Introspective Computing for Prefetching

(CS252 Spring 2000, Final Project)

Andrew Y. Ng and Eric Xing
University of California, Berkeley
Berkeley CA 94720
{ang,epxing}@cs.berkeley.edu

Abstract

In this paper, we consider applying "introspective computing," in which a "primary processor" executes the desired program, and a "secondary processor" monitors the primary processor's execution status and attemps to help it to optimize performance, to the problem of prefetching for a data cache. We examine the space of architectures and prefetching algorithms that may be applied to this task, and propose a simple prefetching scheme using an introspective architecture. We demonstrate that our method can reduce cache misses by up to about 60% on common benchmarks, and conclude that this, and a number of its natural extensions, may be a fruitful method for significantly improving CPU performance.

1 Introduction

With ever-increasing numbers of transistors available on a single die, recent years has seen a growing interest in novel computing paradigms for exploiting the additional available complexity to build faster processors. One of these paradigms, known as "introspective computing" [5], is investigating new ways in which the traditional hardware functionality of a CPU is replaced by feedback-driven, continuous dynamic compilation and execution.

In the form of introspective computing that we study on in this paper, a "primary processor" executes the desired program, and a "secondary processor" monitors the execution status of the primary processor, and attempts to assist it to optimize its performance [5]. There are several ways in which the secondary processor can do this. One example is for it to attempt to track the behavior of branches taken, and to try to improve the branch prediction performance of the primary processor. A second example, and the subject of this paper, is a scheme where the secondary processor tracks the behavior of memory accesses, and attempts to improve the performance of the primary processor by appropiately issuing prefetches for it to reduce processor stall time due to cache misses.

To our knowledge, such a primary/secondary processor form of introspective computing has not been studied in detail, though our particular use of it for prefetching is reminescent of architectures where a separate processor is be responsible for memory accesses (see, e.g. [7]). Of course, prefetching itself is an important and well studied technique, with hardware-oriented forms of prefetching such as miss caching, victim caching, stream buffer prefetching, [4] and even more sophisticated schemes such as [3] employed to great effect; similarly, some compiler-oriented forms of prefetching have also been proposed and seen some degree of success [2]. However, our approach is quite different. First, unlike the hardware oriented schemes, our prefetching decisions are made by a seprate processor; this allows us to consider *significantly* more ambitious and complex algorithms than those typically considered for straightforward hardware implementations. Second, our approach also differs from the compiler-based ones that statically insert prefetching instructions into a program, since our secondary processor has access to and can base its prefetching decisions on runtime information that is typically unavailable to static optimizers.

In this paper, we describe an architecture for using introspective computing for prefetching, and carry out a number of explorations of the design space to examine its utility and feasibility. Our experiments show that, on standard benchmarks, our method can reduce data cache misses by up to about 60%. We also discuss methods for scaling up our introspective architeture, and conclude that introspective computing may well be a fruitful way to significantly improving CPU performance. The remainder of this paper is structured as follows: In Section 2, we describe our proposed architecture, including the mechanisms by which the secondary processor monitors and affects the program execution of the primary processor. Section 3 then examines candidate prefetching algo-

rithms, paying particular attention to the real-time constraints inherent in this problem. Section 4 describes our experiments exploring the design space, and finally, Section 5 discusses a number of issues central to practical implementations of introspective computing such as embedded-processor-style design of the secondary processors, and the use of multiple primary or secondary processors, and closes with future work.

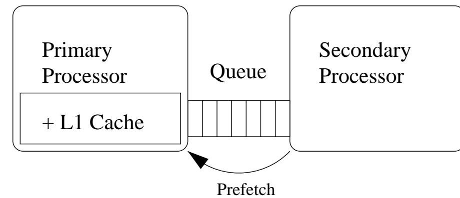

Figure 1: Introspective architecture.

2 Introspective Architecture

For the sake of simplicity, we focus in this paper on the problem of prefetching for the L1 data cache, though there are no difficulties to generalizing our method (using one or more secondary processors) to prefetching for other caches as well. In designing an introspective architecture for prefetching, the two main decisions are regarding the design of:

- The mechanism by which the secondary processor monitors the primary processor's executions status.
- The mechanism by which the secondary processor causes prefetches to occur.

