configurations, identifying the element in the list with the lowest energy state. It is interesting because existing paralleliz-ing compilers will not parallelize it (Waterland et al., 2014). It is parallelized by predicting values referencing later nodes of the linked list.

- `mm`: Naive integer matrix multiplication. It is adapted from the `2mm` multiple matrix multiply kernel in Polybench/C reported by Waterland et al. (2014). It is parallelized by having different workers compute different sections of the final matrix. Unlike the previous kernels, it is extremely memory-intensive, with memory use scaling quadratically with input. It therefore tests the overhead of ASC on high-memory computations (Our experiments used 4000x4000 matrices, meaning each worker process used over a gigabyte of memory).

The next kernel is adapted from Polybench/C:

- `cov`: Covariance matrix calculation. It is adapted from the `covariance` kernel in Polybench/C. It is parallelized by having workers compute different sections of the final matrix. Like `mm`, it is extremely memory-intensive. Like `3sum`, it has nonuniform distance between breakpoints (the breakpoint is reached more frequently over time) because the final covariance matrix is symmetric, so only its upper triangle needs be calculated.

The remaining two kernels we developed ourselves. The first does not require data dependency tracking; the second does:

- `3sum`: A naive cubic solver for the 3-subset sum problem. It generates a large random list of integers and iterates through it looking for a set of three integers that sum to zero. It is parallelized by having different workers test different sets of integers. It has nonuniform distance between breakpoints (the breakpoint is reached more frequently over time) because the size of the search space decreases over time.

- `readmap`: A simple but extremely computationally intensive map operation that repeatedly adds random values to entries in an array. It is parallelized by having different workers perform the map on different elements of the

Kernel	Bits Before	Bits After
`3sum`	80	80
`readmap`	2520	120
`collatz`	224	224
`ising`	280	280
`mm`	1024000072	72
`cov`	576000352	352

Table 1: Sizes of kernel state spaces before and after state space reduction though data dependency tracking.

array. This program is a demonstration of an embarassingly parallel problem that ASC should be able to speed up, but cannot speed up without data dependency tracking.

5.2 Methodology

We ran all experiments on a Microway 1U Xeon server with two Intel Xeon E5-v4 processors. Each of the two processors has 22 cores, for a total of 44. We disabled Intel Turbo Boost on all cores to guarantee consistent clock speeds between cores. The server has 256 GB of main memory. We designed all kernels to randomize behavior between runs (for example, `mm` randomly generates its matrix every run). This randomization is to demonstrate that ASC is actually making predictions about states and not simply memorizing end states. The exception is `collatz`, whose nature is not amenable to randomization. We experimented on `collatz` by training it on one set of numbers and testing it on a different set of numbers. We found all breakpoint instruction pointers manually. In addition to this, `ising` and `mm` required register liveness analysis to determine which registers were dead and did not need their values predicted; we also did this manually. We trained all decision trees offline. Prior to experiments being run, we ran each kernel twenty times on different inputs to generate a training set from which we trained decision trees. We then used these trees for all experiments.

5.3 State Space Size Reduction

We demonstrate the effectiveness of data dependency tracking at reducing the size of the state space of our kernels. We define the size of the state space of a kernel as the total number of bits whose values ASC must predict. We report our results in Table 1. Three of the kernels (`collatz`, `ising`, and `3sum`) do not need data dependency tracking because they do not modify large areas of memory. `3sum` and `ising` look for list entries with certain properties, so they perform huge numbers of reads but few writes. `collatz`, meanwhile, performs only arithmetic. All three kernels could be sped up by the bare-metal version without data dependency

tracking, although the amount of speedup was limited by the bare-metal version's lack of a true cache or scheduler.

