mdp report

Introduction

This benchmark is a pure-Python dynamic-programming simulation of a simplified battle which is being simulated as a Markov Decision Process.

It uses the value iteration algorithm, to solve the MDP.

It models combat turns as state transitions with probabilities, evaluates optimal choices under uncertainty, and computes the player's win probability to a target tolerance.

This is the second benchmark I picked, after AES, and I picked it mainly because of the following:

- It offers a bigger control flow (small functions, branches – if else)
- Arithmetic with fractions, which is different from the math in AES.
- Graph traversal, which is one of the main reasons I wanted to look at it, since graph theory is a very different math, but also have a huge application (movie recommendation system, friends' suggestions on Facebook) – they are all using graph theory.
- The benchmark itself is pure python code – therefore we have a pure python overhead, so I thought it would be nice to have a hands-on optimization on Python.

Besides what already mentioned, MDP has vast number of applications on itself – robotics, machine learning, finance, games. So, speeding up this process might also have a big impact on world real live applications.

MDP overview

The game is modeled as a directed acyclic graph, that includes abstract states like:

- A state for each fighter HP / stats, and whose turn it is now
- Actions which transform a state into next states with some probability functions for example damage randomness, critical hit chance, etc'.
- The benchmark evaluates the win probability from the initial state, by going over the state graph, until the estimate is "close enough"

On a more formal note, MDP is a very popular model for a lot of reinforcement learning problems, and consists of:

- S: set of states.

- A: set of actions.
- P: transition probability function: $P(s'|s,a)$ which give us the probability of moving to state s' after an action a during state s.
- R: the reward function, $R(s,a)$ which gives the reward for taking action a in state s
- γ: discount factor $(0 \leq \gamma < 1)$ which prioritizes immediate rewards over future ones.

The main goal is to find the optimal policy that maps states to actions which will maximize the expected discounted sum of future rewards. When using value iteration, it works by iteratively updating the value function, which in the end represents the maximum future reward some agent can get when the agent starts from state s.

This algorithm uses the bellman update equation, for every state, until the function reaches convergence:

$$V_{i+1}(s) = \max_{on\ all\ actions} \left(\left(R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V_i(s') \right) \right)$$

Benchmark initial analysis

The MDP benchmark in the pyperformance library builds the MDP problem from scratch.

The benchmark operates as follows:

1. Runs a " Pokemon's battle" for a certain number of loops.
2. Assert that the result is the same expected results, allowing a floating-point small precision error $(1 \cdot 10^{-6})$

The main code can be divided into three main parts:

1. State representation defines the game state using namedtuple objects – like Pokemon's HP, stats, and status.
2. Action and transition logic: _applyAction and getSuccessors functions define the main core logic. They also calculate the probability outcome for each possible move. This is basically building the transition probability matrix.
3. Value iteration: the evaluate method does the main heavy lifting. It initializes the values for win (1) and losing (0) and then it iteratively computes the probability of winning for all of the states.

By looking at the code, it seems that most of the logic sits under big while loop under the evaluate code. I suspect that this is where my main focus should be in order to improve the code runtime. In order to make sure which lines are the hot lines, I will use the py-spy library which shows which lines the benchmarks spends the most time on, on a python level.

Unoptimized flame graph version

I ran the benchmark, without the overhead of pyperformance, again, like AES, I wanted to look at only the MDP code.

This is the flamegraph I got:

As we can see, the flamegraph is dominated mainly by the dynamic programming part – which is evaluate function.

At the start, we can see a decent time spent on the building of the graph – which is the topological sort function.

When looking at the relevant code, we can understand why its so heavy:

```
def getSuccessors(self, statep):
    try:
        return self.successors[statep]
    except KeyError:
        st = statep[0]
    if st == 0:
        result = list(self._getSuccessorsA(statep))
    else:
        if st == 1:
            dist = self._getSuccessorsB(statep)
        elif st == 2:
            dist = self._getSuccessorsC(statep)
        result = sorted(dist.items(), key=lambda t: (-t[1], t[0]))
    self.successors[statep] = result
    return result
```

It calls getSuccessors, which triggers:

Multiple function calls (_getSuccessorsA/B/C), contains a lot of dictionary allocations, and, Fraction arithmetic. The fraction arithmetic is extremely slow, since it offers exact arithmetic for rational numbers, it stores each number has two integers: numerator and denominator, and also uses GCD algorithm. All of this stuff, including a lot of layers of indirection (we can see that in the flame graph) suggests that this library is rather slow (compared to float).

