جلسه 7

ادامه‌ی packet classification

هدف: تشخیص جریان های ترافیکی به منظور اعمال مکانیزم های امنیتی مثل کنترل دسترسی، اعمال مکانیزم های میرایی براساس سیاست گذاری و اعمال محدودیت های ترافیکی ← به طور کلی مدیریت ترافیک داخل شبکه

در packet classification ما یک classifier داریم که rule ها را داخل اون تعریف می‌کنیم. هر rule یک جریان ترافیکی را توصیف می‌کنه و می‌گه روی اون بسته ها، میرایاب چه اقدام (action) انجام دهد؟ بپذیرد، حذف کند، از پورت خاصی بفرستد و...

پیچیده‌ترین کار، تشخیص جریان های ترافیکی است ← می‌آییم آدرس‌های لایه Net و transport (فیلدهای) را استفاده می‌کنیم. 4 تا 6 فیلد می‌تونه باشه. وقتی جریان را تشخیص دادیم، حالا action را اعمال می‌کنیم.

روش های packet classification : ص 7

روش های مبتنی بر Trie :

- hierarchical Trie

در backtracking IP address، یک فیلد هدر (مقصد) را برمی‌داشتیم براساس اون، گام بعدی را از جدول forwarding به دست می‌آوردیم. الان مجبوریم k فیلد را برداریم. انگار باید چندین درخت را پیمایش کنیم. اول براساس فیلد 1، به برگ که رسیدیم، براساس فیلد 2 و ...

p.9 : اگه فیلد 1، 2 را درنظر بگیریم، ابتدا براساس فیلد 1، یک درخت جستجو درست می‌کنیم. حالا نگاه می‌کنیم هر شاخه‌هایی منشعب می‌شه از فیلد 1. (مثلاً از *، 00* را داریم که می‌شه R_7)

(3 تا 0* داریم در فیلد 1، می‌تونه بعدش 01* باشه R_4 یا 1* باشه R_6 یا 10* باشه R_5)

⇐ یک درخت سلسله مراتبی تولید می‌شه. ابتدا مثلاً فیلد 1 را از بسته برمی‌داریم جستجو می‌کنیم، هرجا رسیدیم به برگ درخت اول می‌ریم با فیلد بعدی پیمایش می‌کنیم تا برسیم به یک برگ. ولی کار تموم نیست، باید اولویت گذاری رعایت بشه. به آخرین برگ که رسیدیم، باید برگردیم شاخه‌های قبل هم پیمایش کنیم. مثلاً به R_1 و R_2 و R_6 رسیدیم، اون که بالاترین اولویت را داره، باید action‌شو اعمال بشه. این کار، با توجه به تعداد فیلدهای packet classification، عمق زیادی دارد درخت، ثانیاً باید تقریباً کل درخت را پیمایش کنیم، هرجا به برگ رسیدیم، باید برگردیم شاخه‌های قبل هم پیمایش کنیم. ⇐ حافظه‌ی زیاد، پیمایش سخت

برای بهبودش :

- set-pruning Trie

P.10 : شبیه hierarchical trie هست با این تفاوت که وقتی به آخر شاخه می‌رسیدیم برمی‌گشتیم بالا، الان برنمی‌گردیم، اون شاخه‌های قبلی که داشتیم (مثلاً 00* داشتیم رسیدیم به D، ممکن بوده 0* یا * بوده) شاخه‌ی *، 0* را عیناً کپی می‌کنیم در شاخه‌ی بعدی. بعد پیمایش را دیگه یکبار انجام می‌دهیم. به این ترتیب، حجم حافظه را زیاد کردیم چون شاخه‌های

maxnote

Scanned with CamScanner

از برگ های متقارن را کپی کردیم، اما یکبار بیمارش می‌کنیم به آخرین نتیجه که برسیم لو نو اعمال می‌کنیم.

برای بهبود این روش :

Grid of Trie

P.11 : به جای اینکه کپی کنیم ، لینک بدهیم به اون گره ها.

مشکل : گره ای داریم که مثلاً 2 تا والد دارد مثل x' . از 2 مسیر می‌شه به اون گره رسید. این دیگه درخت نیست. یک grid است که براساس درخت trie ساخته شده.

مزیت : حافظه ی کمتری مصرف می‌کنه.

ولی به هر حال چون همه ی این درخت ها عمق زیادی دارد، زمان جستجو را زیاد می‌کند.

FLTC : P.12

فیلد به فیلد درخت را ایجاد می‌کنیم. فیلد اول را نگاه می‌کنیم می‌گیم براساس اون، این مثلاً مجموعه 7 تا rule که داریم را به 4 گروه تقسیم کنیم. مثلاً R_4 و R_5 و R_6 . هر 3 تا شون $0*$ هستند و می‌تونه در گروه $0*$ قرار بگیرد. R_7 ، $*$ است یعنی هرچی باشه. پس R_7 در همه ی گروه ها تکرار شده. $00*$ هم R_1 و R_2 هستند $0*$ ها می‌توانستن $00*$ باشند پس R_4 و R_6 هم هست. R_7 هم که توهم بود. $10*$ هم R_3 و R_7.

=> براساس فیلد اول، rule ها را به 4 گروه تقسیم کردیم.

حالا براساس فیلد دوم، یک تقسیم بندی می‌کنیم. مثلاً R_4 ، R_5 ، R_6 را درنظر بگیریم، $01*$ ، $10*$ و $1*$ هستند. پس مثلاً اون $0*$ را که لوده بودیم به $\boxed{R_4-R_7}$ ، حالا اگه فیلد دوم $1*$ بود، می‌تونه بشه R_6 یا R_7 . $10*$ باشه، می‌تونه R_5 یا R_6 باشه، $01*$ باشه، R_4 است.

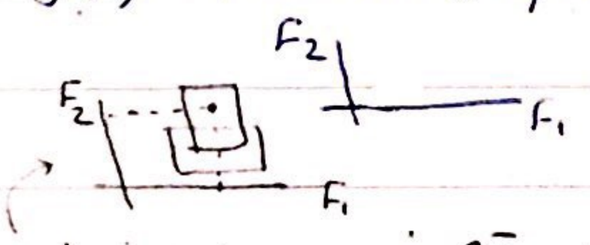
می‌ایم سراغ فیلد سوم، تقسیم را انجام می‌دهیم و فیلد چهارم، نهایتاً rule ها از هم classify می‌شوند ← P.14

حالا یک سبته ی درختی که داریم، اول براساس فیلد اول، می‌آییم در یک نود در level = اول، براساس فیلد دو، می‌اوریم به یک فیلد در level = 2 .... ← براساس فیلد 4، rule را پیدا می‌کنیم.

این روش از نظر تئوری مناسبه ولی در عمل پیاده سازی پیچیدگی‌هایی داره چون با یک فیلد اگر اینا کاملاً مستقل باشند rule ها، یعنی تعداد گره های فیلد اول می‌تونه با تعداد rule ها از یک order باشد. پس خود جستجوی level = اول یه مکانیزم می‌خواد تا level = آخر ← پیچیدگی پیاده سازی بالا.

P.15 روش های هندسی

روش هایی هستند که براساس یک فضای هندسی عمل packet classification را انجام می‌دهند. مثلاً اگر برای packet classification ، 2 تا فیلد را استفاده می‌کنیم (مثلاً فیلد 1 ، فیلد 2) :



هر rule داره یه فضایی را مشخص می‌کنه. مثلاً .

هر سبته ای که وارد می‌شه، یک نقطه از این فضاست. (براساس F_1 و F_2) . باید تشخیص دهیم این نقطه در کدام نواحی (در کدام rule ها) قرار می‌گیرد. اگر در 2 تا rule هست، ببینیم کدوم rule باید روی این اعمال سبته

روش های مختلف هست ← ص 15

یعنی اون درخت جستجو یا استفاده از حافظه‌های TCAM را باید مفهوم هندسی ایجاد می‌کنند. به هر حال وقتی اینا می‌خوان پیاده بشن باید یا درخت بسازن یا از حافظه TCAM استفاده کنند یا روش های DIR باشند. به هر حال از یه مفهوم هندسی برای این کار بهره می‌گیرند.

(جستجو در flow table در SDN عمدتاً مبتنی بر این روش و TCAM و کاهش مصرف انرژی مثل index ها)

p.16 : اگر 2 تا فیلد اول را در نظر بگیریم F_2 F_1 . وقتی 7 تا rule را در فضای هندسی نمایش می‌دهیم، می‌شه شکل اسلاید.

یک سطح که دریافت می‌کنیم می‌شه یک نقطه در این فضا. چون overlap دارن، ممکنه بیش از یک rule بهش بخوره، بین اونا، اونی که بیشترین اولویت را دارد. که معمولاً اونه که از همه بالاتره.

p.17 : روش Cross-producting

می‌گه در مقابل اومدن rule ها را گذاشتن. مثلاً در اون، محور افقی 4 دسته شد (r_0 تا r_3) ، محور عمودی هم 5 ناحیه. میگه بیا یه ماتریس درست کن ← M[4][5] . یک سطح که دریافت می‌شه، در محور افقی و عمودی به طور موازی، index ناحیه‌اش را پیدا کن. توی اون خونه ببین چه rule ی باید اعمال بشه، اونو ذخیره کن در اون خونه‌ی حافظه.

=> از هر فیلد، index یه اون فیلد را برمی‌داریم، بعد مراجعه می‌کنیم به حافظه می‌گیم $M[I_0][I_1]\cdots[I_k]$ می‌شه مثلاً rule یه RP . => یک سطح که میاد می‌شه به طور موازی index شو را پیدا کنیم و با یکبار مراجعه به حافظه، rule یه مورد نظر را پیدا کنیم.

این روش از نظر زمان جستجو خیلی مناسبه چون پیدا کردن index ها در هر فیلد موازی هست و بعدش با یکبار مراجعه به حافظه، rule پیدا می‌شه؛ اما نکته‌ی منفی این روش، حجم حافظه‌ی مصرفی است. چه قدر حافظه مصرف می‌کنه؟ تعداد نواحی در هر فیلد از $O(n)$ است. => حجم حافظه مصرفی : $O(n^d)$ ← تعداد فیلد ها برای classification

با افزایش تعداد فیلد ها، حجم حافظه نمایی افزایش پیدا می‌کنه. مثلاً اگه 10000 rule داشته باشیم، ممکنه هر ناحیه به 1000 تکه تقسیم بشه. اگه 4 فیلد مصرف کنیم، می‌شه $10^{12} = 1000^4$ ← 1TByte .

=> در برابر افزایش تعداد فیلدهای مورد استفاده برای pkt classification ، مقیاس پذیر نیست.

p.18 : روش Bitmap-intersection

سعی کرده scalable نبودن روش قبلی را حل کنه. در شکل ص 18 . مثلاً 4 تا rule بوده که با هم overlap داشتن. محور افقی و عمودی را به نواحی شکسته که محل تقاطع rule ها هستن. مثلاً در x_1 ، یا rule ی 4 می‌تونه اعمال بشه یا 3 . x_2 : یا 1 یا 3 یا 4 x_3 : 1، 4 x_4 : 1، 2، 4 و ...

=> آمده یک bitmap درست کرده که تعداد بیت ها برابر تعداد rule هاست. اون rule ها که در ناحیه بوده 1 گذاشته، بقیه را صفر گذاشته. این جا یک تعداد ناحیه با $O(n)$ داریم و تعداد بیت ها هم $O(n)$ ← $O(n^2)$ بیت

maxnote

Scanned with CamScanner

مصرف می‌شه . در y ها هم $O(n^2)$ بیت مصرف می‌شه. اگه فیلد دیگه‌ای هم بود، باز $O(n^2)$ می‌شه.

=> کل حجم حافظه‌ی مصرفی $O(d\, n^2)$ ← تعداد فیلدها

تعداد rule ها

در مقایسه با روش قبلی، اگه 1000 rule داشته باشیم، نواحی با یه ضریبی مثلاً 1000 بشه ← 4 فیلد 4×10^6

در مقایسه با روش قبلی که 1TByte بود.

چطوری این روش کار می‌کنه؟ هر packet که دریافت می‌کنیم، می‌آییم در محور x ها نگاه می‌کنیم در کدوم ناحیه قرار گرفته؟ مثلاً x_4. در ناحیه y ها، مثلاً y_3 .... حالا می‌آییم bitmap ها شو با هم And می‌کنیم:

یعنی روی این سه تا هم می‌تونه rule 6 اعمال شه هم 4. => $y_3 : 1011 \xrightarrow{and} 1001$ $x_4 : 1101$

بین اینا، اون که اولویت بالاتر داره. اگه اولویت، ترتیب باشه ← R_1.

P.19 : P^2C

این روش می‌خواد از حافظه های TCAM استفاده کنه ولی چون تعداد فیلدها زیاده (32 بیت مبدأ، 32 بیت مقصد، 8 بیت پروتکل، 16 بیت پورت مبدأ، 16 بیت پورت مقصد ... ← عدد بزرگی می‌شه. همین 5 تا می‌شه 104 بیت. اگه بخواهیم حافظه TCAM با اندازه طلاست 104 بیت، به تعداد rule ها داشته باشیم، هزینه حافظه ی بزرگی می‌شه)

روش P^2C که با ایده الگوریتم های هندسی شکل می‌گیره. دوباره ناحیه بندی می‌کنیم محورها را. هر محور را کدگذاری می‌کنیم. مثلاً x ها، 9 ناحیه شده با 4 بیت می‌شه نشون داد. هر ناحیه را یه کد می‌گذاریم به جای اینکه 32 بیت را در نظر بگیریم، با یک روش کد می‌کنیم با تعداد بیت های کمتر. (2^{32} نیست. N است.) محور عمودی هم $O(N)$ است. پس با یک روش کد می‌کنیم که تقریبا $n = \log_2 N$ می‌شه و خیلی کوچکتر از مثلاً 32 هست

بعد اینارو در حافظه‌ی TCAM کنار هم می‌چینیم و سرچ را انجام می‌دهیم.

P.20 : مثال ← یک روش coding (روش کدکردن نواحی یک فیلد). مثلاً محور x ها x_0 تا x_8 را می‌تونیم با 4 تا بیت کد کنیم. 4 تا ناحیه داریم. R_1 و R_4 که اصلاً باهم overlap ندارن. اینارو می‌گذاریم 01 و 10، بقیه جاها 00. R_3 را می‌گذاریم 1، اونطرف را 0. R_4 هم 1، بقیه صفر.

الان کد ناحیه x_0 در میاد: 0000 x_3 : 0001 x_5 : 1011

2 تا روش دیگه هم پیشنهاد داده. مثلاً در b، گفته اونا که overlap ندارن 1، بقیه صفر... ← با این روش، x_5 : 111

فرقی نمی‌کنه. با یک روش این نواحی را کد می‌کنیم.

P.21 : در جدول براساس روش 1 و 2 و 3، اینا نوشته شده.

P.22 : حالا براساس rule ها مشخص می‌کنیم مثلاً R_4 در محور x ها، چی همه 5 بیت 1 ـ آخر =>

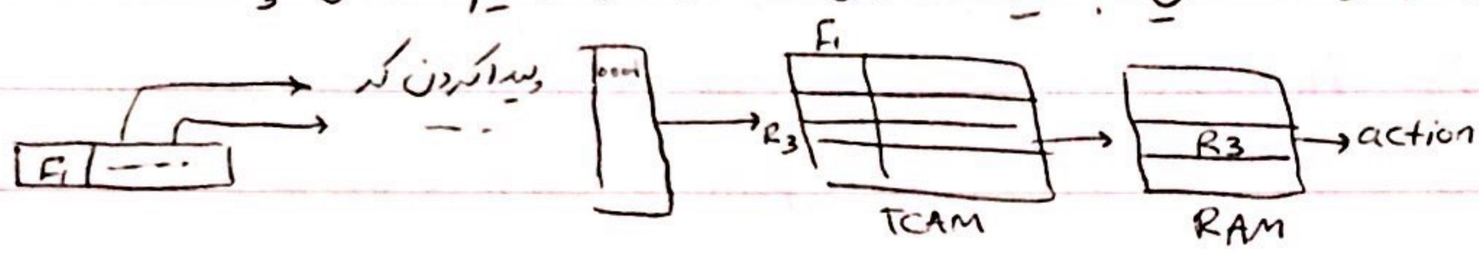
در محور xها، کدها داخلش 0 و 1 و don't care است. اینارو در حافظه‌ی TCAM می‌نویسیم.

مثلاً با روش 1 :

	F_1	F_2
R_1	10xx	
R_2	01xx	
R_3	xx1x	
R_4	xxx1	

فیلد 2 را هم با همین روش و... . اینارو در حافظه‌ی TCAM قرار می‌دهیم.

حالا یک بسته که وارد می‌شه، فیلدهاشو جدا می‌کنیم، F_1 را کدش را با همون روش coding به‌دست می‌آوریم. مثلاً می‌شه 0001. فیلد 2 هم همینطور.... این عدد کدشده را می‌آییم در TCAM می‌گردیم با کدوم match می‌شه؟ مثلاً با R_3. این جا یک حافظه‌ی RAM می‌گذاریم، داخل R_3، rule3 را نوشتیم action ش را اعمال می‌کنیم:



P.23 : روش Area-based Quadtree

می‌دانیم سریع‌ترین جستجو binary search است. زمان جستجو $\log_2^n$ است.

در این روش همین کار را می‌کنیم. می‌گیم محور افقی را 2 قسمت می‌کنیم، محور عمودی هم 2 قسمت. اگه 2 بعدی باشه، می‌شه 4 تا ناحیه. اگه سه بعدی باشه 2^3 ناحیه. 2^d که تعداد فیلدها در classification.

← از ریشه شروع می‌کنیم، 4 تا شاخه پیدا می‌کنیم. اگه اونجا جواب را به‌دست آوردیم (اگه در هر گره، تعداد rule‌ها کمتر از یه عددی بود، search می‌کنیم.) ولی اگه زیاد بود، 2 مرتبه ناحیه را به 4 ناحیه دیگه می‌شکنیم. انقدر تکرار می‌کنیم که تعداد rule های درون ناحیه از یه عددی کمتر باشه، داخل اون یه سرچ خطی می‌کنیم و rule ← مورد نظر را پیدا می‌کنیم.

P.24 : به عنوان مثال، اول به 4 قسمت فیلد 1 و فیلد 2 تقسیم کردیم: (F_1, F_2)

در هر قسمت چه rule‌ای قرار گرفته؟

R_1, R_2 R_5, R_6	R_3
R_4, R_7	R_7

R_7 که شامل همه شده، در ریشه است. R_7، بعد تقسیم‌ها را انجام می‌دهیم. R_1, R_2, R_5, R_6 را باز می‌شکنیم.

درخت را پیمایش می‌کنیم، هر جا متوقف شدیم، داخل همونو می‌گردیم پیدا می‌کنیم، اگه نه می‌ریم پایین‌تر.

(اونایی که شدند که در نود بالایی می‌گذاریم. مثل R_7)

ایراد این روش: اگه جمع rule‌ها در یک ناحیه زیاد باشه، در یک ناحیه خاص، درخت نامتوازن می‌شه.

روش بعدی می‌گه به جای چشم بسته، هوشمندانه تقسیم کنیم براساس جمع rule‌ها. (به جای اینکه هر ناحیه را 4 قسمت کنیم، هر بار به تعدادی قسمت تقسیم کنیم که تعداد rule‌های هر قسمت تقریباً برابر بشه. مرحله‌ی بعد، همون محور یا یک محور دیگه را درنظر بگیریم:

P.26 : Hierarchical intelligent cutting : اگر R_1 تا R_7 را درنظر بگیریم، هرکدوم یک ناحیه در شکل ص 26 هستند

maxnote
Scanned with CamScanner

حالا اگه می‌خواهیم بشکنیم، بهتره محور افقی را به 4 قسمت تقسیم کنیم.

=> در هر مرحله یک محور را در نظر می‌گیریم، تعدادی cut انجام می‌دهیم، rule ها را بر اساس آن تقسیم می‌کنیم. دفعه‌ی بعد، یک محور دیگه با تعداد cut ها. هر دفعه محوری را می‌گیریم که با تعداد cut ها، گروه‌های مساوی بشه => عمق درخت حداقل می‌شه.

این روش، با یک روش هندسی. درخت جستجو را شکسته رسیده به برگ‌هایی که مثلاً k تا rule بیشتر نداره و جستجوی خطی این جا انجام می‌شه.

در حافظه‌های TCAM سراغ indexing نمی‌رفتیم چون اگه کل حافظه را می‌خواستیم فعال کنیم، مصرف انرژی زیاد بود، پس partition بندی می‌کنیم. از همین روش، Hierachical می‌شه index درست کرد و rule هایی که در یک ناحیه قرار می‌گیره، در یک TCAM partition ذخیره کنیم. اول با پیمایش درخت، id ـ partition را پیدا کنیم بعد جستجو را انجام دهیم.

Scanned with CamScanner

Recursive ...

الگوریتم های هیوریستیک

Tuple space search

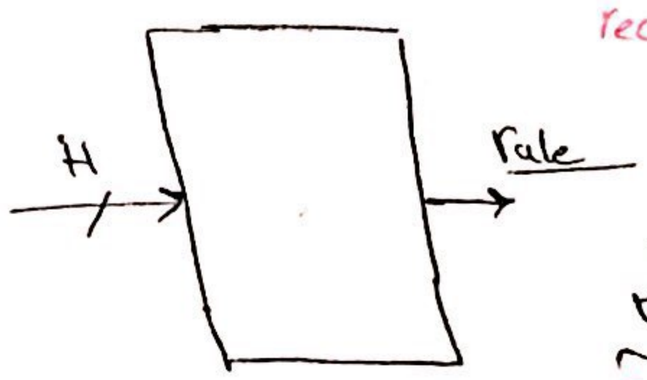
Recursive Flow Classification:

فضای ست وجود بزرگ داریم از تعدادی از فیلدهای هدر استفاده می‌کنیم برای تشخیص جریان ، ما H بیت از هدر داریم استفاده می‌کنیم.

برای یک کلاسیفیکیشن. اگر فرض کنیم مجموع تعداد بیت هایی که از هدر ما برای classification Packet استفاده می‌کنیم در حافظه ای 2^{128} (128 بیت بود) داشتیم

اگر از روش direct استفاده می‌کردیم ← اگر حافظه ای داشتیم که H بیت خط آدرس داشت باید مراجعه به حافظه شید (اگر از روش داشتیم)

Rule ریتیرن کرد



Recursive ایده ی

تعداد rule های درون classifier محدوده. مثلاً 2000 تا.

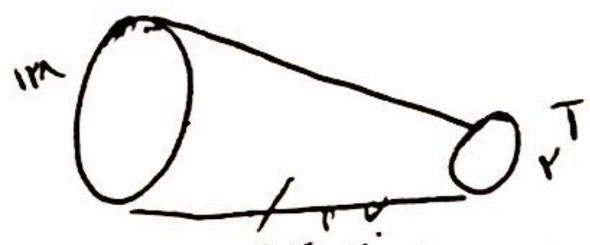
اگر نگاشت داشتیم که حافظه ی 2^H را نگاشت می‌کردیم به 2^T می‌شد

کل حافظه ی پیوسته: خب این حافظه خیلی خیلی بزرگ است از حافظه DRAM استفاده کرد. 2^{128}

2^{10}

$2^{10} = 1000 \times 1000 \times 1000 \times \dots \times 1000$ (2^{128} ، 39 تا صفر) ← 13 بار

پس همچین حافظه ای اصلاً نمی‌توانیم داشته باشیم.



ایده: یک نگاشت ابتکاری پیشنهاد بدهیم که این فضا را رکوییک کنیم

این روش هیچ شما که فیلد تا rule دارید ؟ 1000 تا ؟ 2000 تا ؟ ← 1000 تا ← 2^{14} که بود به سمت Direct منتهی انجام شد

2000 ← 2^{12}

هیچ درسته شما H بیت از هدر رو استفاده کنید و سعی کنید بین این 2000 تا داری سرچ کنی. $\frac{2^T}{4Kbyte}$ if $T=12$

← اگر شما بیای این فضای بزرگ 2^{128} رو نگاشت کنی باید تکلیفم به یک فضای کوچکتر و بعد به روش مستقیم به این فضای کوچک مراجعه کنی. (مستقیم مراجعه کنی به حافظه)

مرحله به مرحله نگاشت انجام بده. ← نگاشت بازگشتی

← این روش شبیه بیاتو چند مرحله انجام بده.

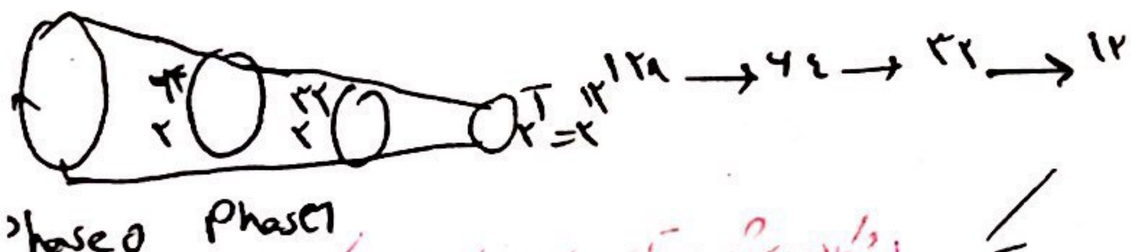
قابل تبدیل باشد.

Chunk: قسمت P.31 قسمت بندی هم اینطوری نیست بهترین تقسیم بندی رو انجام میدم. هدف اینکه به حداقل حافظه ای که در دست داریم

- این Map کردن بسته به محتوای جدول Classifier داره. هر دفعه محتوای جدول تغییر کنه این نگاشت هم تغییر میکنه.

روش Recursive شبیه بیاتو چند مرحله این نگاشت رو انجام بدی

یک نگاشت بازگشتی پیشنهاد کنیم.



فیلد های هدر ← آدرس مبدا، آدرس مقصد، پورت نامبر مبدا، پورت نامبر مقصد، پروتکل

اشکال روش: قاعده ای نداریم برای کاهش بیت ها ممکن هم هیچ کاهشی اتفاق نیفتد

P.28 در این روش Recursive شبیه بیاتو این فیلدها رو بندی کن مثلاً با ارقام بیا نگاشت کن به فضای کوچکتر اتو در این کار را انجام بده تا به نگاشت برسیم

(1) (2)

یعنی 88 بیت را نگاشت کردیم به 8 بیت.

مثال: P.31 : 4 تا فیلد داره. مثلاً آدرس های مبدا را هم که نگاشت کن ... مثلاً شماره پورت. 2^{16} پورت نامبر داریم. اومده تقسیم کرده در 4 گروه مساوی. هر 2 بیت کافیه. مثلاً P.32 بیت همینکار

Scanned with CamScanner

مدیریت ترافیک: مکانیزم‌های کنترل ترافیک ← برای اعمال سیاست‌ها

در روش‌های Throughput را در نظر داشته باشیم و در عین حال QoS رو حفظ کنیم ← تعریف

چالش: Quality of Service رو داریم سیستم، به ما نمی‌ذاره Throughput برسیم چون اجازه QoS رو ازمون میگیره

کیفیت‌های ارسالی (پیامون‌ها) و سرویس‌های (مثل پخش زنده) همه شون برای ما خیلی مهمن. برای این به وجود آمده که ترافیک رو مدیریت کنیم.

این مدیریت شد که دنبال ایجاد مدل ← در این شبکه Throughput بیشتری داشته باشیم و در عین حال QoS رو حفظ کنیم.

مانع و دیر شدن روی ترافیک داشته باشیم ← هدف

برای این کار سه تا مکانیسم داریم. مثلاً اومده صف و بافر درست کرده (مرتب پیامون‌ها رو آماده می‌کنه) اما باید منطقی این کار رو بکنه

فرض کنید هدایت‌کننده‌ها اطلاعاتی ← اتصال‌ها داریم ← ۲ نفر دارن حاضر می‌شن از یه مسیر استفاده کنند

داریم سراغ شبکه کامپیوتری. همین شباهت داریم شبکه‌ها رو بررسی می‌کنیم.