The other three kernels, `readmap`, `mm`, and `cov`, experience large reductions in the sizes of their state spaces. In the case of `readmap`, this is due to the data dependency tracking realizing that the bits in the array to which the map writes do not need to be predicted. This is identical to the squaring map example in Section 1.3. The only bits that have to be predicted are loop counters. Much the same is true for `mm` and `cov`, where the size reduction is even more extreme. We had `mm` multiply two 4000 x 4000 matrices of `longs`, meaning the output was over a gigabyte in size. However, because matrix multiplication does not depend on its own output, predicting elements of the output matrix is unnecessary. The bare-metal version does not realize this, with disastrous results when it tries to predict the values of a billion bits. However, the new version of ASC recognizes that only the bits in the loop counters have to be predicted correctly, leaving only 72 bits to predict, regardless of the size of the matrix. In `cov`, the same is true, but with a slightly smaller matrix (3000 x 3000).

5.4 Methods of Analysis

Our analysis of ASC's performance uses two key metrics. The most important of these is speedup. Speedup is defined as the time it takes for a kernel to run natively divided by the time it takes to run under ASC. This is the core measure of how well ASC performs on a given program. A closely related metric is the computational cache hit rate. Computational cache hit rates are defined as the percentage of lookups (including both synchronous and asychronous iterated lookups as per Section 4.2) into ASC's computational cache that hit. Computational cache hit rates can also be thought of as the percentage of computation done on the workers instead of on the main process. A computational cache hit rate of 80% means that the workers did 80% of the overall computation. Computational cache hit rates can be used to estimate speedup. For example, a computational cache hit rate of 80% implies 5x speedup because only 20% of the overall computation is done by the main process while the remaining 80% is done "for free" in parallel by workers on auxiliary cores. High levels of overhead can cause speedup to be less than that estimate, however.

Crucial to interpreting speedup and computational cache hit rate is the notion of *CPU efficiency*. CPU efficiency is a measure of the overhead of the Pintool. We perform data-dependency tracking by instrumenting workers using Pin, which inserts analysis code after all reads and writes to record the locations read from and written to. While this is effective, it is also slow; we have to insert multiple analysis instructions after every instruction that reads from or writes to memory. As a result, all of the workers (but not the main process, which does not need instrumentation) run much more slowly than they would uninstrumented. We call the ratio of a kernel's uninstrumented runtime to its instrumented runtime the kernel's CPU efficiency. A kernel's CPU efficiency is inversely proportional to the percentage of

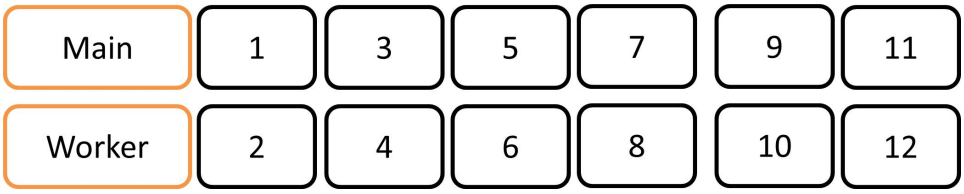

Figure 6: An example of how ASC works with a single worker with 100% efficiency. The numbers in the boxes are timesteps. The main process computes the first timestep while the worker computes the second. This lets the main process fast-forward to the third timestep and the worker compute the fourth. Then the main process can fast-forward to the fifth time-step while the worker computes the sixth, and so on. As a result, the main process finishes the computation twice as quickly as it would natively, achieving 2x speedup with a 50% computational cache hit rate.

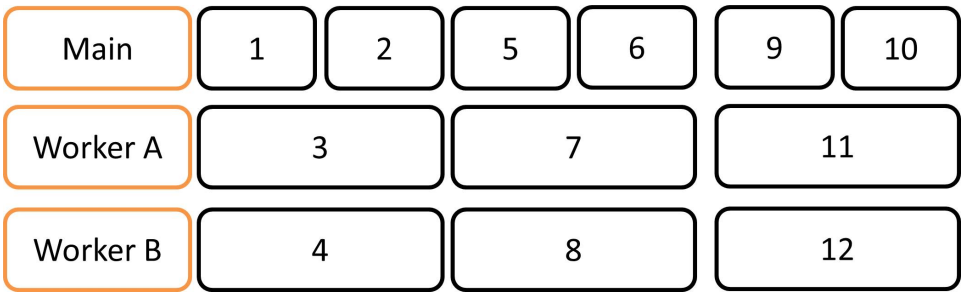

Figure 7: An example of how ASC works with a pair of workers each with 50% efficiency. The numbers in the boxes are timesteps. The main process computes the first and second timesteps while the workers compute the third and fourth. This lets the main process fast-forward to the fifth timestep and compute both it and the sixth timestep while the workers compute the seventh and eighth. Then the main process can fast-forward to the ninth time-step while the workers computes the eleventh and twelfth, and so on. As a result, the main process finishes the computation twice as quickly as it would natively, achieving 2x speedup with a 50% computational cache hit rate.