Also, there is zero optimization on this step. Even if a state has already been visited, its successors are recomputed multiple times before the node is marked as visited.

In addition, most of the samples are inside the nested iteration that updates dmin and dmax dicts for each state. Let's look at this code:

```
222         while dmax[initial_statep] - dmin[initial_statep] > tolerance:
223             itercount += 1
224 
225             for sp in stateps:
226                 if sp in frozen:
227                     continue
228 
229                 if sp[0] == 0:
230                     # choice node
231                     dmin[sp] = max(dmin[sp2] for sp2 in self.getSuccessors(sp))
232                     dmax[sp] = max(dmax[sp2] for sp2 in self.getSuccessors(sp))
233                 else:
234                     dmin[sp] = sum(dmin[sp2] * p for sp2,
235                                    p in self.getSuccessors(sp))
236                     dmax[sp] = sum(dmax[sp2] * p for sp2,
237                                    p in self.getSuccessors(sp))
238 
239                 if dmin[sp] >= dmax[sp]:
240                     dmax[sp] = dmin[sp] = (dmin[sp] + dmax[sp]) / 2
241                     frozen.add(sp)
242         return (dmax[initial_statep] + dmin[initial_statep]) / 2
243 
```

This loops runs as long as the initial state reaches converges.

dmin[s] and dmax[s] track lower/upper bounds on the probability of eventually winning from state s. Convergence means we've narrowed the value close enough – until we reach a certain tolerance.

Stateps is a topological order of all the accessible states, doing the iteration in this order helps with the converges (since we visit the most recent value faster)

There are two node types – one which is choice nodes (sp[0]), for choice nodes, the agent will pick the best successor – which is why those bounds take the maximum value out of it's successor (it makes sense, since we want to take the greedy approach on this case)

The other type of nodes, which is chance nodes, calculate the probability of the random outcome determined by the environment, this is not the agent decision, but the probability of what happens to him in this state. This correlates to:

$$V(s) = \sum_{s'} P(s'|s,a)V(s')$$

It can be clearly seen in the code.

After each update, if $dmin \geq dmax$ we reach a resolved state solution, se we "freeze" this state to make sure we won't continue iterating over it.

This heavy loop is the main time spent on the benchmark, and it makes sense –

We call getSyccessors which is heavy as it is, we perform a lot of dict lookups, tuple creation, and fraction arithmetic, as discussed on the AES report, these operations in Python are heavy, due to its native interpreter execution algorithm. And I expect that the byte code of this loop will be heavy.

optimizations pool

First, like AES, I'll focus on the optimizations that I didn't choose to do:

Usage of threads – again, most of the time we can speed up things using threads. (Look an Nvidia net worth. 😊) As stated in AES, threads in python are limited to Python's GIL. Again, I don't have a native I/O bound in this benchmark, so threads won't give me much benefit besides pure more computation power. Also, making this code threaded is not necessarily an easy task. So, again, I decided not to use threads for my optimization.

Usage of JIT + NumPy – on the AES benchmark, that was a huge improvement, and I could probably use the same in the inner evaluation loop, since it could be vectorized. However, it is still more complicated than it was in the AES benchmark, and I would prefer to try something else that is not hard to implement.

The approaches I took to improving the benchmark can be divided into two options:

On the first version, the focus was removing the Python interpreter overhead by restructure the state transition creation and caching them.

Also, I removed the usage of the fraction model (on some versions that I tried it, I failed in the final max_diff check, so I will also show versions that do not include removing the usage of the library. I also changed some of the dynamically constructed dicts to pre-built deterministic lists. This version will be examined more thoroughly later in this report.

