Function-based thread scheduling using critical path

Proposal draft - v0.1

**CONCEPTUAL DESIGN - PHASE 1**

**1) Mapping of the application**

*Dynamic tracing is better than static because it allows us to see what the hotspots are and the entire process can be made online. Plus, if the software starts behaving in an unexpected way, the scheduler may have some heuristic to adapt. This step is supposed to output the most intensive functions.*

*One way to implement it is through tools like jvmtop. By profiling it during a sample time with jvmtop, you may have all the load-intensive functions. perf-map-agent is also a (easier, but maybe not reliable) solution.*

**2) Determine the critical function**

*I think human intervention here is probably a must because not always the most used function is the most relevant. By choosing a certain function, we can traceback it and set it as the “critical path” of that application. Of course, that’s assuming there’s only one CP.*

*The critical path is not really a tree. May be simply a list sorted on ascending order (which means = higher the index, the near the thread is to the most intensive function).*

**3) Traceback the chosen function.**

*If “E” is the function we chose on “2”, then the critical path would be A -> B -> C -> D -> E (and others “branches”). The order number to traceback is dependent on the application and basically goes with testing.*

*The traceback can be easily done through static mapping (java-callgraph) or online through jstack.*

**CONCEPTUAL DESIGN - PHASE 2**

**High-level algorithm:**

1. *A signal to the scheduler is sent each time a thread enters the beginning of the critical path, moves on it or exits it.*
2. *If the actual thread is entering or moving the critical path, check if there’s free big cores. If yes, check if there are others threads looking for allocation and determine if it should be allocated or not based on the index. If not, runs on small core until next signal.*

*3. If the actual thread is exiting the critical path (through some branch or by end of execution), the space on big/small core is freed and the one with highest CP-index + time at CP is allocated on big core .*

*4. If there’s free space on the big core and no threads on CP, allocate the oldest thread of the running application.*

If the idea sounds okay, the instrumentation may be done with ASM which can send signals to the C/Python scheduler each time certains functions are activated.

**Pseudo-code: (Partly based on EETL paper)**

Input*: number of big cores, number of small cores, critical path list*

Output*: core allocation for current request*

Variables:

*F = set of Fast Cores; t = actual thread;*

*T = other active threads; a = time of the current thread at CP*

|  |  |
| --- | --- |
| *on SIGNAL\_RECEIVE:*  *if t ∈ CP:*  *check if F is full;*  *if false:*  *check if ∃T ∈ CP and T ∉ F;*  *if true:*  *allocate the one with highest index and a;*  *if false:*  *allocate t in F;*  *if true:*  *allocate t in S;*  *if t ∉ CP:*  *free F;*  *check if ∃T ∈ CP and T ∉ F;*  *if true:*  *allocate the one with highest index and a;*  *if false:*  *allocate t in F;* | // Activation of scheduler  // Check if our current thread is part (i.e.: entering or exiting) the critical path  // Are there any free space in F?  // If F is not full, then…  // Are there any others threads at CP and not running at big core?  // if there are others threads at CP, then…  // we weight the CP-index along the time it already spent on CP.  // If the actual thread is the only one at CP  // We simply allocate it on small core.  // If F is full, we simply let the thread run at small core.  // if t is not at CP (exiting it):  // Free the space used by that thread |

This algorithm is async, but I think if we define some arbitrary threshold, it might work on the same idea (check the threads with highest index/age and allocate it).