FAKULTA
APLIKOVANÝCH VĚD
ZÁPADOČESKÉ
UNIVERZITY
V PLZNI

Semestrální práce - Single-Task výpočet

KIV/OS

Jakub Mladý
scrub@students.zcu.cz

Prosinec 2024

Obsah

1 Zadání **2**

2 Analýza a návrh řešení problémů **3**
2.1 Učení a výpočet 3
2.2 Operační systém 6
2.2.1 Čtení z rozhraní UART 6
2.2.2 Minimalizace spotřeby energie 7
2.2.3 Podpora reálných čísel 7
2.2.4 Dynamická paměť 8

3 Implementace **8**
3.1 Učení a výpočet 8
3.1.1 Optimalizace výpočtu 10
3.2 Čtení z rozhraní UART 11
3.3 Minimalizace spotřeby energie 12
3.4 Konverze řetězce na reálná čísla 12
3.5 Dynamická paměť 14

4 Testování **15**

5 Závěr **15**

1 Zadání

Na platformě Raspberry Pi Zero vytvořte operační systém, resp. doplňte potřebnou funkcionalitu kostry operačního systému (např. KIV-RTOS), a napište pro něj program vykonávající specifickou úlohu.

Úloha bude provádět učení modelu specifikovaného následující hypotézou:

$$y(t + t_{\text{pred}}) = A \cdot b(t) + B \cdot b(t) \cdot (b(t) - y(t)) + C, \tag{1}$$

kde:

$$b(t) = \frac{D}{E} \cdot \frac{d}{dt} y(t) + \frac{y(t)}{E}, \tag{2}$$

$(A, B, C, D, E) = \Pi$ jsou parametry modelu, jež je třeba optimalizovat, a t_{pred} je predikční okénko – určuje tedy, jak hodně do budoucna model predikuje. Takto popsaný model odhaduje budoucí hladinu glukózy v intersticiální tekutině na základě předchozích měření.

Další specifikace:

- Program po spuštění vypíše informace o této práci a následně popíše očekávaný vstup,
- vstupy jsou:
  1. celé číslo `t_delta`, které určuje časový interval mezi měřeními vstupních hodnot v minutách
  2. celé číslo `t_pred` popsané výše
  3. reálné číslo (naměřená hodnota), nebo jeden z příkazů „stop“ nebo „parameters“
- program komunikuje přes rozhraní UART
- po zadání prvních dvou vstupů program čeká na vstupy třetího typu.
- přijde-li na vstup reálné číslo, program začne učení modelu a vypíše predikci $y(t + t_{\text{pred}})$, je-li to možné
  - pokud to možné není, vypíše řetězec „NaN“
  - reálné číslo ze vstupu odpovídá argumentu t – mezi dvěma vstupy implicitně uplyne doba odpovídající `t_delta`
- přijde-li během výpočtu na vstup příkaz „stop“, učení se zastaví a program vypíše predikci podle starých parametrů
  - během výpočtu musí být program responzivní
- přijde-li na vstup příkaz „parameters“, program vypíše hodnoty parametrů Π
- pokud neprobíhá výpočet, systém minimalizuje spotřebu energie (např. instrukcí `wfi`)

- veškerá paměť na výpočet bude alokována dynamicky najednou po startu programu
- učení modelu bude realizováno metodou nejmenších čtverců nebo genetickým nebo evolučním algoritmem

2 Analýza a návrh řešení problémů

2.1 Učení a výpočet

Cílem je učit model daný hypotézou určenou rovnicí 1 s parametry $\Pi = (A, B, C, D, E)$ pomocí buď metody nejmenších čtverců, nebo genetickým/evolučním algoritmem. Metoda nejmenších čtverců optimalizuje cenovou funkci:

$$J(\Theta) = \frac{1}{2n} \sum_{i=0}^{n-1} \Big(h_\Theta(x_i) - y_i \Big)^2, \tag{3}$$

kde n je počet trénovacích dat, h_Θ je hypotéza, která závisí na parametrech Θ, x_i je i-tá vstupní hodnota a y_i její správná hodnota (poskytnutá učitelem). Pro náš konkrétní model bude potřeba korektně doplnit zmíněné komponenty cenové funkce. Prvním poznatkem je, že trénovací data lze odvodit z posloupnosti měření, vzhledem k úloze tedy budou stačit zadávaná reálná čísla. Platí totiž, že v čase t model predikuje hodnotu, kterou bychom měli naměřit v čase $t+t_{\text{pred}}$. V čase $t + t_{\text{pred}}$ naměříme další hodnotu, kterou využijeme jako vstup programu. Tento vstup jednak použijeme pro predikci další hodnoty (v čase $t + 2t_{\text{pred}}$), ale jednak také jako správnou odpověď pro predikci z času t. Odpovědi učitele tedy dostáváme s určitým zpožděním, které odpovídá t_{pred}. Důsledkem tohoto poznatku je mj. to, že s učením musíme počkat až právě do času t_{pred}, předešlé vstupy neslouží jako odpověď žádné predikci. S každým dalším vstupem se pak rozrůstá trénovací množina.

Dalším důsledkem je, že doba t_{pred} musí být násobkem intervalu t_{delta}. Jinak by se totiž časy predikcí nikdy neprotínaly s časy měření a predikce by tak neměly odpovídající odpovědi učitele.