On the second version of the optimization, I made significant changes to the algorithm. And now the graph is fully based on arrays. This allows me to replace the dicts lookups with preallocated arrays, eliminating Python hash overhead in every iteration, which was the bulk part of the execution time, as seen in the flame graph. For this optimization, I made two versions:

One that does not use the Fraction module (which is the fastest, but of course lower precision, **still passes the max diff check)**

and another version that still uses the Fraction module, since I wanted to see if most of the performance boost came from switching the algorithm to be an array based (hint: indeed, that is the case).

First version optimization deep dive

After viewing the clean version, it is clear that the clean version performs so slowly, mainly due to pure python interpreter overhead - frequent creation of temporary objects, repeated attribute lookups, and redundant control flow inside small loops. Thus, my main optimization strategy is aimed to mainly reduce the Python interpreter overhead. It will not include any huge algorithmic changes, the first major change I made was replacing the usage of Fraction model with floating point arithmetic. As stated before, this Fraction model is slow compared to pure floating-point logic.

Mainly, key changes in this domain look like changing lines such as:

```
def getCritDist(L, p, A1, A2, D1, D2, B, stab, te):
    p = min(p, Fraction(1))
```

To this:

```
def getCritDist(L, p, A1, A2, D1, D2, B, stab, te):
    p = min(p, 1.0)
```

Next, I added @lru_cache decorator heavy helper functions (as observed in the flame graph) such as getDamages, getCritDist. @lru_cache decorator caches results of function calls, which will eliminate the need for re-running the function with the same output.

```
23  @lru_cache(maxsize=None)
24  def getDamages(L, A, D, B, stab, te):
25      x = (2 * L) // 5
26      x = ((x + 2) * A * B) // (D * 50) + 2
27      if stab:
28          x += x // 2
29      x = int(x * te)
30      return [(x * z) // 255 for z in range(217, 256)]
31
32  @lru_cache(maxsize=None)
33  def getCritDist(L, p, A1, A2, D1, D2, B, stab, te):
34      p = min(p, 1.0)
```

Classic trade off between space and time complexity (since we use the internal CPU caches for results of functions).

Lastly, I re-wrote some of the logic of the battle class, it mainly included optimizations on Python level, removal of branches, removal of try except (which is very expensive approach, though popular in Python, and can be done using simple if else statements)

The main function call schemes in the clean version is:

getSuccessorsList -> getSuccessors -> _getSuccessorsB/C.

but now, we cache both of the views:

```python
def getSuccessorsList(self, statep):
    if statep[0] == 4:
        return []
    cached = self.successors.get(statep)
    if cached is not None:
        return cached[1]  # states_only
    # if not cached yet, compute once
    self.getSuccessors(statep)
    return self.successors[statep][1]
```

This means that for the same result, we will only call those functions once.

In the clean version, even if the same calculation is repeated, getSuccessorsList reconstruct the list each time!

```python
def getSuccessorsList(self, statep):
    if statep[0] == 4:
        return []
    temp = self.getSuccessors(statep)
    if statep[0] != 0:
        temp = list(zip(*temp))[0] if temp else []
    return temp
```

That is a huge difference. This should eliminate thousands of small list and tuple allocations per iteration.

As we can see, this optimized version was straightforward and didn't include doing some heavy lifting. It was pure Python overhead reduction.

Second version optimization deep dive

As we clearly saw before, the usage of Fraction module is indeed a bottleneck. However, there might be more bottlenecks out there. This is why for the second version I created two sub versions:

1. Main algorithm changes, while also removing the usage of the fraction module
2. Main algorithm changes, but **keeping** the fraction module, so can evaluate the importance of removing this module.

For this version, I will still cache important functions like the first version, while also removing the unneeded calls is the same state was visited before, but more importantly, I also added two key changes:

- A new build_graph() routine constructs the entire state graph once and assigns each state a unique integer ID. During evaluation, value iteration operates on pre-allocated Python lists (dmin_arr, dmax_arr, succ_states, succ_pairs) instead of dictionaries. This should drastically reduce per-iteration overhead from hashing, object creation, and dictionary growth.
- The critical inner loops were rewritten to avoid branching and repeated attribute access. Dictionary operations inside the probability accumulation were replaced by plain array indexing, and the freeze condition check was kept minimal, these changes should make the inner loop far more cache efficient.

On my new version, we build the state graph once, and not each loop. In addition, there is no usage of dicts, which are more expensive than python arrays.

