Profiling the AV Stack for Processor Design

Kevin Rao

Abstract

The development of Autonomous Vehicles has begun to place more and more emphasis on software hardware co-design, where software is written with the underlying hardware in mind, and hardware is developed to be software specific. This practice often leads to better results than the two areas working in ignorance of the other. Unfortunately, co-design is often slow and costly.

To help architects understand how AV software will perform on a library of cores, we present the Critical Heuristic Earlist Finishing Time (CHEFT) algorithm, which schedules tasks in a dataflow on a set of homogeneous processors while respecting precedence requirements. We also introduce a novel benchmarking method to evaluate the performance of our algorithm.

The algorithm and all code used to generate the diagrams in this work are available at `https://github.com/kevinrao99/cs249-cheft`

1 Introduction

Recently, the development of Autonomous Vehicles (AVs) has put more and more emphasis on software and hardware co-design. Software should be written with the underlying system and hardware in mind, and hardware development should be guided by software requirements. Unfortunately, a priori it's often difficult to understand how software and hardware will interact. In particular, given the design and structure of AV software, it's difficult but important to know what specs are required of the underlying hardware to achieve certain performance thresholds. Developing hardware first and testing performance experimentally is often expensive and wasteful, so a theoretical framework for understanding how a piece of software will perform on the underlying hardware would save time, effort, and give architects clear goals to work towards.

Given a library of cores with specified homogeneous processing speeds, we construct a scheduling algorithm that maps tasks on a subset of the cores to minimize end to end latency. In order to evaluate the quality of our scheduling algorithm, we also describe a novel benchmarking method based on mathematical properties of the workflow. Finally, we demonstrate how our software can be used by running it on a workload used in MAVBench [3].

2 List Scheduling

Scheduling tasks on processors with precedence conditions is an important and well studied problem both in theoretical and applied settings. Unfortunately, in most settings the scheduling problem is simply a variation of the Job-Shop Scheduling problem, which is known to be NP-hard because it's the general form of the Travelling Salesman problem. Due to the complexity of optimal scheduling, most work focuses on restricting the problem domain and finding suboptimal, lower complexity approximation algorithms. One commonly employed technique is identifying important features of the dependency graph and using heuristic methods. In [2], Topcuoglu et. al. propose scheduling tasks by their upward rank in the dependency graph and assigning each to the processor that minimizes earliest finishing time with an insertion approach. They also try combining upward and downward rank, which performed either better or worse, depending on the benchmark and setting. Our algorithm is a modified version of the Improved Predict Earliest Finish Time (IPEFT) algorithm, proposed in 2016 by [1] and based on the critical node heuristic.

2.1 Our problem

In AVs, the data flow through the software can be modelled as a directed acyclic graph, where each task has some processing weight and certain tasks depend on the output of others. Most literature in the theoretical community focuses on computing in a heterogeneous environment, where processors may run at different speeds on different tasks. For our problem we assume that processors are homogeneous and each task has a processing weight, so the processing time of a task on a processor is simply the weight of the task divided by the processing speed. We also make the simplifying assumption that the cost of communicating between tasks and processors is 0 for two reasons: 1) In the workflow of AVs the amount of data that needs to be passed between tasks is often small compared to the computational complexity of the tasks themselves, and 2) The cost of communicating may depend on such factors as the cores' relative physical positions, which can be difficult to model in a theoretical setting. We also assume that the only concern is latency from input to output, where in applied settings scientists are also concerned with pipelining a continuous stream of inputs.

3 Critical Heuristic Earliest Finishing Time

Working off of ideas employed in the IPEFT algorithm [1], we construct and implement the Critical Heuristic Earliest Finishing Time algorithm (CHEFT), which is specifically designed for the homogeneous computing setting. Given a set of tasks of specified processing weights and dependendies, and given a set of processors, it computes an almost-optimal schedule of tasks onto processors that minimizes the makespan of computing our entire task graph.