Díky předešlé analýze tedy můžeme upravit cenovou funkci tak, aby odpovídala naší úloze:

$$J(\Pi) = \frac{t_{\text{delta}}}{2(T - t_{\text{pred}})} \sum_{\substack{t:=t_{\text{pred}} \\ t:=t+t_{\text{delta}}}}^{T} \Big(y(t - t_{\text{pred}}) - \tilde{y}(t) \Big)^2, \tag{4}$$

kde $\tilde{y}(t)$ je uživatelův vstup (měření) v čase t. Hodnota $y(t - t_{\text{pred}})$ je predikce do doby t.

Před optimalizací se ještě podívejme na jeden problém strojového učení: přeučení. Tento stav je často chápán jako nastavení parametrů takové, že model velmi dobře predikuje na učících datech, ale není schopen dobře předpovídat pro data mimo učící množinu. Přeučení můžeme předejít zavedením tzv. regularizačního členu v cenové funkci. Ten odpovídá součtu druhých mocnin parametrů

– cenová funkce tedy „trestá“ za vysoké absolutní hodnoty parametrů, což je častá příčina přeučení. Důležité ale je neregulovat absolutní člen modelu, ten v našem případě odpovídá parametru C. Regularizační člen se násobí atributem λ, který tak určuje míru „potrestání“. Nastavením $\lambda = 0$ regularizaci vypneme.

Konečný tvar cenové funkce metody nejmenších čtverců pro naši úlohu tedy vypadá následovně:

$$J(\Pi) = \frac{t_{\text{delta}}}{2(T - t_{\text{pred}})} \left[\sum_{\substack{t:=t_{\text{pred}} \\ t:=t+t_{\text{delta}}}}^{T} \Big(y(t - t_{\text{pred}}) - \tilde{y}(t) \Big)^2 + \lambda \sum_{\substack{i=1 \\ i \neq 3}}^{5} \Pi_i^2 \right]. \quad (5)$$

Kromě hodnot t_{pred} a t_{delta} bude programu potřeba dodat také hodnotu λ. Takto cenová funkce však implementována nebude, více níže v sekci 3.1.1.

K optimalizaci takto zadané cenové funkce nelze použít analytické řešení jako např. normální rovnice pro klasickou lineární regresi, je tedy nutné použít tzv. gradientní sestup. Abstraktně řečeno tento algoritmus posouvá vektor parametrů ve směru největšího poklesu po cenové funkci. Směr největšího stoupání je dán operátorem gradient, opačný směr pak odpovídá největšímu poklesu. Gradient vrací vektor parciálních derivací funkce a směr tohoto vektoru, jak bylo zmíněno, určuje maximální růst. Odečtením gradientu od nynějších parametrů tak získáváme nový vektor parametrů, jehož cena je nižší než předchozího. Tento proces se opakuje, dokud cena není dostatečně nízká. Pro tento algoritmus je tedy potřeba vyjádřit parciální derivace cenové funkce podle jednotlivých parametrů.

Dalším způsobem optimalizace je genetický algoritmus. Tyto algoritmy vytvoří populaci „chromozomů“ a zkoumají jejich „životaschopnost“. Ty nejvíce odolné vytvoří „potomky“ do další generace a proces se opakuje, dokud nejlepší z chromozomů nepodává uspokojivé výsledky. Tyto algoritmy bývají intuitivní a snadno implementovatelné, avšak poměrně silně závisí na jejich konkrétní podobně. U genetických algoritmů je potřeba stanovit následující:

- podoba chromozomu
- fitness funkce – obdoba cenové funkce, určuje chybu nebo „životaschopnost“ chromozomu
- selekce – jakým způsobem jsou chromozomy vybírány do další generace nebo ke křížení
- křížení – jak se vytváří populace nadcházející generace z aktuální
- mutace – změna chromozomu po křížení

Snadná implementovatelnost a absence potřeby derivovat jsou důvodem, proč k optimalizaci zvolíme genetický algoritmus.

Chromozom v našem případě může odpovídat vektoru parametrů a fitness funkce může zůstat cenovou funkcí z metody nejmenších čtverců – čím nižší

hodnota, tím lepší chromozom. Cílem je „vyšlechtit“ chromozom s co nejnižší hodnotou této funkce. Prvotní generace je populace rovnoměrně náhodně vygenerovaných vektorů parametrů v rozsahu $-R$ až R, tedy $\Pi_i \sim \mathcal{U}\left[-R;\ R\right]$.

Potomek dvou chromozomů zdědí geny po obou svých rodičích, ale geny silnějšího by měly převládat. Velikost parametrů potomka bude tedy odpovídat váženému průměru jeho rodičů – a váhy odpovídají nadřazenosti chromozomu, tedy cenové funkci. Lepší chromozom však má nižší cenovou funkci a proto jeho váha nemůže být přímo jeho fitness. Všimněme si, že z páru má mít větší váhu ten, jehož cena je nižší. Tím, že za váhu zvolíme fitness druhého chromozomu, posuneme potomka blíže k lepšímu rodiči. A to ještě tím více, čím větší je rozdíl v kvalitě genů rodičů. Potomek bude tedy dán předpisem

$$P = \frac{c_M O + c_O M}{c_M + c_O} \tag{6}$$

kde P, M, O jsou potomek, matka a otec a c_i je cena chromozomu. Tímto se následující generace budou více schylovat k lepším rodičům a algoritmus bude konvergovat[1].