Main changed logic is adding this function:

```python
def build_graph(self, initial_statep):
        from collections import deque
        q = deque([initial_statep])
        id_of = {initial_statep: 0}
        states = [initial_statep]
        kinds = []
        succ_states = []
        succ_pairs  = []

        while q:
            sp = q.popleft()
            i = id_of[sp]
            st = sp[0]

            if st == 0:
                nxt = self.getSuccessors(sp)
                ids = []
                for sp2 in nxt:
                    if sp2 not in id_of:
                        id_of[sp2] = len(states); states.append(sp2); q.append(sp2)
                    ids.append(id_of[sp2])
                kinds.append(0)
                succ_states.append(ids)
                succ_pairs.append(None)

            elif st == 4:
                kinds.append(4)
                succ_states.append([])
                succ_pairs.append([])

            else:
                nxt = self.getSuccessors(sp)
                pairs = []
                for sp2, p in nxt:
                    if sp2 not in id_of:
                        id_of[sp2] = len(states); states.append(sp2); q.append(sp2)
                    pairs.append((id_of[sp2], p))
                kinds.append(st)
                succ_states.append(None)
                succ_pairs.append(pairs)

        return id_of, states, kinds, succ_states, succ_pairs
```

This function, precomputes the entire MDP graph once, this function does a BFS of all of the reachable states from the initial state. (instead of dynamically calling getSuccessors() and building lists on every iteration)

The function returns few key variables that are later integrated into the code, I'll summarize it in a table:

Var	Description
Id_of	Maps each state tuple to a unique integer ID
States	Reverse mapping of IDS back to the original Python tuples
Kinds	Type per node (for example choice, chance, terminal..)
Succ_states	List of successor nodes IDs for each choice node, None for the rest.
Succ_pairs	List of successor IDs for each chance node – pair of ID, probability. None for the rest.

Later in the main evaluate function, the key changes were made:

```
id_of, states, kinds, succ_states, succ_pairs = self.build_graph(initial_statep)
n = len(states)

dmin_arr = [0.0] * n
dmax_arr = [1.0] * n
frozen_a = [False] * n

# seed terminals exactly as before
if self.loss in id_of:
    i_loss = id_of[self.loss]
    dmax_arr[i_loss] = 0.0
    frozen_a[i_loss] = True
if self.win in id_of:
    i_win = id_of[self.win]
    dmin_arr[i_win] = 1.0
    frozen_a[i_win] = True

i_init = id_of[initial_statep]

order_ids = [id_of[sp] for sp in topoSort([initial_statep], self.getSuccessorsList)]


while dmax_arr[i_init] - dmin_arr[i_init] > tolerance:
    for i in order_ids:
        if frozen_a[i]:
            continue

        k = kinds[i]
        if k == 0:
            # choice node
            best_min = float('-inf')
            best_max = float('-inf')
            for j in succ_states[i]:
                vmin = dmin_arr[j]; vmax = dmax_arr[j]
                if vmin > best_min: best_min = vmin
                if vmax > best_max: best_max = vmax
            dmin_arr[i] = best_min
            dmax_arr[i] = best_max

        elif k == 1 or k == 2:
            # chance node
            smin = 0.0
            smax = 0.0
            for j, p in succ_pairs[i]:
                smin += dmin_arr[j] * p
                smax += dmax_arr[j] * p
            dmin_arr[i] = smin
            dmax_arr[i] = smax

        if dmin_arr[i] >= dmax_arr[i]:
            mid = 0.5 * (dmin_arr[i] + dmax_arr[i])
            dmin_arr[i] = dmax_arr[i] = mid
            frozen_a[i] = True

return 0.5 * (dmax_arr[i_init] + dmin_arr[i_init])
```

Main changes include the init of python arrays, and not dicts like the clean version. In addition, the value iteration (big while loop) doesn't create a new graph each iteration but uses the prebuilt graph. In addition, we are operating on arrays, which is more efficient than dicts.

Main changes in the code include:

What have changed	Clean version	My version
State representation	Tuple	Int – ID
Value	Dicts (dmin, dmax)	Lists
Graph access	Multiple creation under each value iteration (using getSuccessors)	Creates a precomputed graph - succ_states[i] and succ_pairs[i]
Frozen states	Set	Bool array

There were some extra small optimizations: removal of div operations, avoid accessing objects inside the loop if I can access them once outside the loop, avoid unneeded max function – moving to manual min max tracking, etc.'