its executed instructions that are reads or writes because only reads and writes are instrumented.

CPU efficiencies are important because they determine the maximum amount of speedup ASC can achieve with a given number of cores. Assuming ASC is supplied with n workers each with a CPU efficiency of e_c, ASC can attain a maximum speedup of $ne_c + 1$. We demonstrate this by example in Figures 6 and 7. Figure 6 shows ASC operating on a program with a single worker with 100% efficiency. Every time the main process reaches the breakpoint, ASC fast-forwards one timestep using the result of the worker's last speculation. This involves two cache lookups, the first of which succeeds at finding the result of the speculation and the second of which tries to iterate from the speculation in the cache as explained in Section 4.2 but fails. As a result, the main process finishes the computation twice as quickly as it would natively, achieving 2x speedup with a 50% computational cache hit rate.

Kernel	CPU Efficiency
`3sum`	15.3%
`readmap`	16.1 %
`collatz`	31.7 %
`ising`	65.4 %
`mm`	10.7%
`cov`	16.9%

Table 2: CPU efficiencies of all kernels. CPU efficiencies are defiend as the ratio of the runtime of the kernel when uninstrumented to when instrumented.

Figure 7, for comparison, shows ASC operating on a program with two workers each with 50% efficiency. Because each worker is half as fast as the main process, ASC can only fast-forward the main process every other time it reaches the breakpoint. However, because there are two workers, each time it does fast-forward, it fast-forwards by two timesteps. As a result, the main process still finishes the computation twice as quickly as it would natively, achieving 2x speedup with a 50% computational cache hit rate. However, it took two workers with 50% efficiency to achieve this speedup, while a single worker with 100% efficiency could achieve this speedup on its own.

In the remainder of the experiments section, we analyze the performance of ASC both absolutely and relative to CPU efficiencies. First, in Section 5.5, we report the CPU efficiencies for each kernel. Next, in Section 5.6, we report computational cache hit rates attained by ASC on all kernels. Finally, in Section 5.7, we report speedups on all kernels and compare them to the kernels' CPU efficiencies. We show that while the relatively low CPU efficiencies limit the raw speedup we can achieve, our speedups for most kernels are close to the theoretical maximums given our CPU efficiencies. This means that ASC's speedups are only limited by the instrumentation overhead from Pin, and if that can be eliminated (likely through hardware support), ASC can reach its goal of near-linear automatic speedup.

5.5 CPU Efficiency

We report CPU efficiencies for all kernels in Table 2. As we explained in Section 5.4, CPU efficiencies are the ratios of kernels' uninstrumented runtimes to their instrumented runtimes. In other words, they are the ratio of the (instrumented) workers' runtimes to the (uninstrumented) main process's runtime. They provide an upper bound on the amount of speedup ASC can achieve. Assuming ASC is supplied with n workers each with a CPU efficiency of e_c, ASC can attain a maximum speedup of $ne_c + 1$ (for example, as we showed in Figure 7, two workers each with 50% efficiency can attain a maximum speedup of 2x).