3.1 Definitions

Suppose we have a dataflow with N tasks. We model the dataflow as a directed acyclic graph where each task is a vertex, and if task b depends directly on the output of task a, we include the edge (a, b) in our graph. We label the tasks $v_1, v_2, \ldots v_N$ and define w_i to be the processing weight of task i (if the meaning is clear, we also write the weight of a vertex v as `w[v]`). We often refer to `succ[v]` and `pred[v]`, the array of immediate successors and predecessors of vertex v, respectively. For convenience, we add dummy entry and exit nodes, `V_ENTRY` and `V_EXIT` each with weight 0, where `V_ENTRY` is a predecessor of every task and `V_EXIT` is a successor of every task.

The Earliest Start Time (EST) of a vertex v is defined recursively as

$$EST(v) = \max_{v' \in \texttt{pred[v]}} EST(v') + \texttt{w[v']}$$

or the maximum over all predecessors v' of v of the earliest start time of v' plus the processing time of v'. The base case is $EST(\texttt{V\_ENTRY}) = 0$.

We note that this notion of EST is borrowed from the IPEFT algorithm, where they compute the Average Earliest Start Time (AEST) by averaging a task's processing time over all available processors. If we do the same in the homogeneous setting, we would simply be scaling all task weight values by the same amount. For example, if we have homogeneous processors p_1, p_2, p_3 with speeds 3, 5, and 10, the average processing time of a task v is

$$\frac{1}{3}\left(\frac{v}{3}+\frac{v}{5}+\frac{v}{10}\right)=\frac{1}{3}\left(\frac{10v+6v+3v}{30}\right)=\frac{19}{90}v$$

. Since all task weights are scaled by the same amount, computing the AEST is the same as just computing the EST by using each task's weight as a proxy for its average processing time.

The Latest Start Time (LST) of a vertex v is similarly defined as

$$LST(v)=\min_{v'\in\texttt{succ[v]}} LST[v']-w[v]$$

where our base case is $LST(\texttt{V\_EXIT})=EST(\texttt{V\_EXIT})$.

We say a node v is *critical* if and only if $EST(v)=LST(v)$, and we define the Critical Node Predecessor (CNP) of a node v to be true if and only if some successor of v is a critical node. The intuition behind the definition of a critical node comes from imagining the "flexibility" of when we process each node. On one extreme end, if we have a task v_0 that is totally independent of all others, its earliest start time is 0 and its latest start time is $LST(\texttt{V\_EXIT})-w[v_0]$, so it can be scheduled whenever there happens to be a free block of time in a processor. On the flip side, if our dataflow is on single linear path, every single node is considered critical because we know precisely when it needs to be scheduled to minimize the makespan of our dataflow. In this case, if we try to schedule a node later than its EST, we just delay all nodes down the line. Thus, a node is critical if minimizing the makespan of the whole system depends on scheduling it precisely at its earliest start time. If a task is a critical node's predecessor, we must prioritize completing it as early as possible so that we can start processing all critical nodes as soon as possible.

We define the Pessimistic Cost Table (PCT) of a node v to be the longest path from any of the successors of v to the exit task, defined recursively as

$$PCT(v)=\max_{v'\in\texttt{succ[v]}} PCT(v')+w[v']$$

We define the Critical Node Cost Table (CNCT) of a node v to be $PCT(v)$ v is not a CNP, and

$$CNCT(v)=\max_{v'\in\texttt{succ[v]}\cap EST(v')=LST(v')} PCT(v')$$

If v is a CNP. The $rank_{PCT}$ of a node $[v]$ is simply the pair $(PCT[v], w[v])$, and we use it because later it will be useful to break ties in PCT by the node's processing weight. We also compute the topological sort, reverse topological sort, and maximum weight path of our graph with the standard algorithms. We specifically define the above features of our graph to be completely independent of the speed and number of our processors, so unlike in the IPEFT algorithm they can be computed a priori and don't need to be recomputed on a different set of processors given the same graph.

We now move on to defining values that need to be computed during the execution of our scheduling algorithm. We implement each processor's schedule as an array of triples `(start_time, end_time, task_id)`, sorted by start time (which is the same as sorting by end time, since a processor's schedule is a set of disjoint, left-closed and right-open intervals). The Actual Finishing Time (AFT) of a vertex is the end time assigned to it by our scheduling algorithm.