مکانیسم‌هایی که به کار گرفته می‌شن (در 3 سطح هستند)

- چراغ‌های راهنمایی: پیامون‌ها رو پیگیری می‌کنه
- سوئیچ‌ها در دو ترد: multiplex شده ترافیک
- حذف ترافیک: کیفیت سرویس رو پایین میاره
- حذف ترافیک: Throughput رو
- Packet loss: با دیدن Cues
- سرویس‌های با اولویت: Priority services
  یا سرویس‌هایی که کاربران مختلف در نوبت قرار دارند
- حداقل Throughput رو هم کار داریم
  حداقل پهنای باند رو نیاز داریم
- Priority
  Real time ترافیک‌ها
- Enterprise: پهنای باند تخصیص شده مشتریان تضمین قرار دارند، دارند Msec

مکانیسم‌ها

اعمال سیستم

- چه ابزارهایی داریم که بتونیم اهدافمون رو بسیم؟ این مکانیسم‌ها در ۳ سطح قابل اعمال

① buffer mgmt, pkt scheduling

④ کنترل ازدحام

⑤ TE

Packet level: متمرکز بر پیگیری بیشتر مکانیسم

Flow level: سیستم روی جریان ترافیک

Flow Aggregate level: مکانیسم‌ها روی کلاس‌های از جریان‌ها

روی هر لینک خروجی یه بافر ایجاد می‌شود. طرز مدیریت این بافر سعی می‌کنیم QoS رو

در سطح Packet level. در سطح مسیریابی داده‌ها تشخیص بسته‌ها در ورودی صف. ما فرض می‌کنیم بسته‌ها وارد شبکه پر شده یا اومدن به تصمیم‌گیری

مدیریت‌ها. شبکه‌ها داره داره می‌شه، در پردازش که انجام می‌شه مثل lookup کردن آدرس مقصد، پورت خروجی مشخص می‌شه و بسته‌ها در بافر خروجی قرار می‌گیرند. روی هر لینک خروجی، یک بافر ایجاد می‌شه. بسته‌ها توسط جریان‌های مختلف، متوجه می‌شیم با plot distribution که یه بسته مثلاً از نظر delay، اولویت بالاتری داره. با مدیریت صف و با plot scheduling، سعی می‌کنیم QoS رو تأمین کنیم. در سطح بسته، مکانیسم‌های مدیریت صف

Scanned with CamScanner

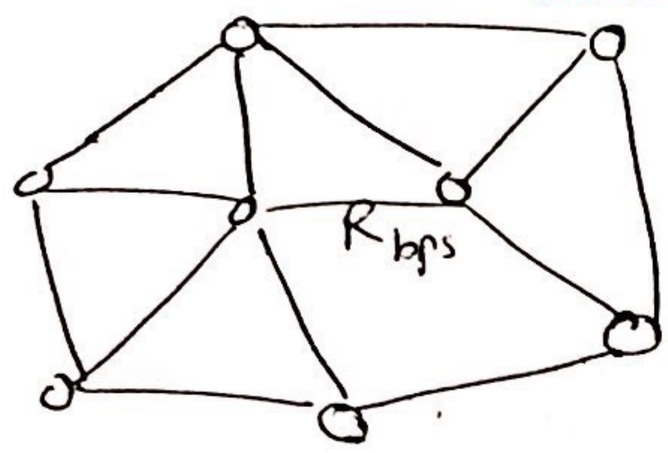
فاصله‌ی زمانی بین ۲ بسته در صف است. پس باید Packet Scheduling, Buffer management در صف باید تصمیم بگیری

۳. این مکانیسم‌ها داخل Packet level سوئیچ‌ها و مسیرها انجام می‌شه.

Flow level : در مقیاس ms ← مکانیزم‌های کنترل از دام ← مدیریت در سطح کلان از دام. با کنترل کردن نرخ ورودی به شبکه

سطح بالاتر: جریان ترافیکی داریم که از مبدا به مقصد هست. این جریان‌های ترافیکی بسته‌ها مشترک در سطح کار داشته باشند

closed loop , open loop مکانیزم‌های

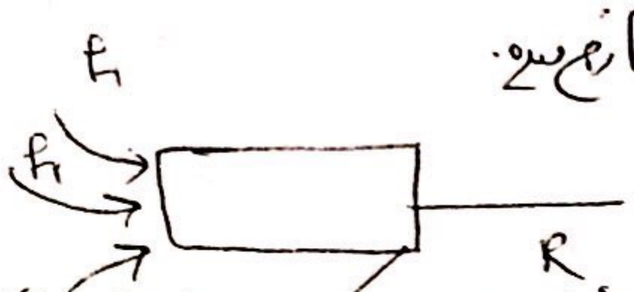


روی هر سیستم تعداد جریان ترافیکی داریم جمع می‌شن.

هر جریانی از مبدا یه نرخ تولید بسته داره.

f_1 bps , f_1 bytes

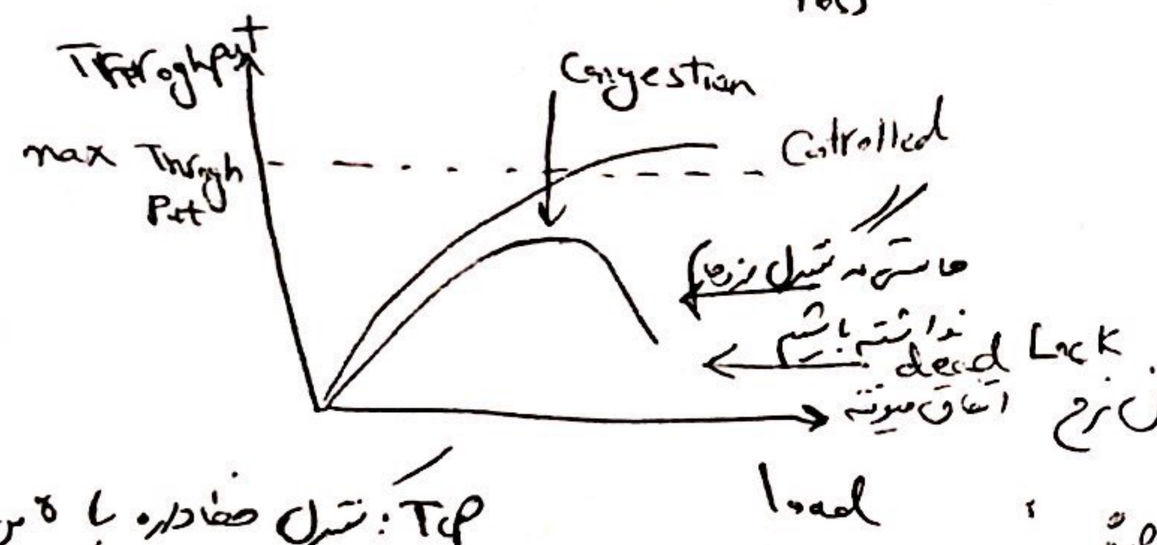
اگر مجموع جریان‌هایی که از یک لینک میخواهد بگذره از ظرفیت لینک بیشتر باشه ← از دام رخ میده.



$\sum f_i < R_{bps}$ تا جریان‌ها بتونن از لینک عبور کنن

$\sum f_i > R_{bps}$ → P-L (از دام) Packet loss رخ میده. هم تأخیر افزایش پیدا می‌کنه چون صف برای پردازش

اثر این Congestion کنترل نشه، از دام بدتر میشه چون Packet میشه، Retransmit میشه با مکانیسم‌های ARQ.



اختلاف load و Throughput ← شبکه اون بسته‌هایی که loss میشن.

جریان‌های ترافیک نمونه‌ای هستند. در نقطه بالا پایین شبکه چون نرخ ترافیک متغیر با زمان هست و ما به دنبال متوسط میگیلن مثلاً f

TCP: کنترل خودکار با ۵ تا تون ← مجدد ارسال بسته ← از دام بیشتر میشه

Congestion : با افزایش load کاهش Throughput پیدا کردیم. که از دام باید کنترل بشه. مکانیسم‌های کنترل از دام وجود دارن

مکانیسم‌های Flow level هستند. چون روش‌های کنترل از دام رو کنترل و تیرو کنترل، همیشه باید در شبکه از دام کنترل شود

Flow Aggregate level : مثلاً پلیس راهنمایی که پنج سه سه جمعه میره چیزی کنترل می‌کنه آمده (در حد دقیقه) روز رفته می‌گه براساس گذشته ترافیک تقسیم می‌کنیم، ترکیب داره. هنوز اتفاق نیوفتاده ولی از روی تجربه گذشته برای آینده استفاده کنیم. order

ما دنبال چی که بیوان برن چیکار کنن؟ میاد الگوریتم مسیریابی از این مسیر خاص‌ترین استفاده کنه.

میاد جریان‌های ترافیکی که قرار بوده بیاد در پیش بینی می‌کنه و مسیرهای از پیش تعیین شده تعیین میکنه.

Extract base routing

مدیریت ترافیک بسته لطفاً نیست ۳۶ سرویس ما با این بیاد ولی دوره مدت هم مدیریت است.

این‌ها TE هم می‌گویند.

پیش بینی می‌کند در آینده چه اتفاقی می‌افتد. اگه فقط کوتاه‌ترین میگیریم ممکنه از دام بگیره ولی اگه از قبل پیش بینی می‌کردیم ما جریان را به شایعه مسیریابی جریان اول را تقسیم با این بهینه‌تر بخوای نمی‌گوییم.

Scanned with CamScanner

ترافیک در شبکه‌های کامپیوتری بیش از حد سنگین شود این مسائل [illegible] است

مهندسی ترافیک: مسائل ترافیک در شبکه‌ها که محول که مورد انتظار است براساس سه سری اصول و قواعد نیازه

باید از قبل مسیر پیش بینی شده ← بهترین مسیر رو براساس محدودیت‌هایی که داریم انتخاب کنه.

پس اولین مکانیسم ما در رسیدن به اهداف شبکه. بیشتر رو مکانیسم اول کار می‌کنیم.

مکانیسم: مهندسی ترافیک

می‌دانیم QOS همواره End to End همه. مثلاً برای صحبت تأخیر Delay میگه نزدیک End to End همه

مثلاً Delay ← تأخیرها جمع می‌شوند

Packet loss ← ضرب می‌شوند (اگر احتمال یکی زیاد باشه، احتمال کلی هم افزایش پیدا می‌کنه)

Throughput ← min گرفته می‌شود.

مدیریت شبکه: مدیریت کنیم اگر قرار است برای لاما شود کدوم بسته از بین بره؟ (P.6)

صف با اولویت ← بسته‌ها سرعت صف نه تا حدودی تبعیض بقیه کلاس رو تحمل کنه

صف بدون اولویت: اگر همه بسته‌ها رو ارزش دیدیم و زیاد فرشی معین به حذف کردن

Packet loss ← انتخاب کدوم بسته رو لاما کنیم.

مثلاً اگر ۲ تا دسته داریم کلاس ۱ و کلاس ۲

تا احتمال بقیه کلاس‌ها کلاس ۱ کمتر بدانند

class 1 — class 2 ← تبعیض کلاس رو تحمل کنه

Threshold ← میزان

Packet Schedeling: اگر ۲ تا بسته داشتیم که یکی تأخیر نیاز دارد و دیگری حساس نیست. مسئله اولویت قرار دادن. بهتر است به بسته با اولویت پایین‌تر سرویس بدهیم که بسته‌ای با اولویت بالا وجود نداشته باشد. ← با نوبت بندی (P.7)

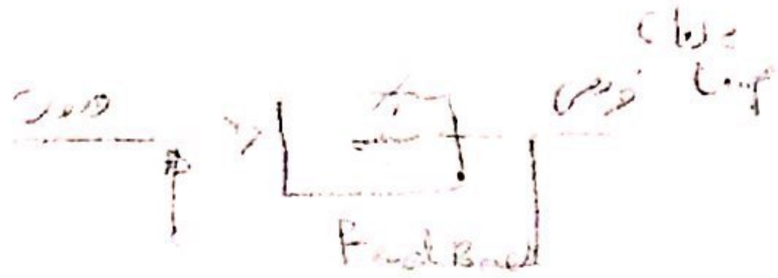
اولویت Delay و P.L. تعریف بزنند

بسته بزنه ابتدای صف یعنی به سرعت انتقال به

هر ۲ تا این مکانیسم‌ها را فوق ۱ باشیم Taggint unit انجام داد.

پس جریان‌ها تشخیص داده شوند و تفکیک شوند و وارد صف شوند

با این کار حجم ترافیک بیشتری را می‌توان پذیرفت.



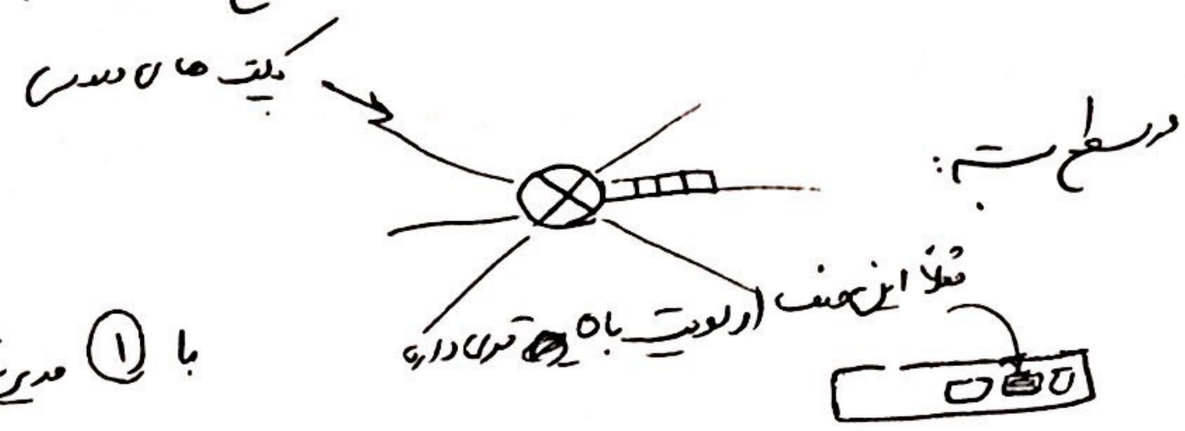
مشخصات ترافیک ← maximum burst size
peak Rate
Averg Rate

Leaky bucket ← الگوریتم کنترل ترافیک که جریان شدید ترافیکی آن که قرار داشتند رو رعایت کنه

Scanned with CamScanner

مکانیزم های کنترل ترافیک:
- در سطح بسته ← در حد Link براساس رسیدن بسته ها. از O(NLS) در روترهای core شبکه
- در سطح جریان ← مکانیزم های کنترل ازدحام
- در سطح تجمیع جریان ← مکانیزم های TE: براساس لایه شبکه گذشته پیش بینی می کنند و ترافیک را در شبکه طوری می سازد که به هدف ماکزیمم کردن روهی برسیم.

در سطح بسته:



با ① مدیریت صف و ② زمانبندی خروج بسته ها سعی می کنند QoS بسته ها را تأمین کنند و به بالاترین Throughput برسیم.

پس مکانیزم هایی در سطح بسته ها:
- Buffer managment: اینکه وقتی بافر پر شود کدام بسته Drop شود.
- Packet scheduling: برای ارسال بسته کدام بسته برای ارسال انتخاب شود.

Flow level. اگر مسیر یابی به درست انجام نرسد بعد از مسیر یابی به ازدحام نرخ بده مکانیزم های سطح بسته کاری از دستشون برنمیاد

مکانیزم های Flow level مکانیزم های کنترل ازدحام هستند. تا از ازدحام جلوگیری کنند. با مسیر یابی و کنترل کردن نرخ ورودی

Flow aggregate: پیش بینی می کنند که در آینده چه اتفاقی می افتد.

این مکانیزم ها Traffic Engineering

فرض جریان n تا روتر پشت سر هم از دهانه بگیریم. اگر پیش بینی کردیم که n تا جریان داریم. ممکن بود مسیر بعضی مسیر یابی با محدودیت انجام میدیم و شکل ترافیکی شبکه را مهندسی می کنیم.

* کیفیت سرویس همواره End to end مدنظر هست. مثلاً مدت Delay: تأخیر کل مسیر end to end

مهم: برخی از پارامترهای کیفیت سرویس Additive هستند. مثل Delay End to end

برخی کیفیت نامکمل: احتمال Packet loss ضرب می شوند. اگر بخواهیم احتمال یک کاش بالا باشد، به شدت اثر می گذارد

اتفاقاتی هستند که به صورت min min تغییر هست مثل Throghput

ما با این سه کار دها مکانیزم های مدیریت بافر، زمانبندی بسته ها را انجام میدیم.

یعنی مدیریت کنیم اگر تراز بسته ها از بین برود دیگه شود کدام بسته از بین برود.

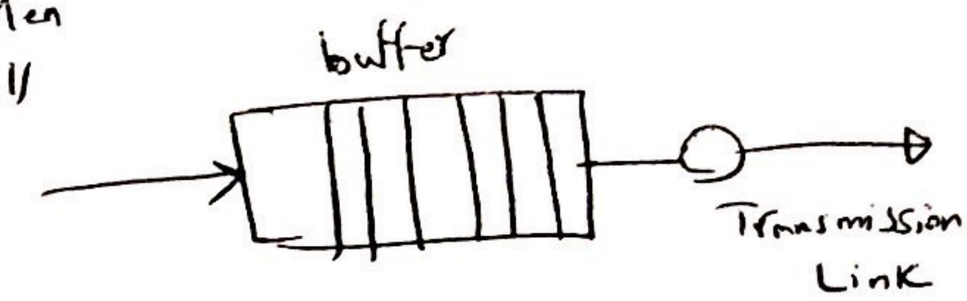
اگر همه ی بسته ها را با یک اولویت ببینیم و تصمیم بافر پر شود شروع می کنیم به کاس شدن بسته هایی که از راه میرسند. بدون توجه به اولویت آنها

اما اگر جریان ها با ترافیک مختلف با اولویت های مختلف داشته باشیم جریان هایی که به دلیل صاحب ترافیک کاس اولویت بالاتری دارند.
صف با اولویت

Scanned with CamScanner

Class 2: discard when Threshold exceeded

class 1: discard when full



کلاس ۲ اولویت PL رو بیشتر تحمل نمی‌کنه اگر بافر تا یه حد آستانه‌ای پر شد از اینجا به بعد سیستم بسته‌های کلاس ۲ رو drop میکنه و از احتمال drop شدن بسته‌های کلاس ۱ رو کاهش میدیم.

Packet sche : اگر ۲ تا جریان داریم که یکی تأخیر حساس و دیگری حساس به پهنای باند است. بین این ۲ جریان باید اولویت قرار داد و فاصله‌ای که جریان با اولویت بالا بسته‌ای تو بافر داره به بقیه بسته‌های جریان با اولویت پایین سرویس بدیم

نوبت‌بندی با اولویت انجام دادیم این‌جوری.

Packet loss & delay

هر ۲ مکانیسم مدیریت بافر و زمانبندی مد نظر هستیم Tagging unit انجام داد. یعنی بسته‌ها رو براساس اولویت Tag بزنیم روی stored Packet (Buffer) داشته باشیم. مثلاً اگر بسته‌ای اولویت و بافر پر بود و این بسته به سلام حساس بود باید یکی بزنیم که یه بسته از صف حذف شود و این بسته وارد شود یا اگر به تأخیر حساس بود Tag ای بزنیم که بره اول صف

کارکرد: جریان‌های وارد شده با Packet Classification شناسایی میشوند و براساس مشخصه جریان‌ها برچسب میخورند و براساس برچسب خورده شده بافر مرتب شده و روند و بالین برچسب‌ها میتوان احتمال PL و تأخیر بسته‌ها رو مدیریت کنیم و با این کار ما هم ترافیک بیشتری پذیریم و QoS را تأمین کنیم.

Generalized Processor Sharing / Fair Queueing

fair Queueing

وقتی روی بافر چند جریان اشتراک میگذاریم هدف ما این است محاسبه‌ای عادلانه داشته باشیم و سهمی که هر کدوم از این جریان‌ها از این منابع برخوردار باشند. مثلاً n تا جریان فعال داشته باشیم و هر جریان $\frac{C}{n}$ بدیم. n بازه‌ی تغییر کنه. بین سهم جریان‌ها هم متوازن تغییر کنه.

ساده‌ترین کار

اگر اندازه بسته‌ها مساوی باشد RR می‌تونه عدالت رو تضمین کنه Fair Queueing باشد و اگر اندازه بسته‌ها مساوی نباشد

اگر اندازه بسته‌ها مساوی نباشد این عدالت حفظ نمی‌شود. در شبکه اینترنت اندازه بسته‌ها مساوی نیست و RR براساس Packet Base کار میکنه اگر بیاییم RR بیت بیت انجام بدیم براساس بیت بود، عدالت داشتیم

Tagging sys مکانیسم استفاده کنیم. یک بیت یک کار میکنیم ولی اندازه‌ی بسته را در نظر میگیریم. اگر یه بسته 1000 بیت است، این انگار 1000 round در حالت بیت به بیت داشته. حالا tag میزنیم سرعت خروج از بافر را 1000، یک بسته اگر 100 بیت باشد round ... میشه.

Scanned with CamScanner

پس در Tag نیز نیم‌ساعت زمان خروج این بسته برابر ۱۰۰۰ دور.

بعد براساس Tag بسته‌ها را انتخاب می‌کنیم.

(P.11)

الان هستیم

Finishing Time = Round numb of + اندازه بسته / نرخ

اگر با نرخ‌های با شکل ← ... آخرین بسته این جریان Finishing Time + ...

$F(i,k,t)$: Finishing Time بسته kام از جریان iام که در زمان t وارد می‌شود. برابر است با:

$$\max\{F(i,k-1,t), R(t)\} + \frac{P(i,k,t)}{r_i}$$

اگر صف پر باشد — فعلی — اندازه بسته برحسب bit

اگر بخواهیم به جریان سهم بیشتری بدهیم بعلی ضریب می‌دهیم. (وزن)

$$\max\{F(i,k-1,t), R(t)\} + P(i,k,t)/w_i$$

اگر بخواهد ۲ برابر سرویس بگیرد پس زمان ارسالش برحسب اندازه بیت نصف می‌شود. این می‌شود WFQ

به جای 1 بیت در هر round، 2 بیت ارسال کند.

waited fair queueing

پس برای برقراری عدالت می‌شود از WFQ استفاده کرد و QoS را محقق کرد.

ایراد WFQ و FQ : به ازای هر بسته‌ای که وارد می‌شود باید فرمول فوق را محاسبه کرد. ← باید state جریان ترافیکی را نگه داریم و state را به روزرسانی هم کنیم.

در Core شبکه میلیون‌ها جریان ترافیکی وجود دارد و باید به ازای هر بسته میلیون‌ها state اضافی نگهداری شود که خیلی پردازشی زیادی می‌طلبد.

پس در سرویس‌دهنده‌های Core شبکه امکان پذیرش برای این در Core شبکه (سرویس‌دهنده‌های) FQ و WFQ امکان پذیرش برای این برای Buffer management در Core، مکانیزم‌های ساده‌تری هم پیشنهاد شده ← RED : Random Early Detection

RED (P.15) : قبلاً بیشتر ترافیک‌های شبکه اینترنت پروتکل TCP است. باید ۷۰٪ ترافیک شبکه‌ی اینترنت TCP خودش مکانیزم کنترل ازدحام دارد که نرخ ترافیک وارد شده به شبکه را کنترل می‌کند ← از نوع مکانیزم‌های Close Loop هستند.

Feedback این که به صورت ضمنی در TCP رخ می‌دهد تشخیص ازدحام با بیت کاپ است ← اگر pkt loss نباشد، ازدحام نیست. در TCP اگر loss شد، نرخ را کاهش می‌دهد. وقتی Ack می‌گیرد، نرخ بالا.

الگوریتم RED براساس هیستریک‌های زودهنگام به TCP عمل می‌کند.

در TCP تفاوتی بین جریان‌ها وجود ندارد ← FQ رقم پیش‌فرض می‌دهد از ازدحام جلوگیری می‌کند و QoS را تأمین می‌کند.

بین عدالت رعایت می‌شود

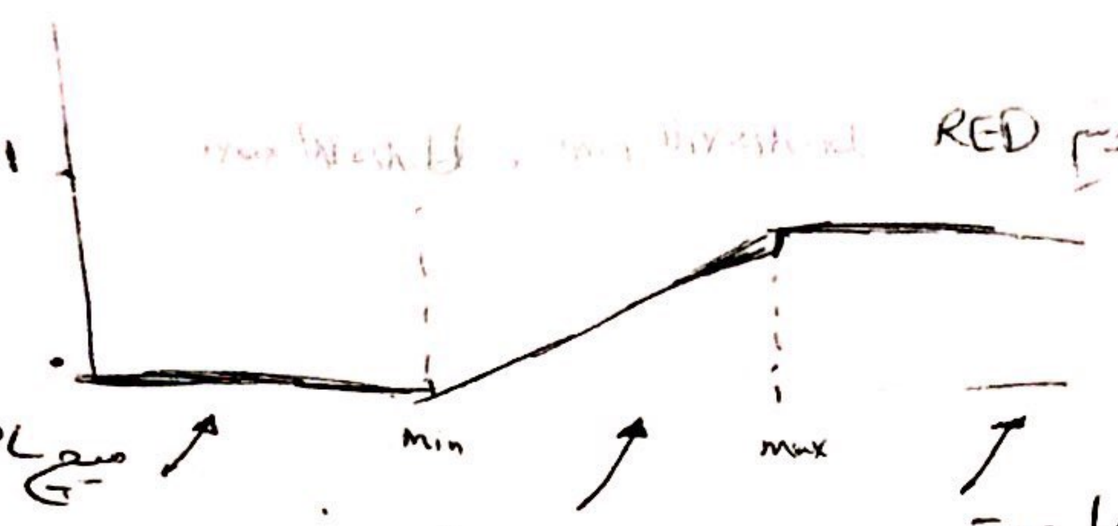
در الگوریتم RED اندازه‌ی لحظه‌ای صف تشخیص می‌دهد که آیا این بسته را وارد صف کنیم یا نه. بیشتر براساس اندازه‌ی لحظه‌ای صف ← احتمال PL عددی است که RED ایجاد می‌کند براساس طول صف هست.

طول صف

loss کنیم

Scanned with CamScanner

احتمال حذف بسته در الگوریتم RED



آیا Threshold دارد الگوریتم RED

(P.18)

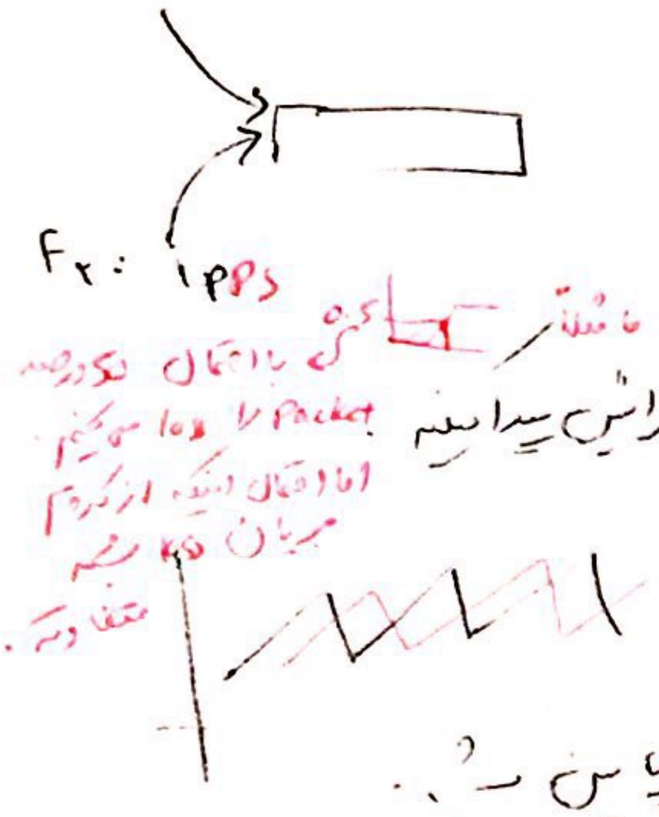
میچ PL اتفاق نمی‌افتد

براساس اندازه صف پر شده. احتمال PL افزایش پیدا میکند.

با احتمال ۱ به drop می‌شود.

F_1: 10 PPS

F_2: 1 PPS



در الگوریتم RED بسته‌ها براساس احتمال دور انداخته می‌شوند. به احتمال بسته‌ها دور می‌ریزند.

★ نکته: زمانی لازم است ان که دریافت کنیم $\frac{1}{11}$ از جریان ۲ و $\frac{10}{11}$ از جریان ۱

اول جریانی که کمیت‌های قبلی را از بین F_2 بیشتر به صورت موفق ارسال شود، نرخش افزایش پیدا میکند

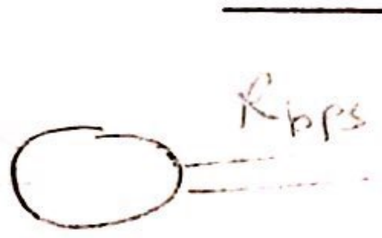
و سرویس Fair Queue ارائه می‌شود و سهم ۱ جریان برابر می‌شود.

اگر بگه 2 تا جریان داشته باشیم یکی 10 بسته در ثانیه، یکی 1 بسته در ثانیه. ما فقط کمی با احتمال یکدرصد 10 تا Packet می‌کنیم اما احتمال اینکه از کدوم جریان 10 بسته متفاوته.