While these choices may initially seem easy, there is a number of important issues that affect the utility of various designs. First, regarding the monitoring mechanism, it is important to carefully select only features of the execution status that are relevant to making prefetching decisions, and it is certainly not the case that the more data about the execution status the secondary processor is given, the better. More subtlely, the secondary processor is faced with the task of learning, by monitoring the primary processor's execution status, where the future cache misses will be, so as to issue appropiate prefetches; if it is given "too much" information about the primary processor's status (say, if it is provided the contents of all of its registers), it is a well-known phenomenon in Machine Learning that this will make it extremely difficult for most adaptive/learning algorithms to learn the required cache miss predictions. [6] Intuitively, this is because the learning algorithms would be so overwhelmed with irrelevant information, that it becomes extremely difficult to detect the signal that allows one to make the required predictions.[1]

Hence, in the spirit of ensuring relevance, we chose a scheme where the information available to the secondary processor was only the location of the (L1 data) cache misses that occurred in the primary processor. More precisely, we place a queue between the primary and the secondary processors, and each time there is a cache miss, the address (or rather, the cache tag) corresponding to the miss is pushed onto the queue. The secondary processor, running in parallel with the primary processor, then repeatedly pops an element off the queue, does some appropiate processing on it, and perhaps decides to insert a prefetch request into the memory access stream of the primary processor. This architecture is shown in Figure 2. In this paper, we assume that prefetched blocks are inserted directed into the primary processor's L1 data cache; note therefore that a secondary processor that indiscriminately issues many useless prefetches may actually hurt the primary processor's performance.[2]

Later, we will also examine issues such as the length and type (e.g. fifo vs. lifo) of the queue type, but for now, we move on to describe possible prefetching algorithms that the secondary processor may use.

3 Prefetching algorithms

The secondary processor, running our prefetching algorithm, repeatedly pops information about cache misses off the queue, does some appropiate processing, and possibly issues one or more prefetches. On each step, it therefore has to make a decision about whether or not to issue a prefetch (and if so, what to prefetch). In this section, we briefly review some powerful algorithms from the Machine Learning literature for learning to make these decisions, discuss problems with their use arising from real-time constraints, and finally describe the simple prefetching algorithm that our explorations converged upon.

3.1 Machine Learning/Adaptive control algorithms

Reinforcement Learning [8] is a subfield of Machine Learning, that offers powerful methods for learning to make optimal or near-optimal control decisions over time. It is based on estimating the *Bellman Equations*

$$C(s) = T(s) + E_{s'}[C(s')] \qquad (1)$$

where, in our context, s is the state of execution of the primary processor, $C(s)$ is the cost to completion

[1] As we will later see, another crucial issue is real-time constraints, which imply we must ensure that the secondary processor does not have so much information that it does not have enough time to process it all.

[2] However, we do assume that if a prefetch instruction tries to prefetch a block of data already in the cache, then it is just ignored by the cache. This takes some of the burden of filtering out "spurious" prefetches from the secondary processor.

of execution from s, $T(s)$ is the immediate time-cost for a single step from s, and s' is the successor state. In essence, this says that the total cost of execution is the single-step cost plus the remaining cost after that single-step.

Based on this, Reinforcement Learning algorithms then use entirely standard estimation algorithms to approximate appropiate relations between the quanties in Equation (1) with the information available about the execution status, and then immediately derives from them (via a few simple, standard, identities) optimal or near-optimal control (prefetching) decisions. [8, 6] Hence, Reinforcement Learning, which incidentally comes with theoretical guarantees of converging to near-optimal decisions under mild assumptions, seemed an ideal algorithm for the secondary processor to use.

Unfortunately, this turns out not to be the case. In simulating our introspective architecture, we found by examing traces on several standard benchmarks (`compress`, `gcc`, `go`, `hydro2d`) that even optimized implementations of Reinforcement learning would be about an order of magnitude too slow for this problem. More specifically, as the primary processor is executing, cache miss events happen often enough that if the secondary processor is too slow in responding to them, then (a) the queue will fill up and start dropping data, and the secondary processor will never get to know about and process many of the cache misses, and/or (b) the secondary processor will be issuing prefetches too late to make a difference (e.g. the cache miss that the prefetch was supposed to prevent would have already happened). Very roughly, we found that for each "miss event" that the secondary processor pops off the queue, it can afford to do at most about 100 cycles of processing if is to be able to "keep up." Other standard learning algorithms that we considered (including Support Vector Machines and neural-network based approaches [6]) were similarly about an order of magnitude too slow in view of this real-time constraint, leading us to consider simpler, faster learning algorithms.