While the CPU effiencies are relatively low, this is only a reflection of ASC's newness and immaturity as a technology. Pin is an effective tool for demonstrating

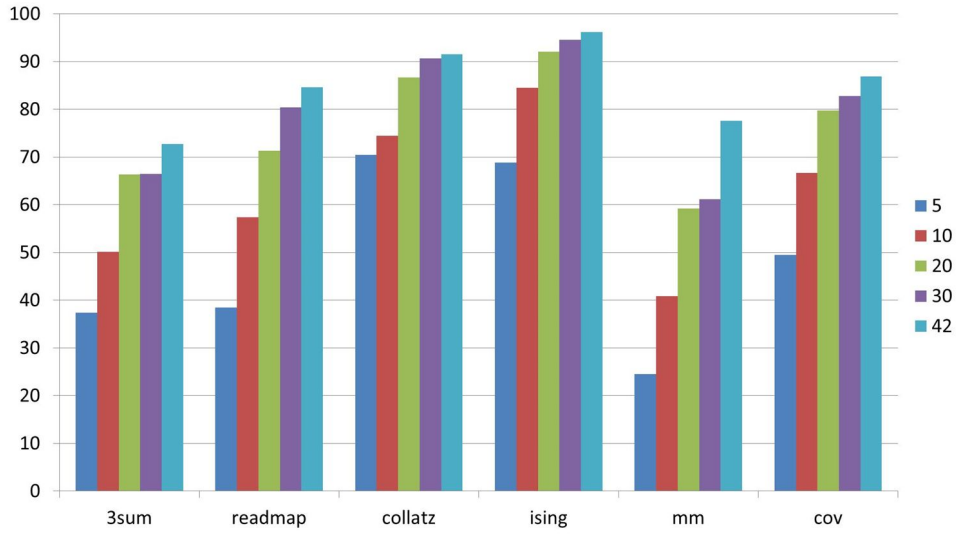

Figure 8: A diagram of computational cache hit rates for all kernels given various numbers of workers. Computational cache hit rates are defined as the percentage of lookups into ASC's computational cache that hit.

that data-dependency tracking can work on real hardware and achieve real speedup, but it is not an optimal way of performing data-dependency tracking. More sophisticated future generations of ASC will be able to integrate themselves more tightly into the virtual memory system to automatically record data dependency operations without incurring as much overhead. This could take the form of software-level instrumentation working within the operating system's virtual memory layer or of hardware instrumentation that automatically flags addresses as read from or written to and allows easy querying and resetting of those flags.

5.6 Computational Cache Hit Rates

We now report computational cache hit rates on all kernels in Figure 8. We report hit rates for 5, 10, 20, 30, and the full complement of 42 workers (reserving one of our server's 44 cores for the main process and another for ASC itself).

For four of the kernels—`readmap`, `collatz`, `ising`, and `mm`—the computational cache hit rates represent exactly what they ought to, the percentage of computation done by the workers instead of by the main process as explained in Section 5.4. For three of these, the computational cache hit rate serves as a good estimator of the speedup we report in Section 5.7 (the exception, `mm`, achieves less speedup than one would expect due to complications from its massive memory usage, as we explain in Section 5.7). Out of these kernels, `ising` and `collatz` have the highest computational cache hit rate because of their high CPU efficiency and `collatz`'s auto-memoization, which allows it to reuse the results of earlier speculations. `readmap` performs slightly worse. `mm` has a lower computational