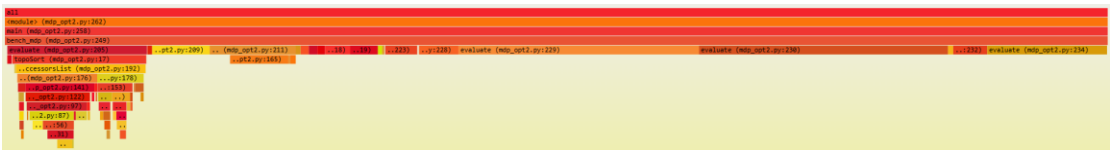
Results section

To gather the results properly, what I did was to create a script with three stages (the same I used for AES):

1. Run the native python MDP implementation
    a. Run that creates the flame graph
    b. Run perf stat for 5 times (between each run I flush the cache)
    c. Run the benchmark using pyperformnace wrappers.
2. Run the first optimized python MDP implementation
    a. Run that creates the flame graph
    b. Run perf stat for 5 times (between each run I flush the cache)
    c. Run the benchmark using pyperformnace wrappers.
3. Run the second optimized python MDP implementation
    a. Run that creates the flame graph
    b. Run perf stat for 5 times (between each run I flush the cache)
    c. Run the benchmark using pyperformnace wrappers.
4. Run the third optimized python MDP implementation (the same one as the second optimized version, but with the Fraction module brought back)
    a. Run that creates the flame graph
    b. Run perf stat for 5 times (between each run I flush the cache)
    c. Run the benchmark using pyperformnace wrappers.

Optimized flame graph version 1

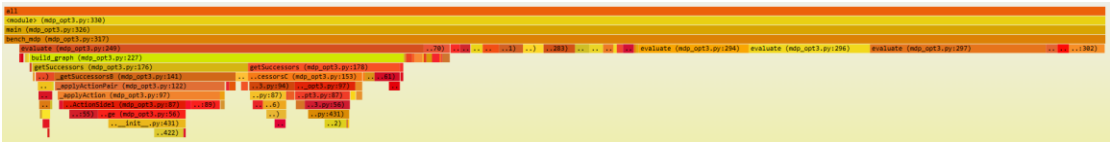
This is the first version flame graph:



We can clearly see that a lot of previously wasted time on toposort, has now decreased a lot. This makes sense completely, since the main changes we focused on this version was the remove unneeded calls to the hot functions, usage @lru_cache decorator for the same results. So, I'm expecting to see a decent decrease in the time spent on this version, as well as branch reductions, instructions reductions (less repeated function calls).

Optimized flame graph version 2

This is the second version flame graph:



Now, the build graph indeed takes quite a bit of time, but we will see later in the results that the overall time was reduced dramatically.

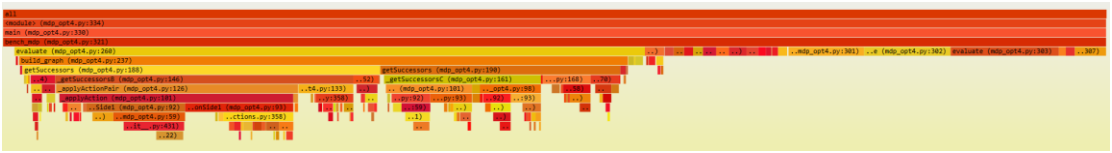
One interesting case is that on the first optimization version, the last if which did this:

```
233
234                 if dmin[sp] >= dmax[sp]:
```

Took a lot of time, but here, using arrays, it takes less time. I suspect that this is due to the fact that we are using arrays. Overall, the time the benchmark runs has decreased by quite a bit, this is mainly because we changed the algorithm and used arrays and not dicts. It can be seen in the flame graph – main time is now spent on pure CPU bound stuff – such as a for loop sum (which is needed for the algorithm).