اون که کمتر بود، موفق می‌رسه، سرعتش رو می‌بره بالا. اون یکی میاره پایین. (Fair Queue)

نرخ ارسال هر یک در پایین است.

این مکانیسم‌هایی که اینجا توضیح دادیم ← Packet Level هستند.



مجموع جریان‌ها — ظرفیت لینک — flow-level

$\sum F_i < R_{bps}$ ازدحام نداریم — کنترل ازدحام

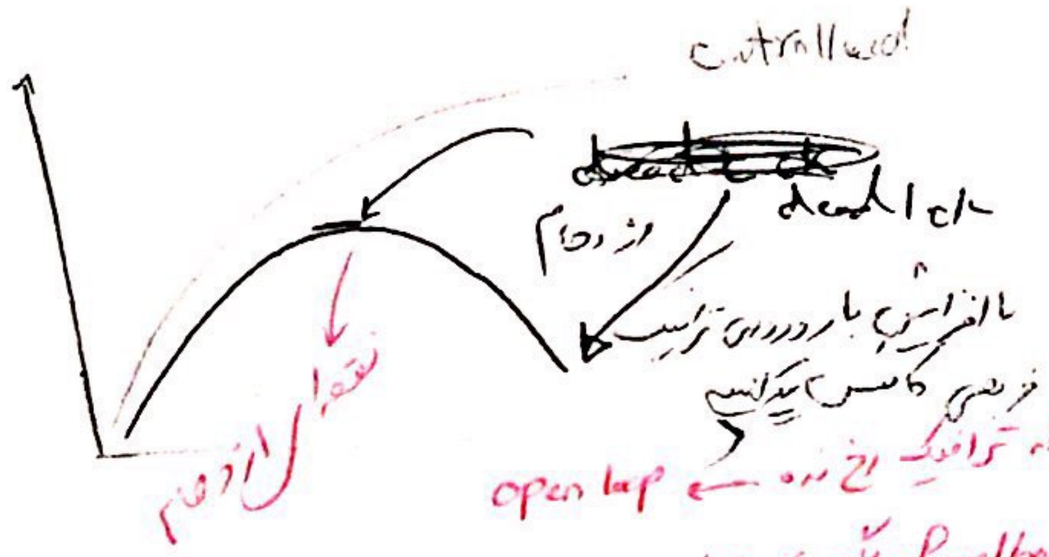
ترافیک اضافه حذف می‌شود — pkt loss ⇐ if $\sum F_i > R_{bps}$ ازدحام — این همه ازدحام

PL داریم: ازدحام: ترافیکی میخواهد وارد لینک بشود که این لینک توانایی انتقال این حجم ترافیک را ندارد. باعث می‌شود صف‌ها پر شود PL اتفاق بیفتد. هم تأخیر افزایش پیدا می‌کند (delay) و هم PL افزایش پیدا می‌کند (Pkt loss) ⇐ سرریز

نکته: بدترین که اگر این بسته‌ها مربوط به پروتکل TCP باشند بسته حذف شده دوباره ارسال مجدد می‌شود. یعنی F_1 از 10 PPS به سرانجام می‌رسد. اگر به بسته از شبکه دور بیاندازیم ← 11 PPS پس صرفاً باید یک کنترل (ازدحام) صورت بگیرد

(P.19)

در غیر اینصورت شبکه با پدیده deadlock مواجه می‌شود ← ترافیک دارد افزایش پیدا می‌کند و ازدحام دارد گسترده‌تر می‌شود ← ازدحام بدتر می‌شود به خاطر ارسال مجدد



(P.20)

کنترل ازدحام: open loop / closed loop

دو راه: ① ترافیک ورودی را کنترل شده بدهیم. به شرط اجازه دهیم که ترافیک رخ بدهد ← open loop

② اجازه ورود را آزادانه بدهیم ولی از خروجی شبکه feedback بگیریم. بفهمیم ازدحام رخ داده یا نه. اگر رخ داده، ترافیک ورودی را کاهش دهیم تا feedback ← مطلوب.

a) Acc ← مشخصات ترافیک ورودی را می‌گیریم (آماری) ← (loss, delay, max burst size, average rate, Peak rate)

b) policy: نظارت، اگر لازم به تغییر شکل ترافیک

c) ...

۳۶+۴

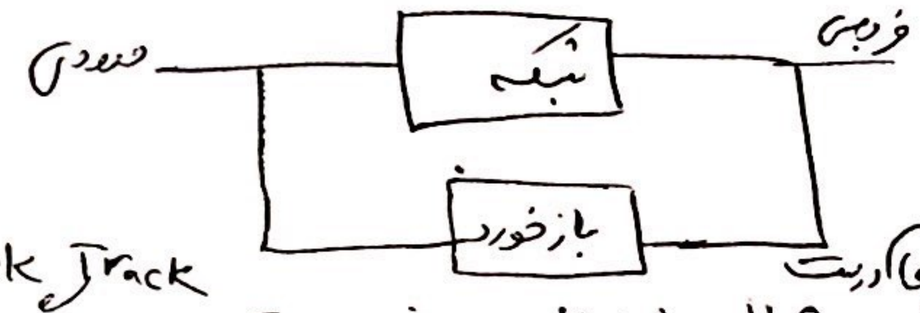
Scanned with CamScanner

جلسه نهم

* بنام خدا *

Policing : حالت است که این توافق را از طرف کاربر نقض نشه. اگه نقض شد، می‌تونه اون ترافیک اضافه را حذف کنه. (ادامه پایین صفحه)

مکانیزم کنترل ازدحام Close Loop : جریان ترافیکی قبل از ورود کنترل می‌شود. ترافیک آزادانه می‌تواند وارد شود. (حداکثر تا زمانی که بهش بتوانیم پیش‌بینی داشته باشیم)



تا ازدحام نشه

اگر مبدا براساس چیزی که دریافت کرده نرخ ارسال خودش را مدیریت کند.

صریح: گرفتن این که با ازدحام روبرو شده یا نشده. مستقیماً یک پیغام دریافت کند implicit

Back Track

Hop by Hop : خودش از طریق مسیریابها برمی‌گردونه بازتو

Feedback

Explicit : مقصد: گره source خودش از وضعیت شبکه ازدحام را تشخیص بده

End to End : نیازمند به مقصد که مقصد باز خورد را به مبدا بفرستد

(P.27)

ازدحام یک پدیده نامطلوب است اثر نامطلوب دارد ← کیفیت کاهش پیدا می‌کند

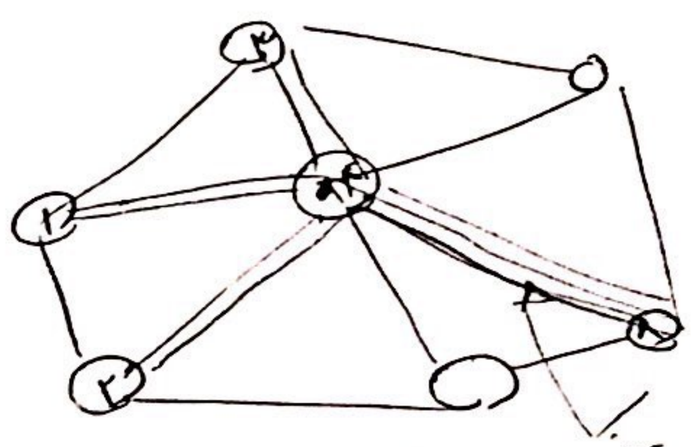
در TCP این Feedback به صورت ضمنی است

مهندسی ترافیک : flow aggregate (P.28)

مالک شبکه با تحلیل ترافیک داخلی شبکه یک پیش‌بینی از آینده دارد. به سطح مسیر تعریف می‌شود که مسیریابی با محدودیت انجام میشه

ترافیک را طوری که دوس داریم میسازیم. که جریان‌های ترافیکی همه جریان‌ها به تمام بهترین مسیرش عبور نکنه ولی مجموع این جریان‌ها بهترین مسیر رو استفاده بکنند و به حداکثر کارایی میرسیم

مثال:



۱ → ۸

۲ → ۸

۳ → ۸

ترافیک شلوغ شود. ازدحام

برای Policing از الگوریتم‌های متفاوتی استفاده می‌شه مثل leaky bucket.

(P.26) یک شبکه مثل A، یک درخواست به B می‌ده برای پذیرش ترافیک. Policing در ورودی چک می‌کنه که توافق نقض نشه.

خود A هم با نظارت داخلی، تغییر شکل ترافیکی سعی می‌کنه ترافیک را نقض نکنه.

P.25 : در مدیریت ترافیک، اگه داخل روتر را ببینیم، ترافیک ورودی وارد می‌شه با classifier، جریان‌های ترافیکی تشخیص داده می‌شه، بعد Pkt scheduling و buffer mgmt روش اعمال می‌شه. برای پذیرش بحث‌های admission control و از رو کردن بهای بانه، agent های routing برای مسیریابی و مدیریت در قسمت control plane اجرا می‌شه و در قسمت data path : Pkt scheduling، buffer mgmt، سرویس ارائه می‌شه.

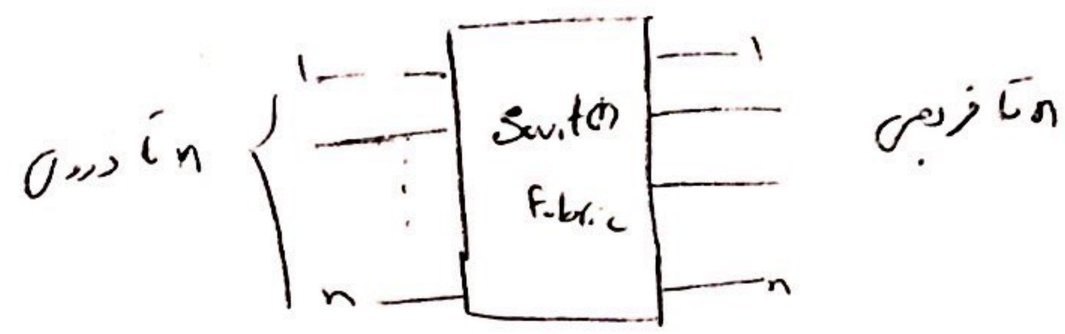
Scanned with CamScanner

Basic of Packet switching

تا ساعت شبکه‌های Grid ما صحبت کردیم

switch fabric : عمل switching را انجام میده. اینکه بسته پورت ورودی اش مشخص شده، به این پورت ورودی فرستاده

یک سوئیچ n×n : همزمان n بسته می‌تواند وارد شود و n بسته خارج شود.
طول بسته‌ها می‌تواند متفاوت باشد.



switch fabric چگونه می‌تواند برنامه‌ریزی پیاده‌سازی شود.

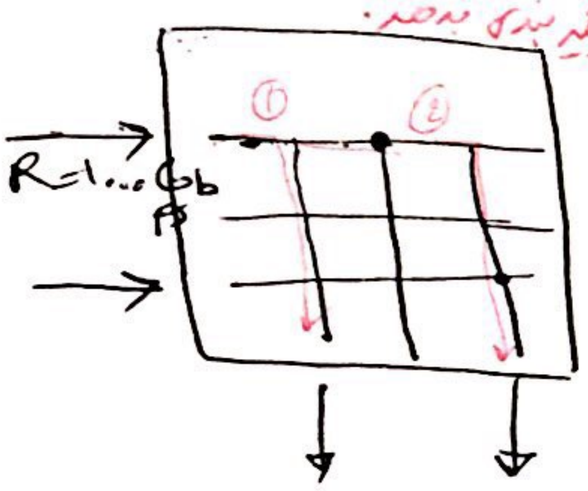
switch در : switching & Routing

Routing (P.4) : یعنی پیدا کردن مسیر در شبکه (که از کدام گره‌ها میان‌ها رد شویم تا به مقصد برسیم)

switching (P.5) : در داخل یک گره محاسبه کند. بسته از یک پورت ورودی آمده، از یه پورت خروجی باید خارج شود. این عمل switching

unicast & multicast :

برای switching فقط به اندازه زمان رسیدن یک بسته (بسته بعدی) فرصت داریم. (مدت زمان دریافت بسته بعدی)



بسته بعدی باید خارج بشه) در زمان خیلی کوتاه باید تغییر پیکربندی بدهد.

$L = 1000$ bit $\quad t = \frac{1000}{1000 \times 10^9} = 10^{-8} = 10\,ns$

یعنی عمل switching باید در حد $O(ns)$ تصمیم‌گیری شود

مشکل عمده در switching اینکه زمان تصمیم‌گیری در مورد اینکه چه پیکربندی‌ای انجام شود بسیار کوتاه است.

یک بسته از یک پورت وارد و چند مقصد. در switch یعنی یک بسته از یک پورت وارد شده می‌خواد از چند پورت خارج بشه، تکنیک سرویس multicast باشه ولی switch unicast.

در سرویس switch :

unicast : بسته‌ای از یک پورت وارد و بخواهد از یک پورت خارج شود.

multicast : بسته‌ای وارد شود و ۳ تا یا n تا پیش فرستاده شود و همزمان بخواهد از چند پورت خارج شود.

unicast : یعنی سرویس ما یک مبدا دارد و یک مقصد.

multicast : یعنی سرویس ما یک مبدا چند مقصد دارد

ممکن است یک بسته‌ی خاص: در یک سوئیچ چهار تا m.cast ارسال شود در سوئیچ بعدی ایکه سرویسش مست و فقط multicast خارج شود

Throghput (P.7) : تعداد ورودی: همیشه نزدیک ناظر بیرونی در نظر است.

ناظر

ترافیک خارج می‌شود از ورودی

اختلاف m و k یا داده‌ها می‌شوند یا باید صف بی‌نهایت داشته باشیم

$$\text{Throghput} = \frac{m}{k} \le 1$$

در بازه‌های زمانی که ما صف بی‌نهایت نداریم پس هیچ وقت نباید پورت‌های ما 100% load داشته باشند باید همیشه load کمتر از Throghput باشد تا صف بی‌نهایت اتفاق نیفتد.

Scanned with CamScanner

speed up : افزایش سرعت

مثلاً اگر بر یک switch، speed up = ۲ باشد : یعنی این switch توانایی دارد در مدت زمانی که یک بسته به switch میرسه ۲ بار عمل switching انجام دهد. یعنی سرعت switching ۲ برابر نرخ ورودی است. در یک بازه زمانی که یک packet میرسه توانایی دارد ۲ تا packet خارج کند.

اگر speed up > ۱ باشد صحیح باید در پورت های خروجی با فرض داشته باشیم تا بتوانیم ادامه شود. (وگرنه، اون یکی را چی کار کنیم؟)

(P.8) Blocking (internal) : حالتی است که اگر شرایطی وجود داشته باشد که نتوانیم یک پورت آزاد ورودی را به یک پورت آزاد خروجی متصل کنیم قوانین موقعیت Blocking می‌گویند.

اسلایدی ۸: یک switch حالتی stage هست. اسلایدی ۹: جایی که collision در سوئیچ های Banyan رخ بدهد بهش Blocking می‌گویند (به ☒ که مدام باید هر ۲ تا از یک پورت خارج بشه تصادم)

در مراحل سوئیچ ها باید طوری عمل کرد که internal Blocking نداشته باشند تا ۹ اتفاق نیوفته.

(P.10) output port contention : رقابت بر سر پورت خروجی : در یک زمان بیش از یک بسته مخابرات بخواهند به یک پورت مقصد بروند

اگر هر ۲ را فرمان سوئیچ کنیم مقصد خروجی Conflict رخ میده.

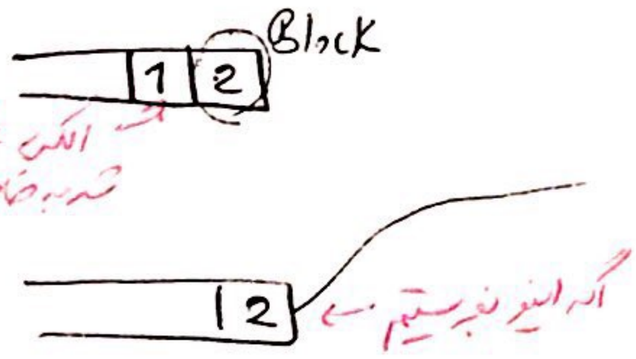
حل کردن مشکله رقابت بر سر پورت خروجی ← تنها یک را حل دارد ← استفاده از بافر برای نگهداری بسته هایی که پورت خروجی مشترک دارند ولی همزمان نمی‌توانند به پورت خروجی شان برسند. که یکی را نگه داریم switch کنیم تا بعدی را بافر کنیم (در ورودی)

* اینکه بافر را کجا بذاریم خودش یک مسئله است. پس ما مکانیزم های بافرینگ داریم. بافر در ورودی - در خروجی - در تکنیک های دیگر

Speedup switch به دهیم که منتقل کنه در خروجی بافر کنه.

یکی را انتقال دهیم، اون یکی را در ورودی نگه داریم.

(P.11) اگر بافرها را در ورودی قرار دهیم یک مسئله دیگری به نام Head of Line Blocking اتفاق می‌افتد.



یکی بلاک شده صف.

اگه اینو نفرستیم

یعنی سرصف بلاک شده سلولی که می‌توانست به یک پورت آزاد برود نمی‌تواند برود.

الان در مثال، به جای ۹ بسته در ثانیه، 3 تا فرستادیم پس گذردهی می‌شه 75 درصد.

این باعث کاهش Throughput می‌شود.

ثابت می‌شود که مسئله HoL، گذردهی را به ۰.۶۸ کاهش میدهد یعنی بنابراین Throughput از ۱.۶۸ بیشتر نمی‌شود چون باعث به وجود آمدن صف بی‌نهایت می‌شود.

چون تغییر پیکربندی همزمان نیست

بسته های ورودی، طول های متفاوتی دارند ← زمان switching متفاوت ← از جایگاه سلولی خارج می‌شه

(P.12) switching & packet mode & Cell mode switching : به سبب سخت سازی switching می‌شود چون طول بسته ها متغیر باید packet mode باشیم!

در ورودی در هر ثانیه input segment module : ISM تقطیع می‌شوند به Cell هایی با اندازه ثابت که در Queue cell داریم

وقتی سلولی شد، دستمون باره، موازی سازی مثل pipelining انجام دهیم

در یک time slot ها یک Cell گوهیترین بیش بسته دهیم که اندازه ثابت داره.

بعد در قسمت خروجی output : ORM، Cell ها Reassemble می‌شوند و یک بسته ساخته می‌شود.

Reassemble module

برای اینکه switching با سرعت بالا داشته باشیم باید cell mode باشیم.

اندازه، همه سرعت بالا داشته باشیم

Scanned with CamScanner

Shared medium — مسیر مشترک — Bus مشترک

Time Division — Switching — تقسیم زمانی انجام میدن (مثل چراغ راهنمایی)

مثل چهار راه که تقسیم را با چراغ سبز و قرمز انجام میدن.

Shared memory : حافظه مشترک

Space Division : فضایی — بیشتر مدارهای موازی دارند.

در چهار راه مثلاً پل های غیر همسطح میزنن

مسیرهای موازی ایجاد شده مثل پل های متقاطع هم سطح. هست با همان سوئیچ کریدور جابجا شیم.

مسیریابی نداریم. Self Routing هست.

Single path : مشکل هست با چند مسیر دارد. از هر ورودی به هر خروجی فقط یک مسیر دارد. یعنی Routing نداره.

قصد مشخص باشد، یک مسیر فقط دارد. نیاز به Routing نداریم. سازگار تر

مشکل: اگر یک Hult در یکی از مسیرها اتفاق بیوفته، switch از کار میفته.

Space Division

switching Element Fail شود

مثال: Fully interconnected — Banyan — Crossbar

multi path : از هر ورودی به هر خروجی بیش از یک مسیر وجود دارد. پس برای هر بسته یک مسیر تصمیم میگیریم. پس خرابی پرواز مشکل داریم و نیز پیاده سازی پیشرفته است. Throghput بیشتر داریم و

را اجرا function

حسن: با افزایش Switching Element هرچند با کاهش کارایی، switch از کار می افتد. اما Throghput میتوان

درخواست عادی از همه مسیرها میشود استفاده کرد و Performance بیشتر داریم.

مثال: Augmented — Clos — multi plane — Recirculation — Banyan

دلایل ایجاد سوئیچ های multi path :

① افزایش ظرفیت انتقال با هر خروجی هر سوئیچ میتواند کارش را انجام دهد.

② افزایش کارایی: افزایش سرعت switch

③ با افزایش مسیرها ریسک switch میتوان Blocking را از بین برد.

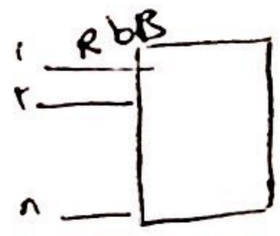
جلسه دهم

بررسی سوئیچ های shared medium (بس مشترک)

۲ مورد اصلی سوئیچ های Time Division محدودیت اندازه دارند.

اندازه یک switch؟ اگر switch، n پورت ورودی داشته باشد و هر پورت $\frac{R}{b}$ ظرفیت داشته باشد اندازه سوئیچ

محدودیت اندازه دارند یعنی $N.r < K$ ⟸ K: یک مقدار ثابت — $N.r$: اندازه سوئیچ — مجموع ترافیک ورودی — مقدار ثابت



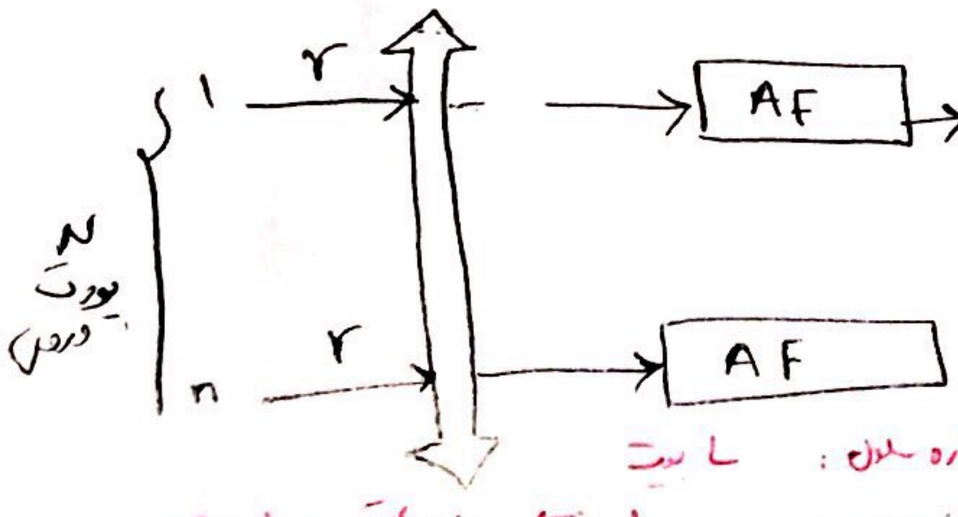
محدودیت اندازه: یعنی یا تعداد پورت ها نمی تواند یک مقدار بیشتر شود یا ظرفیت ارسال نمی تواند از یک مقداری بیشتر شود

این مورد جزء معایب Time Division هست. ضرر: هزینه پیاده سازی کمتر

سرعت Bus > N.R ← باید Bus سرعت این Bus مشترک داریم که Shared medium : یک

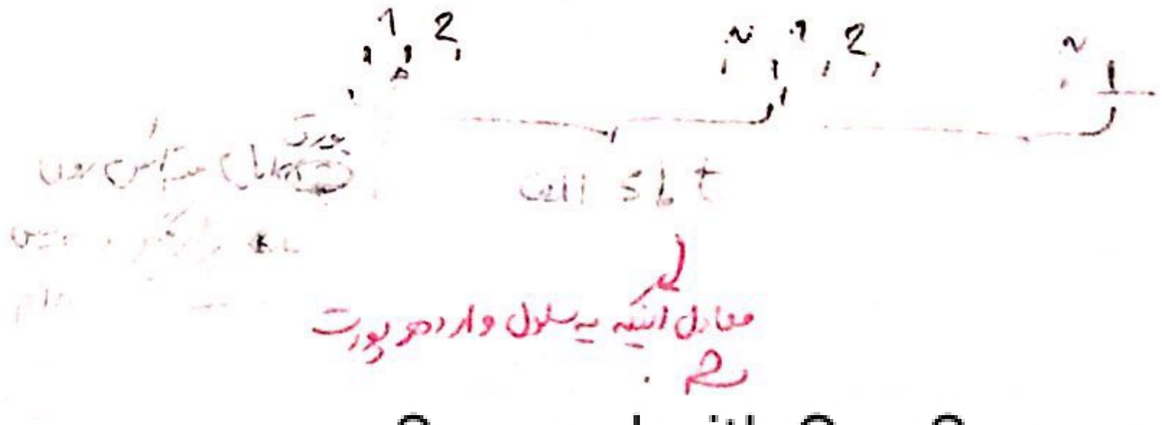
یک سیگنال تزریق میکنیم بدون زمان. اندازه این سیگنال یک Cell slot است.

مدت زمانی که یک سلول وارد پورت می شود. اگر اندازه یک Cell L bit



اگر اندازه سلول: L بیت ⟸ زمان سلول $\frac{L}{r}$ ← چون سرعت وسط بالا تره r

Slot بندی میکنیم به N Slot



معادل اینکه یک سلول وارد هر پورت

Scanned with CamScanner

پورت عددی 1 در slot (Cell slot) خودش بسته اش رو روی باس میذاره. چون سرعت باس از N.r بیشتر نیست تو یه cell

slot هر پورتی میتونه بسته عددی اش رو روی باس بذاره.

وقتی یک سلول را پورت عددی 1 سوی باس می‌گذارد همه‌ی خروجی‌ها پیدایش می‌کنند.

AF: مشخص می‌کند که عملاً بسته‌ای که از پورت 1 وارد شده می‌خواهد به کدام پورت خروجی برود. مثلاً اگر بخواهد پورت n برود. AF تمام پورت

به غیر پورت 1، اجازه عبور این بسته و تمامی به غیر بقیه پورت‌ها را نمی‌دهد و فقط AF خروجی n اجازه عبور می‌دهد.

و حذف می‌کند.

به سلول‌هایی که مقصدشون این پورت باشد اجازه عبور می‌دهند.

از AF که عبور کند در ماتریس خروجی نوشته می‌شود و سلول سرعت ارسال می‌شود.

Read 1 + write N تا

اگر همه‌ی سلول‌ها بخواهند به یک پورت خروجی هم بروند هم این امکان وجود داره.

بدترین حالت در بافر خروجی n تا خانه برای هر کدام که همه‌ی n تا در روی مقصدشون یک پورت خاص باشد + ! دونه هم برای انتقال

سرعت را بخوانیم ارسال کنیم. پس در یک cell slot ← N+1 دسترسی به حافظه (بافر) داریم. Read

T_{mem}: در حد چند ns ثابت است.

$\rightarrow (N+1)$ Access memory

وقتی T_{mem} طول می‌کشد $\rightarrow T_{mem}(N+1)$

باید از cell slot تمام باشد. T_{cell} $\rightarrow (N+1)T_{mem} < T_{cell} \rightarrow (N+1) < \frac{T_{cell}}{T_{mem}}$

$T_{cell} = \frac{L}{R} \rightarrow (N+1) \leq \frac{L}{R.T_{mem}} \Rightarrow (N+1)r \leq \frac{L}{T_{mem}}$ ← ثابت ← ثابت

$N.r + r \leq K \rightarrow N.r \leq K$ صرف نظر

اثبات کردیم اندازه سوئیچ‌های shared medium ثابت است.