3.2 A fast, simple, prefetching algorithm

Upon popping a information about a cache-miss event off the queue, the secondary processor has to (i) update its stored statistics regarding cache misses appropiately, and (ii) possibly issue one or more prefetches. We now describe these two steps in turn for the algorithm we converged to after some amount of exploration and experiments.

Recall that the information the secondary processor sees is a stream of cache tags corresponding to the misses. For some value of n (defaulting to 5 in our experiments), we consider that a cache miss at tag x following later by a miss at tag y are "close" if they occurred within n of the observed cache misses of each other. Our algorithm then used a large hash table that, for each cache tag x at which there was a miss, has a table of m (also defaulting to 5 in experiments) entries for keeping track of the number of times misses at other tags y occured "closely" after it. i.e. If a miss at some tag x_0 is almost always immediately followed by a miss at some other tag y_0, then the statistics gathered by this algorithm will reflect this, and this information may then be used to (say) issue a prefetch for y_0 whenever there is a miss at x_0. We will later also consider different strategies for dealing with conflicts in the hash table, but for now, the default strategy is to throw away data that results in conflicts (e.g. not gather statistics about misses at x_1 if it hashes into the same entry as some previous miss x_0 about which we are gathering statistics).

The decisions for whether to issue a prefetch are now straightforward. With the gathered statistics, the secondary processor will, upon a miss at cache tag x, look in its hash table to see if it has observed at least T times a closely following miss at any cache tag y, for some integer T (defaulting to 2, and whose effect we study in our experiments). If so, then it issues a prefetch at tag y.

The specific details regarding the implementation of this algorithm are not interesting, but we note that it is important, because of the real-time aspect of this problem, to use a highly optimized implementation. For example, even though we have described the algorithm as first updating its stored statistics (upon observing a cache miss) and then possibly issuing some number of prefetches, we actually implemented the two steps in *reverse* order, so so that it first issues its prefetches (if any), and then updates its stored statistics. This allows the prefetch to be issued a few cycles earlier, and actually results in small improvements (about 2-10% reductions in cache misses) in the performance of our algorithm on benchmark tests, some of which we describe in our next section.

On that note, even with this simple algorithm, there is still a large number of interesting parameters that may be tuned to give the best possible performance. In the next section, we describe our experiments using this algorithm, that represent an exploration in the extremely rich design space implied by even this simple prefetching algorithm.

4 Experiments

We now describe a number of experiments exploring the design space of the introspective architecture and prefetching algorithm described in Sections 2 and 3. The experiments described in this section were carried out using the SimpleScalar simulator [1], which we modified carefully and extensively to simulate the simultaneous execution of the primary and secondary processors, as described earlier. Our experiments were carried out using the `compress`, `gcc`, `go`, and

`hydro2d`benchmarks from SPEC95 set.[3]

For our first experiment, we tried varying the queue type (of the queue between the primary and secondary processors) between fifo and lifo. While a fifo queue seems a natural choice, we also conjected that a lifo queue might be better, since it allows the secondary processor to get to handle the most recent (and hence presumably the most "urgent," in terms of requiring a prefetch soon) cache misses so far. The results of this experiment are shown in Figure 2a. As we see, our conjectured turned out not be be correct, and in fact a fifo queue did much better. Examining the trace of the cache misses under the two algorithms, it appears that the fifo queue scheme, which processes cache misses in order, is often getting needed prefetches issued just in time, and hence did much better on the first benchmark; in contrast, by the time the lifo scheme got around to the last element on the queue, it was often processing fairly old data, for which it was far too late to do anything useful.

We next examined the effect of different policies for what to do upon an attempted push onto the queue when the queue is already full. The simplest solution would be to simply drop the push; but given the application, it seems natural to expect that a better solution would be to drop the oldest data on the queue (something easily done in a fifo, or a lifo, queue) to make room for the push. Thus, the queue is, with its finite memory, trying to keep track of the most recent data for the secondary processor. The result of this experiment is shown in Figure 2b, and indeed, we do see that the latter policy does seem to result in a slight increase in the number of cache misses avoided.