Optimized flame graph version 3

This is the third version flame graph:



There is nothing special about this, but we can see that the re-addition of the Fraction module has its cost – not the build graph takes quite a bit of time (55%~ of the time the benchmark is running!) this can clearly be explained by the fact that the fraction module is far slower than the usage of normal floating point, as explained before.

Numerical results

Full results will be attached to this report on an HTML and excel, here is the full results attached as a table:

	mdp_clean	mdp_opt2	mdp_op4	mdp_opt3
time	89.12	49.17	23.85	16.68
instructions	472,798,271,875	268,486,449,356	129,175,090,586	94,424,121,599
cycles	231,334,954,673	127,634,468,823	61,884,809,547	43,292,747,562
IPC	2.04	2.10	2.09	2.18
context-switches	261.00	152.00	78.00	60.00
page-faults	5971.00	6424.00	6791.00	7486.00
branches	112,590,248,650	63,217,148,827	30,662,176,589	22,294,886,655
branch-misses	847,069,184	427,698,832	225,468,079	145,791,914
L1-dcache-loads	127,674,335,928	70,781,229,380	36,903,230,679	27,084,594,396
L1-dcache-load-misses	2,460,060,767	921,255,002	769,171,203	356,565,880
LLC-loads	294,814,849	235,675,242	157,876,992	129,009,581
LLC-load-misses	1278518.00	581456.00	859491.00	678415.00
dTLB-loads	127,698,191,835	70,744,535,008	36,833,951,064	27,216,481,664
dTLB-load-misses	56683536.00	17230771.00	14644673.00	8839893.00
iTLB-loads	108,064,221	14744982.00	44608244.00	11861893.00
iTLB-load-misses	2233451.00	1011329.00	1949767.00	729008.00
Speedup	1.00	1.81	3.74	5.34

Mdp_clean is running the benchmark without any modifications to the MDP. Opt2 is the first version described here, op4 is the third version described here (with the usage of Fraction module) and opt3 is the second version described here.

Looking at the results, it is clear that the best version is number three. Which, both removed the Fraction usage, and changed the algorithm. IPC for versions 2,4 are relatively close, while the IPC for the version 3 is the best. I suspect that version number 4 and 2 have similar IPC, due to their native tradeoffs – in version number 4 we removed the heavy dict lookups, but kept the Fraction module usage – which uses GCD for example which is a heavy compute instruction (for example, it includes div), while on version number 2 we removed the Fraction module, but kept the dict lookups – which is expensive as well.

Starting by analyzing the results version by version, only removing the Fraction module, and adding some minor optimizations, we gain 1.81 speedup, which is a very good speedup. It can be clearly seen that almost all of the key metrics contain a decent drop. However, one metric that went up on this version is the amount of page faults, which increases by ~500. I suspect that is due to the optimizations made in this version: we added some caching to heavy functions, and this caching caused the program to have a bit more sparce code and

memory footprint. I want to clarify that the overall code footprint and memory footprint is smaller (as we can see that the number of loads has decreased) but the location of each data that is needed sits on different pages, causing a small increase in the number of page faults. The usage of lru_cache increases the heap size. However, this is only metric that has increased and overall get got a good speed up.

The next version I'll focus on, is number 3. Version number three is the best in terms of speedup – we achieve a very good speed up of 5.34x compared to the baseline. Nothing surprising, since we used the optimizations on version number 2, and added on top of it a change in the algorithm, which dramatically reduced the number of works needed to be done on the benchmark, as we can see that the number of instructions to run is the lowest. IPC wise, this version is still the best and reaches an IPC of 2.18. looking at all of the metrics, version number 3 has the best metrics possible (lowest when it should be lowest, highest when is should be highest). However, two metrics that we "lose" to version number 2, we can see a slight increase in the number of page faults (also when comparing to the baseline in this case), and the amount of LLC loads misses increases, when comparing to version number 2. It is important to understand why this might happen. My main assumption is that version number 3 had a decent trade off between time and memory, I allocated large python lists, instead of creating a very small dicts in each iteration, (the way version 2/ baseline did the algorithm), thus, in this version, we heavily rely on contiguous python arrays. So, most of the work, the CPU can operate on L1/L2. We can see that the amount of LLC loads in general is smaller, so, ironically, the fact that we have a bigger LLC load miss, basically means that our memory usage is better than version number 2. In version number 2, probably most of the memory is small enough to sit in the LLC (which is bigger than L2/L1) however, in our version, the memory layout is more efficient, so most of the work is being done on L1/L2 level. LLC misses increase since we increase the number of compulsory misses (we don't have any reuse of a cache line right now – we are looping each cache line once). Unlike version 2, where we might reuse the same dicts, but it is less efficient, since we need much more memory (as we can see that the amount of LLC loads is bigger in version 2). This can also be clearly seen by the amount of L1 load misses going down by a third.