Divergenci se pokusíme zamezit v kroku selekce. Výběr chromozomů budeme provádět značně elitářsky – pouze několika z nich dovolíme přežít a rozmnožovat se. Tento přístup je diskutabilně nemorální, avšak povstání reálných čísel se nemusíme obávat[2], k vyšlechtění ideálního nad-chromozomu tedy využijme této metody[3]. Mutace nám pak pomohou v případech, kdy se chromozomy začnou ocitat v lokálním minimu. S určitou pravděpodobností změníme každý z parametrů potomka až o jeho poměrnou část: $\Pi_i := \Pi_i + (\Pi_i + c) \cdot \mathcal{A}(p) \cdot \mathcal{U}\left[-q;\ q\right]$ (s vhodným „rozsahem mutace“ q a posunem c), díky čemuž se vektor může posunout o značný kus a překlenout hranici konvergence do lokálního minima. $\mathcal{A}(p)$ značí alternativní rozdělení, které nabývá hodnoty 1 s pravděpodobností p (ppst. mutace). Pravděpodobnost i rozsah mutace necháme opět vybrat uživatele po začátku programu. Velikost posunu nechť je $c \sim \mathcal{U}\left[-R;\ R\right]$. To je vše potřebné pro implementaci algoritmu.

Před dokončením analýzy výpočtu si ještě povšimněme derivace v předpisu funkce modelu, konkr. ve vzorci 2. Analytické řešení derivace by bylo poměrně komplikované, jelikož se jedná o funkci danou rekurentně, sáhneme tedy po aparátu numerické matematiky a derivaci odhadneme zpětnou diferencí: $\frac{dy}{dt} \sim \frac{y(t)-y(t-h)}{h}$. Musíme akorát vyřešit hodnotu kroku h. Z definice derivace víme, že h by se mělo limitně blížit nule. Hodnoty $y(t)$ přicházejí s rozestupy t_{delta}, což je zároveň nejmenší granulární jednotka. Logicky tedy h nejbližší nule odpovídá právě hodnotě t_{delta}. Model je však stavěn tím způsobem, že očekává čas ve frakcích dnů [4]; jelikož t_{delta} je měřena v minutách, vydělíme ji číslem $24 \cdot 60 = 1440$, které odpovídá počtu minut v jednom dni. Celou derivaci tak odhadneme

[1] snad

[2] alespoň než umělá inteligence nenabyde vědomí

[3] případné debaty směřujme na fakultu filozofickou

[4] Toto mi bylo odhaleno v jedné konverzaci se zadavatelem.

takto:

$$\frac{dy}{dt} \sim \frac{y(t) - y(t - t_{\text{delta}})}{\frac{t_{\text{delta}}}{1440}} = 1440 \frac{y(t) - y(t - t_{\text{delta}})}{t_{\text{delta}}}. \tag{7}$$

Nyní je problém výpočtu a učení vyřešen.

2.2 Operační systém

Zadání dovoluje použití kostry operačního systému, která bude pouze doplněna o potřebné služby. Pro potřeby práce budeme volit kostru KIV-RTOS, vyvíjeného na cvičeních předmětu KIV/OS. Pro tuto volbu se rozhodneme z důvodu značného ulehčení práce – jednak není potřeba vyvíjet nový systém a jednak je s kostrou autor obeznámen díky zmíněným cvičením. Tato kostra však postrádá některé služby potřebné ke splnění zadání – jmenovitě čtení z rozhraní UART, minimalizaci spotřeby energie, podpora počítání s reálnými čísly a správa dynamické paměti procesů.

2.2.1 Čtení z rozhraní UART

Systém KIV-RTOS obsahuje ovladač pro tzv. MiniUART přítomný v systému na čipu BCM2835 používaném cílovou platformou Raspberry Pi Zero. Uživatelské procesy k této verzi UARTu mají přístup skrze souborový systém (voláním již implementovaného systémového volání `open`). Ovladač má však naimplementovaný pouze výpis na tuto periferii.

Pro vlastní návrh čtení je potřeba znát některé vlastnosti této periferie. Stejně jako všechny ostatní vstupně-výstupní zařízení v BCM2835 má MiniUART své registry namapované do paměťového prostoru. Jeden z těchto registrů slouží pro přístup do bufferu MiniUARTu pro psaní (při zápisu do registru) a do bufferu pro čtení (při čtení z něj). Oba tyto buffery mají však kapacitu pouze na jeden znak (1 byte). Tato skutečnost při zápisu nevadí, znaky jsou dostatečně rychle vypisovány, pro čtení však vzniká problém – pokud CPU zrovna z UARTu nečte, znaky zprávy jsou přemazávány a v bufferu zůstane pouze poslední znak. Tento problém je možné vyřešit pomocí systémového bufferu a vyvolání přerušení v momentě přijetí znaku, což MiniUART umožňuje. V obsluze přerušení se nově příchozí znak uloží do systémového bufferu uvnitř jádra operačního systému. Čtení v uživatelském procesu pouze překopíruje znaky v systémovém bufferu do bufferu uživatele a posune systémový buffer o počet přečtených znaků.

Tento návrh může být ještě mírně optimalizován – můžeme totiž předpokládat (mj. i z povahy úlohy), že uživatel nebude na UART psát jednotlivé znaky, ale celé řetězce. Pro dosavadní návrh to znamená, že do systému přijde několik přerušení v relativně krátkém intervalu. S každým přerušením přichází také jeho režie – zjištění, odkud přerušení přišlo, a volání funkcí jádra. Tuto přidanou režii je však relativně snadné odstranit. Pokud uvažujeme s oním předpokladem, v obsluze přerušení můžeme nějakou dobu čekat a během ní se snažit z bufferu UARTu číst znaky (jeden z jeho registrů říká, zda je v bufferu nový znak). Místo

mnoha přerušení tak potažmo celý řetězec přečteme pouze v jednom „hromadném přerušení“.