مثال: اگر L = 64 Kbyte و R = 10G و $T_{mem} = 4ns$ در این سوئیچ چند پورت می‌تواند داشته باشد؟

$\frac{L}{R} = \frac{64 \times 8}{10 \times 10^9} = 51,2ns$

R / Throughput

$N+1 < \frac{51,2}{4} \rightarrow N = 11$

اگر بخواهیم N را افزایش بدهیم باید R را کاهش بدهیم. پس این $N.r < K$ مشکل سوئیچ‌های Time Division است

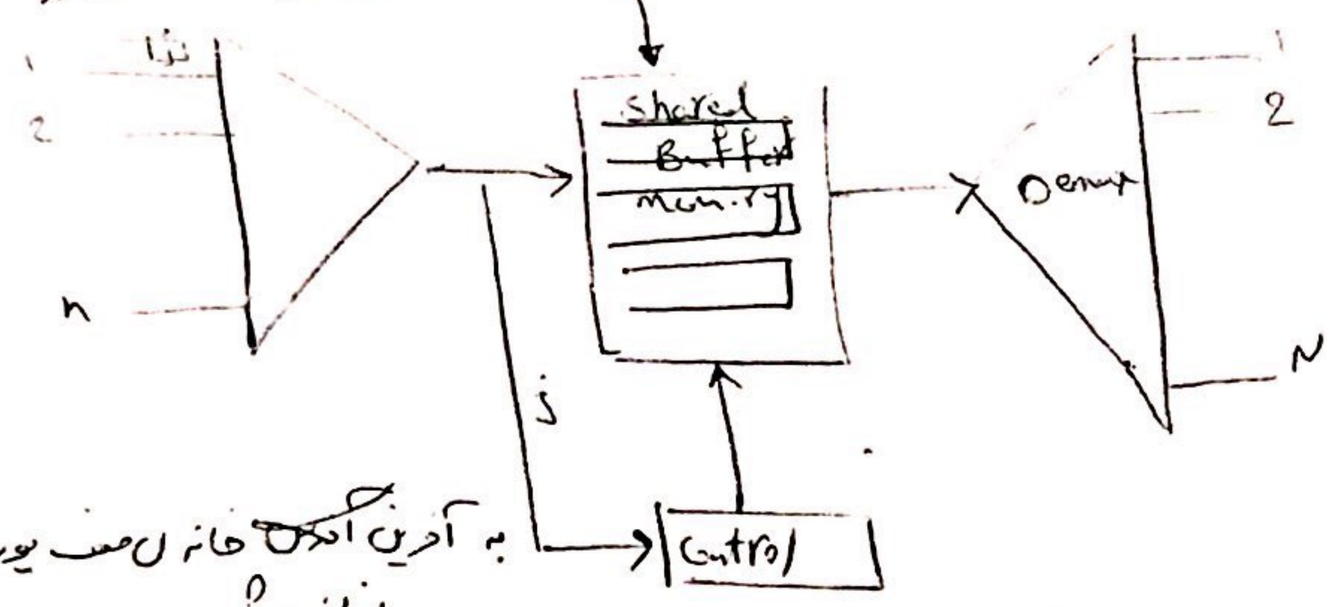
در این سوئیچ‌ها حافظه بافری که برای هر پورت گذاشتیم پر شود دیگر اتفاق می‌افتد. در حالی که بافر سایر پورت‌ها فضای خالی بدون استفاده دارد و این خوب نیست. چون حافظه‌ها را از آن پورت‌های خروجی بگیریم Packet ها ریخته شده در صورت وجود فضای خالی هم اتفاق می‌افتد

بیفتد.

در shared memory، این مشکل رفع می‌شود. یک حافظه برای تمام پورت‌های خروجی داریم که 1 تا صف خروجی مجازی برای هر خروجی ایجاد می‌شود.

Scanned with CamScanner

صف های بافری (FIFO) که دو طرفه یک واحد کنترل این صف



به آدرس آمدن خانه ی صف یورت لازم اضافه میشه.

در هر Cell time ، N تا دستور برای نوشتن در Buffer و در هر Cell time در حداکثر این تا فرض

N تا Read از سر هر صف خواهیم داشت *

این صف ها تقطیعی هستند با اندازه ثابت و Fix شدند *

$2N.T_{mem} \leq T_{cell} = \frac{L}{r}$

$\Rightarrow 2Nr \leq \frac{L}{T_{mem}}$ ثابت ← محدودیت حافظه

اینجا محدودیت بیشتر روی حجم حافظه ای که مصرف میکنیم است ولی با هزینه کمتر میتونیم shared memory برای باشیم

اگر محدودیت اندازه سوئیچ ها Time Division ، مشکلی برای کاربرد ما ایجاد نمی کند. از این نوع SW ها استفاده میکنیم که بازم shared memory ها فاصله کمتری مصرف میکنند.

پس مسیر پیشگیری محدودیت کاربرد ما

اگر محدودیت شدید باشد و نتوانیم کار از Time Division SW ها استفاده کنیم میریم سراغ Space Division SW ها.

$2N$ عدد زیاد باشد

با تعداد پورت زیاد شده ← سرعت پایین ($2N.r$)

از لحاظ توری محدودیت اندازه ندارند اما از نظر عملی (پیاده سازی) این مدارات VLSI هستند که باید داخل chip دیزاین شوند که صفحه محدودیت فضای دیزاین سازه داشته باشیم که تکنولوژی پیشرفت کرد پس میریم محدودیت اندازه ندارند

Single Path : هر سلول، یک پورت مقصد دارد و یک مسیر طی میکند تا به پورت مقصد برسد. ⇐ مسیریابی داخل سوئیچ نداریم. (Self routing)

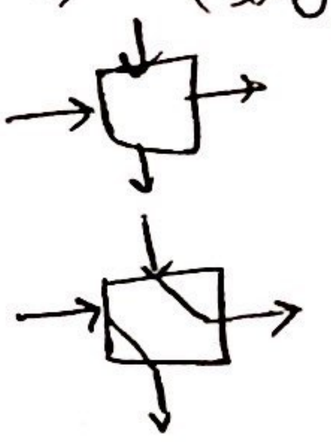
multiple Path : وقتی یک سلول به مقصد پورت j وارد می شود، می تواند چندین مسیر j داشته باشد. (افزایش reliability) (افزایش کارایی)

Cross bar switches : جای switch ها که blocking دارد، افزونگی کمک می کند از بین برود

پورت های ورودی را خطوط افقی در نظر بگیریم.

— عمودی — — فرودی — —

در محل تقاطع پورت های ورودی و خروجی (ماتریسی که تشکیل شد) مناسبی



Cross state

Bar state

Cross Point switch قرار میدهیم.

Bar state ها ۲ تا دارد.

Cross Point ها N^2 داریم

اگر یک سلولی روی هر پورتی وارد شود و مقصدش هر پورتی باشد. میتونه با حرکات افقی عمودی و Bar به خروجی مورد نظر برسد. (P.20)

این switch : internal Blocking اصلاً نداره. تنها دو حالتی که ممکنه Blocking اتفاق بیوفته که ۲ یا چند سلول بخواهند به یک پورت خروجی بروند که بحث output contention پیش میاد ← استفاده از بافر / اگر در Cross Point ها بذاریم N^2 تا

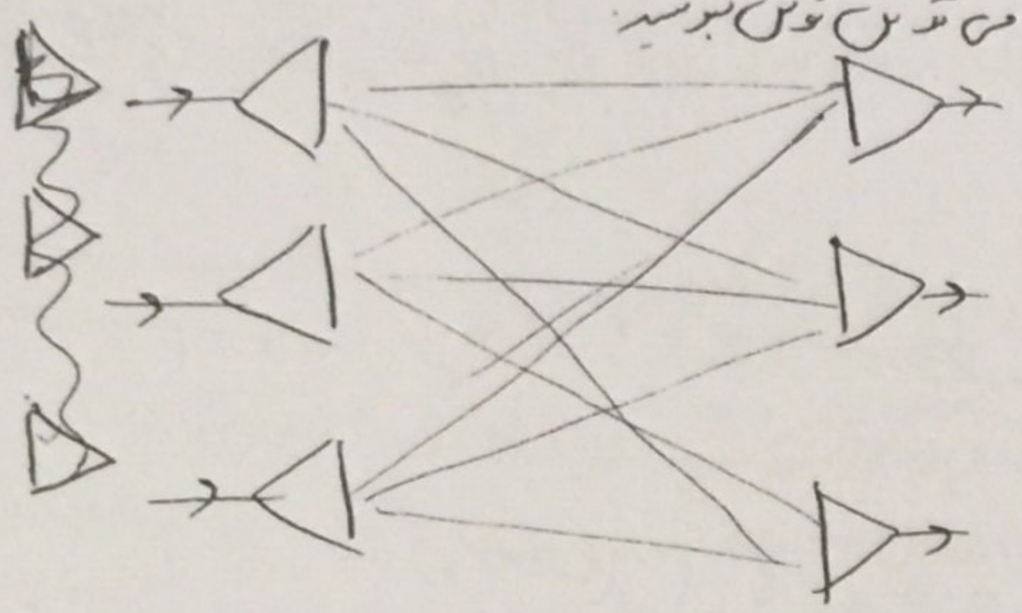
N^2 حافظه جزء نیاز داره که هزینه پیاده سازی به شدت افزایش میده / بافر در پورت ورودی

چون N ورودی داریم، N خروجی، N^2 تا Cross Point داریم.

Scanned with CamScanner

Fully interconnected switches :

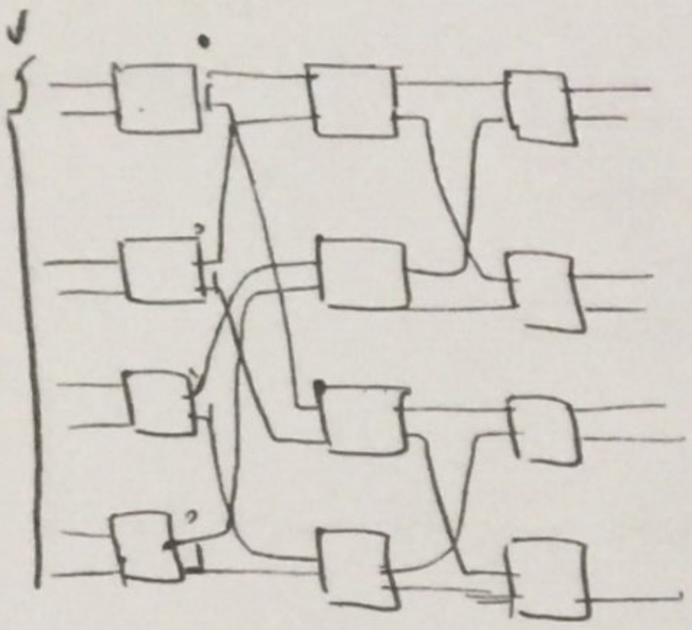
تمام پورت‌های ورودی و خروجی به هم متصل اند. اگر switch N×N باشد با N باس می‌توان پیاده‌سازی کرد. به ازای هر خروجی 1 باس استفاده می‌کنیم که همزمان هر N ورودی می‌توانند روی آن بنویسند



(پیاده‌سازی N باس خیلی پیچیده است به طوری که به هم برخورد نکنند)

در VLSI عملاً قابل پیاده‌سازی نیست.

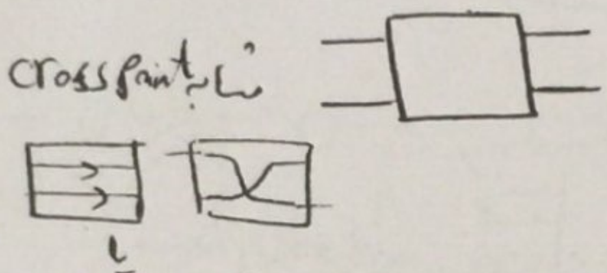
Banyan_Based switches :



این سوئیچ‌ها دارای stage هستند. اگر switch ها N پورت داشته باشد

تعداد stage ها = $\log_2 N$

در هر stage، $N/2$ عدد 2x2 switching Element (2x2 SE) داریم



شبیه crossbar

با بیت تصمیم‌گیری شود که در کدام حالت باشد پس خروجی

a) Delta network

stage 1 2 3

پس تعداد SE های به کار استفاده شود $N/2 \times \log_2 N$ $O(N \log N)$

برای شکل اگر N=100 ، $100 \times \log_2 100 = 700$ SE

$1.4 \rightarrow$ SE

پس از نظر پیچیدگی پیاده‌سازی به شدت کم هزینه‌تر نسبت به crossbar

در crossbar ← N×N ، SE استفاده می‌شود. $O(N^2)$

ساختارش براساس یک قاعده است و منظم ← می‌شه در VLSI الگوریتم نوشت که ساختارش را بده.

(a) Delta Network : اتصالاتی که در این شبکه‌ی Banyan switch کشیده شده، یک اتصالات کاملاً منظم و با قاعده‌ای هستند

یک شبکه منظم با قاعده است و برای پیاده‌سازی اتصالات در VLSI جواب مناسب و پیچیدگی ندارد.

تمام شکل‌های Banyan Based ← internal Blocking دارند ← روش‌هایی پیشنهاد شد که این مشکل رو برطرف کنند که

جلوتر بررسی می‌کنیم ← multi path (P.25)

Augmented Banyan :

(a) Augmented Banyan : همان Banyan است که تعداد stage هایش افزایش پیدا کرده (P.26)

در Banyan تعداد stage ← $\log_2 N$ ولی می‌شه در Augmented اینجوری نیست. اضافه کردن stage باعث شکستن internal Blocking از بین برد.

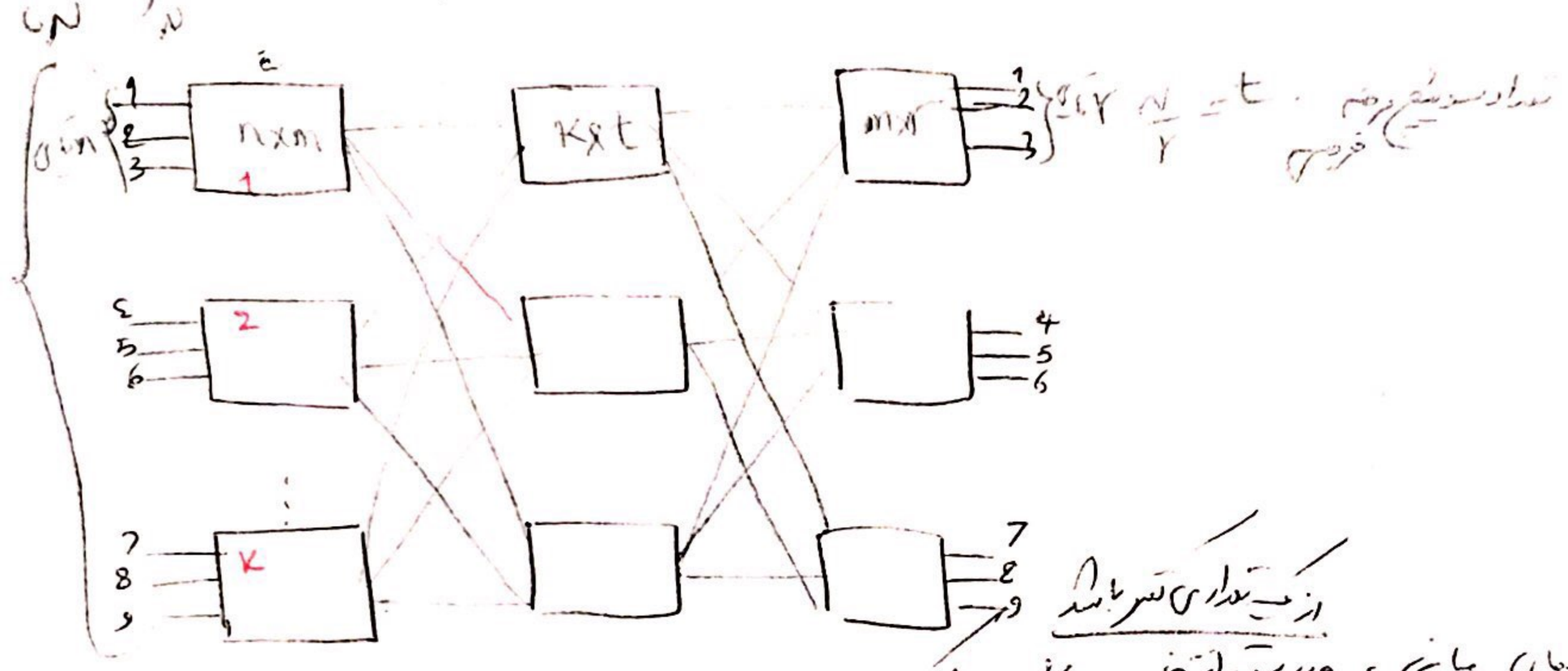
Scanned with CamScanner

Three stage clos — سه مرحله‌ای — مرحله ابتدایی — مرحله میانی — مرحله خروجی

ورودی‌ها را در گروه‌های n تایی دسته‌بندی می‌کنند.

$K = \frac{N}{n}$ ← تعداد سوئیچ‌های هر مرحله

$t = \frac{N}{r}$ — در مرحله میانی n تا سوئیچ داریم که از همه‌ی ورودی‌ها بهشون وصل میشیم. و همه خروجی‌ها وصل می‌شوند.



از یک تعداد کمتر باشد

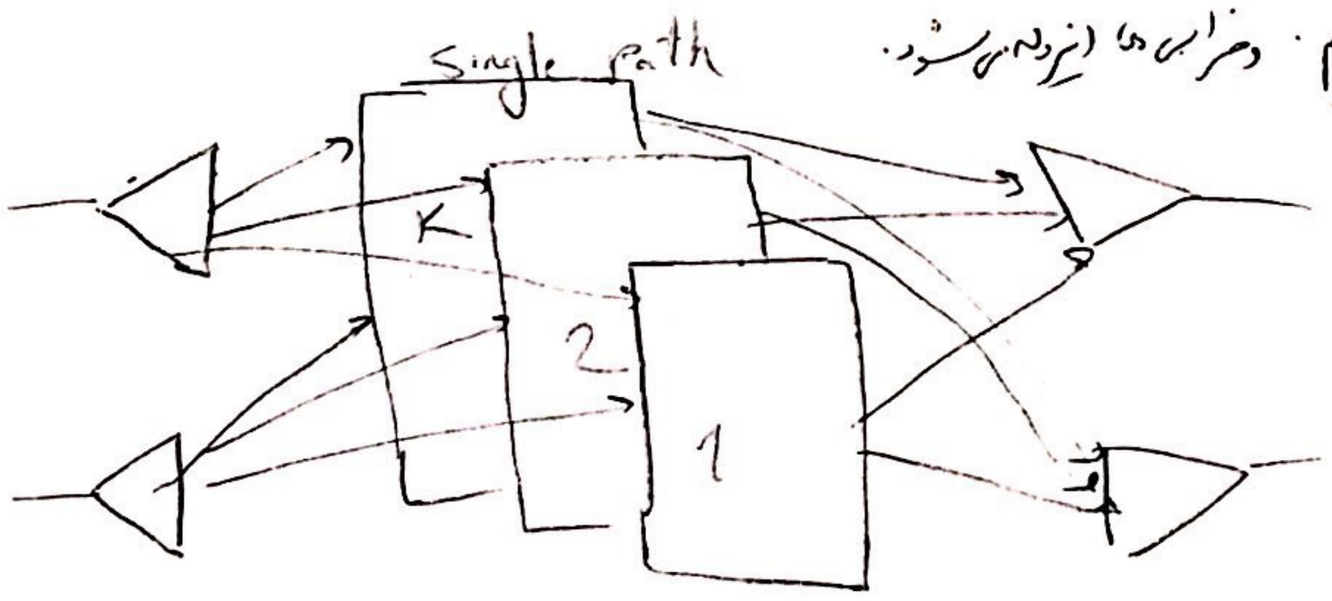
در clos اگر تعداد سوئیچ‌های میانی، در بستر انتخاب نشود مشکل internal Block دارند. در این شکل پورت آزاد ورودی 9 به پورت آزاد خروجی 4 نمی‌توان وصل کرد.

(P.28)

Multi plane switch : K تا Single path switch را موازی گذاشتند. K تا مسیر از هر ورودی به هر خروجی می‌توانیم Load balancing بین این K تا انجام بدیم، سرعت سوئیچینگ K برابر می‌شوند. این K وقتی به صورت مستقل کار می‌کنند.

Single point of failure — نداریم fault tolerant داریم. و خرابی‌ها ایزوله می‌شود.

① سرعت بیشتر

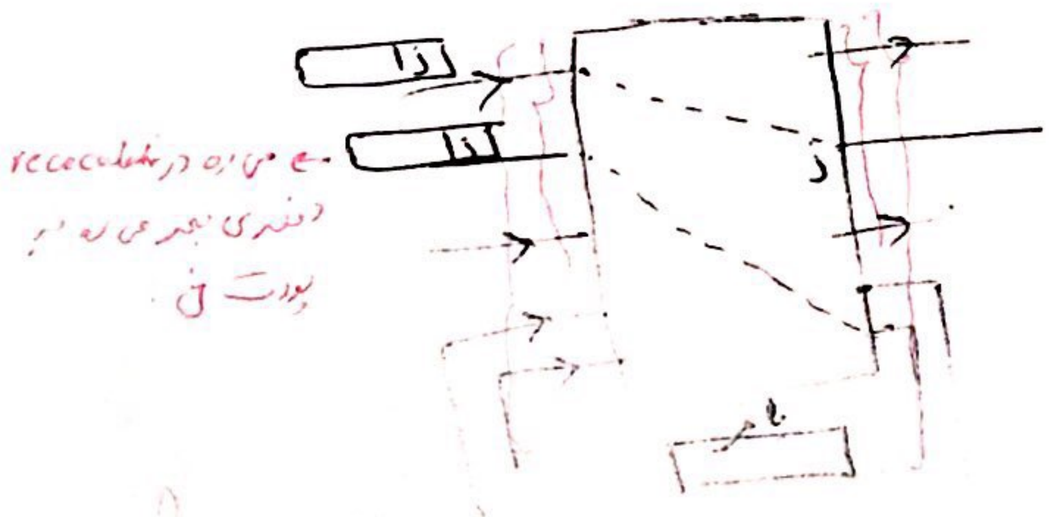


(P.29) ⓒ

Recirculation switch :

با جدا این switch از تعداد پورت‌ها بیشتر، جابجا انجام شده

output citation, internal block نداشته باشیم.



Scanned with CamScanner

یک بسته ممکنه چندین بار بره تو مسیر Recirculation . برای اینکه TTL اش منفی نشه و drop نشه ← میاد internal priority تعریف می‌کنند که هر بار بسته میره تو این مسیر ← اولویت داده میشه بهش.

جلسه یازدهم

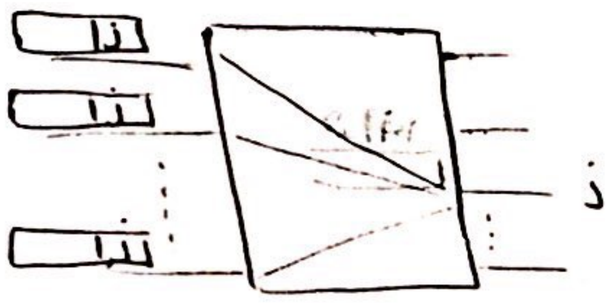
Buffering strategies

به دلیل مشکل رقابت بر سر پورت خروجی مجبوریم از حافظه استفاده کنیم. چون بسته‌هایی که به یک پورت در یک time slot رسیده‌اند فقط یکی رو میشه ارسال کرد و باقی باید در بافر نگهداری شوند. وقتی این بافر کجا گذاشته شود. از حافظه مشترک استفاده شود یا نه.

1 shared Buffer Queueing ، 2 output Queueing ، 3 input Queueing ، 4 virtual output queueing (VOQ)

بافر رو تو پورت ورودی بذاریم و صف‌بندی براساس پورت خروجی انجام بدیم

5 Combined input & output Queueing ، 6 cross point Queueing



در یک cell slit فقط یک بسته را می‌توان ارسال کرد پس برای حل مشکله output contention نیاز به بافر داریم.

P.51

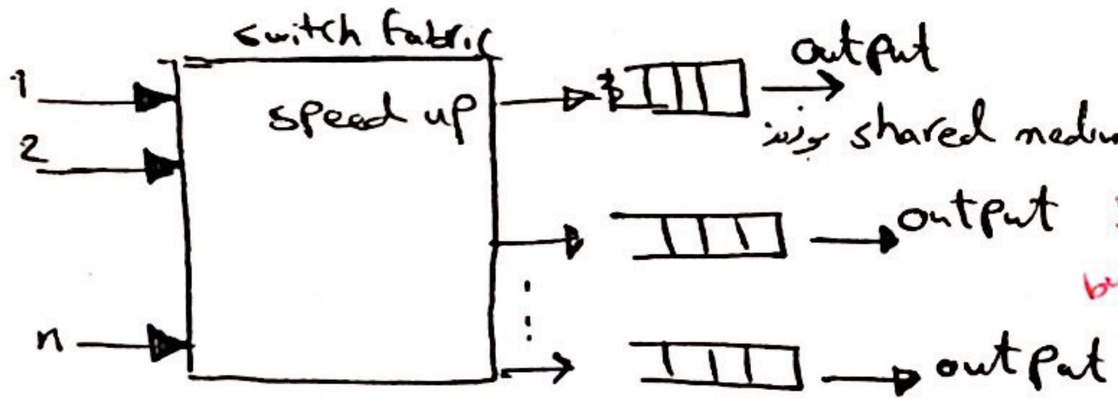
- shared memory Queueing : صف‌بندی با حافظه مشترک ← مثل سوئیچ‌های shared memory که یک حافظه مشترک داریم.

درون این حافظه مشترک، صف‌های منطقی مجزا برای هر پورت خروجی داریم.

① از نظر حافظه مصرفی یکی از بهترین روش‌هاست. در واقع بهترین روش هست. هرجا یک منبعی رو استفاده اشتراکی ازش انجام بدیم بهتر در مورد مزایا

② سادگی پیاده‌سازی ③ تأخیر در این سوئیچ‌ها عادلانه است.

معایب: محدودیت حافظه داریم ← $N.r < K$ ← همین باعث می‌شود ما احتمال cell loss داشته باشیم. اگر محدودیت اندازه نداشته باشیم هیچ انتخاب ما shared memory هست.



shared medium بودند

- output Queueing : یکی از مثال‌های این نوع سوئیچ‌ها، سوئیچ‌های bus base

این سوئیچ‌ها باید speedup داشته باشند که در یک کلاک بتوانند چند بسته را منتقل کنند

در سوئیچ‌های shared medium ، speed up = N است. وقتی اگر همه‌ی بسته‌های ورودی به یک پورت خروجی بخواهند بروند می‌شود منتقل کرد. مشکل. محدودیت حافظه (N.r)

① در از space Division های sw با speed up = K داشته باشیم اگر $K < N$ باشد ← احتمال cell loss خواهیم داشت چون اگر بیش از K تا سلول بیاد که بخواهند به یک پورت خاص بروند cell loss داریم.

معایب: دسترسی به حافظه داریم ← $T_{cell} > (K+1) T_{mem}$ اندازه سوئیچ محدود می‌شود. (K+1) write read

② احتمال cell loss به دلیل پر بودن (محدودیت) بافر خروجی وجود داره. این محدودیت در سوئیچ‌های output queueing نسبت به shared memory شدیدتره. سهم هر پورت خروجی در output queue ← $\frac{x}{n}$

Scanned with CamScanner

Space Division سوئیچ ← input buffering

در هر cell slot حداکثر ۱ بسته وارد می‌شود و یک بسته خارج می‌شود ← بین فقط ۲ دسترسی به حافظه داریم

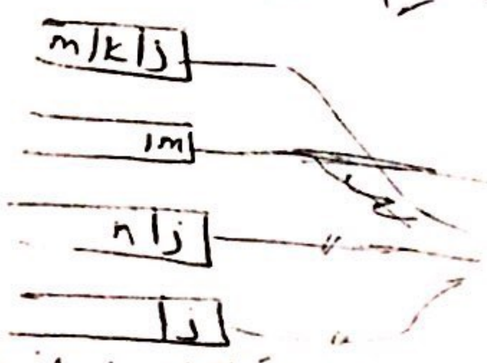
مزایا: محدودیت اندازه نداریم (بخاطر اینکه دسترسی به حافظه)

- سلول هایی که سر صف هستند باید انتخاب شوند تا به پورت خروجی بروند. برای این کار نیاز به یک الگوریتم scheduling داریم

یعنی زمان پردازش لازم داریم که این یک عیب برای این سوئیچ هاست

عیب ①: داشتن الگوریتم scheduling که باعث داشتن سربار زمان پردازشی می‌باشد که باید در زمان محدود cell slot اجرا شود.

زمانی که به دلیل محدودیت حافظه نتوانیم از shared memory، output queue استفاده کنیم باید سراغ این روش



این ۱ تا انتخاب نمی‌شود و سلولی از صف مربوط به آن ها انتخاب نمی‌شود

این ۲ بسته ای که انتخاب نشدند - خاطره رقابت بر سر پورت خروجی انتخاب شدند و منتظر time slot بعدی انتخاب شوند

- همیشه انتخاب سلول از سر صف است. صورتی که انتخاب نشود از سر صف

اگر از n تای سر صف تعدادی را انتخاب کنیم قطعاً پورت خروجی هم خالی می‌ماند.

همچنین مشکل HOL قرار می‌دهد باعث کاهش Throughput می‌شود.

سوئیچ های input buffer، احتمال PL ندارند چون آستانه گذاشتیم

② تأخیری که به سلول تحمیل می‌کنیم از زمانی که وارد سوئیچ می‌شود تا زمانی که خارج می‌شود فقط مربوط به بسته هایی که در دیگر از آن وارد شده به پورت خودش هستند که به دلیل انتخاب شدن سلول سر صف هم هست

پس تأخیرش از انتظار داریم.

