Final Project 4 Preliminary Report

Yanping Chen, Lekha Kuhananthan, Frank Li, Albert Wang, Chen Zhao

Profile Data for Reference Implementation

Perf Statistics Data

We ran perf on our program searching to different depths.

Depth 1:
```
 Performance counter stats for './khetplayer':

            256809982  branches                  #      0.000 M/sec
            399085  branch-misses                #      0.000 M/sec
            951759763  cycles                    #      0.000 M/sec
            1155672596  instructions             #      1.214 IPC

            0.041270387  seconds time elapsed
```

Depth 2:
```
 Performance counter stats for './khetplayer':

            2630286291  branches                 #      0.000 M/sec
            20991802  branch-misses              #      0.000 M/sec
            9469884249  cycles                   #      0.000 M/sec
            11844542731  instructions            #      1.251 IPC

            0.360605955  seconds time elapsed
```

Depth 3:
```
 Performance counter stats for './khetplayer':

            26158065346  branches                #      0.000 M/sec
            236633640  branch-misses             #      0.000 M/sec
            96020300246  cycles                  #      0.000 M/sec
            117729341952  instructions           #      1.226 IPC

            3.614371323  seconds time elapsed
```

Depth 4:
```
 Performance counter stats for './khetplayer':
         654039011259  branches              #         0.000 M/sec
         975779697  branch-misses            #         0.000 M/sec
         2286770651768  cycles               #         0.000 M/sec
         2944552551875  instructions         #         1.288 IPC

  85.810726360  seconds time elapsed

Depth 5:
 Performance counter stats for './khetplayer':

         2589244513345  branches             #         0.000 M/sec
         5376323916  branch-misses           #         0.000 M/sec
         9038687939830  cycles               #         0.000 M/sec
         11655148394441  instructions        #         1.289 IPC


  339.156201666  seconds time elapsed

Depth 6:
 Performance counter stats for './khetplayer':

         52562064899771  branches            #         0.000 M/sec
         247535132690  branch-misses         #         0.000 M/sec
         186067098999651  cycles             #         0.000 M/sec
         236594018658336  instructions                 #         1.272 IPC

 7217.922479786  seconds time elapsed
```

Callgrind Data

In addition to running perf, we also ran callgrind (a valgrind tool) on the initial implementation, which we considered a larger profile coverage than gprof. Callgrind measures the performance cost of all executed code, and displays the costs for each called functions and a breakdown of the cost of each line in those function. The performance costs Callgrind measured were the number of calls/instruction fetches, and data reads/writes. Data was reported inclusively, meaning a functions performance cost included the cost of subroutines within that function, and as a percentage of total program cost.

Most costly functions in a depth 3 search:
(all values are in terms of percent of total program time out of 100%, inclusive. So, A() - 80% means that the function A, and all the functions it calls within itself, take up 80% of the total program time)

Instruction fetch:
ABSearch::ABSearch - 99.61
ABSearch::ABRootSearch - 99.59
ABSearch::search - 99.26
KhetState::getPossibleStates - 97.52
KhetState::KhetState constructor (2 arguments) - 86.15
KhetState::makeMove - 84.43
KhetState::alg - 62.12
KhetState::gen - 18.82
KhetState::KhetState constructor (1 argument) - 10.78

Data Read/Write Accesses:
ABSearch::ABSearch - 99.58
ABSearch::ABRootSearch - 99.58
ABSearch::search - 99.25
KhetState::getPossibleStates - 97.42
KhetState::KhetState constructor (2 arguments) - 84.15
KhetState::makeMove - 82.56
KhetState::alg - 58.20
KhetState::gen - 20.91
KhetState::KhetState constructor (1 argument) - 12.68

Optimizations already done

We implemented a bunch of optimizations based around cutting out the wasteful operations of KhetState, and to a lesser extend, changing the order of operations on ABSearch.