2.2.2 Minimalizace spotřeby energie

K minimalizaci spotřeby slouží v architektuře ARM instrukce `wfi`, popř. `wfe`. Tyto instrukce pozastaví celý systém do příchodu přerušení nebo nějaké definované události a proto jsou privilegované. Kostra KIV-RTOS neumožňuje procesům tuto instrukci volat (např. systémovým voláním). Jednoduchým řešením je takové systémové do jádra volání implementovat. To však z pohledu operačního systému není ideálním řešením – obecně může v OS běžet více procesů zároveň a pozastavení výkonu procesoru omezí ostatní procesy (jimž by teoreticky mohl uplynout deadline). „Správným“ operačně-systémovým přístupem je, aby se proces buď vzdal zbytku svého kvanta, nebo čekal zablokovaný na nějaký zdroj. V našem případě proces nevykonává žádnou užitečnou činnost právě když neprobíhá výpočet a přitom čeká na vstup od uživatele – jinými slovy čeká na naplnění systémové fronty. Touto logikou by se tedy měl zablokovat, pokud nemá vstup, a k CPU připustit další proces. O tom, zda je potřeba něco vykonávat, by měl rozhodovat pouze operační systém – a vykonávat je potřeba právě když na svůj přidělený CPU burst čekají některé připravené procesy. Jestliže jsou v operačním systému spuštěny pouze procesy, kterým nelze přidělit procesor (např. kvůli blokaci), pak je možné systém uspat.

V rámci této práce bude v OS načten pouze jeden uživatelský proces – a když začne čekat na vstup, nemá operační systém jiné povinnosti a může volat např. instrukci `wfe`. Typicky se v praxi do OS zavádí další systémové procesy. V našem případě to může být proces, který systém v cyklu uspává. Tento proces se naplánuje po zablokování uživatelského procesu nad rozhraním UART.

Naším úkolem je tedy umožnit čekání na vstup z UARTu a probuzení procesu po přijetí znaků. Kostra KIV-RTOS již obsahuje podporu pro schéma čekání na zdroje, avšak pro UART není implementována. To bude naším dalším úkolem.

2.2.3 Podpora reálných čísel

Systém na čipu BCM2835 využívá CPU ARM1176JZF-S, jehož VFP koprocesor poskytuje instrukce nad IEEE 754 čísly (podporována je jak jednoduchá, tak dvojitá přesnost). Tento koprocesor je potřeba aktivovat pomocí instrukcí procesoru. Následně jsou všechny instrukce koprocesoru zpřístupněny programátorovi, v důsledku čehož je možné v jazyce C a C++ používat typy `float` a `double` a operace nad těmito typy. Překladači je však potřeba úmysl koprocesor používat sdělit příznaky `-mfloat-abi=hard` (hardwarové instrukce pro čísla) a `-mfpu=vfp` (instrukce provádí koprocesor). Kód pro aktivaci koprocesoru je:

```
1   mrc p15, 0, r0, c1, c0, 2
2   orr r0, r0, #0x300000
3   orr r0, r0, #0xC00000
4   mcr p15, 0, r0, c1, c0, 2
5   mov r0, #0x40000000
6   fmxr fpexc,r0
```

Tento kód vložíme do rutiny pro restart, aby byl koprocesor aktivní po celou dobu běhu zařízení.

2.2.4 Dynamická paměť

Úloha má paměť potřebnou k výpočtu alokovat dynamicky po spuštění, kostra KIV-RTOS však haldu procesů neimplementuje. Vzhledem k úloze, ale také ke vhodné velikosti a rychlosti takového OS lze přistoupit k méně komplikované implementaci – např. pouze přiřazování celých stránek. Vnitřní fragmentace a správa paměti po menších blocích může zůstat na zodpovědnosti uživatele nebo standardní knihovny. Operační systém pouze přidělí požadovaný počet stránek, je-li to možné.

Uživatelský proces si o nové stránky žádá prostřednictvím systémového volání, bude jej tedy potřeba založit. Vstupem volání bude počet stránek a výstupem počáteční adresa první z přidělených stránek, nebo neplatný ukazatel v případě chyby. Systémové volání musí najít požadovaný počet volných rámců v RAM, alokovat je a následně zapsat mapování z virtuální adresy stránky na fyzickou adresu rámce do tabulky stránek procesu. Kostra již obsahuje funkce pro alokaci jedné stránky a namapování do tabulky stránek. Stačí dopsat funkci na alokaci více stránek. Ta buď přidělí požadovaný počet, nebo žádnou v případě, že neexistují volné rámce, nebo proces spotřeboval všechnu jemu dostupnou paměť. Je kritické, aby v metodě nevznikal únik rámců – tedy alokace rámce, který nebude nikdy dealokován. Při zjištění chyby v průběhu této metody musí být vráceny všechny již alokované rámce. K zamezení úniku je pak také potřeba vrátit všechny přiřazené rámce po skončení procesu. Tuto funkcionalitu je také nutné doimplementovat. Strukturu informací o procesu tak bude potřeba rozšířit o pole adres přidělených rámců.

Standardní knihovna může programátorovi procesů pomoct při alokaci arbitrárního množství paměti. Je potřeba naimplementovat funkci `malloc`. Tu je možné uskutečnit pomocí čítače a algoritmu, který jej inkrementuje o velikost požadované paměti. Pokud je potřeba přidělit stránky, algoritmus ji odhalí a požádá o stránky operační systém. Následně algoritmus vrací první adresu bloku nebo neplatnou adresu, pokud není možné nové stránky přidělit.