بیرون: ① PL نداریم ② محدودیت اندازه هم نداریم

- اگر مدل ترافیک ورودی را بدانیم برای مثال اگر مدل پواسون باشد با مدل سازی می‌توان حجم بافر را تخمین زد به طوری که خیلی بی نهایت (بزرگ) تشکیل نشود. بنابراین حجم بافر با محاسبات و قابل پیش بینی است و با احتمال درست می‌توان PL را از بین برد.

← مشکل HOL باعث می‌شود Throughput به 58,6 کاهش پیدا کند. (برای ترافیک uniform) بینهایت switch

۱۰۰ باشد بیشتر از 58,6 نمی‌توان ازش استفاده کرد.

مشکل HOL را با تکنیک windowing می‌توان کاهش داد. به جای اینکه تنها سلول اول صف بررسی کنیم به اندازه w=2 یا 4 می‌توان دید انتخاب کرد.

w=1 ←	58,6	انتخاب از بین N سلول سر صف
w=2 ←	70	2N
w=3 ←	78	3N
w=4 ←		4N

اگر

اما هر چه پنجره بزرگتر کنیم مشکل انتخاب ما پیچیده‌تر

Scheduling پیچیده‌تر

هزینه اجرای الگوریتم انتخاب $O(N^3)$ هست. $N > 2N$ مشکل به صورت عادی افزایش می‌یابد.

پیچیدگی عادی افزایش پیدا می‌کند.

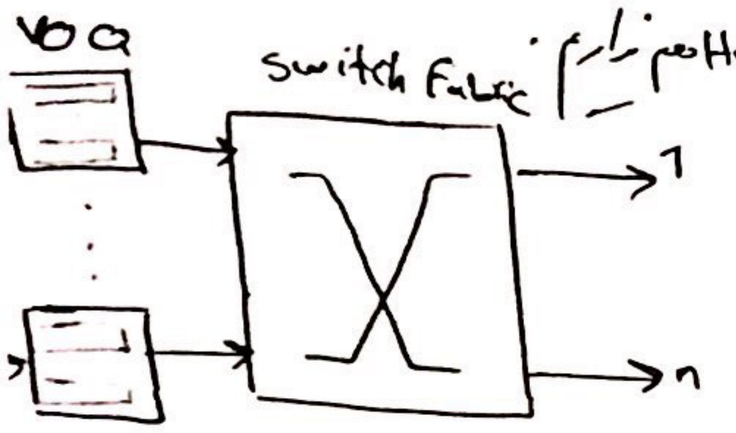
Scanned with CamScanner

Virtual output Queuing : مشابه input Buffer هستند. این بلوک ورودی براساس پورت فرودی، داخلش n صف منطقی ایجاد کرده

برای هر پورت فرودی یک بافر نداریم که این حافظه مثل حافظه ی shared memory ، صف های منطقی داره. در هر پورت ورودی

برای هر پورت فرودی یک صف منطقی داریم. این نوع سوئیچ ها بین بافر ورودی نداشته شده. در هر Cell slot یک خانه رقابتی

نوشتن داریم یعنی ۲ دسترسی به حافظه داریم که این یعنی محدودیت حافظه داریم. مسئله Hol هم داریم ①



سلول که وارد می شود بررسی می شود به کدام پورت خروجی باید برود سپس ی صف مربوطه نوشته

می نویسند. ② احتمال Cell Loss هم داریم

برای انتخاب: اگر بیفوام برای پورت خروجی 1 انتخاب کنیم، از صف های مربوط به پورت خروجی 1، n صف، از بین این ها انتخاب کرده

و فرودی باید یک ورودی match شود در هر ورودی هم فقط به یک خروجی می تواند میچ شود.

برای خروجی 1 ← N انتخاب داریم. برای خروجی ۲ ← (N-1) ← همینطور ← N! انتخاب داریم.

① از بین فضای جواب باید بهترین فضای جواب match انجام شود ← $O(N^3)$ برای به دست آوردن جواب بهینه

فقط زمان یک Cell slot که در هر ثانیه دارد صرف

② تأخیر متغیر و زیادتر از Out Put Queuing.

(P.36)

combined input and output Queuing:

هم بافر ورودی هم بافر خروجی وصل می کنیم با speed up دادن به سوئیچ

مزیت هر دو روش را داشته باشیم. پس هزینه پیاده سازی بالاست. هم بافر ورودی، هم

بافر خروجی، هم الگوریتم scheduling هم speed up داشتن سوئیچ باعث افزایش هزینه پیاده سازی

می شود. اما مزیت این روش اینکه فرایاری هر ۲ روش قابل حصول است.

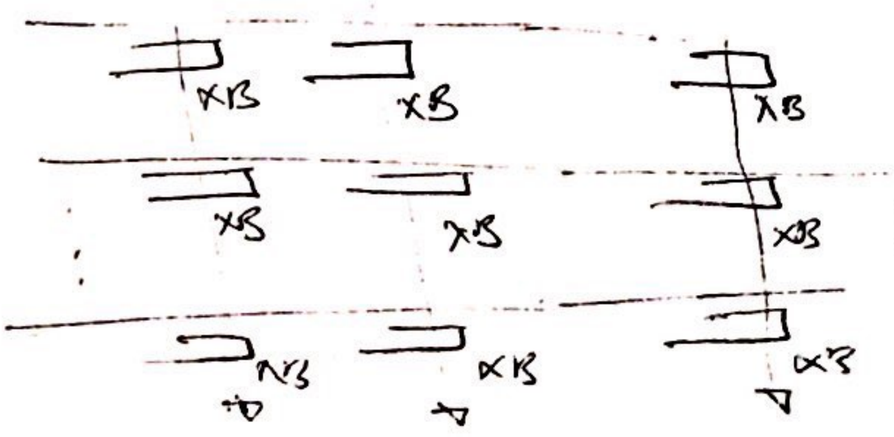
متری می شود: جای input Buffer ، VOQ داریم.

(P.38)

Cross point Queuing.

ما N^2 ، Cross point داریم پس N^2 حافظه داریم. به ازای هر پورت خروجی N تا حافظه داریم. رابطه بین این N تا باید انتخاب

کنیم در هر Cell slot کدام وصل به پورت خروجی برود. هزینه پیاده سازی بسیار بالا و از حافظه ها استفاده بهینه نمی شود.



سرعت و زمان بخش زیادی از این حافظه ها خالی می مانند.

برای مثال اگر از سه پورت ورودی به یک پورت خروجی حافظه buffer داشته باشیم

در این پورت به ورودی به سایر پورت های خروجی سهمی نداریم پس این

حافظه ها خالی می مانند (N^2 حافظه هم داریم)

Scanned with CamScanner

Parallel switch استفاده میکردن از چند

برای افزایش سرعت switching ← تکنیک موازی سازی

Banyan-Based
Clos

مراحل را موازی می‌کنیم به شکل Pipelining

سوئیچ‌های multistage می‌شوند

switching کامل است و سوئیچ‌ها switch

هر سوئیچ برای خودش می‌

Parallel Packet switching : K تا روتر موازی می‌کنیم برای اینکه سرعت سوئیچینگ را زیاد کنیم. با یک load متوازن را به صورت روا انجام داد.

متوازن بین این K تا انجام دار.

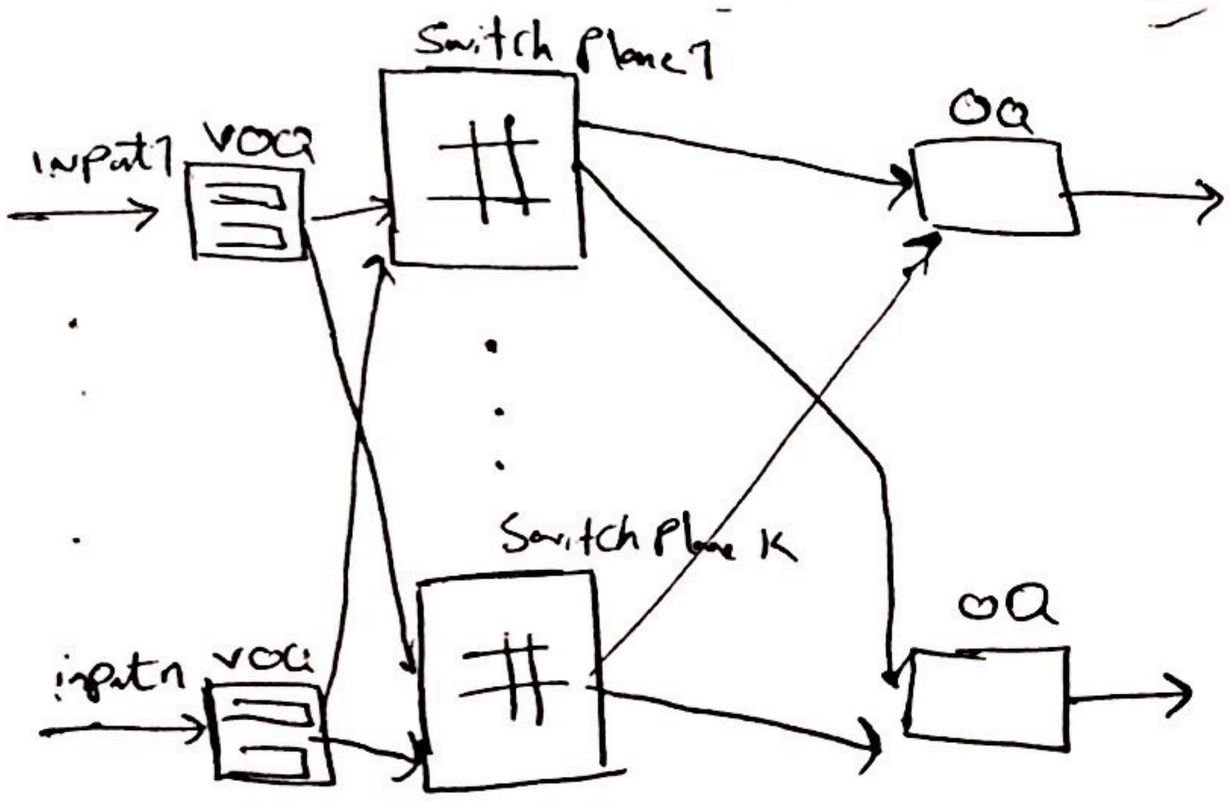
① تقسیم بار متوازن بین K سوئیچ دو راه نیاز به الگوریتم دارد.

نیازمندی ها

② نیاز به حفظ ترتیب بسته‌ها در خروجی داریم. این ترتیب را با Tag زمان یا Time snap زمان انجام میدن. و متناسب با میاد عمل ترتیب و در هنگام ورود خروجی بر اساس همین ترتیب مرتب می‌کنند

اگر یکی از سوئیچ‌ها خراب شود باید این سوئیچ را کنار بذاریم و با K-1 سوئیچ این عمل انجام شود.

چون سوئیچ‌ها Speed up بیشتر هستند و خروجی نیاز به بافر دارم سوئیچ هم بافر داریم که مشکل رقابت بر سر پورت خروجی رو حل کنیم



K تا سوئیچ در یک slot می‌تواند انجام شود.

برای توزیع بار نیاز به الگوریتم داریم در این که این الگوریتم پیچیده‌تر شکل الگوریتم دارد.

→ multiplane

← نسبت به حرامی، تحمل پذیر هستند

(P.41)

multi Stage

Banyan Based ← Pipelining

پیچیدگی هر SE را با تعداد X تعداد ورودی نشان می‌دهند.

اگر در هر مرحله یک بافر (register) بگذاریم که وقت داره کارشو انجام میده. نتیجه ثابت باشه برای Stak زیادی. ← Pipeline میشه

(در هر زمان Stage، یک خروجی داریم.) ← سرعت 3 برابر

ممکنه حرامی‌ها را تحمل نکنند

$n-1+n-1+1=2n-1$

switching Clos ← اگر n-1 پورت ورودی و خروجی اشغال باشد. (2(n-1)). حالا اگر یک SE دیگه بود،

internal Blocking نداشتیم:

اگر $m \geq 2n-1$ باشد مشکل internal Blocking حل می‌شود. ← سوئیچ‌های لایه میانی را افزایش دهیم

پیچیدگی یک عنصر switching با تعداد پورت ورودی x خروجی نشون می‌دهند

$\frac{N}{n}=k$ ← تعداد SE های ورودی.

m " SE های میانی

k " " خروجی

اگر مقایسه کنیم مرتبه پیچیدگی را با Clos که $O(n^2)$ بود:

$O(nxm)$

$O(k^2)$

$O(mxn)$

P 45 : اگر $m=2n-1$ ، k تا SE ورودی داشتیم، m تا میانی، k تا خروجی. می‌دانیم $\frac{N}{n}=k$. جایگذاری

mn ، k^2 ، nm ← پیچیدگی

$N_c = knm + mk^2 + kmn = 2Nm + m(\frac{N}{n})^2 \xrightarrow{m=2n-1}$ ... N^2 ... $n = \sqrt{N/2}$

Scanned with CamScanner

: Performance of Basic switches

ارزیابی کارآیی بعضی از سوئیچ های اساسی

input buffer ← مشکل اصلی: کاهش Throghput بخاطر HoL

output buffer ← مشکل PL (Cell Loss) رو داریم.

shared buffer ← switches

به صورت تئوری و تقریبی با روش های ریاضی تحلیل می کنیم میزان کارآیی این سوئیچ ها و از نظر Throghput , Cell Loss بررسی کنیم.

* درس ارزیابی کارآیی * خوانده های کاربرد مارکوف و تئوری صف *

یکی از بحث های مهم در ارزیابی کارآیی بحث مدل کردن ترافیک ورودی است.

زمانی که یک سیستم را مورد ارزیابی قرار میدهیم ما این سیستم یک سیستم تصادفی هست

سیستم یک فرایند درونی دارد. یک ورودی ای انجام میدهد. یک فرایند خروجی دارد.

یک فرایند تصادفی دارد. ترافیک ورودی مشخص نیست که از چه نوعی باشد.

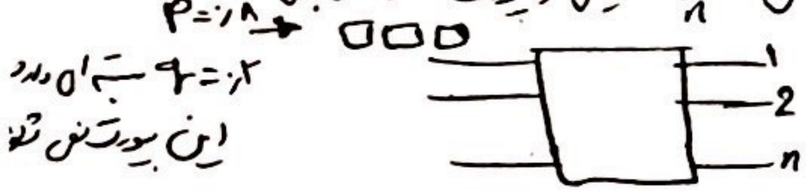
به عنوان مثال ترافیک ورودی از نوع پواسون، برنولی، ON-OFF، فرایندهای مارکوف و...

- هر کدام از این مدل ها روش مدل خودش رو داره.

معمولاً در سوئیچ ها برای ترافیک ۲ مدل در نظر می گیرند. ① Bernoulli ② on-off

Bernoulli: اگر n ورودی داریم (ورودی) را مستقل از هم در نظر می گیرند. مثلاً اگر Load ما ۰/۸ باشد در هر Cell slot با احتمال ۰/۸ یک سلول وارد می شود و ۰/۲ وارد نمی شود. و هر ورودی با یک احتمال مساوی $\frac{1}{n}$ یکی از پورت های خروجی را انتخاب می کند.

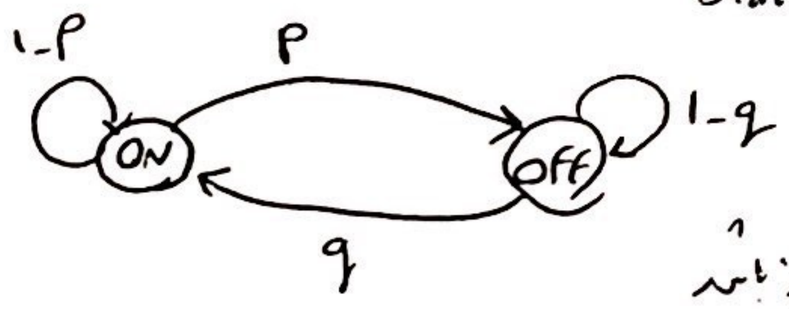
Random Traffic



on-off: عموماً برای ترافیک Bursty مورد استفاده قرار می گیرد.

برای یک تعداد slot time cell بیت هم سلول وارد می شود. ← state ON

(یک مدت زمان) سلول وارد نمی شود ← state off



این مدل به مدل واقعی نزدیک تره.

به مدت زمانی اشغاله، به مدت زمانی بستگی وارد نمی شود. اگر Load ما برابر ۰/۸ باشد میانگین بار ورودی ۰/۸ * ۰ می شود. ورودی ترافیک بیت هم هست

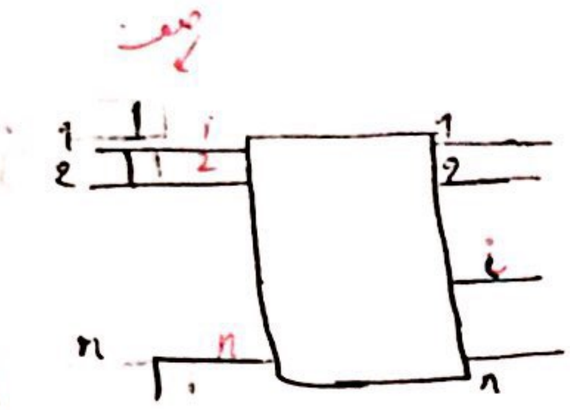
با p و q می شه میزان burstyness را تعیین کرد.

$$Load = \rho = \frac{q}{q+p-pq}$$

در تحلیل های بعدی برای ساختن ترافیک بر روی برای مدل سازی استفاده می کنیم

Scanned with CamScanner

بررسی سوییچ های با بافر ورودی: (P.46)



وقتی میخواهیم برای یک پورت خروجی مشخص مثل پورت i انتخاب انجام بدهیم. از سلول های سرصف میتوانیم انتخاب کنیم. اگر n پورت ورودی داشته باشیم. n تا سلول سرصف داریم.

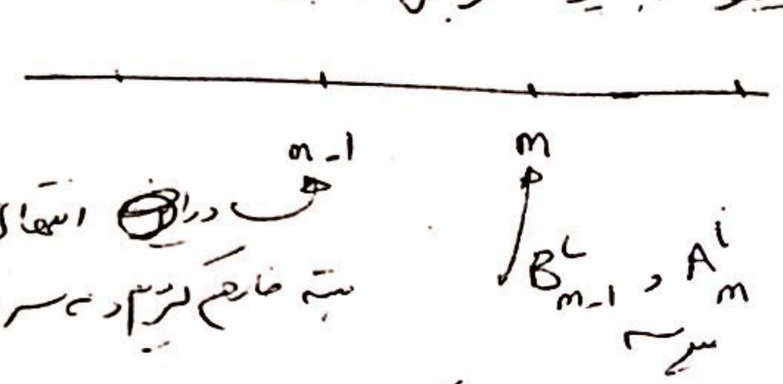
یک متغیر تصادفی به نام B^i.

B^i_m : در برش زمانی m ام چند سلول سرصف هستند که میخواهند به پورت خروجی i ام بروند.

فرض کنیم $B^i_{m-1} = 2$: 2 تا سلول سرصف بودند که میخواستند به پورت i بروند.

A^i_m : در برش زمانی m ام چند سلول به سرصف میرسند که میخواهند به پورت خروجی i بروند.

محور زمان به صورت Time slot تقسیم شده



در انتهای time slot m-1 ، m هم در انتهای time slot m-1 به سری بسته خارج کردیم و به سری بسته خارجی بودند

در انتهای time slot m ، m هم (تست کیو رو نشان کنیم) بقیه باقی هستند.

$$B^i_m = B^i_{m-1} + A^i_m - 1 \xrightarrow[\text{شود}]{\text{به شرطی که}} B^i_m = \max(0, B^i_{m-1} + A^i_m - 1)$$

یعنی بسته ای هم داشته به بسته خارج شده به سمت پورت i

اگر 1 باشد (1 نمی تونه بشه) اگر بیشتر باشه هنوز بسته میشه

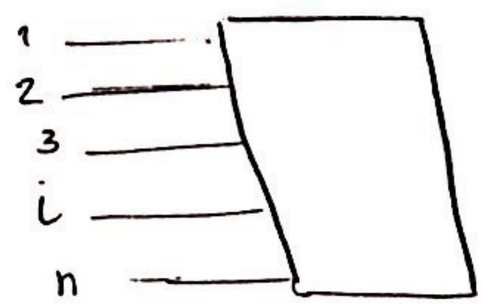
اگر چیزی باشد نمی شه 1، میشه 0

احتمال اینکه در time slot ، m ام ، k تا سلول وارد شوند برای پورت خروجی i (یعنی برسند سرصف بخواهند به پورت i بروند)

$$Pr[A^i_m = k] = \binom{F_{m-1}}{k} \underbrace{\left(\frac{1}{N}\right)^k}_{\text{k تا پورت i میخواهند بروند}} \underbrace{\left(1-\frac{1}{N}\right)^{F_{m-1}-k}}_{\text{به پورت i نمی خواهند بروند}} \qquad k = 0, 1, 2, \dots F_{m-1}$$

F_{m-1} : تعداد خانه های خالی شده سرصف ها (که k تا از سلول های وارد شده میخواهند به پورت i بروند) هر سلول به احتمال $1/N$ میخواهد برود به پورت i

$$F_{m-1} \simeq N - \sum_{i=1}^{N} B^i_{m-1}$$



و

$$F_{m-1} = \sum_{i=1}^{N} A^i_m$$

مجموعه سلول هایی که میرسند به سرصف برابر مجموعه خانه های خالی سرصف هاست.

اگر تعداد پورت ها را میبریم به سمت $N \to \infty$ ، توزیع A^i_m یک توزیع پواسون میشود. با نرخ متوسط ρ^i_m

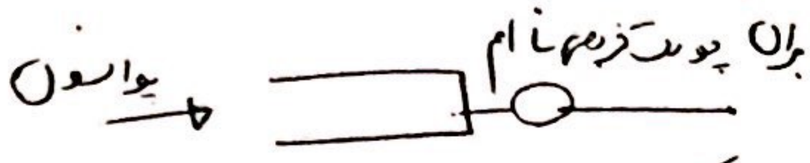
$$\rho^i_m = \frac{F_{m-1}}{N}$$

$\rho_0 = \frac{\bar{F}}{N}$

$\rho_0 = \frac{\bar{F}}{N}$ ، اگر در زمان متوسط بگیریم ، steady state

Scanned with CamScanner

یک صف داریم که برای یک پورت خاص مثل پورت خروجی ام ، اندازه سفارش ها ثابت، زمان برداشتن ثابت.



یک فرایند داریم و دوباره توزیع پواسون داره.

M/D/1 ← صف

زمان سرویس Deterministic

فرمول مارکوف

M/D/1 رو داریم توی کتاب هست.

☆ برای هر پورت خروجی ای :☆

(P.50)

فرمول ارزیابی کارآیی تعیین صف های

$\overline{B^i}$ متوسط تعداد سفارش ها در صف ←

$$\overline{B^i} = \frac{\rho_0^2}{2(1-\rho_0)}$$

حداکثر کارآیی انقدر برای N های بزرگ $\rho_0 = 2-\sqrt{2} = 0.586 \rightarrow 58.6\%$

باشه اردهال

$$\bar{F} = N - \sum_{i=1}^{N} \overline{B_G^i}$$

$$\Rightarrow \overline{B^i} = 1-\rho_0 = 1-\frac{\bar{F}}{N}$$

$$\overline{B^i} = \frac{\rho_0^2}{2(1-\rho_0)}$$

$\Rightarrow \rho_0 = 2-\sqrt{2}$

این نتایج بر مبنای این بوده که همه ما فرض کردیم در ورودی مدل پواسون داره و هر

طبق جدول ، $N = 9 \text{ و } 11$ به هم نزدیک ۵۸٪ میرسونه.

اگر از سوئیچ با بافر ورودی استفاده نکنیم نباید head ورودی از ۵۸٫۶٪ بیشتر داشته باشیم اگر head افزایش پیدا کند

صف بی نهایت تشکیل می شود و اینجا عملا تاخیر بی نهایت

P.52 ، اگر N را از ۶ زیاد کنیم ، N ، اگر یک باشه دیگه سوئیچی نداریم ، throughput = 1 است... از بعد از ۸ ، throughput

می شه 58 درصد.

P.53 ، براساس تئوری صف های هندسی (توزیع هندسی دارد سیستم ها (شمارشی)). طبق مزول P.K :

$\bar{S}$: متوسط service time P : لود

$$\Rightarrow \bar{W} = \frac{P\bar{S}(\bar{S}-1)}{2(1-P\bar{S})} + \bar{S} - 1$$: متوسط تاخیر

براساس این رابطه ، نمودار ص 54 را داریم. محور افقی P ، است ، محور عمودی تاخیر

وقتی به لود نزدیک 0.58 می رسیم ، delay می ره به سمت ∞ .

⇐ اگر از سوئیچ input buffering استفاده می کنیم که مشکل HoL داره ، لود ورودی باید کمتر از 0.58 باشه وگرنه : صف بی نهایت و تاخیر بی نهایت

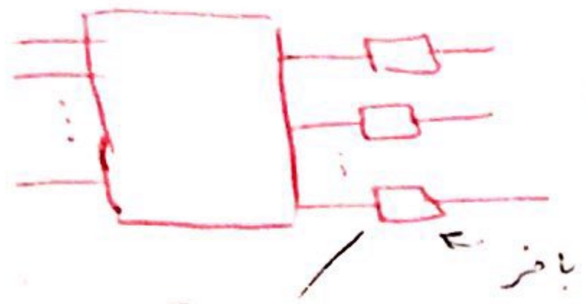
Scanned with CamScanner

بررسی سوییچ های output buffer

تا از ورودی که به پورت مقصد خروجی مثل m هست برسیم

تکلیف ما در ورودی ها به پورت مشخص خروجی مثل پورت j اتمام میدهیم

برای تمام پورت های خروجی قابل استفاده



speed up : N

ظرفیت بافر خروجی ثابته ← مثلاً b : (b تا سلول میتونه داشته باشه)

از این پورت های ورودی به سرعت سلول به این بافر در یک Time slot وارد میشوند ← مثلاً K تا و احتمال اینکه K تا سلول بیاد A_K

یا تا سلول قبلاً در بافر بوده و در هر تایم اسلات یک یکت میتونه خارج شود.

یکت کامل زمانی رفع میشه که بافر پر شود. میخوایم این احتمال را حساب کنیم

اگر سلول برابر P باشد. و ترافیک ورودی بزرگتر باشد در سلولی وارد شده با احتمال P/N این پورت خروجی را انتخاب کند

پس هر پورت ورودی با احتمال P/N یک سلول میاد در میخواسته، پس احتمال خروجی نداشتن را $1-P/N$ میخواسته، پورت های ورودی دیگر

$$a_K \triangleq Pr[A=K] = \binom{N}{K}\left(\frac{P}{N}\right)^K\left(1-\frac{P}{N}\right)^{N-K} \qquad K=0,1,2,\dots N$$

Arrival

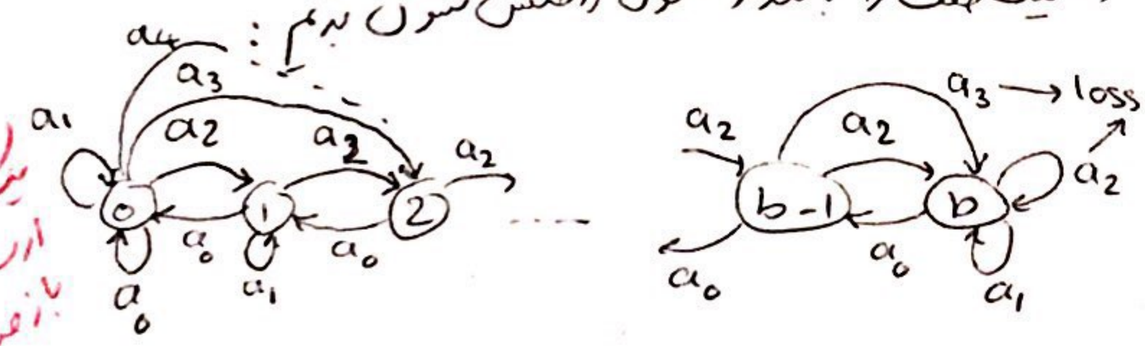
اگر $N\to\infty$: ترافیک ورودی به پواسون میل میکند.

مثلاً N از ۱۰۰ تا بیشتر شبیه واقعی میشود.

$$a_K \triangleq Pr[A=K] = \frac{P^K e^{-P}}{K!}$$

احتمال اومدن K تا ورودی

اگر فرض کنیم هر صف خروجی، b تا سلول را می تونه ذخیره کند. وضعیت صف را با تعداد سلول داخلش نشون بدیم:



یک سلول میاد، ارسالش می کنیم یا صف خالیه

احتمال بودن در هر state را با q_i نشون میدهیم.