The earliest time available of scheduling task v on processor p, written $T_{av}(v, p)$, is defined as the earliest time we can insert v into p's schedule and have a contiguous block of processing time long enough to complete v, under the condition that v starts after all predecessors of v are finished computing. On an implementation level, we accomplish this by first iterating over all predecessors v, finding the latest finishing time, and then searching the processor's schedule from then onward. The Earliest Finishing Time (EFT) of a node v on a processor p is simply the time available plus the time it takes to complete v on p.

3.2 Algorithm

Now that we've defined all the necessary terms, we can describe the way our algorithm works. Given that we've precomputed all the features of the graph, we push all vertices onto a max heap ordered by $rank_{PCT}$. While our max heap is not empty, we pop off the top node v and attempt to schedule it. If v is a CNP, we prioritize only its EFT. Otherwise, we also care about the distance to `V_EXIT`, so we compute `EFT(v, p) += CNCT[v]`. We then assign v to the processor that minimizes the EFT and continue with the next task. This pseudocode gives a rough overview of the CHEFT algorithm:

Algorithm 1: CHEFT

```
Data: The dataflow as a graph, each task's processing weight, and each
      core's processing speed
Result: A schedule on the set of cores
Compute the topological sort, maximal antichain, and max weight path
 of the graph
forall nodes in the graph do
    Compute $EST, LST, CNP, CNCT, PCT, rank_{PCT}$
end
**
Sort nodes in descending order by $rank_{PCT}$
while There still remain unscheduled tasks do
    Let $v$ be the next node
    for Each processor $p$ do
        Compute the EFT of assigning $v$ to $p$, or EFT(v, p)
        if $v$ is not a $CNP$ then
            EFT(v, p) += CNCT[v]
        end
    end
    Assign $v$ to the processor that minimizes EFT and update AFT[v]
end
Return the AFT of V_EXIT
```

Notably, the actual CHEFT algorithm starts on the line with the double asterisk, while everything before is precomputed for the task graph. The step of computing the maximal antichain will be discussed in benchmarking.

4 Benchmarking

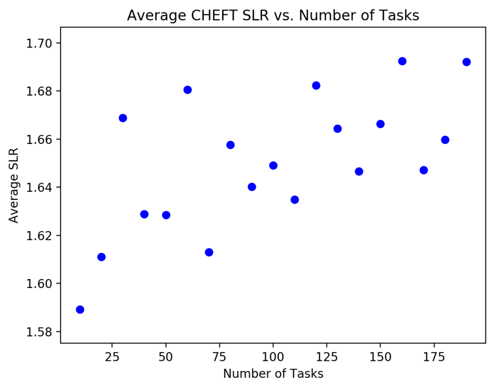
Naturally, because CHEFT is an approximation algorithm, we naturally wish to evaluate how well it does. We write a random graph generator to create lots of test cases. We define the function `binomial_init_graph(n, p)` that generates a directed acyclic graph with n vertices, labelled $1 \ldots n$. For all $i < j$, the edge (i, j) is included in the graph with probability p. We also define the function `unif_init_weights(a, b)` that initializes all task processing weights uniformly at random in the interval $[a, b)$.

4.1 Comparing Directly to IPEFT

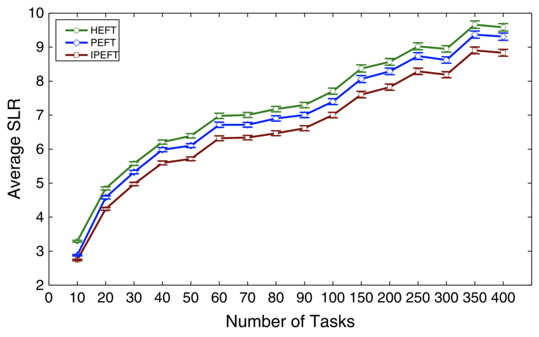
The intuitive first step to evaluating CHEFT was to run it and the IPEFT algorithm side by side on the same test cases. This turned out to be pretty meaningless, because when we restrict the inputs to IPEFT to our problem domain, IPEFT just becomes a clunkier, slower, but mathematically identical version of CHEFT. The two performed identically on all inputs.