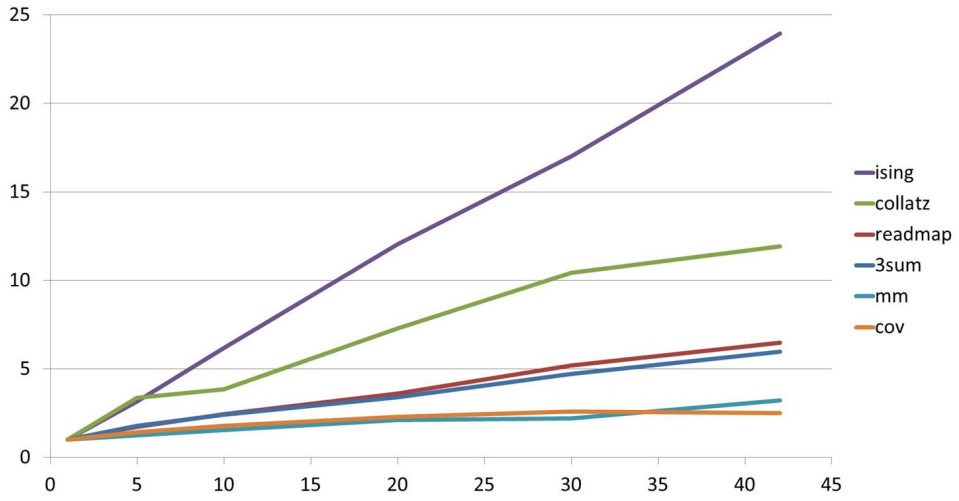

Figure 9: A diagram of speedups attained by all kernels given various numbers of workers. Speedup is defined as the time it takes for a kernel to run natively divided by the time it takes to run under ASC.

cache hit rate than any of the preceding three because of its low CPU efficiency.

For the remaining two kernels, `3sum` and `cov`, the computational cache hit rates are somewhat misleading because in both kernels the distances between breakpoints are non-uniform. Both programs reach the breakpoint more frequently as they run. In `3sum` this occurs because the size of the kernel's search space shrinks. In `cov` this occurs because the height of the column of the triangular matrix the kernel calculates shrinks. Because the amount of computation done between breakpoints is non-uniform, the linear relation between computational cache hit rate and speedup breaks down. In `cov`, the computational cache hit rate is an overestimate of the amount of speedup achieved because later workers add a large number of cache entries that correspond to little computation but still register as cache hits. In `3sum`, which reaches its breakpoint more frequently than `cov`, the exact opposite happens, causing the computational cache hit rate to underestimate the amount of speedup. Towards the end of computation, it queries the cache more quickly than the workers can fill it. This causes a large number of misses, but does not reduce speedup because al those cache misses are for trivial amounts of computation.

5.7 Speedup and Analysis

We now report our main results, the speedups attained by ASC on all kernels. We report raw speedups in Figure 9. This figure shows the speedups attained by all kernels given 5, 10, 20, 30, and the full complement of 42 workers (reserving one of our server's 44 cores for the main process and another for ASC itself). While the differences between the kernels' speedups may appear large, they correspond closely to the differences in CPU efficiencies reported in Section 5.5.

To clarify this correspondence, we compare in Figure 10 the speedups attained by ASC to the maximum possible speedups given the kernels' CPU efficiencies. As explained in Section 5.5, the maximum possible speedups are extrapolations from the CPU efficiencies assuming that n workers each with efficiency e_c can attain a maximum speedup of ne_c+1 (for example, as we showed in Figure 7, two workers each with 50% efficiency can attain a maximum speedup of 2x). We report these speedups for every kernel in Figure 10. In Figure 10, the blue lines are the speedups we actually achieve, while the green lines are the theoretically maximal speedups.