احتمال جابجایی بین state ها (P_{ij}) را می توانیم به دست آوریم:

$$P_{ij} = \begin{cases} a_0 + a_1 & i=0,\ j=0 \\ \vdots & \end{cases} \qquad (2.12)$$

P.56) Q_m می شه وضعیت بافر خروجی که حداکثر b است، حداقل 0.

می شه وضعیت مثبت + سلول هایی که وارد شده منهای 1

P.57) در ذخیره های مارکف (چون ورودی پواسون بود)، یک balance equation داریم که احتمال وارد شدن به state با احتمال خروج از state با هم برابره. ⇐ معادله ی q ها را می نویسیم. جمع همه ی احتمالات هم یک است ⇐ q_0 به دست میاد.

P.58) حالا P_0 یا throughput می شه حالتی که صف اشغال است:

$P_0 = 1 - \text{احتمال خالی بودن صف} = 1 - q_0 a_0$

⇐ درصد خروجی: $\frac{P_0}{P}$ ← لود

$$Pr[\text{cell loss}] = 1 - \frac{P_0}{P} \quad \Leftarrow$$

P.59) حالا می آییم نمودار می کشیم. اگر $P=0.8$، احتمال cell loss به سایز بافر. با کاهش بافر، cell loss افزایش.

P.60) اگر لود را زیاد کنیم، این احتمال افزایش پیدا می کنه. $1-\frac{P_0}{P}$. اگر بافر صفر باشه، cell loss، صد می شه. (یه درصد کمی عبور می کنه)

P.61) اگر فرض کنیم N عدد بزرگیه (بزرگتر از 32 پورت). بافر اگه یه مقداری باشه، هر چی لود افزایش، cell loss افزایش.

Scanned with CamScanner

- در مورد تاخیر در بحث output buffering :

وقتی buffer size محدوده ، به تعداد سلول (b) می‌شه ، اونایی که وارد می‌شه به تاخیری دارد . حداکثر تاخیرشون اینه که با صف پر (به اندازه‌ی b تا time slot) مواجه کنند . حداقل هم صف خالی . متوسط می‌شه $\frac{b}{2}$ time slot .

(p.62) اگه بود را از صفر تا 100 افزایش دهیم، اگر b=1 باشه ، تاخیر حدوداً $\frac{1}{2}$ time slot می‌شه و...

وقتی صف خروجی اندازش محدوده ← سلول هایی که موفق به ورود به صف می‌شوند، تاخیر محدود دارن . متوسط می‌شه $\frac{b}{2}$ time slot .

shared memory switches

ما یک حافظه مشترک داریم که توش به ازای هر خروجی ، یک صف منطقی تشکیل می‌شه . ⇐ اگه بخواهیم ببینیم per port چه قدر حافظه داریم ، اگه حجم حافظه M باشه ، $b=\frac{M}{N}$ ← حافظه به ازای هر پورت . ولی چون حافظه مشترکه . اگه یک صف طولانی بشه ، چون shared است ، احتمال cell loss کاهش پیدا می‌کنه .

← هر چی اندازه‌ی سوییچ را زیاد کنیم ، چون میزان به اشتراک گذاری حافظه بیشتر صورت می‌گیرد ، احتمال cell loss کاهش می‌یابد .

(p.64) برای بود 90 درصد هست . مثلاً اگه اندازه سوییچ 16 باشه ، با حافظه‌ی 15 خونه‌ی حافظه به ازای هر پورت می‌شه به احتمال 10^{-10} رسید . اگه اندازه switch ، 256 باشه ، با مثلاً 4 خونه حافظه به ازای هر پورت ، به کمتر از 10^{-10} می‌رسیم

تاخیر بیشتره ، cell loss کمتره . (نسبت به output buffering)

جلسه 13

اسلاید shared memory switches

حافظه مشترک داره که تمام سلول‌ها روش نوشته می‌شه .

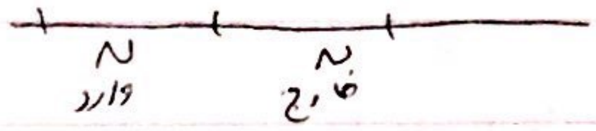
از انواع time division هست .

اگر زمان دریافت یا ارسال یک سلول = cell slot

$$\Rightarrow \text{cell slot} = \frac{\text{cell length}}{\text{link speed}}$$

در مدت زمان یک cell slot ، اگر switch ، N پورت ورودی داشته باشه ، N سلول وارد می‌شه .

" N سلول از پورت های خروجی باید خارج بشه در cell slot بعدی .



⇐ در زمان یک cell slot ، 2 تا access به حافظه می‌خواهیم داشت .

⇐ زمان خوندن نوشتن : $2N \times t_{mem}$ که cell slot باید بزرگتر مساوی این باشه .

$$\Rightarrow \underset{\frac{L}{R}}{t_{cell}} \geq 2N.t_{mem} \Rightarrow N.r \leq \underset{\text{عدد ثابت}}{L/2t_{mem}}$$

⇐ ظرفیت محدود می‌شه . ← مشکل

maxnote

Scanned with CamScanner

P.3) مثال : اندازه سلول : 53 بایت سرعت خط : 149.76 Mbps

$$t_{cell} = \frac{53 \times 8}{149.76 \times 10^6} = \frac{423}{149.76 \times 10^6} = 2.83\ \mu s$$

if $t_{mem} = 10ns$ $t_{cell} > 2N . t_{mem} \Rightarrow N \leq \frac{t_{cell}}{2 t_{mem}} \Rightarrow N \leq 141$

اگر 141 برای ما قابل قبول باشد، این switch، گزینه‌ی مناسب است.

P.4) پیاده سازی switch های shared memory :

N پورت ورودی - حافظه مشترک - N پورت خروجی.

پورت ها را با علامت گذاری مشخص کرده (هاشور)

باید در حافظه ی مشترک، به ازای هر پورت خروجی، یک صف منطقی باید ایجاد کنیم.

=> احتیاج به یک مدار (بخش) کنترل کننده تا صف ی منطقی در حافظه مشترک را ایجاد کند.

ایده ها : ① استفاده از linked list : (P.5) : بخش کنترلی در شکل بخش پایینی.

← به ازای هر پورت خروجی داخل حافظه مشترک، یک صف با linked list ها تولید میکنیم.

هر سلولی که وارد می‌شه به مقصد پورت i، می‌آییم اون سلول را در انتهای صف i ام می‌نویسیم.

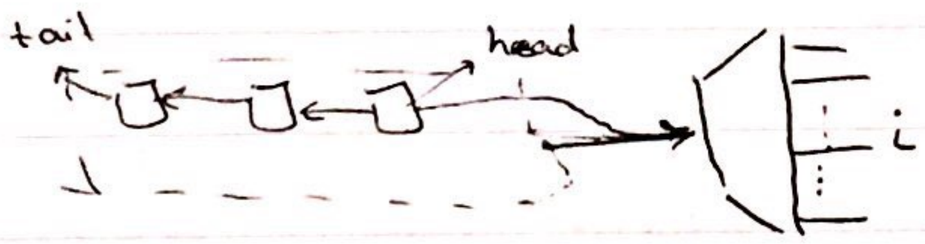
(انتهای صف، یک خانه‌ی آزاد است که اون جا می‌نویسیم)

وقتی می‌خواهیم یک سلول را خارج کنیم، از ابتدای صف می‌خوانیم خارج می‌کنیم.

=> 2 تا ptr نیاز داریم. ← head ptr , tail ptr

اگر N خروجی داریم، به ازای هر خروجی، این 2 تا را داریم.

پس وقتی سلولی را می‌خواهیم بخوانیم، با head ptr حافظه را آدرس می‌دهیم، سلول را می‌خوانیم خارج می‌کنیم.



وقتی اینو می‌خوانیم، next ptr، اشاره می‌کنه به خونه‌ی بعدی که با هاش head ptr را update می‌کنیم.

برای نوشتن، با tail ptr حافظه را آدرس دهی می‌کنیم براساس اینکه در کدوم پورت خروجی نوشته می‌شه. (برای هر سلول، پورت خروجی مشخصه) ← سلول وارد شده را در tail ptr می‌نویسیم. بعدش باید صف update بشه به یک خانه‌ی آزاد. (IAF ← صف خانه‌ی آزاد حافظه) => همزمان نوشتن، آدرس یک خونه آزاد هم از IAF گرفتیم و همزمان آدرس و data را می‌نویسیم. این آدرس آزاد، جایگزین tail ptr می‌شه.

Scanned with CamScanner

P.6) حافظهٔ مشترک (cell buffer mem.) را داریم که 2 تا بخش دارد : 1) cell data

2) next ptr ها (خونه بعدی در صف)

الان head ptr = 0 داره به اون خونه اشاره می‌کنه (cell(1)) ، next ptr به خونه‌ی بعدی و... خانه‌ی آخر به یک خونه‌ی آزاد اشاره می‌کنه که tail ptr = 0 به اون اشاره می‌کنه. اگه سلول جدید بیاد، این جا نوشته می‌شه. بعد از صف آدرس های آزاد (Idle Addr. FIFO) یکی می‌گیریم مثلاً آدرس خونه‌ی آخر. هم در next ptr ، آدرس اینو می‌نویسیم هم tail ptr رو update می‌کنیم به این.

معمولاً از حافظه های Static RAM استفاده می‌شه که Access time کم داشته باشند و معمولاً در core نمی‌توانند استفاده شوند چون ظرفیت switching محدوده. (یا تعداد پورت ها کمه یا همپای باند هر پورت) N.P ثابت

P.7) مراحل . صف اولیه را داریم:

(b) برای نوشتن ، اول به tail ptr دسترسی پیدا می‌کنیم ، سلول جدید را توی می‌نویسیم . از صف آدرس های آزاد ، یک آدرس می‌گیریم ، به عنوان next ptr و tail ptr .

← همه با یکبار دسترسی به حافظه

(c) خوندن از سر صف . از طریق head ptr ، آدرس سر صف را می‌گیریم و می‌خونیم. data را از پورت خروجی خارج می‌کنیم ، همزمان ، آدرسش را آزاد می‌کنیم در صف آدرس های آزاد قرار می‌دهیم. اون سلول که خوندیم ، یه next ptr داشت ، اون الان می‌شه head ptr

← یکبار دسترسی به حافظه

$\Rightarrow 2N \cdot t_{mem} < t_{cell}$

⇐ به ازای خوندن و نوشتن ، 2 بار دسترسی به حافظه ✗

- در این روش ، همیشه انتهای صف ، یک حافظه‌ی آزاد (idle) داریم. اگر N تا پورت باشه ، یعنی N خونه‌ی حافظه بلااستفاده. اگه Qos هم باشه که به ازای هر پورت خروجی، چند صف داشته باشیم، تعداد این خونه‌ها زیاد می‌شه. روش دیگه اینه که tail ptr به خونه آزاد اشاره نکنه. به آخرین خونه‌ی پر اشاره کنه. این طوری وقتی می‌خواهیم سلول اضافه کنیم ، اول باید آدرس یک خونه آزاد را بگیریم ، سلول را توش بنویسیم ← یک دسترسی به حافظه بعد ، ptr = خونه قبلی را باید update کنیم ⇐ 2 دسترسی به حافظه . (شکل سمت راستی)

برای خوندن ، مشابه قبلی .

⇐ به ازای خوندن و نوشتن در هر cell slot به ازای هر پورت ورودی و خروجی ، 3 بار دسترسی به حافظه داریم.

← محدودیت شدیدتر : $3 \times N \cdot t_{mem} \leq t_{cell}$ به جایش هدر رفت حافظه نداریم.

P.8) مثال از switch ، shared memory ، 4 × 4 . سلول هایی که از پورت ورودی وارد می‌شه، براساس شماره پورت خروجی، tail ptr را انتخاب می‌کنیم ، انتهای صف می‌نویسیم ، برای خوندن و خارج کردن از پورت های خروجی هم از head ptr استفاده می‌کنیم.

maxnote

Scanned with CamScanner

- یکی از اشکالات switch shared mem. ها که با linked list درست می‌شه، اینه که این switch داره به طور مداوم کار می‌کنه. احتمال داره به دلیل نویز در محیط، یکی از ptr ها دچار مشکل شه، اشاره کنه به یک حافظه‌ی دیگه بجای خونه‌ی بعدی. (۱ بیت خطا) => اگه یک ptr اشتباه باشه، از اونجا به بعد، سلول ها را اشتباه می‌خونیم و خارج می‌کنیم. قابل تشخیص نیست این خطا رخ داده یا خیلی دیر متوجه می‌شویم.

=> یک خطا، باعث انتشار یا اشتباه switch کردن تعداد زیادی سلول می‌شه.

چه کنیم؟

در مسائلی که انتظار داریم که اگر یک خطا به موقع تشخیص داده نشه، باعث انتشار خطا می‌شه، چه طوری کنترل کنیم؟

اگه یه ptr خراب شه، فقط یک سلول اشتباه بشه، سلول بعدی اشتباه نزه.

یک راه: ptr دو طرفه. اگه این به اون اشاره کرده، اون به جای دیگه، نشان دهنده‌ی خطاست. ولی چه طوری recover کنیم؟!

روش پیشنهادی: ptr ها را جداگانه در صف های FIFO ذخیره کنیم. (P.9) => در حافظه دیگه نیازی نیست next ptr ها را ذخیره کنیم. میاریمشون در یک حافظه‌ی FIFO. این حافظه چون محکمه یکی مخه ترافیک بره به سمت یک پورت خروجی، باید اندازه‌ی بزرگی داشته باشه و چون تفکیک شده هستند، این جا استفاده بهینه نداریم. ولی چون طول ptr ها کوتاهه، می‌ارزد که جلوی خطا را اینطوری بگیریم.

اگه خراب هم بشه، ptr به بعدی درسته.

=> به ازای هر پورت خروجی، یک حافظه‌ی FIFO می‌گذاریم، آدرس سلول های حافظه را در این ذخیره می‌کنیم.

(در روش قبلی، فقط پوینترهای ابتدا و انتها را داشتیم، حالا به تمام سلول ها، ptr ها را نگه می‌داریم. اولین ptr، سر صف را نشون می‌ده، آخری انتهای صف را.)

وقتی بخواهیم یک سلول بخونیم، سر صف را می‌خوانیم، حافظه را با هاش آدرس می‌دهیم، اون سلول را می‌خوانیم، از پورت مورد نظر خارج می‌کنیم. همین آدرس (که از FIFO) خوندیم را به آدرس های آزاد، اضافه می‌کنیم.

برای نوشتن، یک آدرس آزاد را می‌گیریم، حافظه را آدرس می‌دهیم، سلول وارد شده را در اون خونه می‌نویسیم. این سلول مثلاً مربوط به پورت خروجی i ام است. این آدرس را انتهای صف i ام قرار می‌دهیم.

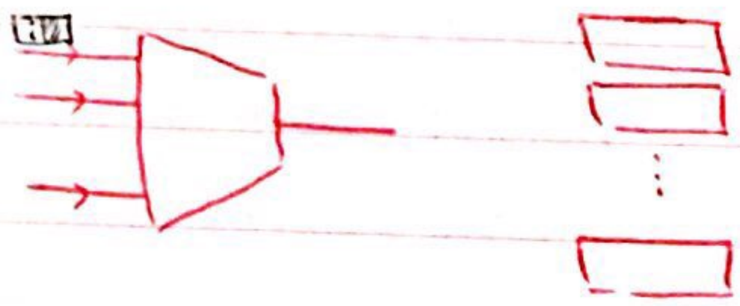
=> اگه یک ptr را خوندیم اشتباه شد، فقط یک سلول را اشتباه فرستادیم. بعدی درسته.

روش دیگر پیاده سازی switches shared memory

2) استفاده از حافظه های CAM

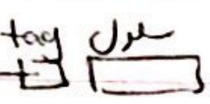
(P.10) سلول هایی که وارد switch می‌شوند را قراره توی حافظه، N صف منطقی درست کنیم به ازای هر خروجی یک صف. سلول ها داره از پورت های 1 تا N وارد می‌شه. اینا وارد می‌دیم به یک multiplexer، در حافظه می‌خواهیم بنویسیم:

Scanned with CamScanner



حالا مثلاً یک سلول از پورت 6 وارد شده به مقصد پورت i :

می‌آییم اینو در حافظه می‌نویسیم. به این سلول یک tag اضافه می‌کنیم.

به هر سلول، یک tag اضافه می‌کنیم  برای پورت i اُم است و شمارهٔ ترتیبش مثلاً s است.

↳ i , s

یعنی به سلول هایی که میاد، یک شماره ترتیب می‌زنیم. (شماره ترتیب ورود به ازای هر پورت خروجی)

=> هر سلول که وارد می‌شه، خروجی اش را نگاه می‌کنیم مثلاً خروجی، پورت i اُم است، یک شماره ترتیب هم پشش اضافه می‌کنیم همراه شماره ترتیب در حافظه می‌نویسیم.

حالا می‌خواهیم یک سلول را از پورت j ، خارج کنیم :

قبلاً مثلاً شماره سلول t-1 را خارج کرده بودیم. الان سلول t را باید خارج کنیم.

=> در حافظه می‌گردیم دنبال tag ی که برای پورت خروجی j هست و t را می‌خواهیم. => یک شماره ترتیب خواندن داریم : (j , t)

شماره ترتیب ورودی — پورت خروجی

=> tag شد : (i , s) . این tag به سلول اضافه می‌شه و در سلول نوشته می‌شه.

وقتی می‌خواهیم یک سلول را بخوانیم، دنبال سلولی که از پورت j می‌خواد خارج بشه می‌گردیم و t : شماره ترتیبی که باید بخوانیم و خارج کنیم : (j , t)

شماره ترتیب خواندن — شماره پورت خروجی

← وقتی اینو پیدا کردیم، می‌خوانیم و از پورت j خارج می‌کنیم.

هر یک سلول که وارد می‌شه، s یکی اضافه می‌شه. هر یکی که می‌خوانیم، t یکی اضافه می‌شه.
چون این دنبال گشتن طول می‌کشه، از CAM استفاده می‌کنیم.

(p.10 حافظه را 2 بخش می‌کنیم : 1) قسمت RAM که سلول ها را می‌نویسیم.
2) قسمت CAM که tag ها را می‌نویسیم

سلول ها که میان، tagشون را در CAM می‌نویسیم، خود سلول را در RAM و می‌روی آدرس یکسان.

maxnote

Scanned with CamScanner

وقتی می‌خواهیم بخونیم، سرچ می‌کنیم لون tag را پیدا می‌کنیم (j, t) ، وقتی پیدا کردیم، آدرس را به دست آوردیم، با اون آدرس، حافظه‌ی RAM را آدرس‌دهی می‌کنیم ، data را می‌خوانیم و ارسال می‌کنیم.

برای نوشتن ، یک خونه‌ی آزاد باید پیدا کنیم ← یک دسترسی به حافظه . بعد سلول و tag را همزمان در RAM و CAM می‌نویسیم

← شد 2 بار دسترسی به حافظه

برای خواندن ، یکبار در حافظه‌ی CAM سرچ می‌کنیم ، tag را پیدا می‌کنیم ، یکبار هم می‌ریم از RAM می‌خوانیم

← 2 بار دسترسی به حافظه .

⇐ مجموعاً 4 بار دسترسی به حافظه .

به ازای هر پورت خروجی ، یک write Seq. num. و یک read Seq. num. ← 2 تا شماره ترتیب داریم .

موقع نوشتن ، سلول وارده ، پورت خروجی را داره (با IP addr. lookup) . موقع خواندن هم باید به تمام پورت‌ها سرویس دهیم . (از کدوم پورت می‌خواهیم خارج کنیم) ← در یک time slot باید از همه‌ی پورت‌ها سلول خارج کنیم .

موقع نوشتن هم تمام ورودی‌هایی که سلول دارن ، label ـِ پورت خروجی دارن ، Seq. number هم داریم ، tag را درست می‌کنیم در حافظه می‌نویسیم .

P.11) برای نوشتن یک write sequence number (از WS[i] ، پورت خروجی i) ، اینو می‌گیریم ، اسمش را می‌گذاریم S . i هم شماره پورت خروجی است ، pid می‌شه tag .

در حافظه‌ی CAM یک خونه‌ی آزاد را پیدا می‌کنیم . (برای خانه‌ها یک بیت می‌گذاریم که empty هستند یا خیر) پس یکبار به CAM مراجعه کنیم ، یک خونه‌ی آزاد را پیدا کردیم . پیداش که کردیم ، می‌آییم در RAM در حافظه‌ی آزاد ، سلول را می‌نویسیم و tag‌ش را در حافظه‌ی CAM ـِ متناظرش می‌نویسیم در همون آدرس .

بعد seq. num را یکی اضافه می‌کنیم . (اگه S بوده ، می‌شه S+1) ← آماده می‌شویم برای write ـِ بعدی . در هر لحظه ، N تا سلول خارج می‌شه در N پورت موازی . ولی سرعت حافظه بالاتره ؛ یعنی ما یک سلول را می‌خوانیم ، از پورت 1 می‌خواهیم خارج کنیم ، سلول 2 را می‌خوانیم از حافظه برای پورت دوم و ... . چون سرعت حافظه ، بیشتر از سرعت لینک‌هاست ، می‌تونیم این کار را انجام دهیم .

سرعت حافظه در shared mem. switches ، 2N برابر سرعت لینک‌هاست . اگه برای switch هایی ، این کمه ، باید 4 برابر باشه . (مثلاً برای نوشتن 2 بار دسترسی به حافظه داره . یکبار خونه‌ی آزاد را پیدا می‌کنیم ، یکبار هم می‌نویسیم . برای خوندن هم یکبار آدرس را پیدا می‌کنیم ، یکبار می‌خوانیم .)

P.12) برای خواندن ، می‌خواهیم از پورت j خارج کنیم. seq. num.j ـِ j می‌خوینم. اسمش را می‌گذاریم t . یک tag داریم : (j, t) . می‌گردیم در CAM پیدا می‌کنیم آدرسش را . از اون آدرس ، سلول را می‌خوانیم و ارسال می‌کنیم ، در CAM هم اون را tag می‌کنیم که آزاد شده .

seq. num ـِ خوندن هم t بود ، می‌کنیم t+1 برای خوندن بعدی آماده بشه .

Scanned with CamScanner

توضیح بیشتر: ما یک حافظه‌ی RAM داریم که مشترکه. در روش CAM، یک CAM هم برای tag ها داریم.
اگر اندازه‌ی حافظه، به اندازه‌ی M سلول باشه، آدرس ها m بیتی می‌شه. ($2^m = M$)
حالا اگه به ازای یک پورت خروجی، صف می‌تونه نصف حافظه شه (حداکثر اندازه‌ی صف به ازای یک پورت خروجی)، seq. number اگه می‌خواست کل حافظه شه، m بیت می‌خواست. الان باید m-1 بیت باشه. <= تعداد بیت های seq. num بستگی داره به ازای یک پورت خروجی حداکثر چه صفی تشکیل می‌شه.

P.13: روش لیست پیوندی و CAM را مقایسه کرده.
فرض کنیم در حافظه مشترک می‌خواهیم 256 سلول را ذخیره کنیم. اندازه‌ی RAM می‌شه: $M = 256$ سلول
اندازه‌ی سلول هم: 53 byte ← 424 bit
تعداد پورت ها: $N = 16$

یک فضای ذخیره سازی سلول ها را داریم: $256 \times 424 = 108.544$ ← در هر 2 مشترک.

در روش linked list، به ازای هر خانه، یک next ptr داریم. 256 تا خونه داریم، پس $m = 8$ (2^8)
(یعنی آدرس ها 8 بیتی هستند) <= 2048 بیت برای نگهداری next ptr ها در حافظه‌ی اصلی
در روش CAM، یک tag داریم که شماره پورت را نشون می‌ده. چون 16 تا پورت داریم، 4 بیتی است.
فرض اینه هر صف خروجی می‌تونه نصف اندازه‌ی حافظه باشه، پس seq. num. ها 7 بیتی است.
<= $11 \times 256 = 2816$ ← حافظه‌ی مورد نیاز برای نگهداری tag ها در CAM.

در linked list به ازای هر پورت خروجی، 2 تا ptr داریم. یک head ptr به tail ptr. 16 تا پورت داریم، هر پوینتر، 8 بیتی: $2 \times 16 \times 8 = 256$
در روش CAM، به ازای هر پورت خروجی، 2 تا seq num داریم یکی write یکی read، 16 تا پورت، هر seq num هم 7 بیتی است: $2 \times 16 \times 7 = 224$

IAF، اول که switch شروع به کار می‌کنه، کل switch، خالیه، اندازه این IAF، می‌شه 256 تا خونه، هر ptr 8 بیتی: $256 \times 8 = 2048$ بیت برای نگهداری آدرس های آزاد حافظه.
در روش CAM، با یک بیت، پر یا خالی بودن را مشخص می‌کردیم: $256 \times 1 = 256$

جمعاً: کدوم روش بهتره؟ اگر فقط تعداد بیت حافظه را درنظر بگیریم، CAM بهتره. از نظر حافظه، زیر یک درصد با هم متفاوته. اما پارامتر های دیگه هم وجود دارد. مثلاً در linked list در هر cell slot به ازای هر پورت

خروجی ، 2 بار دسترسی به حافظه داشتیم . در روش های CAM ، 4 بار دسترسی داریم. ⇐

محدودیت در LL : $N.r \leq K$ در CAM : $N.r \leq \frac{K}{2}$

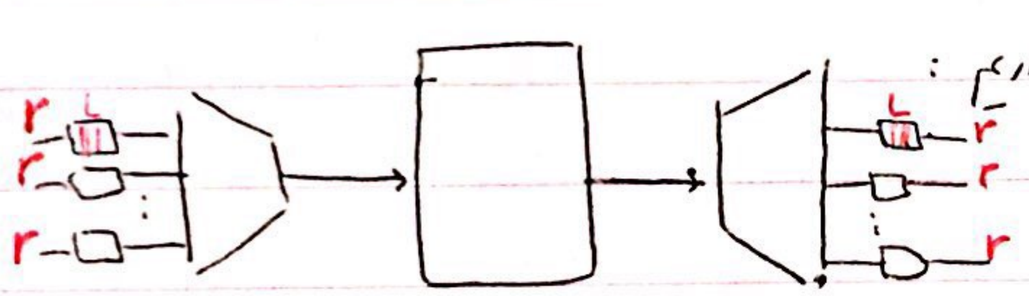
در CAM ، حافظه ها گرون تر هستند و مصرف انرژی بالایی دارند .

اگه محدودیت سرعت نداریم ، هزینه هم مهم نیست ، CAM بهتره . اگه محدودیت سرعت داریم ، قیمت مهم نیست ، می رویم سراغ linked list ← بحث حفظا داشت که رفتیم سراغ اینکه ptr ها را در FIFO نگه داریم ، حافظه مصرفی افزایش پیدا می کرد .

⇐ باید معیار ها را مشخص کنیم ← trade off کنیم ← بهترین روش را انتخاب کنیم .

(به صورت سریال و شیفت هم می خوانیم . برای پورت 1 ، پورت 2 ، ... اما مجموع خوندن ها برای N تا پورت ، از یک cell slot کمتره . تقسیم زمانی می کنیم . مجموع N تا خوندن از سرعت 1 پورت بیشتره) ← ارسال همه در یک time slot انجام می شه .

خلاصه : یک حافظه مشترک ، یک MUX ، یک DEMUX داریم :



فرض کنیم در خروجی ، باخر داریم که فقط یک سلول را می تونه ذخیره کنه . در ورودی هم همینطور .

سرعت ورودی ، r است که همزمان دارن می نویسن .

نوشتن اولی چه قدر طول می کشه ؟ نوشتن بیت ها در باخر ورودی ؟ اگر L بیت داشته باشیم ، $\frac{L}{r}$ طول می کشه :

$t_{cell} = \frac{L}{r}$ ← یک cell slot

در خروجی هم ، L بیت وجود داره که داریم می خونیم ارسال کنیم با سرعت r . چه قدر طول می کشه ؟

(همه هم موازی دارن می خونن)

$t_{cell} = \frac{L}{r}$

⇐ الان سلول میاد در ورودی نوشته می شه ، وقتی تموم شد ، در انتهای time slot ، این سلول در باخر ورودی نوشته شده . اینو برمی داریم در حافظه می نویسیم . با چه سرعتی ؟ با سرعت 2N برابر این . بعد ، بعدی را برمی داریم می نویسیم در حافظه ، بعد بعدی و ... . برای هر کدوم یک بریدگی زمانی کوچکی (mini slot) مصرف می کنیم . یعنی هر N تا رو می تونیم در یک cell slot بنویسیم در حافظه . شیفت سر هم با سرعت بالا .

