Computational Caches

Citation

Waterland, Amos, Elaine Angelino, Ekin D. Cubuk, Efthimios Kaxiras, Ryan P. Adams, Jonathan Appavoo, and Margo Seltzer. 2013. "Computational Caches." In Proceedings of the 6th International Systems and Storage Conference on - SYSTOR '13, June 30 - July 02, 2013, Haifa, Israel, 8. doi:10.1145/2485732.2485749.

Published Version

10.1145/2485732.2485749

Permanent link

Terms of Use

Share Your Story

Computational Caches

Amos Waterland[1] Elaine Angelino[1] Ekin D. Cubuk[1] Efthimios Kaxiras[2,1]
Ryan P. Adams[1] Jonathan Appavoo[3] Margo Seltzer[1]

[1] *School of Engineering and Applied Sciences, Harvard University*
[2] *Department of Physics, Harvard University*
[3] *Department of Computer Science, Boston University*

ABSTRACT

Caching is a well-known technique for speeding up computation. We cache data from file systems and databases; we cache dynamically generated code blocks; we cache page translations in TLBs. We propose to cache the act of computation, so that we can apply it later and in different contexts. We use a state-space model of computation to support such caching, involving two interrelated parts: speculatively memoized predicted/resultant state pairs that we use to accelerate sequential computation, and trained probabilistic models that we use to generate predicted states from which to speculatively execute. The key techniques that make this approach feasible are designing probabilistic models that automatically focus on regions of program execution state space in which prediction is tractable and identifying state space equivalence classes so that predictions need not be exact.

Categories and Subject Descriptors

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

General Terms

Performance,Theory

1. INTRODUCTION

Caching has been used for decades to trade space for time. File system and database caches use memory to store frequently used items to avoid costly I/O; code caches store dynamically instrumented code blocks to avoid costly reinstrumentation; TLBs cache page translations to avoid costly page table lookups. Caches are used to both speed up repeated accesses to static data and to avoid recomputation by storing the results of computation. However, computation is dynamic: an action that advances a state, such as that of a program plus associated data, to another state. We pose the question – can we cache the *act* of computation so that it can be applied repeatedly and in new contexts?

Conventional caches have a crucial property that we would like to replicate in a computational cache: the ability to load their contents at one time and place but use their contents repeatedly at different times and places. There are several ways to exploit a computational cache having this property: we could load the cache and then use it later on the same machine, or we could load the cache using large, powerful computers and stream it to smaller, more resource-constrained computers where the cost of a cache lookup is less than the cost of executing the computation.

Our computational cache design arises from a model of computation that we originally developed to extract parallelism from sequential programs [30]. In this model, computation is expressed as a walk through a high-dimensional state space describing the registers and memory; we call such a walk a trajectory. The model is practical enough that we have built a prototype $x86$ virtual machine and in current work have successfully used it to automatically parallelize programs with obvious structure. The virtual machine embeds in its execution loop learning, prediction, and parallel speculative execution; we call this trajectory-based execution.

Computational cache entries are stored pieces of observed trajectories. The entries can be pieces of trajectories we encountered during normal execution or during speculative execution. In either case, we can think of these entries as begin-state and end-state pairs – any time the virtual machine finds itself in a state matching the begin-state of some cache entry, it can immediately speed up execution by using the cache entry to jump to the corresponding end-state and then continue normal execution.

In the rest of this paper we summarize our model of computation, present the specifics of our computational cache, address the dual challenges of representing cache entries efficiently and allowing each entry to be reused in as many different ways as possible, analyze the relationship between the predictive accuracy of a cache's models and the resultant speedup, and report experimental results.

2. MODEL OF COMPUTATION

The two key ideas from trajectory-based execution that give rise to computational caching are: representing the state of computation as a state vector and representing the execution of an instruction as applying a fixed transition function. A state vector contains all information relevant to the future of the computation. A transition function de-

SYSTOR '13, June 30 - July 02 2013, Haifa, Israel
Copyright 2013 ACM 978-1-4503-2116-7/13/06 ...$15.00.

pends only on an input state vector to deterministically produce an output state vector.