In the next two experiments, we varied the hash size used by the algorithm to store its cache miss statistics, and the size of the queue. The results of these are shown in Figures 3a and 3b. It appears from these experiments that a hash size of 10000 would be a good choice, with further increases given greatly diminishing returns. As for the queue size, this appeared to be a parameter to which our results were fairly insensitive, with values around or slightly smaller than 10 doing best. It was not surprising that extremely small queue sizes (such as 1) did slightly worse than larger ones, but it was interesting that very long queues also did not do well. On reflection this is not very surprising: examing the miss and queue traces again, we find that, with very long queues, the secondary processor is sometimes processing a cache miss long after it had happened, and far too late to do anything useful (such as issuing a useful prefetch) based on it.

Next, we examined the effect of varying the speed of the secondary processor. Since the secondary processor requires extra silicon area and power, it would be nice if we could build it to run only at a slower speed, and still yield good prefetching benefits. The result of this experiment (expressing secondary processor speed as a fraction of the primary processor's speed) is shown in Figure 4a. Unfortunately, it appears that as the secondary processor is made slower, prefetching performance does drop off fairly rapidly. But on the upside, a secondary processor even faster than the primary processor does still shown some gains; while implementing this per se would probably be impractical, this also suggests that using multiple secondary processors to speed up computation, which we discuss in the next section, may be fruitful.

Figure 4b next shows the effect of varying T (described in the previous section). Interestingly, it appears that fairly small values do well. Lastly, Figures 5a and 5b consider randomized strategies for handling conflicts in the hash table used to store the cache miss statistics. We consider a strategy where, upon a conflict, the old data stored in an entry in a table is replaced by the new piece of data with some probability p, where $0 \leq p \leq 1$.[4] Interestingly, it appears that the original scheme of ignoring any new data that results in a conflict (i.e. $p = 0$) did as well as anything else; we believe this is because, if two miss addresses result in a conflict, then storing information about the first one is as good as storing information about the second; moreover, each time we throw away an entry and replace it with a new one, we are actually throwing away some set of statistics that we had spent time gathering, and would need to start from scratch again.

5 Discussion and future work

In this paper, we have demonstrated how introspective computing for prefetching can, on standard benchmarks, reduce the number of cache misses by up to about 60% (which roughly translated into 10-15%

[3] Other experimental details: Unless otherwise specified (such as if an experiment varied the parameter), the queue between the primary and secondary processors was a fifo queue of size 10, and used the "delete oldest" policy (described later) on an attempt to push an element onto it if it was already full. The hash size defaulted to 10000. The L1 data cache was a 4-way associative cache with 64 byte blocks and size 16K, had a hit time of 1 cycle, and a miss penalty (loading from L2) of 6 cycles. Similar to some other authors and remaining in the spirit of using a fixed testbed, the longest running benchmarks were truncated at a very large but fixed number of execution cycles, to make repeated experiments complete in a tractable amount of time. Unless otherwise stated, the speed of the secondary processor was assumed to the same as that of the primary processor, and other parameters took the default values described in the previous section.

[4] Figure 5a shows the effect of this scheme for entries in the hash table. Figure 5b shows the effect of this scheme for entries in each of the small tables that is the data contained in the hash table, described in Section 3. Similar results to Figure 5a were obtained using various different hash sizes.