سمت خروجی هم با سرعت بالا برمی داریم می ذاریم در باخرهای خروجی .

⇐ N تا نوشتن و N تا خوندن را در یک cell slot انجام می دهیم . پس سرعت حافظه باید 2N برابر r باشد . با تقسیم زمانی (به mini slot ها) ، داریم switching انجام می دهیم . (از یه پورت می خونیم ، روی یک پورت می نویسیم .)

برای همین می گیم محدودیت سرعت داره . در یک cell slot ، 2N دسترسی به حافظه داریم که باید کمتر مساوی t_{cell} باشد :

$2N.t_{mem} \leq t_{cell}$ → محدودیت اندازه switch

$N.r \leq K = \frac{L}{2t_{mem}}$

Scanned with CamScanner

جلسه 14

مرور: switch های shared memory را دیدیم و روش های پیاده سازی اش را.
یکی از مزایای switches shared memory ، استفاده ی بهینه از حافظه است => کاهش هزینه ی پیاده سازی
(با حافظه ی کمتری می توانیم به performance metric های خوبمون مثل cell loss برسیم.)

=> ایده هایی مطرح شده که switch های shared memory , space division را با هم ادغام کنیم
که هم محدودیت سرعت switch های shared mem. را نداشته باشیم و هزینه اش را داشته باشیم.

(p.14 switch های time-space approach ← رویکرد استفاده ی همزمان از حافظه ی
مشترک و switching بر مبنای تقسیم زمانی ، space division ، رویکردی هست مورد توجه.

یکی از روش ها: space-time-space approach
سوییچی که میاد برای حافظه ی مشترک از چندین حافظه استفاده می کند. حافظه هایی که به طور فیزیکی از هم مجزا
هستند و امکان نوشتن / خواندن همزمان از این حافظه ها وجود دارد؛ اما کل این حافظه که به صورت
چندین حافظه ی فیزیکی هست، همش به عنوان یک حافظه ی مشترک تلقی می شه و صف هایی که برای پورت های
خروجی تشکیل می شه، درون این حافظه های مشترک شکل می گیرد.

قبلاً که یک حافظه ی مشترک داشتیم، صف ها داخل یک حافظه ی مشترک قرار داشت، برای خوندن و ارسال از
یک پورت خروجی، ما یک head ptr داشتیم که سر صف را نشون می داد و یک tail ptr که
انتهای صف را نشون می داد. => چون یک حافظه ی مشترک داشتیم، 2N ، access به memory
در هر cell slot باید انجام می دادیم که محدودیت برای ما ایجاد می کرد.

در این روش می آییم به جای اینکه از یک حافظه ی مشترک استفاده کنیم، از k تا حافظه ی مشترک استفاده می کنیم
پس در یک cell slot که قراره 2N access memory داشته باشیم، به هر حافظه ی مشترک، $\frac{2N}{k}$ access خواهیم داشت و اگر k را مضربی از N بگیریم مثلاً $k=\frac{N}{2}$ یا $k=N$ یا $k=\frac{N}{4}$
یعنی اگر $k=c\cdot N$ ، می شه نتیجه گرفت به هر حافظه ی مشترک ما $\frac{2N}{cN}=\frac{2}{c}$ دسترسی خواهیم داشت.
مثلاً اگر $c=\frac{1}{2}$ ، می شه 4 تا دسترسی. => با یک مقدار دسترسی مشخص به حافظه های مشترک، می توانیم
این switch را پیاده کنیم.

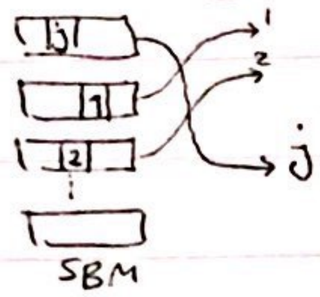
مثال: $k=N$ ، N ورودی و N خروجی داریم. N تا سلول ورودی را باید داخل حافظه ی مشترک بنویسیم
و N تا سلول خروجی را بخوانیم خارج کنیم. یک space division _ switch قرار می دهیم (سمت چپ شکل ←
(Crosspoint switch

maxnote

Scanned with CamScanner

این switch ← space division میاد این N تا ورودی را در N تا خونه‌ی حافظه می‌نویسه. (مثلاً یکی را در SBM ← پایینی، یکی تو SBM ← وسطی و ...) ← این عمل switching را انجام می‌ده.
این در انتهای صف خروجی باید نوشته بشه.

وقتی می‌خواهیم سلول را بخونیم، باید از ابتدای صف بخونیم. مثلاً برای پورت خروجی 1، ممکنه سلول ابتدای صف در SBM 2 باشه. برای پورت 2، یه‌جای دیگه باشه. مثلاً switching پس انجام می‌شه.



برای آدرس دهی، قبلاً می‌گفتیم head ptr مشخص می‌کنه از سر صف بخونیم. الان هم head ptr داریم ولی دو سطحی آدرس دهی می‌کنه. می‌گه از کدوم ① حافظه، چه آدرسی ② از اون حافظه.
یعنی برای نگهداری آدرس های صف، یک زوج عدد باید نگه داریم. ptr قبلاً یک آدرس حافظه بود، الان که k تا حافظه مشترک داریم، شماره‌ی حافظه و آدرس درون اون حافظه را باید نگه داریم.
⇐ ptr های درون Addr. queues، یک زوج عدد را مشخص می‌کنه.

ما به گونه‌ای این حافظه ها را رشد می‌دهیم در مدار کنترلر که اینها باهم رشد کنند و ما از کل فضای حافظه، استفاده بهینه ببریم. همون وضعیت shared memory را این جا به دست آوریم.
(اندازه‌ی حافظه ها می‌تونه برابر یا متفاوت باشه اما عموماً برابر هستند)

اشکال این روش: یک مسأله درست می‌کنه به نام head of line blocking (متفاوته از قبلی) به این شکل که چون صف های منطقی به طور مشترک ایجاد شدند، ممکنه زمانی پیش بیاد که سلول سر صف برای پورت 4 و سلول سر صف بعدی پورت 3، جفتشون در یک حافظه‌ی مشترک بیفتند. اینطوری یکیش را نمی‌تونیم ارسال کنیم. چون اگه تعداد دسترسی ها به هر حافظه مشخص و محدود باشه (مثلاً 2 یا 4)، حالت هایی ممکنه پیش بیاد که سلول‌های سر صف خروجی، برای چند تا خروجی مختلف بیفتند در یک حافظه‌ی مشترک و نتوانیم خارجشون بکنیم.
(مجموع ظرفیت با حالت یک حافظه برابره فقط محدودیت تعداد دسترسی ها را کاهش دادیم و از بین بردیم. قبلاً در یک cell slot باید 2N access می‌داشتیم. حالا کاهش دادیم به 2 یا 4 تا دسترسی. N را از معادله حذف کردیم. چون k را گرفتیم ضریبی از N. اندازه‌ی switch حالا می‌تونه بزرگ بشه.) هر چیز در مدار کنترل کننده با سلول هایی که وارد شده، اگه بتونیم پیش‌بینی کنیم و پیشگیری کنیم از سلول هایی که احتمال داره سر صف بیوفتن؛ اما این پیشگیری نمی‌تونه به طور صد در صد جلوی blocking را بگیرد.)

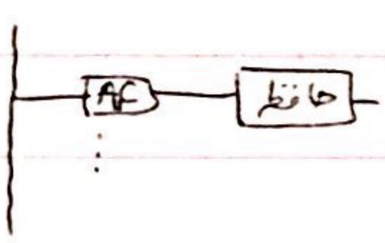
P.16 ② روش multi-stage shared memory switches

Shared mem. switch، محدودیت اندازه داشت، در اندازه‌های کوچک می‌توانستیم درست کنیم مثل 8x8 ولی اگه

switch ← 64 x 64 بخواهیم ، محدودیتش به ما اجازه نمی‌ده .
یکی از روش ها اینه بیایم ماجول های کوچکتر shared memory درست کنیم مثل 8x8 ، بعد چند مرحله‌ای عمل switching را انجام دهیم ← multi. Stage shared mem.

p.17) یکی از نمونه هاش سوییچ WIGAS است که اومده از switch های shared memory 8x8 استفاده کرده و یک switch 64x64 ایجاد کرده .
هنوز این switch های multi stage نمی‌گه داریم استفاده بهینه از حافظه می‌بریم چون وقتی ترافیک می‌ره به سمت یک دسته از پورت های خروجی ، یک دسته‌ی دیگه خالی می‌ماند . مثلاً در دسته‌ی از 8x8 SM استفاده کردیم ولی کل حافظه‌ای که در multi Stage switch گذاشتیم ، برای تمام پورت های خروجی ، مشترک نیستند . مثلاً ماجول آخر مربوط به 8 تا پورت آخر است . ⇐ مثلاً استفاده‌ی بهینه‌ی 100 درصدی از حافظه نداریم ولی نسبت به حالت اینکه به ازای هر پورت خروجی ، یک حافظه‌ی جدا بگذاریم ، این switch وضعیت بهتری دارد .
از نظر تأخیر هم باید بررسی شه آیا این multi stage switch که چندین مرحله حافظه دارد ، آیا تأثیری روی افزایش تأخیر داره . یا خیر .

p.18) روش دیگه ، switch ی هست که ایده‌اش switch های shared medium هست . در Shared Medium یک Bus ← مشترک داشتیم ، address filter داشتیم و به ازای هر خروجی یک حافظه داشتیم :



در این روش می‌گه می‌آییم پورت های خروجی را دسته‌بندی می‌کنیم مثلاً هر 8 تا پورت خروجی را یک گروه در نظر می‌گیریم ← concentrator based growable switch
می‌گه من دسته‌بندی می‌کنم به ازای مثلاً هر 8 پورت خروجی ، یک حافظه‌ی مشترک استفاده می‌کنم . ⇐ نسبت به وضعیتی که به ازای هر پورت خروجی یک حافظه داشتم ، از نظر استفاده از حافظه وضعیت بهتری است .

مبنای کار اینطوره که برای هر گروه k تایی از پورت های خروجی ، یک ماجول سوییچ وجود دارد . (در مورد ATM : شبکه‌ی با سرعت بالا هستند . اندازه‌ی بسته ها ثابت و کوچک است و به بسته ها سلول می‌گویند . پس چون cell mode switching می‌گیم مثلاً switch های ATM را داریم پیاده می‌کنیم .)
⇐ برای هر k تا یک ماجول می‌گذاریم . تمام n تا ورودی به این ماجول وارد می‌شوند . از این N تا ، اینا رو از AF رد می‌کنیم . ممکنه همه‌ی N تا سلول بخواد به این k تا خروجی بره ، ممکنه کمتر بیاد ، ممکنه هیچی نیاد .
N ورودی داریم ، بعد یک Concentrator گذاشتیم که N را تبدیل می‌کنه به 8 .
وظیفه‌ی Concentrator : از این N تا ، اون 8 تا که متعلق به این خروجی هستند انتخاب کنه . اگه و تا سلول می‌خواد به این مجموعه 8 خروجی بیاد ، Concentrator ، نفر‌ی را حذف می‌کنه . (مثلاً با مکانیزم های اولویت دهی) این 8 تا رو می‌بره تو حافظه می‌نویسد . به ازای هر پورت خروجی از این 8 تا یک صف منطقی ایجاد شده و سرویس را می‌دهد . مثلاً حافظه‌ی مشترک را بین 8 تا پورت به طور مشترک استفاده می‌کنیم و استفاده‌ی بهتری از حافظه می‌کنیم .

maxnote
Scanned with CamScanner

اما احتمال دارد در Concentrator سلول را حذف کنیم. اگر بیایم این جا speed up بدهیم به switch یعنی در switch ی 8x8، امکان اینکه در یک cell slot، 16 تا سلول را بنویسیم، 8 تا رو بخوانیم هم وجود داشته باشد، یعنی محدودیت access به memory به ما اجازه بدهد، concentrator می‌تونه بیاد در همین مرحله این کار را انجام بده و احتمال cell loss را ببره به سمت 0.

نکته: توان با انرژی متفاوته. ممکنه یک switch، توان مصرفی‌اش 1000 وات باشه ولی همیشه فعال نباشه. مثلاً در CAM اگه بخواهیم از کل حافظه استفاده کنیم، انرژی زیادی مصرف می‌کنه، توان بالاست ولی اگه درصد استفاده را مدیریت کنیم، انرژی مصرفی میاد پایین. (وات ساعت)

چرا 8 تایی تقسیم کردیم؟ ما یک حافظه مشترک داریم. در یک cell slot، t_{mem} هم مشخصه $\Leftarrow$ $n = \frac{t_{cell}}{t_{mem}}$

$\Leftarrow$ n دسترسی به حافظه می‌توانیم در یک cell slot داشته باشیم.

حالا به حافظه مشترک چند دسترسی می‌خواهیم؟ k ورودی و k خروجی داریم، $2k$ دسترسی می‌خواهیم داشته باشیم $\Leftarrow$

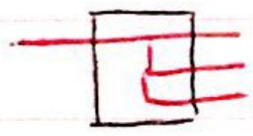
$2k \leq n \Rightarrow k \leq \frac{n}{2}$

در این مثال k شده 8.

مدیریت مصرف انرژی در همه‌ی سطوح: 1) در طراحی 2) مدیریت مصرف نرم‌افزاری مثل OS که می‌ره به sleep

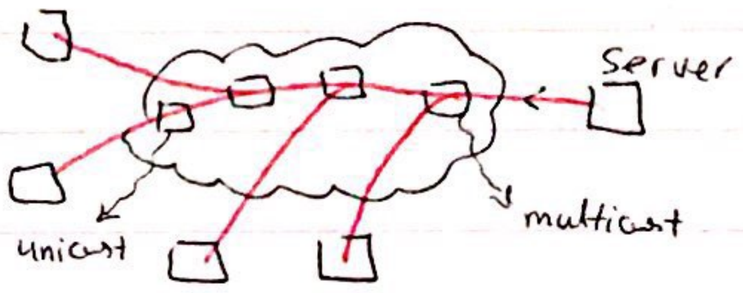
(p.19 multicasting یا چندپخشی

تعریف: اگر یک سلول، از یک پورت ورودی وارد شه، بخواهد از چندین پورت خروجی خارج شه، پخش سلول چندپخشی می‌گوییم.



ما یک سرویس multicast در شبکه داریم که یک سرور می‌خواهد همزمان به چندین کلاینت سرویس دهد. اگر فرض کنیم اینا از switch ها دارن رد می‌شوند $\Leftarrow$ switch ها باید سرویس multicasting ارائه دهند.

مثلاً در شکل، یک بسته وارد شده به گره اول، شده 2 تا.



مثال: پخش یک شبکه تلویزیونی live. هرکس join می‌شه به این گروه، سرور یک Stream داره می‌فرسته ولی در switch ها پخش می‌شه براساس routing انجام می‌شه. سرور ما باید سرویس multicasting را داشته باشه و یک سلول ورودی را از چند پورت خروجی خارج کند.

در switch های shared memory چون حافظه‌ی مشترک داریم، به سادگی انجام می‌شه. یکبار رو حافظه می‌نویسیم

Scanned with CamScanner

چند بار اون حافظه را می‌خوانیم. روش پیاده سازی multicasting در Switch های shared mem، 3 روش وجود دارد: (P.19)

1) روش multicast logical queue
2) روش cell copy
3) روش Address copy → بیشترین مزیت ولی پیچیده‌تر.

P.20: در روش multicast logical queue می‌گیم می‌آییم در اون حافظه مشترک که داشتیم، به ازای هر خروجی یک صف multicast درست می‌کنیم. برای multicast هم یک صف multicast درست می‌کنیم.
<= N+1 صف درست می‌کنیم.

سلول هایی که وارد می‌شوند، در line card ها با پردازش مشخص می‌شه سلول multicast است <= روی سلول های multicast یک label multicast می‌زنیم که در اون label مشخص می‌کنیم این سلول از چه پورت های خروجی باید خارج شه. ← اگه این label را داشت هدایت می‌کنیم به صف multicast.

حالا برای یک خروجی مشخص، 2 تا انتخاب داریم، آیا سلول unicast ش را منتقل کنیم یا اگه multicast داره، اونو.
نکته‌ی دیگه: وقتی یه سلول multicast میاد سر صف، این فقط از یه پورت خروجی قرار نیست خارج شه، از چند پورت خروجی باید خارج شه <= یکجا باید switch شه. اگه قراره یکجا switch شه، هم پورت مثلاً 1 هم k باید همزمان اینو خارج کنند. <= باید به صف multicast اولویت دهیم بگیم اگه multicast زداشت برو سراغ unicast. باید یک strict priority یا اولویت سخت و ثابت (بدون تغییر) بگذاریم همیشه برای multicast ها.
حالا اگه یه سلول با اولویت بالا اومد در صف unicast برای پورت خروجی 1 که براش multicast هم وجود دارد، unicast انتخاب نمی‌شه، QoS لطمه می‌خوره. ← ایراد روش multicast logical queue
برای رفع این ایراد:

P.21) روش cell copy

در این روش می‌گیم اگه یه سلول multicast وارد شد، یک عدد کپی کننده می‌گذاریم، اینو به تعداد پورت هایی که قراره خارج بشه کپی کنه. (مثلاً اگه قراره از پورت های 0 و 1 و 3 خارج شه، 3 تا کپی می‌گیریم). حالا می‌شه 3 تا سلول مجزا و هر کدوم در انتهای صف unicast نوشته می‌شه. ما دیگه صف multicast نداریم. به ازای هر پورت خروجی یک صف داریم. بحث اولویت این سلول با سلول های داخل صف هم می‌تونه مدیریت بشه. (همزمان ممکنه خارج نشوند ولی مشکلی نیست)
روی یک پورت ترتیب نباید به هم بخوره ولی پورت‌های مختلف عیبی نداره

ایراد: هدر رفت حافظه زیاد. یک سلول را 3 بار کپی کردیم. در بدترین حالت اگه روی N تا ورودی، N تا سلول وارد بشه که همه broadcast باشه، N^2 حافظه مصرف می‌شه. <= هدر رفت حافظه زیاد در این روش.

maxnote

Scanned with CamScanner

چه طوری این مشکل را می‌توانیم حل کنیم؟

P.22) روش Address copy

در این روش، یک سلول را فقط یکبار در حافظه می‌نویسیم ولی آدرس این حافظه را انتهای صف‌ها می‌گذاریم. یعنی اگر یه سرویس multicast داریم که از 3 تا پورت خروجی می‌خواهد خارج بشه، آدرسش را در انتهای صف خروجی‌هایی که باید ازشون خارج شه می‌نویسیم => یکبار نوشتیم، 3 بار می‌خوانیم. چون این خوندن همزمان نیست، مثلاً پورت 3 ممکنه زودتر بخونه بعد پورت 5 بعد پورت 6. => خوندن همزمان نیست. قبلاً که یک سلول از حافظه می‌خوندیم می‌گفتیم این آدرس حافظه آزاد می‌شه می‌توانیم سلول جدید را اونجا بنویسیم. الان چون سلول multicast هست وقتی می‌خونیم نباید آزادش کنیم. (سلول جدید جاش نوشته می‌شه، پورت 5 و 6، سلول اشتباه می‌فرستن)

=> در این روش که یک سلول را فقط یکبار در حافظه می‌نویسیم، موقعی می‌توانیم حافظه را آزاد کنیم که مطمئن شویم تمام کپی‌ها ارسال شده. وقتی آخرین پورت سلول را می‌خواند باید آزادش کنیم.

=> وقتی سلول را می‌نویسیم در اون خونه‌ی حافظه، برای اون خونه یک Counter، Set کنیم، مثلاً 3، هر بار که می‌خوانیم یکی ازش کم می‌کنیم. وقتی اون عدد 0 شد، حافظه آزاد باید بشه.

=> در این روش، از حافظه بهتر استفاده می‌کنیم اما باید به ازای تمام خونه‌ی حافظه که سلول‌ها را توش ذخیره می‌کنیم، به ازای هر خونه‌ی حافظه یک Counter در نظر بگیریم. (MCC در شکل ص 23)

P.23) وقتی یک سلول ورودی را در اون خونه‌ی حافظ می‌گذاریم، Counter را براساس تعداد دفعاتی که باید خوانده بشه تا آزاد بشه، Set می‌کنیم.

اگه سلول ورودی unicast باشه، 1، Set می‌شه. یکبار بخونیم آزاد می‌شه. وقتی multicast است به تعداد پورت‌های خروجی که باید خارج بشه اون عدد را Set می‌کنیم. اگه قراره از 2 تا پورت خارج بشه (A_0 و A_0)، در MCC، عدد 2 را Set می‌کنیم. یعنی بعد از 2 بار خوندن باید آزاد بشه. هر بار که می‌خونیم Counter یکی کم می‌شه. وقتی 0 شد، این آدرس می‌ره در آدرس‌های idle قرار می‌گیرد. (صف آدرس‌های آزاد)

=> در این روش از فضای حافظه بهتر استفاده می‌کنیم دیگه کپی نمی‌کنیم در حافظه ولی مسأله‌ای که باید حل کنیم اینه که وقتی یک خانه‌ی حافظه را می‌خواهیم آزاد کنیم باید مطمئن شویم تمام کپی‌هایش ارسال شده => مجبوریم به ازای هر خانه‌ی حافظه، یک Counter در نظر بگیریم. مدار کنترل کننده، پیچیده‌تر می‌شه، هزینه‌ای که می‌پردازیم اینه که به ازای هر خانه‌ی حافظه یک Counter داریم.

MCC می‌گه مثلاً در خانه‌ی حافظه‌ی A_0، این سلول چند بار دیگه خونده بشه، آزاد می‌شه. ارسال (حافظه را که آدرس می‌دهیم، اون Counter فعال می‌شه. اگه دستور read می‌زنیم یکی از Counter کم می‌شه.

موقع write کردن که اون خونه آزاد بوده قبلش، Counter را Set می‌کنیم: 1 ← unicast بزرگتر ← multicast)

۱۶ دقیقه آخر نیم ۱۴ ← شروع اسلاید input Buffered switches

سوئیچ ها را از نظر عملکردی اینطوری بررسی کردیم و دیدیم که گفتیم سوئیچ های shared memory بهترین عملکرد را از نظر هزینه و تأخیر دارد.

اما مشکل سوئیچ های shared memory ← محدودیت اندازه

اگر محدودیت اندازه برای حافظه سوئیچ ما مشکل زا نباشد و بتوانیم از این محدودیت چشم پوشی کنیم (این محدودیت جلوی عملکرد ما در کاربرد ما را میگیرد) گزینه اول ما همیشه shared memory هست.

اگر این محدودیت برایمون مشکل باشه سراغ output Buffer میریم. از نظر هزینه بهتره، shared memory هزینه بیشتری داره اما از نظر تأخیر تقریباً با shared memory برابر است. ← چون به ازای هر پورت خروجی صف مجزا تشکیل میشود

اما سوئیچ های output Buffered هم ما محدودیت اندازه داریم. ← تعداد دفعاتی که میتوانیم Read & Write داشته باشیم محدوده.

اگر سوئیچ ما speedup = N باشد ← N+1 دسترسی به حافظه میتونیم داشته باشیم در output Buffer

محدودیت اندازه در output Buffer کمتر از سوئیچ های shared memory هست.

← اگر این محدودیت به گونه ای بود که نتوانیم از shared memory و output Buffer استفاده کنیم به ناچار مجبوریم برویم سراغ input Buffer

وقتی داریم از سوئیچ input Buffer صحبت میکنیم ← ما میخوایم از این سوئیچ ها در سرعت های بالا استفاده کنیم. یا اندازه سوئیچ بالا (تعداد پورت ها زیاد)

وقتی میگیم سرعت بالا یعنی عدد $\frac{\text{cell size}}{\text{زمان ارسال یا دریافت یک سلول}}$ ما در بسیار بسیار کوچکی باشه. $O(ns)$

در سوئیچ input Buffer محدودیت اندازه نداریم. هر چقدر خواستیم میتونیم ظرفیت رو بالا ببریم اما ۲ مسئله مهم داریم: ۲ تا مشکل داریم:

1 HoL ← محدودیت Throughput ایجاد میکند.

2 scheduling ← بحث الگوریتم های زمانبندی مطرح میشه که خودشون پیچیدگی دارند و زمان میبرند.

برای حل HoL، باید از سوئیچ های سریع تر استفاده کنیم ← هزینه افزایش پیدا کند

یا موازی ← که سرعت های متفاوتی از ورودی به خروجی داشته باشیم ← افزایش هزینه پیاده سازی

پس مشکل HoL ← جدای از مشکل کاهش Throughput، مشکل افزایش هزینه پیاده سازی را هم به همراه دارد.

Scanned with CamScanner

پس در input Buffer ها با یک روشی Throughput مورد استفاده تامین کنیم.

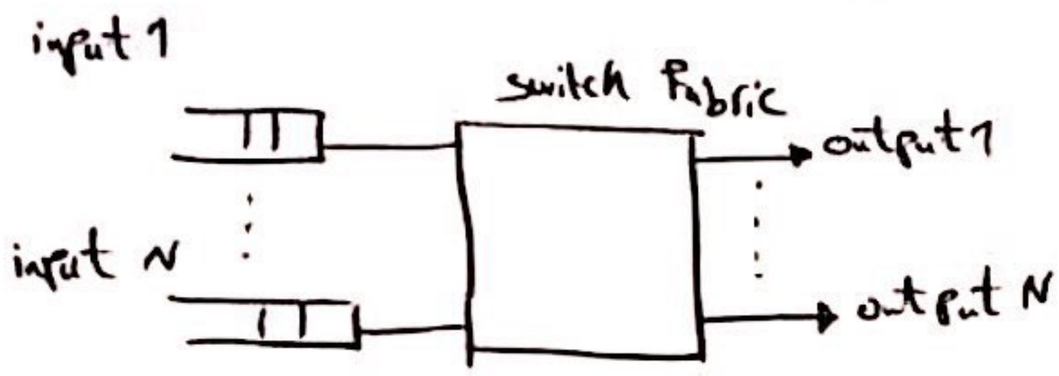
مثلاً سوئیچی درست کنیم که ظرفیتش 1Tbps باشد تا بتونیم Throughput = 600Mbps برسیم.

این محدودیت HOL (محدودیت سر دهی) را با یک virtual output Queuing (VOQ) می‌توان حل کرد.

اما مسئله‌ی Arbitration (scheduling) در جایی که بیایم از VOQ استفاده کنیم مسئله‌ی پیچیده‌تری نسبت به حالت قبلی خودش می‌شود.

در این فصل تمرکز روی مکانیسم‌های scheduling هست. ابتدا یک مرور از VOQ و HOL داریم و بعد سراغ مکانیسم‌های scheduling

- چرا در SW، از بافر استفاده می‌شود؟ (برای مدیریت بار...) به خاطر مسئله output contention یا همان مسئله رقابت بر سر پورت خروجی



سلول‌ها در چند نقطه وارد می‌شوند. و قرار بخواهیم انتخاب کنیم برای ارسال switching به خروجی، از بین سلول‌هایی که سر صف هستند، برای هر پورت خروجی فقط یک سلول را می‌توانیم انتخاب کنیم.

HOL: اگر سلول سر صف انتخاب نشود، تمام سلول‌هایی که پشت این سلول در صف باشند هم نمی‌توان برای ارسال انتخاب کرد.

که راه‌حل‌های برای حل مسئله HOL بررسی شد. 1 windowing (به جای اینکه برای انتخاب تنها به اولین سلول سر صف نگاه کنیم، به 4 سلول سر صف نگاه کنیم و انتخاب کنیم.

2. VOQ

P.4

VOQ: در بافر ورودی، کاری شبیه shared memory انجام بدهیم. بافر ورودی هر پورت ورودی را به صورت منطقی به ازای هر پورت خروجی، یک صف منطقی (در بافر ورودی) تشکیل دهیم. ← با این کار در هر ورودی به ازای هر پورت خروجی یک صف داریم.

پس کلاً مسئله HOL از بین می‌رود.

Q11: صفی که در ورودی i برای پورت خروجی j تشکیل شده.