3 Implementace

3.1 Učení a výpočet

Během analýzy jsme položili teoretické základy učení a navrhli způsob, jímž bude probíhat. V této sekci rozebereme technické možnosti a důsledky. Prvně zkoumejme měření času. Z pohledu systému je pro výpočet čas diskrétní, hodiny tikají vždy při vstupu od uživatele a posouvají se dopředu o `t_delta` minut. Čas tedy můžeme vnímat v násobcích právě `t_delta`, kdy nula odpovídá času prvního vstupu, jedna druhého a tak dále. Podstata tohoto vnímání je, že vstupní hodnoty můžeme ukládat do pole čísel, jež indexujeme touto jednotkou času. V analýze jsme dál zjistili, že doba predikce, `t_pred`, musí být dělitelná krokem

`t_delta` – to znamená, že vydělením získáme zpoždění nebo časový „offset“, který ale také odpovídá offsetu do pole hodnot. Učení může započít až v momentě, kdy aktuální čas odpovídá tomuto offsetu. Zároveň hodnota v poli na indexu `t` slouží jako odpověď učitele pro predikci pomocí hodnoty na indexu `t - offset`.

I přes tento rozbor ale nemůžeme začít trénink v momentě, kdy aktuální čas odpovídá offsetu, což bylo v analýze opomenuto. V čase rovném offsetu sice máme první dvojici dat pro učení, avšak hypotéza modelu (kterou v cenové funkci potřebujeme znát) nelze vyčíslit kvůli zpětné diferenci – nemáme hodnotu před začátkem času, nebo pragmaticky řečeno, přistupovali bychom na index $t = -1$. Algoritmus učení tedy může začít až v momentě, kdy platí podmínka `t - offset >= 1`. Výpočet hypotézy a cenové funkce implementujeme s ohledem na indexy.

Dále se věnujme genetickému algoritmu. Genetické algoritmy vyžadují nějaký generátor náhodných čísel už jen pro krok mutace, vhodný může však být také při inicializaci. Naštěstí systém na čipu BCM2835, který je naší cílovou platformou, obsahuje hardwarový generátor náh. čísel a kostra OS KIV-RTOS k němu má již implementovaný driver, o většinu práce je tedy postaráno. Pro naše účely však výstup generátoru upravíme tak, abychom dostávali reálná čísla z intervalu $[0; 1]$ – generátor vygeneruje celé kladné číslo v rozsahu 4 bytů, to převedeme na reálné číslo (pomocí instrukce koprocesoru VFP) a vydělíme maximálním číslem. K této funkci přidejme ještě jednu, která pak rozsah $[0; 1]$ přemapuje na jiný libovolný interval.

Kromě chromozomů genetického algoritmu bude výhodné uchovávat také hodnoty jejich fitness funkce, abychom je nepočítali stále dokola. Zaveďme tedy dvě spojité oblasti paměti: jednu pro chromozomy a druhou pro jejich fitness. Obě oblasti budou mít stejný počet prvků. Cenu chromozomu určuje stejnolehlý prvek v paměti cen.

V kroku výběru a křížení je potřeba najít určitý počet nejlepších chromozomů, které mají právo se křížit a některé z nichž v populaci zůstanou. Tohoto dosáhneme tzv. algoritmem quickselect, který v průměrném čase $O(n)$[5] dokáže najít k nejmenších prvků tím, že je uvnitř pole přesune na začátek. Takto uspořádané chromozomy můžeme začít křížit – posloupnost začneme procházet odzadu (od nejhorších) a každý chromozom nahradíme jeho průměrem s náhodným z chromozomů, kteří mají dovoleno se křížit. Procházení skončíme před k-tým chromozomem, jelikož lepší chromozomy zároveň přežijí.

Zbývá jen genetický algoritmus poskládat:

```
1  typedef struct {float A,B,C,D,E;} chromozom;
2  chromozom chromozomy[N_CHROMOZOMU];
3  float     ceny[N_CHROMOZOMU];
4
5  chromozom trenuj(){
6      inicializuj_populaci(chromozomy, N_CHROMOZOMU);
7      vyhodnot_populaci(chromozomy, ceny, N_CHROMOZOMU);
8      cyklus:
9          // heapsort + prumerovani
```

[5] a nejhorším čase $O(n^2)$

```
            vyber_a_kriz(chromozomy, ceny, N_CHROMOZOMU);
            // p=ppst mutace, q=rozsah mutace
            zmutuj_populaci(chromozomy, N_CHROMOZOMU, p, q);
            vyhodnot_populaci(chromozomy, ceny, N_CHROMOZOMU);
            if( !konec(chromozomy, ceny, N_CHROMOZOMU))
                goto cyklus;

        return vyber_nejlepsi(chromozomy, ceny, N_CHROMOZOMU);
    }
```

Ukončovací podmínka nechť platí v případě, že nejlepší z chromozomů má ceny pod danou hranicí (např. 10^{-4}), nebo pokud jsme překonali určitý počet generací (1000).

3.1.1 Optimalizace výpočtu

Takto navržený výpočet je korektní, avšak má pro naše potřeby stále některé nedostatky. Hlavním je, že platforma RaspberryPi Zero není příliš výkonným počítačem a tudíž by výpočet s naivní implementací trval příliš dlouhou dobu. Rozeberme tedy, kde můžeme optimalizovat a kde je to vhodné.