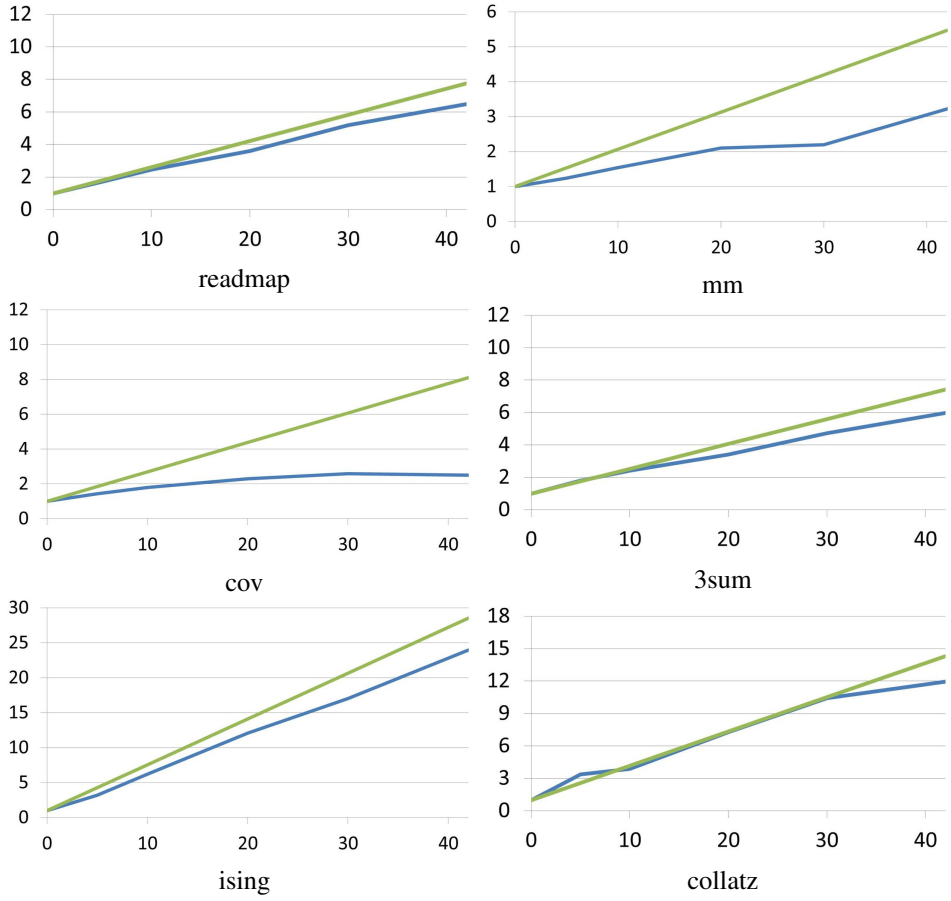

Figure 10: Diagrams of the actual speedups achieved on all kernels (blue) relative to the maximum speedups possible given the the kernels' CPU efficiencies (green).

As we can see from Figure 10, two kernels perform somewhat worse than the other four relative to their theoretical maximum speedups. These two kernels are `cov` and `mm`. Both perform worse than expected due to their gigantic sizes in memory. `mm` uses well over a gigabyte and `cov` several hundred megabytes per worker. The overhead incurred from gathering, scattering, and performing computational cache operations on these gigantic memory spaces causes `cov` and `mm` to attain less speedup than they would otherwise. This is worsened by slowdown

from CPU cache contention between all of the workers, making them compute somewhat slower than they would otherwise. In `cov` these problems are exacerbated by the nonuniform distance between breakpoints explained in Section 5.6, which makes optimal scheduling of workers difficult and wastes large amounts of worker time.

On the other hand, the remaining four kernels perform extremely well. These four are `readmap`, `3sum`, `ising`, and `collatz`. All four have speedups within a small factor of the theoretical maximum given their CPU efficiencies. Two of these kernels have especially interesting results. `Collatz` actually attains more speedup than the "theoretical maximum" for small numbers of cores because its auto-memoization allows it to reuse speculations. `Ising`, meanwhile, attains near-optimal speedup despite its extremely high CPU efficiency, proving that ASC does not rely on its kernels having poor CPU efficiencies to scale and will still scale extremely well once CPU efficiencies are improved. All four kernels' speedups are exactly what we expected given their computational cache hit rates, once we account for `3sum`'s artifically poor hit rate as explained in Section 5.6. This proves that ASC fundamentally works–accounting for the CPU efficiencies of the workers, it is able to achieve speedups comparable to other parallelization systems on kernels with reasonable memory overhead. More importantly, speedups on all four kernels (and `mm`) scale near-linearly with an increasing number of cores. This demonstrates that ASC can achieve its goal of providing near-linear automatic parallelization for large classes of programs.

6 Conclusions and Future Work

6.1 Summary

In this thesis, we present a new version of ASC, a powerful system for automatic parallelization. We prove that ASC is capable of working on unmodified binaries without the use of static analysis, giving it more potential power than other automatic parallelization systems. We demonstrate ASC's ability to automatically parallelize a variety of kernels. These kernels include fundamental paralellizable computations such as maps or matrix multiplication along with problems other automatic parallelization systems struggle with such as linked list navigation and auto-memoization. Moreover, we show that ASC's speedups scale linearly with the number of cores, allowing it to work efficiently on systems with large numbers of cores. This makes ASC potentially useful for many scientific computing applications, where researchers often not trained in parallelization techniques need to run extremely computationally intensive programs on extremely powerful computers.