4.2 Scheduling Length Ratio

A common metric used to evaluate the performance of this type of scheduling algorithm is the Scheduling Length Ratio, define as the makespan of the generated solution divided by the minimum possible makespan of the longest path alone. In Figure 1, we included a plot of the SLR of CHEFT as well as the SLR of other scheduling algorithms [1].



(a) CHEFT



(b) HEFT, PEFT, and IPEFT [1]

Figure 1: Results collected from graphs generated by `binomial_init_graph(i, 0.2)` for $i = 10, 20, 30, 40, \ldots 190$, `unif_init_weights(20, 150)` with 50 samples each, and processor speeds `[10, 15, 20, 40]`

We can see from the plots that CHEFT seems to perform much better on the SLR benchmark than the existing algorithms. This is probably because the plot on the right hand side was taken from Zhou, et. al. [1], and they ran their algorithms in a heterogeneous computing environment, which allows for more "strange" inputs, so it's not surprising that in our setting our algorithm performs better (again, the IPEFT algorithm was identical to CHEFT on the same input, so the SLR was always the same as well).

4.3 Max Parallelism

To get an idea of how CHEFT performs compared to a theoretical lower bound, we computed the max parallelism of the dataflow, which we define to be the largest number of tasks that can be processed simultaneously while obeying precedence. We first discuss how we were able to compute max parallelism by introducing some theoretical machinery, then discuss how we applied the max parallelism to get a baseline theoretical optimal makespan.

4.3.1 Computing Max Parallelism

In combinatorics, a partially ordered set is a structure that formalizes order. A partially ordered set is defined by a set of elements S and a partial ordering $\leq$ on the elements of S which is a binary relation for which $a \leq a$, $a \leq b \Rightarrow b \nleq a$, and $a \leq b \cap b \leq c \Rightarrow a \leq c$. We say elements $a, b \in S$ are *related* if $a \leq b$ or $b \leq a$ and unrelated otherwise. A *chain* in a poset is a subset of S of elements that are all related to each other, and an *antichain* is a subset of S for which no two elements are related to each other.

The elements and their relationship in a poset can be represented as nodes in a directed acyclic graph. In this representation, there exists an edge (a, b) if $b \leq a$ and there does not exist any c in S such that $c \neq a, c \neq b, b \leq c \leq a$ (that is, a and b are directly related). A chain in the poset is represented as a path in our graph, and an antichain is represented as a set of nodes that pairwise do not share any paths in common.

Lemma 1. *A set of nodes is parallelizable if and only if no two of them lie on the same path*

Proof. We give brief, informal arguments for both directions that we think are convincing enough for most readers. The formal proof of this lemma is left as an exercise.

Suppose we have a nontrivial set of parallelizable nodes. Pick any two of them, s_1 and s_2, and assume for the sake of contradiction that they lie on the same path in our dataflow graph. Without loss of generality, suppose s_1 is an ancestor of s_2. This means s_2 depends (perhaps indirectly) on the output of s_1 to run, which contradicts the assumption that we have a nontrivial set of paralleizable nodes. Thus, for any two nodes s_1, s_2 in our set, they cannot lie on the same path.

Now suppose we have a set of nodes that have the property that no two of them lie on the same path in our dataflow. Since the only condition we impose on our scheduling is that if a depends on b we can only process a after processing b has been completed, no two nodes are restricted from being processed at the same time. Thus, we have a set of parallelizable nodes. $\square$

By the lemma above, and with our graph representation of a poset, we can see that the problem of computing the max parallelism of our graph is precisely the problem of computing the size of the largest antichain in the poset!

Theorem 1 (Dilworth's Theorem)**.** *The size of the maximal antichain in a poset is precisely the minimum chain cover of the poset*

The proof of this theorem is outside the scope of this paper, but it tells us that to compute the max parallelism of the graph, we can instead compute the minimum path cover.