Consider the memory and registers of a conventional computer as defining a very large state space. Applying the transition function executes one instruction, deterministically moving the system from one state vector to the next state vector. Program execution thus traces out a trajectory through the state space. For the moment, we exclude external non-determinism and I/O – we assume that execution begins after all input data and the program have been loaded into memory. Programs can still use a pseudo-random number generator; we simply require that it be seeded deterministically. Given these constraints, execution is a memoryless process – each state depends only upon the previous state. This means that any pair of state vectors known to lie on the same trajectory serve as a compressed representation of all computation that took place in between; our computational cache design arises from this fact.

Our prototype $x86$ virtual machine is a parallel program that has hard-wired into its execution loop the tasks of (1) learning patterns from the observed state vector trace of the program it is executing, (2) issuing predictions for future states of the program, (3) executing the program speculatively based on the predictions, (4) storing the speculative execution results in our computational cache, and (5) using the computational cache to speed up execution. The virtual machine smoothly increases or decreases the scale of these tasks as a function of the raw compute, memory, and communications resources available to it. More raw resources cause the virtual machine to query and fill a larger computational cache, learn more complex models, predict further into the future, and execute more speculations in parallel.

As aggressive as these ideas sound, we have previously demonstrated that we can use them in limited cases to automatically speed up certain classes of binary programs [30]. This paper addresses a broader question – how can we go beyond speeding up a single computation to reusing work performed in the context of one program to improve the performance of a different program?

We examine two ways to reuse and generalize information gained from executing a program. The first is to extract the symmetries of a stored trajectory – constraints which if satisfied allow pieces of many other distinct trajectories to be considered equivalent to it and thus able to be sped up by it. For example, the same library function can be called by two different programs. The second is to extract the statistical patterns of a stored trajectory – which when encoded as the parameters of a probabilistic model allow accurate predictions for future states of a wide class of similar programs. For example, similar machine code generated for two different programs allows a predictor trained on one to be immediately useful for speculative computation on the other.

3. CACHE DESIGN

Our computational cache stores two kinds of objects: compressed state vector pairs along with their symmetries, and the parameters of trained probabilistic models. To fill our cache, we take in a sequence of vectors produced by executing some program for some useful number of instructions, and insert in the cache only the first and last vector as a pair. By construction in a state-space model, we lose no information by forgetting the intermediate vectors, no matter how long the sequence. The two vectors are themselves often highly compressible, so in practice a billion-instruction sequence of megabit state vectors often results in just a kilobit cache entry.

Our execution loop continually queries our cache for a pair whose first vector matches our current vector. If the cache hits, we immediately transform our current vector into the second entry of the pair, speeding up computation by fast-forwarding through potentially billions of instructions. If the cache misses, we execute one instruction by applying the transition function as normal, then try again. Sequences of state vectors usually have many symmetries, so our current vector need only match along the causal coordinates of a stored pair in order to get a cache hit. Matching is conservative: we tolerate false negatives in order to guarantee that we will never have false positives. Cache hits speed up execution but produce the exact same computation as normal, and cache misses just result in normal execution with query overhead.

There are characteristics of both execution and the state space that we exploit in our computational cache: trajectories are not random, each instruction modifies only small portions of the state, and even long sequences of instructions frequently depend upon only small portions of the state. Programs are frequently constructed out of components: functions, library routines, loops, etc. Within each one of these constructs, execution typically depends on only a small portion of the space and modifies only a small portion of the space. We'll use these observations to construct useful entries in a computational cache.