Ústím hrdla je výpočet cenové funkce – zahrnuje poměrně náročné operace (násobení a dělení IEEE 754 čísel), s každým vstupem se zvyšuje počet iterací výpočtu a počítá se pro každý chromozom v populaci. Cenovou funkci tedy bude potřeba zjednodušit. Výpočetně snadnější cenová funkce je tzv. střední absolutních chyba (MAE). Tvar cenové funkce 5 je střední kvadratická chyba. Úprava je jednoduchá: druhou mocninu odchylky nahradíme její absolutní hodnotou. Porovnání čísel je rychlejší operací než násobení. Totéž provedeme i v regularizaci, která nám ale také přidává na složitosti. Regularizaci tedy ponechme jako podmíněně překládanou – pokud o ní bude mít uživatel na úkor výkonu, může program sestavit s definicí makra `#define GLPRED_REGULATION 1`. Náročné je taktéž dělení. K určení střední chyby je potřeba sumu přírůstků vydělit jejich počtem. Pokud ji však nevydělíme, získáme potažmo ještě jinou cenovou funkci – celkovou absolutní chybu. Jelikož nás nezajímá konkrétní hodnota střední chyby, ale jen, zda je cena vyšší, nebo nižší, je tato funkce pro nás výhodnější. V nejjednodušší podobě pak cenová funkce fakticky vypadá takto:

$$J(\Pi) = \sum_{\substack{t:=t_{\text{pred}} \\ t:=t+t_{\text{delta}}}}^{T} \Big| y(t - t_{\text{pred}}) - \tilde{y}(t) \Big|, \tag{8}$$

Tuto podobu budeme implementovat. Výpočet odchylky taktéž optimalizujeme na nejnižší počet operací a indirekcí. Možnou optimalizací také je v předpisu 2 parametrem E nedělit, ale násobit. Parametr by se v učení měl této modifikaci přizpůsobit a místo velkých čísel začít nabývat nízkých hodnot (nebo opačně). Uživateli pak ale musíme vypisovat jeho převrácenou hodnotu, aby to odpovídalo původnímu předpisu. Podobné optimalizace můžeme využít také s parametrem D, kterým je násobena diference. Jelikož diference sama o sobě obsahuje dvě konstanty (1440 a t_{delta}), můžeme nechat parametr D se přizpůso-

bit těmto konstantám a ve výpočtu je vynechat. Diferenci tedy budeme počítat pouze jako rozdíl dvou po sobě jdoucích vstupů.

Výpočetně náročné je také námi definované křížení v analýze – obsahuje jak násobení, tak dělení a to pro každý parametr. Tento typ křížení tedy opět nechme jako volitelnou možnost překladu a jako základní algoritmus křížení zvolme aritmetický průměr rodičů. Jedná se o podobný přístup, jen při něm nezohledňujeme fitness. Mutace lze také zjednodušit – jako alternativu při překladu nabídněme posun parametrů o náhodný vektor $\Pi := \Pi + X$, kde každý prvek $X_i = \mathcal{A}(p) \cdot \mathcal{U}\,[-q;\ q]$.

Dalším problémem v rychlosti výpočtu jsou náhodná čísla. Ta generuje hardwarová komponenta, k níž přistupujeme pomocí systémových volání. Pokud bychom chtěli generovat náhodné číslo pokaždé, kdy je v algoritmu potřeba, běh by to značně zpomalilo. Místo toho si po začátku programu necháme vygenerovat mnoho náhodných čísel (např. 1 MB nebo klidně více). Po spotřebě všech můžeme vygenerovat nová čísla anebo je začít používat od začátku – pseudonáhodné generátory beztak také po určité periodě generují stejnou sekvenci čísel, důležitá je velikost periody.

Poslední a nejmenší optimalizací je zbavit se co nejvíce operací dělení a nahradit je násobením. Toto je např. vždy možné u konstant a také u zmiňovaného parametru E.

3.2 Čtení z rozhraní UART

V sekci 2.2.1 bylo nastíněno řešení chybějící služby pro čtení z výpomocného systému MiniUART, avšak byly opomenuty některé problémy, které vznikají při implementaci řešení. Tyto problémy odhalme a vyřešme v této sekci.

Jedním problémem je implementace systémového bufferu. Zjevným řešením je staticky alokovaná cyklická fronta: pole s ukazatelem na začátek fronty a počtem znaků v ní. Pro odhalení dalšího problému se na situaci podívejme jako na známý problém producent – konzument. Fronta znaků je omezený zdroj a přístup k ní je kritickou sekcí. Konzumentem je uživatelský proces, který čte (konzumuje) znaky, a producentem jádro v obsluze přerušení. Tento problém lze v běžné situaci řešit synchronizačními prostředky – např. semafory. V našem případě se však producent nachází uvnitř přerušení, které nelze (jako v běžné situaci) přeplánovat při nezdařeném pokusu o získání přístupu k bufferu. Pokusem o produkci znaků v době jejich konzumace by zamrzl celý systém a nastala by situace značně připomínající uváznutí (deadlock). Toto lze řešit několika způsoby:

- Speciální druh zámku umožňující funkci `trylock()` – při nezdaru se nestane nic, přijaté znaky jsou zahozeny
- zákaz přerušení v době konzumace – synchronizačním prostředkem je v podstatě příznak CPU povolující přerušení, znaky jsou zahozeny
- negace Coffmanových podmínek – např. neodjímatelnosti, následek záleží na implementaci

- problém neřešit – nezavádět žádné synchronizační prostředky, výsledek opět záleží na implementaci

Při vhodné implementaci lze zajistit, že poslední dvou body budou mít při souběhu producenta i konzumenta za následek získání celého řetězce ve dvou čteních. Avšak pravděpodobnost souběhu je poměrně nízká, tudíž nejjednodušším řešením je buď poslední, nebo druhý bod. V implementaci zvolme bod poslední.

Posledním problémem může být doba čekání na uživatelův vstup – pro někoho, kdo píše pomaleji, se by se mohla přednastavená doba zdát krátká, při delší době musí uživatel po posledním znaku čekat. Proto do rutiny driveru `CUART_File::IOCtl` přidáme možnost nastavení této čekací doby s možností vypnutí (při nastavení čekání na nulu).