To do this, we introduce some notation that will help us along. Suppose we have a directed acyclic graph $G = (V, E)$. We can construct a bipartite graph $G' = (V_{in} \cup V_{out}, E')$ as follows:

$$V_{out} = \{v \in V | v \text{ has positive out degree}\}$$
$$V_{in} = \{v \in V | v \text{ has positive in degree}\}$$
$$E' = \{(u, v) \in V_{out} \times V_{in} | (u, v) \in E\}$$

Theorem 2 (Kőnig's Theorem)**.** *G' has a matching of size m if and only if there exists $n - m$ vertex-disjoint paths that cover each vertex in G, where n is the number of vertices in G*

Again, the proof of this theorem is outside the scope of this paper, but it further reduces the problem of computing the max parallelism to maximum bipartite matching. Maximum cardinality bipartite matching is well known to be solvable with max flow min cut, which is the method we use.

4.3.2 Theoretical Optimal Makespan

After computing the max parallelism n of our graph, we get a theoretical optimal makespan by summing the weights of all of our tasks, summing the processing speeds of the fastest n processors available, and simply dividing to find out how long it would take the theoretical maximum number of useful processors to process every task if we completely disregard precedence. Since this baseline effectively treats the entire dataflow as one homogeneous task and the n fastest processors as one super fast processor, it's definitely an exaggerated underestimate of the makespan of the true optimal schedule. Even so, our algorithm performed surprisingly well by comparison. Given a dataflow and a library of cores of varying speeds, we ran CHEFT on all subsets of our library and plotted the makespan against the cost of the subset. We also computed the cost and theoretical optimum for all subsets and plotted them on the same graphs.

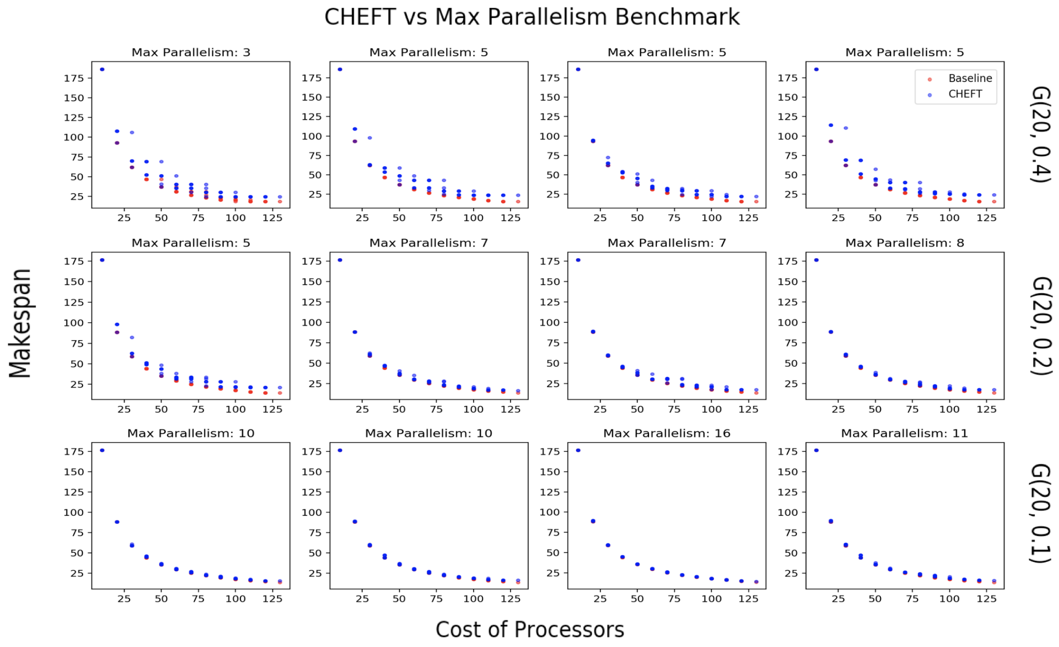


Figure 2: CHEFT compared to the theoretical optimal makespan, run on random graphs of varying density

We use the sum of the processor speeds as a proxy for some model of cost, which depending on what cost is (dollars, power, area) has different models in different settings. Notably, CHEFT performed much better compared on graphs with higher max parallelism, likely because if you have 20 nodes and a max parallelism of 16, there aren't many precedence relationships to violate when computing the theoretical optimal makespan. It's also probably the case that dense graphs tend to not be at their widest for very long (this seems convincing intuitively but we were unable to prove it formally).