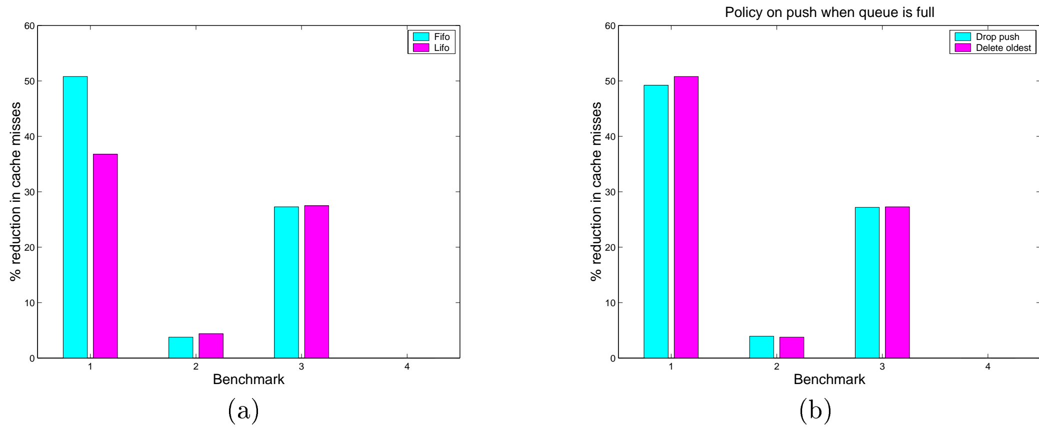

Figure 2: (a) Percentage reduction in number of cache misses for the four benchmarks (`compress`, `gcc`, `go`, `hydro2d`2d), when using a Fifo or a Lifo queue between the primary and secondary professors. (b) Percentage reduction in number of cache misses for two possible policies for what to do on a push() when the queue is full.

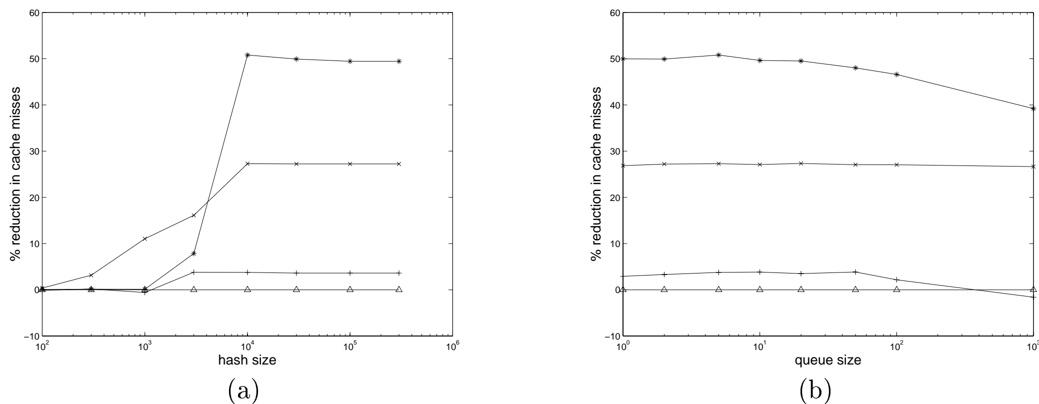

Figure 3: (a) Reduction in cache misses for the four benchmarks vs. size of the hashtable. (b) Reduction in cache misses vs. size of queue between primary and secondary processors. (Legend: $*$:`compress`, $\times$:gcc, $+$:`go`, $\triangle$:,`hydro2d`)

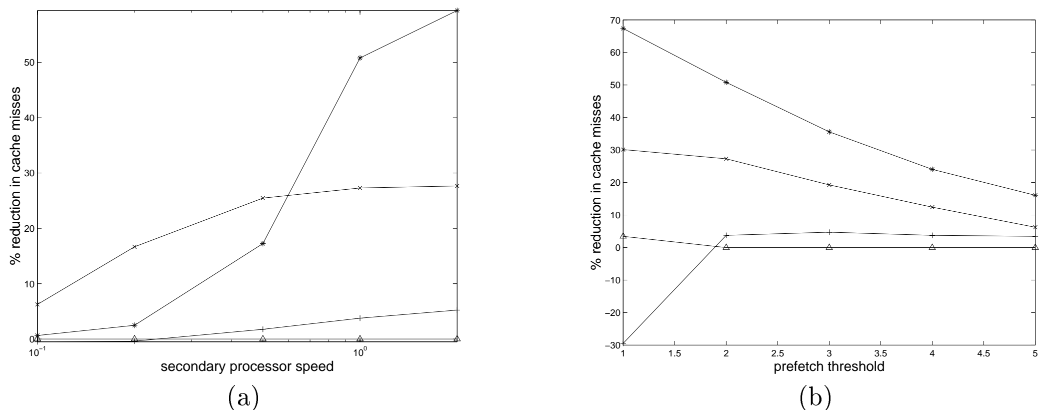

Figure 4: (a) Reduction in cache misses vs. speed of secondary processor (expressed as a fraction of the speed of the primary processor). (b) Reduction in cache misses vs. different values of the "threshold" T at which we will issue prefetches. (Same legend as Figure 3.)