Let's begin by identifying some useful characteristics of a cache entry and then move on to techniques for producing such entries. First, an entry has to encapsulate enough computation that the benefit of using the entry to fast-forward through a certain minimum number of instructions outweighs the cost of looking it up. We do not yet have a comprehensive cost model for the trade-off between caching a sequence of computation and regenerating that sequence [1], but in our prototype virtual machine we find that entries must comprise a pair of state vectors separated by at least 10^4 instructions to make the cache lookup time worthwhile. Second, we would like entries that correspond to units of computation with few dependencies, since they will have larger equivalence classes of matching states. Third, we would like the entries to correspond to building blocks that are likely to be used repeatedly within a single program or frequently among different programs.

Our next task is to identify pieces of a trajectory whose start states are close in state space, as these states often represent the start state of a repeated pattern of computation worth caching and training models on. One approach is to look for a sequence of states that share the same value for the instruction pointer (IP). Such states represent repeated visits to the exact same location in a program, presumably in a loop, although they could also correspond to repeated invocations of a function. By tracking such sequences of states, we can ask if the computation between any two visits is sufficiently long to consider creating a cache entry. If it is not, we might try using every n-th occurrence of the IP. If it is, then we compute the distance between two such states to see if they are close in state space. If they are, we next consider the dependencies and modifications that happen between any two occurrences of the IP.

Given the sequence of states sharing the same IP, we examine execution between subsequent pairs of states. Our virtual machine keeps track of every state vector element that the program reads or writes between those states. The computation depends on any element that it reads before writing – if the number of such elements is relatively small, then we've satisfied our three criteria, and we have identified good candidates for cache entries. Storing trajectories with few dependencies is key to implementing a cache with wide reusability. A cache entry will match all states in the space that match on its dependent elements; entries with fewer dependencies have larger equivalence classes of matching states. Thus, we have reduced our problem from having to match on a ridiculously large state vector to having to match only a few elements in that vector.

When the current state vector matches a cache entry we speed up execution by overwriting the current state with the output elements in the cache entry. Figure 1 illustrates this process, but there are intuitive interpretations as well. Thinking of the state space geometrically: states matching on dependent elements form a hyperplane in the space; the dimensionality of the hyperplane is much lower than that of the space. We use cache entries by translating the entry to any state we encounter that lies on the hyperplane. Alternately, programmatically, imagine that the IP value corresponds to a location inside a loop that operates over an array of data. Execution depends only upon the loop variable and the location being read in the loop, and those two elements are likely perfectly correlated. Better yet, consider the IP to be that of a pure function call encapsulated in a loop – in this case, the function arguments are the dependencies and return value of the function is the output. In this latter case, the cache entry corresponds perfectly to memoization [17].

We can now take this cache entry algorithm one step further. Once we have found sequences of states with matching IP values that are good cache entry candidates, we can begin trying to predict likely future states that will match the entry. We can then use those predictions to launch parallel threads that begin speculatively executing from those predicted states. Each thread executes until it next encounters the right IP value and enters its computation into the cache. This provides a form of speculative, generalized memoization.

We have implemented this cache entry detection algorithm in our virtual machine only for sequences of matching IP values (e.g., program locations that appear in a loop), however, we have experimented with other forms of detection such as function call invocation and return. We expect to pursue a variety of cache entry detection algorithms as future work. Our virtual machine watches the IP looking for repeated values that are sufficiently far apart to warrant further investigation. When it finds such values, it does three things. First, it begins adding entries to the cache for instances it has seen. Second, it dispatches an ensemble of predictors that use the sequence of states sharing the same IP to build models and generate predictions. Third, using the predictions, it launches speculative threads to create cache entries from those predictions. Meanwhile, whenever it encounters the IP value, it queries the cache for matching entries; when it finds one, it then fast-forwards its execution to the state that results from applying the cached entry, then resumes execution.

Our predictive models operate on state vectors of bits or on transformed feature vectors (e.g., 32-bit integers) and are able to find patterns that are difficult to resolve at the semantic level of a programming language. However, for the purpose of exposition we present an example pattern that corresponds to a human-level concept that was automatically discovered by the neural network we discuss in §5.