5 Additional Features

On top of the theoretical work above, we included a few useful features in our code towards making co-design easier. Given a dataflow and library of cores, our program also allows the user to specify the speed of a hypothetical core and visualize the impact that including it would have on the makespan of running on different subsets in our library. We also allow the user to specify a target latency to understand what kinds of cost they should expect to achieve certain levels of performance. Finally, we make it easy for users to specify a processor vs cost model different from simply the speed of the processor. As an example, we ran CHEFT with visualization on a package delivery dataflow from MAVBench [3] with initial processor speeds `[10, 10, 10, 15, 15, 20, 40]`, target latency 12, hypothetical processor speed 60, and on randomized weights.

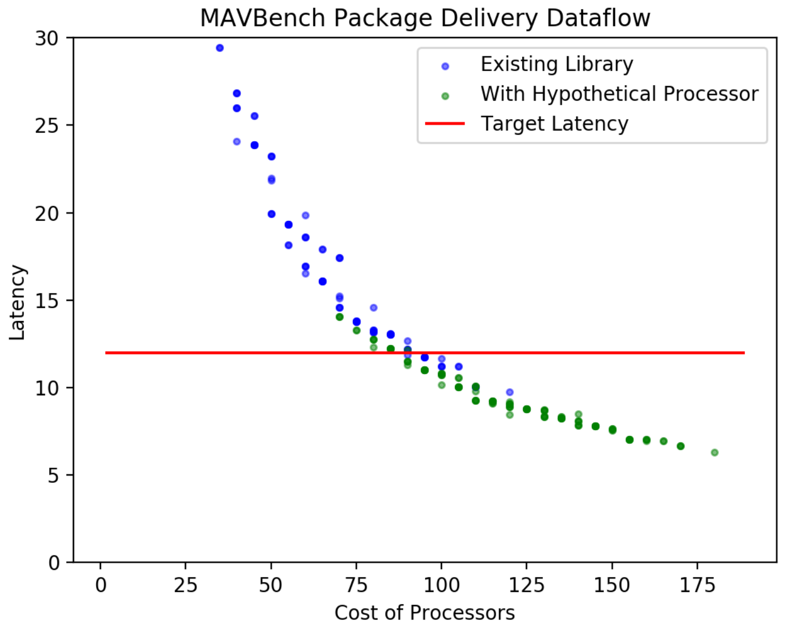


Figure 3: CHEFT on MAVBench Package Delivery

6 Conclusion

In this work we introduced a modified version of existing scheduling algorithms specifically tuned to mapping tasks in an AV dataflow to homogeneous processors. To evaluate its performance, we introduced a novel benchmarking method via max parallelism, and showed that it performs well compared to a theoretical lower bound on the optimal makespan. Finally, we described some additional features of our software included to assist in software hardware co-design.

7 Future Work

There is definitely still work to be done here. In our translation of ideas from IPEFT to CHEFT, there probably are redundant computations that are no longer necessary given a homogeneous processing environment. In particular, it seems like `PCT[v] = CNCT[v]` in every case, and it's even less clear that adding `EFT[v] += CNCT[v]` for non critical predecessors has any effect on the scheduling. In short, CHEFT can probably be optimized further for performance and even packaged into a piece of user friendly software.

We also leave open the problem of modelling the cost of communicating between processors, which would be very useful in many settings. IPEFT already takes this into account, so perhaps some of the same ideas can be translated to CHEFT.

It would also be useful to study how, given a set of processors and a set of tasks, we can reverse engineer adversarial dataflows to investigate co-design in the opposite direction.

References

[1] Zhou, N., Qi, D., Wang, X., Zheng, Z. and Lin, W., 2017. A list scheduling algorithm for heterogeneous systems based on a critical node cost table and pessimistic cost table. *Concurrency and Computation: Practice and Experience, 29*(5), p.e3944.

[2] Topcuoglu, H., Hariri, S. and Wu, M.Y., 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems, 13*(3), pp.260-274.

[3] Boroujerdian, B., Genc, H., Krishnan, S., Cui, W., Faust, A. and Reddi, V., 2018, October. MAVBench: Micro aerial vehicle benchmarking. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (pp. 894-907). IEEE.