Version 4 also tells us a lot about the story, since it is the same as version 3, but with the re-addition of the fraction module (so that we can understand how much gain we have from this added precision). Main key metrics that behaves differently is the amount of ITLB loads goes up by a **lot.** It can be explained do to the fact that now we are using the Fraction module, so the code footprint

increases, since this library includes new instructions to be carried out by the CPU, now the benchmark needs to run the Fraction module code.

In terms of speedup, we gain 3.74x speedup compared to the baseline, so this version sits right in the middle between the best version (number 3) and number 2. This also tells us the removal of the Fraction module is worth extra ~50% speedup (5.7 / 3.7). not a small amount. As said previously, IPC is very close to version number 2. Which tells us that the algorithmic changes + Fraction module addition "cancel" each other out (**at least in terms of IPC!**)

Hardware accelerator proposal

As we saw, the main overhead and compute time goes on the big while loop which gets a step we are currently in and computes the next state value. For choice nodes we take the max out of all of the successors, and for chance node we take the weighted average of the successors. This computation is relatively slow, but we can build a dedicated chip unit that will perform this for each step, in parallel, something like this scheme:

For each state, read its successors, compute max / sum (depending on the type of the state) in the benchmark, we have 7k~ states. So, what we will do is build a simple 5 pipeline unit that will, after pipe warmup, will fetch a new state each clock cycle.

The hardware accelerator will replace the big while loop in code we had, based upon version number 3 of the optimized algorithm. Meaning that we will prebuild the graph, and store everything in SRAM. The hardware will do as follows:

Fetching the successors given a state id. Succ_start is the start address, and succ_end is the end address of its successors. (state fetch at the diagram below)

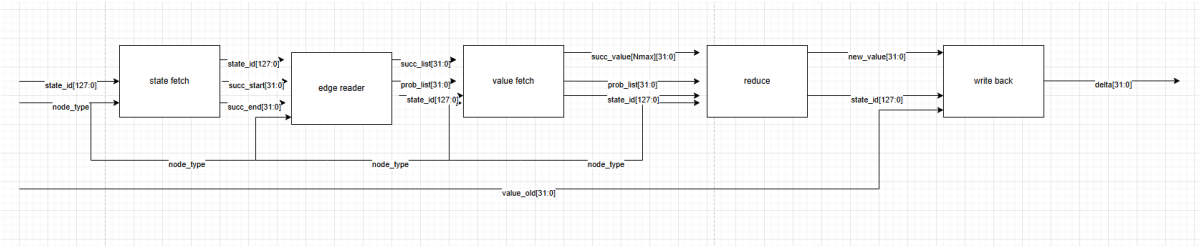
Reading all the successors indices for that state. (called Edge reader at the diagram below)

Getting the current value of each successor (value_fetch at the diagram below)

Reduce: performing the bellman update.

For choice node – we will take the maximum out of neighbors, and for probability we will take the probability times the successors value (weighted sum). (reduce at the diagram below)

Store the computed value, compute the convergence value (the stopping condition) and output it to the software. (write back at the diagram below).



We can also make this more parallel, at the cost of more hardware, for example we can have 16 duplicates so that we can work on 16 states each cycle.

Summary

In this report I provided a through analysis of the MDP algorithm, I showed you how I first analyzed the clean benchmark, provided explanations on why I picked this benchmark instead of others, presented my opinion on what optimizations areas I will emphasize here, provided a deep analysis of all of the improved versions I made, as well as provided explanations on the results we got. Lastly, I provided a suggestion for a hardware accelerator that can be integrated under a unique socket, that can replace the heaviest compute work needed in the benchmark – the value iterations.