3.3 Minimalizace spotřeby energie

Jak bylo zmíněno v analýze, minimalizaci spotřeby bude zajišťovat operační systém v momentě, kdy nemůže naplánovat žádný uživatelský proces. Ze zadání plyne, že tato situace nastává, když naše úloha čeká na vstup od uživatele (mimo dobu tréninku). Operační systém pak naplánuje svůj proces, který systém uspí do přerušení. Pro čekání na zdroj má KIV-RTOS již implementované systémové volání, jen ne pro UART. Virtuální soubor, který UART zastupuje, implementuje abstraktní třídu `IFile`, jež poskytuje už implementované metody pro zařazení procesu do fronty čekajících a notifikaci (probuzení) několika z čekajících procesů. To znamená, že jediné, co je třeba doplnit, je konkrétní implementace metody čekání pro třídu virtuálního souboru UARTu. Ta však bude snadná: stačí zjistit, zda systémový buffer obsahuje požadovaný počet znaků, a pokud nikoliv, zavolat zařazení do fronty čekajících a proces zablokovat. V obsluze přerušení UARTu pak pouze doplníme notifikaci procesu, který má k UARTu exkluzivní přístup. Zablokováním procesu operační systém naplánuje svůj systémový proces, který vykoná instrukci `wfe`.

3.4 Konverze řetězce na reálná čísla

Koprocesor reálných čísel jsme aktivovali v analýze a návrhu řešení, vstupy úlohy jsou však textové řetězce obsahující tato čísla. Bude tedy třeba převést textové řetězce na čísla a zpět, pojmenujme tyto funkce `atof`, resp. `ftoa`. K implementaci funkce `ftoa` se inspirujeme implementací ze stránek GeeksForGeeks.org [6]. Algoritmus z reálného čísla extrahuje celou část a část za desetinnou (nebo spíše binární) čárkou. Desetinnou část vynásobí číslem 10^s, kde s odpovídá specifikovanému počtu desetinných míst po převodu. Obě části pak převede na text, jako by se jednalo o čísla celá, a oddělí je tečkou. Standardní knihovna kostry KIV-RTOS již poskytuje funkce na převod z textu na celá čísla a zpět, `atoi`, resp. `itoa`. Pro účely implementace `ftoa` je ale potřeba funkci `itoa` pozměnit: za prvé musí vracet délku výsledného textového řetězce a za druhé musí

[6] `https://www.geeksforgeeks.org/convert-floating-point-number-string/`

podporovat doplňování nul zleva do požadované minimální velikosti řetězce. Druhá podmínka je potřeba pro čísla jako např. 2.003 s $s \geq 3$.

Funkci `atof` implementujeme následujícím způsobem. Vstupem je textový řetězec začínající číslem a výstupem toto číslo ve formátu IEEE 754 v jednoduché přesnosti. Pokud číslo začíná znakem `-`, do proměnné určující znaménko si uložíme hodnotu $-1.0f$ a ukazatel řetězce posuneme na další znak. Kromě této máme proměnné pro výstup a dělitele nastaveného na hodnotu 0. Následně konzumujeme řetězec, dokud nenarazíme na jiný znak než číslici nebo tečku. Při každé iteraci výstup vynásobíme deseti a přičteme k němu další číslici. Jestliže je znakem tečka, nastavíme dělitele na 1 a v každé další iteraci jej násobíme deseti. Pokud na tečku narazíme podruhé, iterování taktéž ukončíme. Po této smyčce vracíme hodnotu výstupu vynásobenou znaménkem a dělenou dělitelem. Kód vypadá následovně:

```
float atof(const char* input)
{
        float output = 0.0f;
        int divider = 0;
        float sign = 1.0f;

        if (*input == '-')
        {
                sign = -1.0f;
                input++;
        }

        while (*input != '\0')
        {
                if (*input == '.')
                {
                        if(divider) break;
                        divider = 1;
                        input++;
                        continue;
                }

                if (*input > '9' || *input < '0')
                        break;

                output *= 10.0f;
                output += *input - '0';
                divider *= 10;

                input++;
        }
        return sign * output / (divider? (float)divider : 1.0f);
}
```

Takto napsaný algoritmus korektně přijímá i čísla $\in (-1, 1)$ bez nuly před des. tečkou, tedy např. „.03“ nebo „−.15“.

Dalším problémem je, že již implementované funkce `atoi` a `itoa` nepodporují záporná čísla. Tento fakt můžeme vyřešit mírnými modifikacemi: u funkce `atoi` můžeme využít stejný mechanismus s vynásobením znaménkem. U funkcí `itoa` a `ftoa` jednoduše před začátkem algoritmu zkontrolujeme znaménko čísla a

pokud je záporné, na výstup přidáme znak - a číslo vynásobíme -1. Algoritmus následně pokračuje nezměněn.

3.5 Dynamická paměť

Sekce 2.2.4 již dává dobrou představu řešení přidělování paměti, v této části jej doplníme o několik implementačních detailů a důsledků. Analýza se zmiňuje o rozšíření informačního bloku procesu o pole adres rámců. Dává smysl, aby toto pole bylo alokováno v jádře staticky, což ovšem znamená, že procesy mohou vlastnit pouze konstantně omezený počet stránek. To ovšem nebude příliš na závadu, jelikož účelem zařízení není provádět příliš paměťově náročné výpočty nebo vykreslování grafiky. Počet nastavme na 16, v důsledku čehož budou mít procesy k dispozici až 16 MB dynamické paměti.