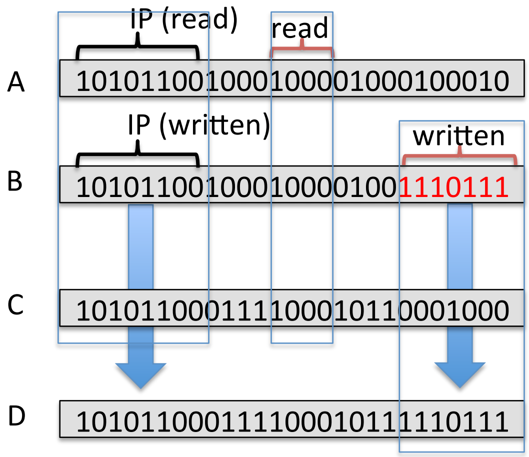

Figure 1: Cache Matching. State vectors A and B represent the entire computational state at two subsequent visits to states sharing the designated IP value. During the execution from A to B, we keep track of the bytes that have been read (marked in A) and written (marked in B). State C matches the cache entry we created from execution from A to B, because it has identical bit patterns for all the bits read. We use the cache entry by replacing C's bits with the bits marked as written in C to produce the state in D.

In Listing 1, we know at compile time that this loop is calling a function, `potential()`, that dereferences pointers. Thus, the function is not amenable to conventional memoization. As we show in Listing 2, this function traverses a dynamic data structure and is computationally intensive. If our models correctly predict a state corresponding to the entry point of `potential` with correct values for the pointer `node` and the contents of the memory to which it refers, then when we later encounter that element of the list, we will be able to use the cached entry. In other words, if we speculatively execute a possible future invocation of this function and cache the computation, we can use it later to accelerate execution.

```
struct node *node = head;

while (node) {
  energy = potential(node);

  if (energy < GROUND + e)
    break;

  node = node->next;
}
```

Listing 1: List traversal.

Having accomplished the basics of caching computation both by capturing actual executions as well as possible executions, we can consider ways to extend this idea. In our current framework, the cache entries depend on the value of the pointer in the list and the value stored in the list entry. It is easy to imagine a hybrid approach that uses semantic information about the relationship between the pointer and the value to do even better (i.e., require matching only on the value of the list element and not the pointer). Another way to extend this work is to attempt techniques to more rapidly train our models. For example, right now we train only on states we have actually observed, however, we could also synthesize training data by first fixing the bits corresponding to the instructions of the program and then sampling the remaining bits from some distribution.

Creating and using cache entries is similar to some conventional parallelization approaches, such as loop parallelization. However, the scope of our technique is more general. Our hypothesis is that while there exist programs for which prediction is impossible, in practice there are a great many interesting programs for which prediction is tractable. Some of those programs are the "embarrassingly parallel" ones that modern compilers can also parallelize. However, it is also possible that many programs are not able to be parallelized by conventional approaches for incidental reasons, such as aliasing, rather than fundamental unpredictability inherent in the problems they are solving. For example, the code from which we extracted Listings 1 and 2 is in daily use in a local research lab and computes on data stored in a linked list that is not amenable to conventional parallelization. However, we show in §5 that a neural network can generate accurate predictions and also be robust to small changes in the program code.

```
int potential(struct node *node) {
  int i, j, spin, energy = 0;

  for (i = 0; i < 1024; i++) {
    for (j = 0; j < 1024; j++) {
      spin = node->spins[i][j];
      /* Calculate energy. */
    }
  }

 return energy;
}
```

Listing 2: Function that dereferences pointers.

The direct cost of filling a cache is mostly that of dependency tracking overhead during speculative execution, which we currently measure at about 13% over and above the cost of running in a virtual machine. Indirectly we can smoothly consume increasing amounts of otherwise unused compute and memory resources by speculatively filling the cache.