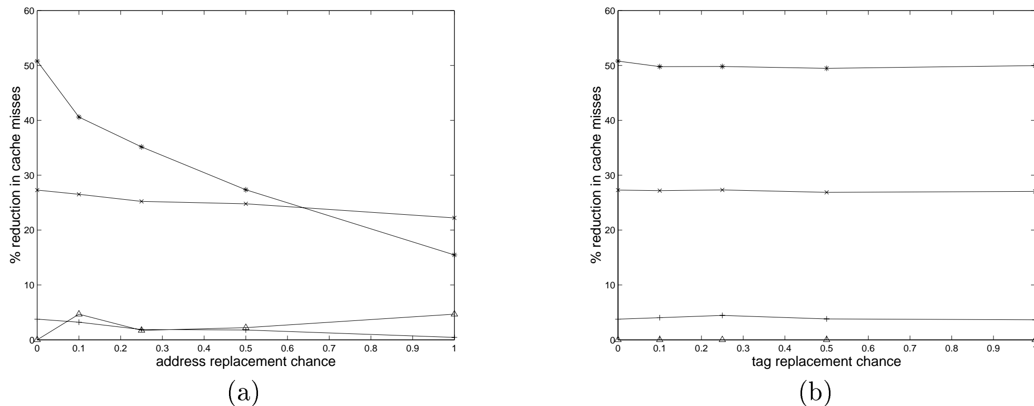

Figure 5: (a) Reduction in cache misses vs. p (chance of replacement on conflict), for main hash. (b) Reduction in cache misses vs. p, for the tables storing statistics of "closely following" misses for a given tag. (See text.) (Same legend as Figure 3.)

reduction in running time). Introspective computing has several other interesting features, one of which is that it scales up very well. For example, our method can readily be applied to multiprocessors, with perhaps one (or less) secondary processor working on behalf of each primary processor. On the converse, it is also easy to construct systems where multiple *secondary* processors service a single primary processor. Indeed, since the secondary processor is currently just repeatedly popping items off the queue and doing processing on that, its task parallelizes nicely, and we can simply have multiple secondary processors all popping jobs off the single queue and working on them.

Another important issue in the application of these ideas is that since the secondary processor is pretty much doing only one thing, it is perhaps best thought of as an embedded processing executing a single application. Taking this point of view, one can then easily imagine cheap DSP implementations of the secondary processor, the use of specialized instructions for the secondary processor's particular application, and so on. A careful study and evaluation of these issues remains important future work, as is the futher exploration of the design space of prefetching algorithms in introspective computing.[5]

Acknowledgments

This work was based on a project suggestion made by Professor John Kubiatowicz. We also thank Scott Weber for helpful pointers and comments.

[5] Space constraints preclude a lengthy discussion, but for example, we found in experiments (not reported here) that if, whenever the queue is nearly full, the secondary processor stochastically neglects to update some of the statistics the original version did, so that it has more time to concentrate on issuing the required prefetches, then performance can be further improved. We believe, from lessons learned from Reinforcement Learning [8], that the exploration of such *stochastic* algorithms is a particularly promising area.

References

[1] Doug Burger and Todd M. Austin. The SimpleScalar tool set, Version 2.0. Technical Report 1342, University of Wisconsin-Madison, Computer Science Department, 1997.

[2] Todd C.Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 62–73, 1992.

[3] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *Proceedings of the twenty-fourth international symposium on Computer architecture*, pages 252–263, 1997.

[4] Norman Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 364–373, 1990.

[5] John Kubiatowicz. Dynacomp. `http://www.cs.berkeley.edu/~kubitron/courses/cs252-F99/projects/suggestions.html`, 1999.

[6] Tom Mitchell. *Machine Learning*. McGraw Hill, 1997.

[7] James Smith. Decoupled access/execute computer architectures. *ACM Transactions on Computer Systems*, 2:289–308, 1984.

[8] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning*. MIT Press, 1998.