6.2 Future Work

Moving forward, a great deal of work can be done to improve ASC and make it an even more powerful tool. Future avenues of research include:

- Implementing an automatic breakpoint recognizer and live register analyzer. For all our experiments, as was discussed in Section 5.2, both breakpoint recognition and live register analysis were done manually. Performing them automatically, however, is possible. Live register analysis is well-understood, though no tools yet exist for easily performing it on a compiled binary as we need to. LASC had a breakpoint recognizer that used Gaussian processes, and while it has not been implemented in the new version of ASC yet, a version of it will be included in the future. Once both breakpoint recognition and live register analysis are done automatically, ASC will be fully automatic and will be able to speed up a parallelizable binary with no user input.

- CPU efficiency improvements. As was discussed in Section 5.5, Pin is an effective but suboptimal way of performing data dependency analysis. Future versions of ASC will do this in a more efficient way, either through better integration with the virtual memory system or, looking forward, with specialized hardware.

- Further state space reduction. Data-dependency tracking dramatically reduces the size of the state space for some kernels, as Section 5.3 makes clear, but it does not actually minimize it. As a result, there are many parallelizable programs ASC cannot speed up. For example, ASC cannot speed up an accumulator that sums the elements in an array because that requires predicting the values of the sum. This could be avoided by giving ASC more information about what operations the computation was actually performing on individual registers or areas of memory. In the case of the accumulator, if we could tell ASC the value of the sum was never queried but only added to itself (an associative and commutative operation), ASC would realize that it did not actually need to predict the values of the sum, only calculate intermediate values and add those to each other.

- Optimization. As was discussed in Section 5.7, ASC's performance relative to theoretical maximums is good, but it can always be better. With more optimized code for major operations such as scattering, gathering, and predicting, we can reduce the constant factor of our linear time-dependence on the memory consumption of worker processes. This will dramatically improve performance on high-memory kernels such as `mm`.

The completion of any of these will make ASC far more powerful, the development of all of them—which is completely possible—will make ASC a fully mature technology capable of outperforming and outscaling any other automatic parallelizer.

REFERENCES

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

Bill Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, et al. 1994. Polaris: The next generation in parallelizing compilers. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, pages 141–154.

Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. 1995. *Cilk: An efficient multithreaded runtime system*, volume 30. ACM.

Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 265–275. IEEE.

Simone Campanoni, Timothy Jones, Glenn Holloway, Gu-Yeon Wei, and David Brooks. 2012. The helix project: overview and directions. In *Proceedings of the 49th Annual Design Automation Conference*, pages 277–282. ACM.

Leonardo Dagum and Ramesh Menon. 1998. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55.

Ben Hertzberg. 2009. *Runtime Automatic Speculative Parallelization of Sequential Programs*. Ph.D. thesis, Stanford University.

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

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, volume 40, pages 190–200. ACM.

Mojtaba Mehrara. 2011. *Compiler and Runtime Techniques for Automatic Parallelization of Sequential Applications*. Ph.D. thesis, Microsoft Research.

Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM.

Arun Raman, Hanjun Kim, Thomas R Mason, Thomas B Jablin, and David I August. 2010. Speculative parallelization using software multi-threaded transactions. In *ACM SIGARCH computer architecture news*, volume 38, pages 65–76. ACM.

Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. 2003. Hybrid analysis: static & dynamic memory reference analysis. *International Journal of Parallel Programming*, 31(4):251–283.

J Greggory Steffan, Christopher B Colohan, Antonia Zhai, and Todd C Mowry. 2000. A scalable approach to thread-level speculation. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 1–12. ACM.

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

Amos Waterland, Elaine Angelino, Ryan P Adams, Jonathan Appavoo, and Margo Seltzer. 2014. Asc: Automatically scalable computation. In *ACM SIGPLAN Notices*, volume 49, pages 575–590. ACM.

Jing Yang, Kevin Skadron, Mary Lou Soffa, and Kamin Whitehouse. 2011. Feasibility of dynamic binary parallelization. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*.