4. SPEEDUP

To further explore the potential of computational caching, we analyze how often cache entries must be used to yield useful speedup. We know that entries created from actual execution are always correct, so let's focus on entries we create speculatively. Assume that we have N cores, one of which is running the master computation and the other $N-1$ are running workers whose job it is to speculatively fill the computational cache. Suppose that a constant fraction α of the speculative computation is correct. It is straightforward to show that with these assumptions the expected speedup S as a function of α and the number of workers has the relationship:

$$S \leq 1 + \alpha(N-1). \tag{1}$$

We obtain equality when the cache entries represent evenly spaced pieces of the full trajectory with no latencies or redundancies. A system that uses decision theory to maximize its expected speedup may be able to get quite close to this even spacing, in which case its speedup is *linear* in the number of cores and accuracy. Figure 2 shows contour lines of constant speedup when Equation 1 is an equality. These "isospeedup" contours give an intuitive view of why a computational cache is so interesting. Imagine that we are currently running on a computer with $N = 26$ cores that has constant predictive accuracy $\alpha = 0.16$. Then our speedup lies on the $S = 4$ contour. Now imagine that we populate the cache with results from a much larger machine that has a constant predictive accuracy of $\alpha = 0.4$. Our speedup now lies on the $S = 10$ contour, which means that our system now performs as if we installed 38 additional cores. To date, most of our experience comes from scientific applications, so we envision a world where we create large distributed caches using supercomputers and make them accessible to your laptop, letting you perform computations locally that would normally far exceed local capacity. In the long run, one might imagine sets of distributed computational caches that contain cached executions of common library functions or other widely used computations. In the extreme, the line between data caching and computational caching blurs.

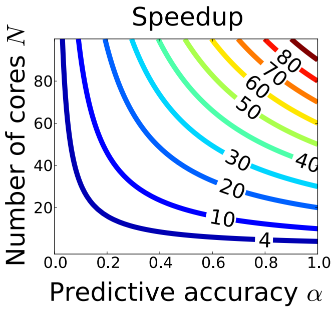

Figure 2: Convertibility of speedup and accuracy.

5. NEURAL NETWORK EXAMPLE

We close this paper with a small, but concrete example of how we speculatively populated a cache using neural networks. This example is intended to show some promising steps toward two goals: (1) that our system can automatically learn models that achieve nontrivial predictive accuracy on interesting programs such as the linked list program we have been discussing, and (2) that it is feasible to populate a cache during the execution of one program and then use that cache to speed up a different program.

Given a sequence of state vectors $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_k$ produced during execution, our goal is to predict the state vector $\mathbf{x}_{k+d}$ that is d steps in the future. We speculatively execute from the predicted state $\hat{\mathbf{x}}_{k+d}$ for some useful number of instructions to the state $\hat{\mathbf{x}}_{k+\ell}$ and then insert in the cache the pair $(\hat{\mathbf{x}}_{k+d}, \hat{\mathbf{x}}_{k+\ell})$.

To do predictions we set up a probabilistic model for the future state $\hat{\mathbf{x}}_{k+d}$ given the current state $\mathbf{x}_k$. Since each coordinate in our n-dimensional state vector is a bit and our transition function is memoryless, we factor the joint distribution over $\hat{\mathbf{x}}_{k+d}|\mathbf{x}_k$ as:

$$\begin{aligned} p(\hat{\mathbf{x}}_{k+d}|\mathbf{x}_k, d, \boldsymbol{\theta}) &= \prod_{i=1}^{n} p(\hat{x}^i_{k+d}|\mathbf{x}_k, d, \boldsymbol{\theta}) \\ &= \prod_{i=1}^{n} \text{Bernoulli}(\hat{x}^i_{k+d}|\boldsymbol{\theta}_i(\mathbf{x}_k)) \end{aligned}$$

This factorization makes our models modular and decomposes the problem of learning $\hat{\mathbf{x}}_{k+d}|\mathbf{x}_k$ into n conditionally independent binary classification problems: one for each bit $\hat{x}^i_{k+d}$. We then plug in n separate binary classifiers $\boldsymbol{\theta}_i(\cdot)$, which allows us to bring to bear a wide range of powerful models, as binary classifiers are one of the best developed areas of machine learning [21]. It also makes model training a parallel process, but has the disadvantage of forcing the models to learn low-level representations for things such as a loop induction variable, which might be much better handled as a feature. In practice, we fit classifiers for only the bits that we have seen change, which is usually a small fraction of n.