ABSearch:
-ABState is changed to instead of returning a list of possible next states, to return a list of possible moves (ABMove) from the given state.
-A method is added to ABState to get the next state given a move (ABMove) on the current state.
-ABSearch is changed to deal with vectors of ABMoves instead of ABStates. Calculation of next state from current state is not done until the next level of the search tree - this means that transitions from the current state by moves that are pruned are not actually calculated.

KhetState:
-The numerous useless calls to alg() are removed, instead a comparator for KhetMove was added.
-Using assumption that all moves passed into KhetState constructor are valid moves from the given states, we instantly make the move instead of checking it against the list of possible moves.
-Caching of moves vector - it is created once and only once.
-Making KhetState (kinda) immutable - makeMove method returns a new KhetState instead of modifying the original state.
-Caching of the key - we finished the zob hashing implementation - the hashBoard is only calculated on creation of a new KhetState form a new board, otherwise, it is updated by only XORing the squares that are changed.
-Caching of KhetState - KhetStates are cached, so that there is only 1 of the same board representation. The caching is done by the key calculated from zob hashing, using 64bit ints, so collisions should be few. The caching method is using a std::map - this may be changed in the future, but was the easiest “correct” thing to do

Some Major improvements and their Impact:

Below, we list some of the more critical improvements, and their impact, backed up with profiling data. Here, most of the data is in terms of percent of Instruction Fetch, e.x. “60% IF” means that a function took up 62.98% of all instruction fetches in the running of the program as Callgrind reports.
-removal of useless alg() calls, making a KhetMove comparator, and directly using moves we know to be safe took out the first bottleneck, which was string.compare() in makeMove - this took up about 60% of original IF (all improvements below are in terms of IF after this change)
-proper caching of the key from hash_board was the second bottleneck, taking up about 20% IF, so removing this again gave a decent speed up
-caching of moves vector (gen()) - after the alg() change, this was another crucial bottleneck, taking up about 20% IF
-Removing unneeded elements from structs - mainly removing KhetPiece from KhetMove - cut down on the memory the program was using (from about 40GB to 30GB for a 6 ply search)
-Caching and reusing KhetStates - This gave a modest speed up (2~4X), but the main performance increase was greatly cutting down on the memory we used. Since we moved to using pointers to KhetStates and did not bother freeing things at the end of a search, each run to depth 6 took up about 20~30GB of RAM. After this change, ram usage dropped to about 200~300MB

Overall, this gave about a 7200 times improvement, since searching to depth 6 only took about 1 second instead of the 2+ hours it used to take.

Profile of (current) Optimized Implementation

We took similar profile data as gathered with the reference implementation. As you can see, many of the functions we targeted with optimizations no longer are as costly as before, and now we have new functions to target.

Depth 1:

```
 Performance counter stats for './khetplayer':

        245494655  branches                #      0.000 M/sec
        307538  branch-misses              #      0.000 M/sec
        916558228  cycles                  #      0.000 M/sec
        1105858367  instructions           #      1.207 IPC

        0.039082563  seconds time elapsed
```

Depth 2:

```
 Performance counter stats for './khetplayer':
```

```
          221333015  branches                  #        0.000 M/sec
          403393  branch-misses                #        0.000 M/sec
          837670845  cycles                    #        0.000 M/sec
          1000602100  instructions             #        1.195 IPC

          0.041902400  seconds time elapsed

Depth 3:
 Performance counter stats for './khetplayer':

          403262747  branches                  #        0.000 M/sec
          1546865  branch-misses               #        0.000 M/sec
          1448498230  cycles                   #        0.000 M/sec
          1830517485  instructions             #        1.264 IPC

          0.062359489  seconds time elapsed

Depth 4:
 Performance counter stats for './khetplayer':

          851003175  branches                  #        0.000 M/sec
          3385934  branch-misses               #        0.000 M/sec
          3014089936  cycles                   #        0.000 M/sec
          3867770153  instructions             #        1.283 IPC

          0.123171200  seconds time elapsed

Depth 5:
 Performance counter stats for './khetplayer':

          3496552180  branches                 #        0.000 M/sec
          15921767  branch-misses              #        0.000 M/sec
          12166427999  cycles                  #        0.000 M/sec
          15912500578  instructions            #        1.308 IPC

          0.465011828  seconds time elapsed

Depth 6:
 Performance counter stats for './khetplayer':

          9755548583  branches                 #        0.000 M/sec
          46193203  branch-misses              #        0.000 M/sec
          34112640360  cycles                  #        0.000 M/sec
```

```
    44467935743  instructions              #    1.304 IPC

    1.291756923  seconds time elapsed
```