Funkce jádra pro přidělování více rámců volá přidělení jednoho rámce opakovaně. Pokud však v průběhu nastane problém, všechna úspěšná volání musí být odvolána. Funkce tak potřebuje vlastní pole již alokovaných rámců, které zároveň může vracet pro připojení k poli již alokovaných stránek procesu. Metoda uvolnění celé haldy prochází pole alokovaných stránek a postupně je uvolňuje.

Pole alokovaných rámců samozřejmě musí obsahovat adresu rámce, nikoliv stránky ve virtuálním adresním prostoru procesu. Můžeme tak zvolit buď fyzickou adresu, nebo adresu v prostoru jádra, která je pouze posunutá o konstantu. Volba je zcela arbitrární, vydejme se např. cestou fyzické adresy.

Pro proces se dynamická paměť však musí jevit jednolitá a souvislá, jeho virtuální adresy musejí navazovat, i když rámce se mohou nacházet kdekoliv v RAM. Jelikož stránky neuvolňujeme během běhu procesu, index do pole alokovaných rámců odpovídá také „indexu“ stránky ve virtuálním prostoru procesu. Adresy stránek tedy můžeme odvozovat právě od tohoto indexu – určíme si, kde bude halda začínat, a k této adrese přičteme konstantu odpovídající velikosti stránky tolikrát, kolik je index. Počátek haldy nastavme na adresu `0x00400000`, jelikož je dostatečně daleko od počátku kódu (`0x00008000`) i počátku zásobníku (`0x90100000`, roste směrem dolů) – tyto hodnoty jsou přednastavené v kostře KIV-RTOS. Horní mez haldy tedy odpovídá virtuální adrese `0x01400000` (maximálně 16 stránek, viz výše).

O alokaci libovolně velikého bloku se stará standardní knihovna. Té stačí znát pouze „vrchol“ haldy a při každém volání přírůstek k alokované paměti. Jestliže se přírůstek vejde do poslední stránky, není třeba volat kernel a stačí posunout vrchol a vrátit adresu původního vrcholu. V opačném případě je potřeba určit počet dodatečných stránek. Ten odpovídá rozdílu počtu stránek před alokací a po alokaci, tedy hodnotě $pocet\_stranek(vrchol + pozadavek) - pocet\_stranek(vrchol)$. Funkci $pocet\_stranek$ můžeme vyjádřit např. takto:

```
1 int32_t pocet_stranek(uint32_t velikost)
2 {
3     int dodatek = (velikost % velikost_stranky != 0) ? 1 : 0;
4     return velikost / velikost_stranky + dodatek;
5 }
```

4 Testování

Pro účely testování byly vytvořeny programy pro operační systém, které využívají nových služeb OS. Správnost výsledků byla ověřována pomocí výpisů na UART nebo blikání LED. K testování i jako cílová platforma bylo zvoleno hardwarové zařízení, nikoliv emulátor.

K ověření správnosti čtení z UARTu byl napsán „echo“ program, který čeká (pomocí SWI `wait`) na vstup z UARTu a tento vstup následně vypíše zpět. Program testuje jak čtení z UARTu, tak SWI `wait` – systémový proces byl přeprogramován tak, aby místo volání `wfe` blikal LED žárovkou.

Přidělování dynamické paměti bylo testováno dvěma úlohami. Jedna si postupně žádá o paměť velikosti hraničních hodnot – jedna celá stránka, 1 byte, 0 bytů, jedna stránka bez jednoho bytu, půl stránky, několik stránek a paměť mimo rozsah haldy – a každý blok zaplní daty. Druhá úloha alokuje jednu stránku a zapíše na ni o jeden byte více, než je její velikost. Tato úloha končí CPU výjimkou `data abort`.

Korektnost operací nad IEEE 754 čísly a převodů čísel na řetězce a zpět byla ověřena pomocí úlohy, která nejprve převede různé zápisy čísel v textovém formátu na číslo a zpět do textu. Následně vypisuje některé prvky aritmetické posloupnosti s rozdílem `3.0f`.

Pro finální úlohu byla taktéž napsána testovací verze, která měří trvání jednotlivých funkcí výpočtu v počtu tiků systémového časovače. Ve výsledném programu je navíc možnost nechat předvyplnit řadu vstupů tak, aby odpovídala konstantní, aritmetické, nebo geometrické posloupnosti.

5 Závěr

Práce nebyla tak složitá, jak se zprvu zdálo, a to zejména díky předpřipravené kostře KIV-RTOS a díky cvičením předmětu KIV/OS, na nichž byla kostra vyvíjena a vysvětlována. Nejnáročnější bylo právě pochopení, jak celý operační systém funguje (zejm. přerušení a virtuální adresní prostor), potřebné úpravy a dodatky – tedy čtení skrze rozhraní UART, podpora dynamické paměti a podpora čísel standardu IEEE 754 – byly následně zjevné. K vytvoření uživatelské úlohy bylo potřeba pochopení genetických/evolučních algoritmů, čemuž poskytují základy např. předměty KIV/UIR a KIV/SU, implementace konkrétních kroků algoritmu však byly úplně vymyšleny, nebo značně modifikovány. Výsledná přesnost modelu pak značně závisí na volbě pravděpodobnosti mutace a jejího rozsahu, na počtu chromozomů a počtu elity, rozptylu počáteční populace a na zvolených implementacích mutace a křížení. „Ideální“ parametry nalezeny nebyly; to je složitý problém sám o sobě a závisí také na vstupních hodnotách.

Co se však operačního systému týče, vše potřebné bylo úspěšně implementováno a s výslednou implementací nebyly odhaleny žádné problémy. Autor tak považuje práci za úspěšnou.