We used the standard GNU toolchain to compile and statically link the program given in Listing 1 and Listing 2. We then ran this program in our $x86$ virtual machine with a dimensionality of state space $n = 8 \times 25000$ bits. Our virtual machine plugged in for each $\boldsymbol{\theta}_i(\cdot)$ one of the simplest useful models: the single-layer neural network. For each bit i it used training data pairs of the form $(\mathbf{x}_k, x^i_{k+d})$, where each $\mathbf{x}_k$ has the IP value of interest and d is the number of instructions to the subsequent occurrence of that IP value, to fit the weight vector $\boldsymbol{\theta}_i$ in:

$$\boldsymbol{\theta}_i(\mathbf{x}_k) = \frac{1}{1 + e^{-(\theta^0_i + \theta^1_i x^1_k + \cdots + \theta^n_i x^n_k)}} \tag{2}$$

Equation 2 has a sigmoidal form that ranges from 0 to 1, whose output is interpreted as the single layer neural network's estimate of the probability, conditioned on state $\mathbf{x}_k$, that the i-th bit of the future state $\mathbf{x}_{k+d}$ will be 1.

To evaluate our predictive accuracy, we gave our virtual machine a test set of state vectors with the IP value of interest from the execution of the same program—but which it had never seen before—and asked it to predict the next such state. In Figure 3, the blue line shows the empirical cumulative distribution function (CDF) of how many bits the model got wrong. About 20% of the time the model predicted the entire state vector perfectly, and 50% of the time it predicted only six or fewer bits wrong. The dashed red line shows the CDF for the number of bits the model mispredicted when asked to make predictions for a variant of the original program that we produced by adding a small amount of random noise to its linked list contents. We find these results highly encouraging as they demonstrate the cache's robustness; we populated the cache using one program and were able to make accurate predictions for a different version of the program. These predictions can then be used for speculative execution, thus speeding up computation.

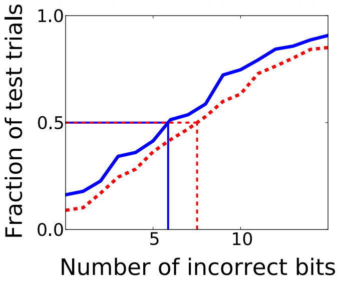

Figure 3: Robustness.

6. RELATED WORK

Identifying good cache entries is closely related to techniques in compiler loop parallelization. While early parallelization techniques relied largely on static analysis [2, 3, 6, 14], the limitations of static analysis have become apparent [10]. Incorporation of runtime checks [22] expanded the reach of parallelizing compilation systems, but these techniques still do not generalize easily to general-purpose programs in the face of interprocedural dependencies and pointer-based access [15]. To date, we have only demonstrated the potential of computational caches on programs that have obvious repetitive structure, but we have shown that they work in the presence of pointer-based codes, which is promising.

Speculatively populating our computational cache can be considered an extreme case of some of the other speculative approaches to parallelization such as Thread-Level Speculation [20, 23, 5] and Decoupled Software Pipelining [25, 27, 19]. Our virtual machine design draws inspiration from trace caches [18], memoizing processors [13, 17, 28], caching cellular automata [9, 26] and perceptron branch prediction [12].

The low-level representation from which we make our predictions suggests analogies between computational caches and binary translation [4] and binary parallelization systems [8, 16, 7, 10, 29, 31, 32]. We believe that hybrid approaches that leverage existing work, such as applying branch prediction to parallelization [11, 24] or code rewriting systems [16, 8, 29] to generate better predictors could further improve the efficacy of computational caches.