Callgrind Data (search to depth 5)

Instruction Fetches:

ABSearch::ABSearch	92.45
ABSearch::root_search	92.32
ABSearch::search	91.70
KhetState::evaluate	45.50
eval()	45.45
KhetState::getPossibleStates	26.87
KhetState::fireLaser	19.89
KhetState::makeMove	15.85
KhetState::getKhetState	15.81
adjacentEmptySquares	13.60

Data Read/Write Accesses:

ABSearch::ABSearch	92.16
ABSearch::root_search	92.15
ABSearch::search	91.50
KhetState::evaluate	44.24
eval()	44.18
KhetState::getPossibleStates	29.59
KhetState::fireLaser	19.55
KhetState::makeMove	14.81
KhetState::getKhetState	14.78
adjacentEmptySquares	13.84

Planned Optimizations

Looking forward, the optimizations we have planned are (in order of priority):

ABSearch
-implementing heuristics to cut down on search space - we really have not done much in this regard, so we should be able to get a decent speed up in this category, this includes:
--Don’t search repeat states
--Search first the best moves in each ply
--Proper caching of results
--Quiescence search
--Killer moves

While this optimization doesn’t target any actual program bottleneck, it’s an algorithm change that should prune up to 99% or more of the tree we are searching, hence resulting in a very good speedup. As we can see in our profile data, ABSearch functions are currently quite costly.

KhetState
-Change datatypes - KhetPiece, KhetMove, KhetBoard could be made more efficient, most specifically, piece and move can fit in 32 and 64 bit ints respectively, and there should be better ways of representing the board as well - This optimization should drasticly reduce the amount of memory we use, and might have the nice side effect of speeding up the program through passing smaller structs by value
-Calculating all next moves can be made more efficiently, perhaps implement a system where moves are incrementally calculated, since pruned moves are unneeded - it is unsure how much of a performance benefit this might have, but could be useful to implement as part of the changes to ABSearch (incremental move generation)

We’ve already made some fruitful changes, but there is more to be made. As the profile data shows, behind ABSearch functions, the remaining costly functions are mostly in KhetState.

Eval
-Calculation of the laser could be done more efficiently - Perhaps with bithacks or matrix operations backed by Intel’s linear algebra library this could give a decent/good speedup
-Eval calculation could be made more efficient, might require tie-in code change in KhetState to facilitate this - eval is taking up about 25% IF right now, so an improvement here could give a good speedup
-Experiment with different evals - not so much a performance change but a game play/strategic change.

This should be a pretty helpful optimization, being that eval is now one of our top expensive functions according to our profile data.

khet
-Time control - we currently have none, and this will be crucial in actual matches
-Parallelize - we decided to not parallelize this for now, since parallelizing (the easy way) did not give any performance improvements, but in the long run, we will need to put this in. However, compared to say cutting down on number of nodes searched or removing inefficiencies in KhetState, this is a second priority since at best, parallelization would only give a 12X improvement on 12 cores.

This is just game management stuff that we should be dealing with. We expect some improvements from these optimizations but not a ton.

Work Division:

This is a tentative work division for the next week of the project that will change as people become more/less free to work on this project:
-Chen, Frank, Yanping will be working on ABSearch, implementing heuristics to cut down on the number of nodes searched.
-Albert, Lehka will be working on optimizing the eval and associated functions, time control, and parallelization.