7. CONCLUSION

Computational caches have enormous potential. We have described them here in terms of making a large cache accessible to a small computer, but as we are exploring in concurrent work, they also have the ability to automatically parallelize programs that are not currently amenable to conventional parallelization techniques. In the long run we imagine a computational infrastructure where an entire network of computers coordinate and collaborate by sharing model parameters and cache entries, thus harnessing their combined compute power, automatically, without any special problem-specific middleware. Realizing these visions requires that we develop robust models with sufficient predictive accuracy to produce useful speculative trajectories. The results from our simple neural network predictor demonstrate that such accuracy may be possible.

8. ACKNOWLEDGMENTS

The authors would like to thank Gerald Sussman, Miguel Aljacen, Jeremy McEntire, Liz Bradley, Benjamin Good and Scott Aaronson for their contributions. This work was supported by the National Science Foundation Graduate Research Fellowship under Fellow ID 2012116808, the Department of Energy Office of Science under its agreement number DE-SC0005365, and the National Institutes of Health under Award Number 1R01LM010213-01.

9. REFERENCES

[1] I. F. Adams, D. D. E. Long, E. L. Miller, S. Pasupathy, and M. W. Storer. Maximizing efficiency by trading storage for computation. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[2] V. S. Adve, J. Mellor-Crummey, M. Anderson, J.-C. Wang, D. A. Reed, and K. 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.

[3] S. P. Amarasinghe and M. 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] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[5] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '02, pages 99–108, New York, NY, USA, 2002. ACM.

[6] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. 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.

[7] M. K. Chen and K. 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.

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

[9] B. Gosper. Exploiting regularities in large cellular spaces. *Physica D. Nonlinear Phenomena*, 1984.

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

[11] E. Ipek, M. Kirman, N. Kirman, and J. 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.

[12] D. A. Jiménez and C. 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.

[13] Y. Kamiya, T. Tsumura, H. Matsuo, and Y. Nakashima. A speculative technique for auto-memoization processor with multithreading. In *Parallel and Distributed Computing, Applications and Technologies, 2009 International Conference on*, pages 160 –166, dec. 2009.

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

[15] H. Kim, N. P. Johnson, J. W. Lee, S. A. Mahlke, and D. 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.

[16] A. Kotha, K. Anand, M. Smithson, G. Yellareddy, and R. 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.

[17] D. Michie. Memo functions and machine learning. Nature.

[18] Z. Purser, K. Sundaramoorthy, and E. 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.

[19] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 65–76, New York, NY, USA, 2010. ACM.

[20] L. Rauchwerger and D. Padua. The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 218–232, New York, NY, USA, 1995. ACM.

[21] D. S. Richard Duda, Peter Hart. *Pattern Classification (Second Edition).* Wiley-Interscience, 2001.

[22] S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. *Int. J. Parallel Program.*, 31(4):251–283, Aug. 2003.

[23] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The stampede approach to thread-level speculation. *ACM Trans. Comput. Syst.*, 23(3):253–300, Aug. 2005.

[24] D. Tarjan, M. Boyer, and K. Skadron. Federation: repurposing scalar cores for out-of-order instruction issue. In *Proceedings of the 45th annual Design Automation Conference*, DAC '08, pages 772–775, New York, NY, USA, 2008. ACM.

[25] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[26] T. Toffoli. *Action, or the fungibility of computation*, pages 349–392. Perseus Books, Cambridge, MA, USA, 1999.

[27] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 49–59, Washington, DC, USA, 2007. IEEE Computer Society.

[28] C. Verbowski, E. Kiciman, A. Kumar, B. Daniels, S. Lu, J. Lee, Y.-M. Wang, and R. Roussev. Flight data recorder: Monitoring persistent-state interactions to improve systems management. In *OSDI*, pages 117–130. USENIX Association, 2006.

[29] C. Wang, Y. Wu, E. Borin, S. Hu, W. Liu, D. Sager, T.-f. Ngai, and J. 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.

[30] A. Waterland, J. Appavoo, and M. 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.

[31] 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.

[32] E. Yardimci and M. 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.