با این کار، حالا اگر بخواهیم برای خروجی j، یک سلول انتخاب کنیم، فقط به صف‌های i در ورودی‌های مختلف نگاه می‌کنیم. با این کار برای پورت‌های خروجی دیگر می‌توانیم از صف‌های i انتخاب کنیم

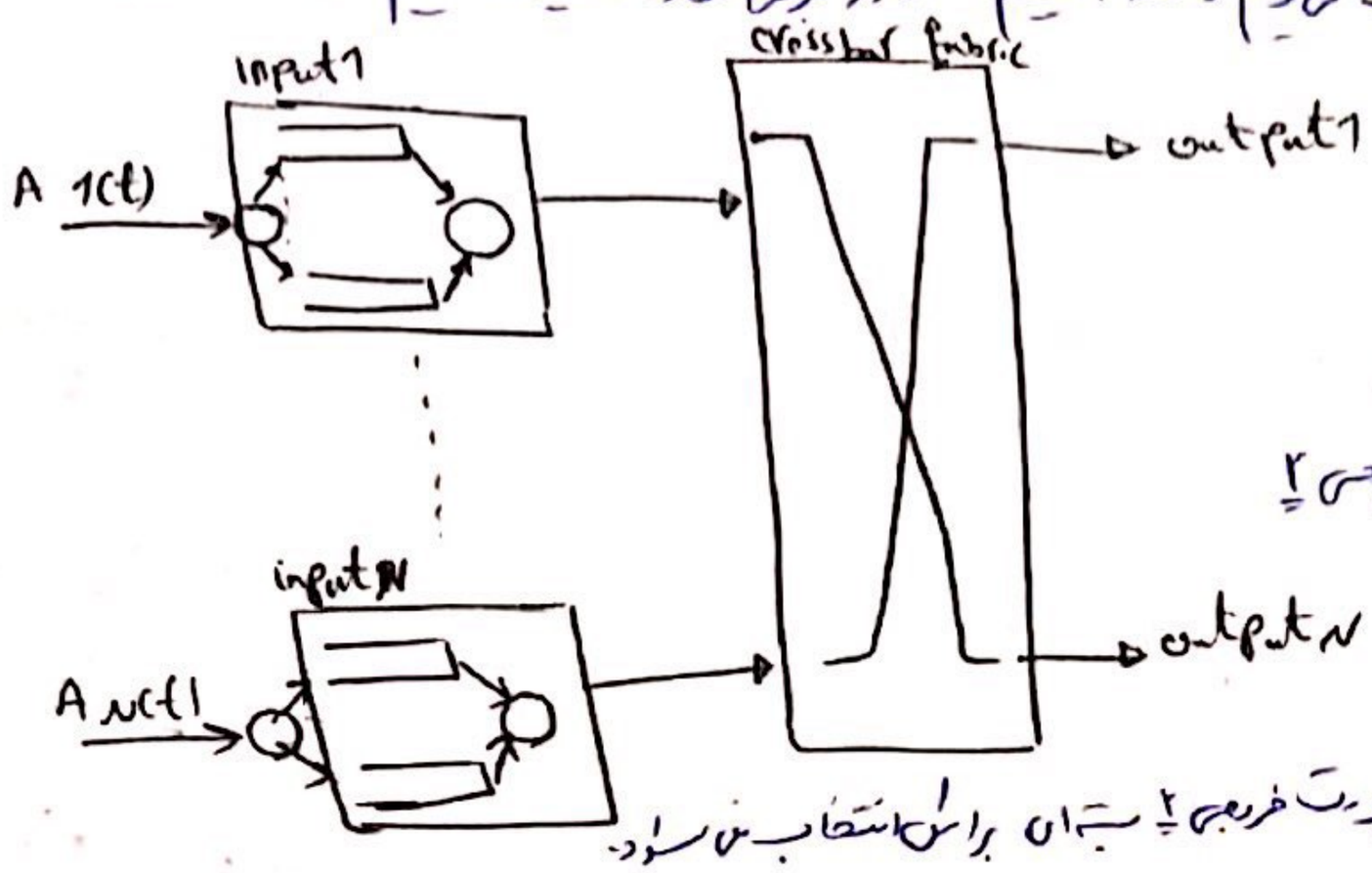
⇐ بنابراین عدم انتخاب یک ورودی برای یک خروجی، مانع انتخاب اون ورودی برای خروجی‌های دیگر نمی‌شود.

Scanned with CamScanner

مسئله VOQ هم مسئله Arbitration (انتخاب دارد). باید هر ورودی روی یه خروجی match کنیم و هر خروجی به یک ورودی match شود. بنابراین نیاز به الگوریتم scheduling داریم که در واقع عمل matching را انجام میده.

این matching یک به یک هست یک ورودی فقط به یک خروجی میچ میشود و یک خروجی هم فقط به یک ورودی میچ میشود.

- اگر همه‌ی صف‌ها پر باشند، n تا سوئیچینگ می‌توانیم داشته باشیم. اما در برخی موارد میشه که نتونیم n تا سوئیچینگ داشته باشیم



مثلاً رشته input 1 از صف خروجی 1 میخواهد سلول انتخاب بشود و در حال ارسال به خروجی 1 باشد

و تنها در بافر 1 Q_{12} سلول برای پورت خروجی 2 وجود داشته باشد و سایر ورودی‌ها سلولی برای پورت خروجی 2 نداشته باشند. یعنی صف 2 همه‌ی ورودی‌های دیگر خالی باشند. در این صورت خروجی 2 سلولی برایش انتخاب نمی‌شود.

چون از ورودی 1 (بافر input 1) سلولی میچ شده به سمت دیگری نمی‌تواند میچ شود.

پس ما نیاز به الگوریتم scheduling داریم.

سیستم IQ. ادامه‌ی input Buffer

میخوام مسئله matching رو حل کنیم. قبل اینکه بخوام وارد بشم ببینم چه الگوریتم‌هایی برای matching وجود داره.

یک سری تعاریف اولیه داشته باشیم و توضیح بدم یک الگوریتم scheduling تحت چه شرایطی باید جواب داشته باشه:

$A_i(t)$: اگر فرض کنیم در هر پورت ورودی مثل i، فرایند ورودی بسته‌ها که یک فرایند تصادفی هست. $A_i(t)$ می‌شود فرایند تصادفی رسیدن بسته‌ها به پورت ورودی i ام.

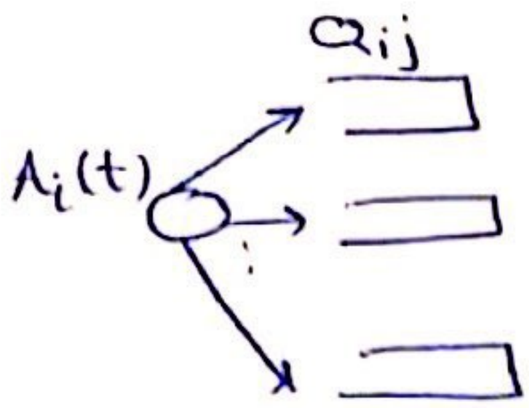
a_{ij} : به ازای هر خروجی، در هر ورودی یک صفی تشکیل بشه که ما این رو با a_{ij} نشون میدیم. a_{ij} می‌شود صفی که در ورودی i ام برای خروجی j ام تشکیل شده.

$L_{ij}(t)$: طول صف a_{ij} در لحظه‌ی t.

Scanned with CamScanner

$A_{ij}(t)$: فرایند ورودی به ورودی i ام به خروجی j ام.

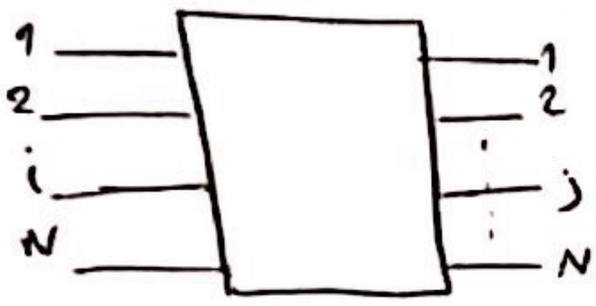
پس در هر ورودی برای هر خروجی یک صف درست می‌کنیم، ترافیک ورودی i ام به بخش j ام می‌رود به صف خروجی 1، بخش دوم به صف خروجی 2 و ...



λ_{ij} : نرخ متوسط ورود بسته‌ها به صف Q_{ij}

اگر λ_{ij} را جمع بزنیم از $j=1$ تا N ← $\sum_{j=1}^{N} \lambda_{ij}$ = نرخ ترافیک وارد شده به ورودی i ام

اگر λ_{ij} را براساس پورت‌های ورودی بزنیم از $i=1$ تا N ← $\sum_{j=1}^{N} \lambda_{ij}$ = نرخ ترافیکی که باید از خروجی j ام خارج شود.



می‌دونیم داریم، به هر پورت ورودی یه ترافیکی وارد می‌شود. به طور متوسط به این میگیم λ_i

از همه‌ی ورودی‌ها هم که به خروجی j، ترافیک داشته باشیم. جمع این‌ها میشه λ_j

اگر λ_i و λ_j (λ_{ij}) را نرمال شده در نظر بگیریم، یعنی فرض کنیم ظرفیت بیشتر از 1 نمی‌تواند باشد، در نتیجه λ_j نمی‌تواند از 1 بیشتر باشد. چون ترافیک نمی‌تواند از خروجی خارج شود

فرض کنیم یه لینکی داریم 1Gbps ← در نتیجه مجموع ترافیکی که وارد می‌شه و که بخواد از لینک j ام خارج بشه، نمی‌تونه از این 1Gbps بیشتر باشه چون باعث می‌شه ترافیک انباشته توی صف‌ها بمونه و صف بی‌نهایت تشکیل میشه.

همچنین ترافیکی که به یک لینک ورودی وارد می‌شود نباید از ظرفیت لینک بیشتر شود.

* پس شرایطی که ترافیک ورودی به یک پورت ورودی و ترافیک ورودی به یک پورت خروجی به ازای تمام پورت‌ها از 1 بیشتر نباشد را شرایط (ترافیک) Admissible (ظرفیت لینک) می‌گوییم

$$\sum_{i=1}^{N} \lambda_{ij} < 1 \quad \forall j \qquad \text{And} \qquad \sum_{j=1}^{N} \lambda_{ij} < 1 \quad \forall i$$

پس شرط اینکه بتوانیم scheduling داشته باشیم این که ترافیک Admissible داشته باشیم.

یعنی ترافیکی که وارد شود می‌تواند خارج شود. نه از ظرفیت لینک‌های ورودی بیشتر نه از ظرفیت لینک‌های خروجی بیشتر.

Scanned with CamScanner

پس ما در شرایط Admissible دنبال یک الگوریتم scheduling می‌گردیم. در غیر این حالت اصلاً نمی‌توان ترافیک را خارج کرد و یک سوئیچ un-stable یا ناپایدار خواهیم داشت.

تعریف الگوریتم scheduling :

الگوریتم زمانبندی، فرایند داوری کردن یک سوئیچ است تا تصمیم بگیرد سلول‌هایی که در بافرهای ورودی هستند منتقل بشوند به پورت‌های خروجی. فرایند تصمیم‌گیری که چه سلول‌هایی از حافظه ورودی انتخاب بشوند و به خروجی منتقل شوند را الگوریتم زمانبندی می‌گویند.

در هر بازه زمانی (cell slot) یک ماتریس تصمیم داریم که این ماتریس $N \times N$ است.

$$\begin{array}{cc} & \text{پورت‌های خروجی } j \\ & \begin{array}{ccccc} 1 & 2 & & & N \end{array} \\ \text{پورت ورودی } i \Rightarrow \begin{array}{c} 1 \\ 2 \\ \vdots \\ N \end{array} & \begin{bmatrix} & & & & \\ \cdot & \cdot & s_{ij} & \cdot & \cdot \\ & & 0 & & \\ & & 0 & & \end{bmatrix} \end{array}$$

$$s_{ij}(t) = \begin{cases} 1 & \rightarrow \text{در این بازه زمانی سلول ورودی i ام به خروجی j ام منتقل می‌شود} \\ 0 & \rightarrow \text{سلول انتقال پیدا نمی‌کند} \end{cases}$$

پس $s_{ij}(t)$ اگر یک باشد یعنی matching صورت گرفته

طبیعی است که پیننگ 1 به 1 است. پس اگر در هر ستون یک درایه مقدار 1 بگیرد (یعنی برای آن پورت خروجی یک match اتفاق افتاد) پس سایر درایه‌ها باید صفر بشوند. و همینطور برای آن سطر هم سایر درایه‌ها باید صفر بشوند.

در هر یک ستون حداکثر، یک دونه 1 داریم. در هر سطر هم حداکثر یک دونه 1 داریم.

اگر در سطری تمام درایه‌ها از این پورت ورودی به هیچ پورت خروجی ای سلول منتقل نمی‌شود. صفر باشند.

شرط Admissible :
$$\begin{cases} \sum_{i=1}^{N} \lambda_{ij} < 1 \\ \sum_{j=1}^{N} \lambda_{ij} < 1 \end{cases}$$

تحت شرایطی که ترافیک ورودی Admissible باشد،

مقدار متوسط مورد انتظار = جمع متوسط طول تمام صف‌ها $E\left[\sum_{ij} L_{ij}\right]$ از بی‌نهایت کمتر باشد. یعنی صف بی‌نهایت تشکیل نشود.

یک الگوریتم scheduling را stable می‌گوییم اگر در شرایطی که ترافیک ورودی ما Admissible است، صف بی‌نهایت تشکیل نشود.

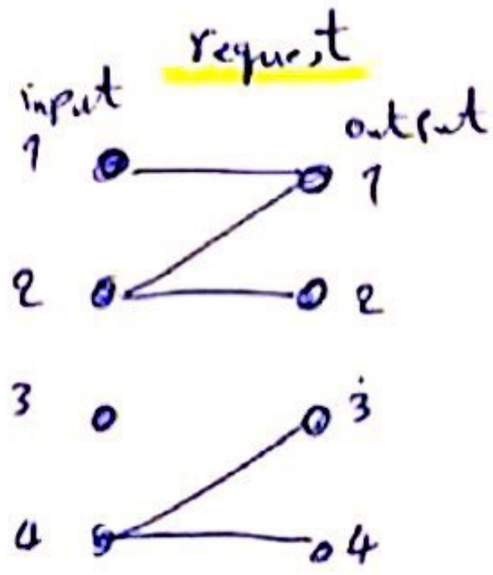
$$E\left[\sum_{ij} L_{ij}\right] < \infty$$

Scanned with CamScanner

تفاوت الگوریتم scheduling در سوئیچ‌های input Buffer مثل الگوریتم‌های matching است.

صورت الگوریتم‌های matching را با گراف نمایش می‌دهند.

- می‌توانیم درخواست‌های ورودی را با گراف نمایش بدهیم.



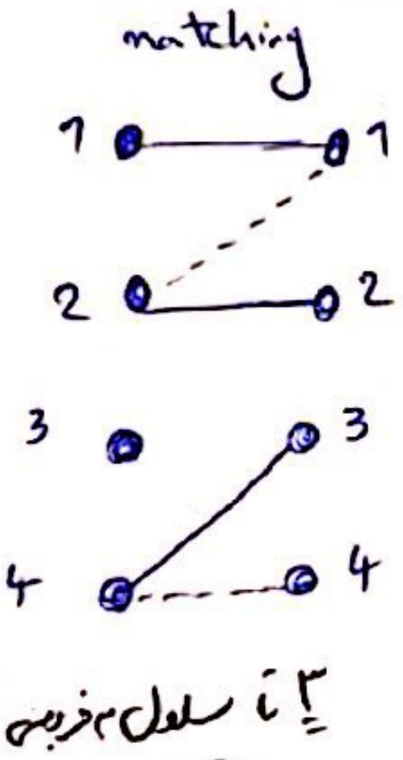
از پورت ۱ برای پورت خروجی ۱ سلول وجود دارد.

در ورودی ۲ برای خروجی‌های ۱ و ۲ درخواست وجود دارد. یعنی سلول وجود دارد.

۳ که به هیچ جا وصل نشده یعنی هیچ درخواستی در این صف وجود ندارد. تا کمی صف‌هایش خالی است.

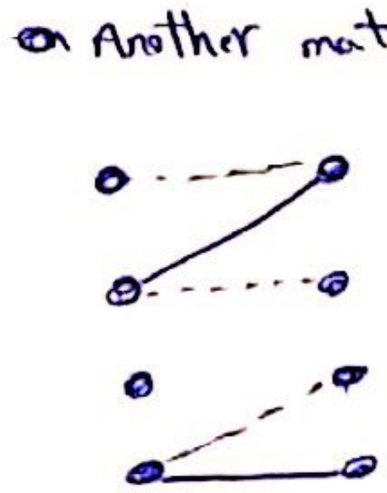
۴ و ۲ تا صف غیرخالی دارد. Q_{43} و Q_{44}

الگوریتم matching ما هر پورت ورودی را به یک پورت خروجی و هر پورت خروجی هم فقط از یک پورت ورودی سلول می‌گیرد.



۳ تا سلول به خروجی منتقل می‌شوند.

یکی از چیزهایی که می‌تونه اتفاق بیوفته شبیه این موارد باشه.



۲ تا سلول به خروجی منتقل می‌شوند.

نتیجه می‌گیریم scheduling (matching) می‌تواند جواب‌های متعدد داشته باشد.

سوال: کدام جواب بهتر؟

اگر جواب ۳ را انتخاب کنیم چون تعداد سلول بیشتری منتقل می‌شود گذردهی بیشتری داریم.

اگر جواب ۲ را انتخاب کنیم از منظر Throughput، گذردهی کمتری داره. اما از نظر تأخیر چطور؟ پس فاکتور و شاخص دیگری برای QoS اش نقش بازی می‌کند.

الگوریتم scheduling نیاز داریم.

- فاکتورهای انتخاب scheduling

1. Efficiency: کارآمدی که در بحث زمانبندی منظور ماکزیمم گذردهی می‌باشد.

2. Fairness: عدالت: سعی با حداکثر گذردهی عدالت رعایت شود و برخی سلول‌ها رجیا رمتقی بشوند یا با تأخیر زیاد منتقل شوند یا کم‌تری از سوئیچ بهره‌مند شود.

Scanned with CamScanner

stability : در شرایطی که ترافیک Admissible هست ، صف بی نهایت تشکیل نشود .

*. implementation complexity (مهم ترین فاکتور) پیچیدگی پیاده سازی

الگوریتم های scheduling باید در زمان یک cell slot تصمیم گیری کنند . مدت زمان اجرای یک الگوریتم زمانبندی برای بدست آوردن ماتریس s(t) حداکثر یک برش زمانی است .

برش زمانی ما چقدره ؟ تصمیم ما اگر محدودیت نداشته باشیم اصلا سراغ سوئیچ های input Buffer نمی آیم .

پس ما کلا سوئیچ های shared memory و output Queuing محدودیت سرعت داشتیم که سراغ این روش اومدیم .

اندازه سوئیچ بزرگ هست .

$$T_{cell} = \frac{L}{r}$$

$N \cdot r$

باید در یک cell slot پس إلا جواب ، جواب بهینه پیدا کنیم . باید با سرعت انجام شود . پس عملا ما نمی توانیم از روش های نرم افزاری استفاده کنیم ← باید الگوریتم scheduling ما یه نوعی باشد که در مدارات سخت افزاری قابل پیاده سازی باشد تا بتواند در سرعت بالا در مدت زمان یک cell slot انجام شود

maximum matching ← الگوریتم هایی که جواب بهینه رو بدست میارن

الگوریتم های زمانبندی به دو دسته تقسیم می شوند :

sub optimal ← الگوریتم هایی که جواب نزدیک به بهینه رو بدست میارند اما جواب بهینه رو به دست نمیارن ؛ اما پیچیدگی محاسباتی کم و قابل اجرا هستند .

N = تعداد پورت ها

از بین maximum matching ، N! جواب فرایم جواب بهینه رو پیدا کنیم ← محاسبه کردن و دیدن این الگوریتم از $O(N^3)$ می باشد . یعنی حتی اگر تعداد پورت ها کم باشد مثلا N=100 باشد اونوقت یک بیلیون محاسبات در مدت زمان یک cell slot که عملا قابل اجرا نیست . یعنی implementation complexity بالا زیاد که قابل پیاده سازی نیست

وقتی نمی توانیم جواب بهینه رو بدست بیاریم به ناچار مجبوریم رو بیاریم به الگوریتم های sub optimal

maximal matching ← ما در اوقات میایم سعی می کنیم جواب انتخاب کنیم بعد می بینیم اگر زمان داریم باز هم دنبال جواب بهتر می گردیم تا وقتی که به cell slot برسیم . به این الگوریتم ها maximal matching می گویند (جلوتر توضیح میدیم) اسم این الگوریتم ها تکرار شونده هستند . با هر بار تکرار جواب قبلی بهتر می شوند .

Scanned with CamScanner

روش هایی که عملاً اعمال می‌شوند در الگوریتم های زمانبندی الگوریتم های maximal matching هستند.

Randomized matching Algorithm : فرض یه سوئیچی داریم که N تا پورت ورودی و N تا پورت خروجی داره. برای هر خروجی در بافر ورودی یک صف داریم ← پس N^2 صف داریم در ورودی ها.

بعد از هر Time slot که الگوریتم matching اجرا می‌شود. اگر فرض کنیم وضعیت صف ij ما (تعداد سلول های موجود در صف Q_{ij}) با $L_{ij}(t)$ نمایش دهیم. آنگاه در Time slot بعدی $(t+1)$ ۳ حالت می‌توان تصور کرد:

$$L_{ij}(t+1) = \begin{cases} L_{ij}(t) - 1 & \text{: سلول جدیدی وارد نشود و یک سلول از سر صف ارسال شود} \\ L_{ij}(t) + 1 & \text{: هیچ سلولی از سر صف ارسال نشود و یک سلول هم وارد شود} \\ L_{ij}(t) & \text{: یک سلول ارسال شود و یک سلول وارد شود} \end{cases}$$

پس وضعیت صف ها اینجوری می‌شه که یا تغییر نمی‌کنه یا اضافه میشه و یا کم می‌شود. برای اکثر صف ها تغییری نمی‌کند N^2 صفی که داریم. در بیشترین حالت اینطوری که N تا سلول وارد شود و N تا سلول خارج شود یعنی 2N صف وضعیتشان تغییر کند. پس اکثر صف ها وضعیتشان ثابت باقی می‌ماند.

معنی این اینکه اگر الان یک matching انجام دادیم و matching خوبی بود، چون وضعیت تغییر زیادی نکرده، پس matching بعدی ای هم که انجام میدیم باید نزدیک به هم و معمولاً مشابه قبلی باشد و اختلاف زیادی نداشته باشد.

$s(t)$

matching بعدی: $s(t+1)$

الگوریتم های Randomized می‌گویند matching = $s(t)$ را برگیریم. با یک جست‌وجوی تصادفی، حول وحوش matching قبلی، با محاسبات کم، matching جدید را بدست بیاریم (چند نمونه رو جلوتر استاد درس میدن)

Frame-based matching : سری روش های دیگه داریم به اسم Frame-Based که الگوریتم هایی هستند که برای matching هایی به کار می‌رن که وقتی این سوئیچ ها میخوان matching اشون رو درست کنند خود = عملیات Reconfiguration کردن سوئیچ زمان ببره. (مثل سوئیچ های optical) در سوئیچ های

سوئیچ:
- optical : در سوئیچ های نوری، عناصر مکانیکی توی این سوئیچ ها قرار داره تحت عنوان mems که تغییر پیکربندی رو زمانبر می‌کنه.
- Electrical : زمان پیکربندی زمان زیادی نمی‌بره.

زمان انتقال + زمان تغییر پیکربندی : برای مواقع

به جای اینکه بیایم برای هر time slot ، ماتریس تطابق انجام بدیم، برای مثلاً چا تا هزار انجام بدیم.

به حداقل تغییر پیکربندی برسیم ← بعد با جابه‌جا کردن scheduling

ایده frame-based matching: می‌تونیم بیایم برای یک فریم بزرگ، مثلاً K تا برش زمانی سلول، یک جا scheduling رو انجام بدیم بعد

scheduling هایی که مثل هم هستند رو لیست هم بذاریم که تغییر پیکربندی به حداقل برسه (جلوتر بررسی می‌شود.)

stable matching with speed up - برای الگوریتم‌های scheduling ای که هم بافر ورودی هم خروجی بافر دارند

بیایم یه الگوریتمی ارائه بدیم که هم بشه اجراش کرد تو سوئیچ‌های input buffer و هم حداقل تأخیر را

تأخیر شبکه سوئیچ‌های output queuing (که بهترین تأخیر کم‌ترین تأخیر رو داشتند) برسیم

P.11 بررسی الگوریتم‌های maximum matching (که گفتیم به خاطر پیچیدگی‌شون قابل اجرا نیستند.)

بیایم به هر request یه وزنی بدیم. چون ما میخوایم الگوریتم ما fairness هم باشه، در نتیجه درسته تعداد سلول‌های انتقال داده شده (گذردهی) براش اهمیت داره اما عدالت هم براش مهمه.

پس بیایم به هر request یک وزن بدیم و بعد matching ای رو انجام بدیم که بیشترین وزن رو داره (و نوعاً بهینه‌ترین جواب رو پیدا کردیم. پس الگوریتم ↓

MWM : Maximum weight matching ← $O(N^3)$

وزن‌ها رو براساس مدت زمان همه‌ی پورت‌ها و داشتن حداکثر گذردهی انتخاب کنیم. پس باید روش‌هایی برای وزن دهی به درخواست‌ها داشته باشیم.

الگوریتم‌های وزن دهی به Requests :

LFQ (Longest Queue First) : صفی که طول بیشتری داره، وزن بیشتری بدیم. این روش می‌تونه Throughput رو افزایش بده.

OCF (Oldest cell First) : از صفی انتخاب کنیم که قدیمی‌ترین سلول داخل اون صف باشه. این روش تضمین عدالت رو برقرار کنه.

یکی از روش‌هایی که ترکیب ؟ روش بالاست ← LPF (Longest Port First)

$$w_{ij}(t) = \begin{cases} R_i(t) + C_j(t) & L_{ij}(t) > 0 \\ 0 & \text{otherwise} \end{cases}$$

Scanned with CamScanner

$R_i(t) = \sum_{j=1}^{N} L_{ij}(t)$: $R_i(t)$ مجموع سلول هایی که در ورودی i ام برای همه ی خروجی ها وجود دارد

$C_j(t) = \sum_{i=1}^{N} L_{ij}(t)$: $C_j(t)$ مجموع سلول هایی که در همه ی ورودی ها برای پورت خروجی j ام وجود دارد.

اگر براساس این L_{ij}، هم عدالت و هم مسئله توزیع ورودی رو رعایت کردیم.

معنی کردیم { یعنی ورودی ای رو انتخاب کردیم که بیشترین سلول داخلش وجود داره
و یعنی خروجی ای رو انتخاب کردیم که بیشترین سلول براش وجود داره

P.13 maximum size matching (MSM) : $O(N^{2.5})$ - $O(N^2\sqrt{N})$ بازم پیچیدگی بالایه.

وزن همه درخواست ها را یک در نظر بگیریم ← به یک نوعی انگار فقط ماکزیمم Throughput رو در نظر می گیریم.

← ما الگوریتم های ماکزیمم matching رو بخاطر پیچیدگی بالا شون اجرا نمی تونیم کنیم.

* * * * * * * *

مسیم سراغ بررسی الگوریتم های maximal matching

P.14 یعنی بیشینه

یعنی این جواب حداکثر بهترین جواب، اگر تکرارش کنیم ممکنه بهتر هم بشه. این الگوریتم ها، الگوریتم های تکرار شونده هستند. با هر تکرار جواب رو بهتر می کنیم ولی بازم ممکنه بهترین نباشه. بعضی بیشینه حداکثری می گویند.

برای اینکه از تکرار بیشتری بتونیم استفاده کنیم توی سری الگوریتم ها از تکنیک Pipline استفاده می کنند.

EX چون ورودی ما یک ست ریکت مدر ورودی می شگونیم، سلول های سادی،

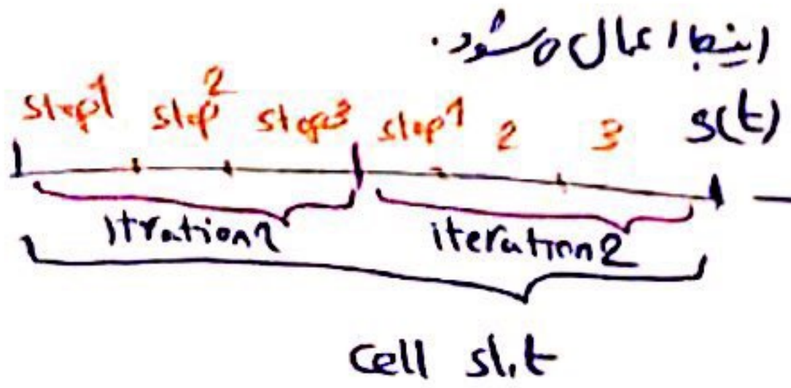
ل با سیفوان با نگاه انکیه سلول های پشت هم متعلق به یک پشت هستند این مچینگ رو بهبود بدهند.

P.15 Parallel iterative matching (PIM) :

مرتب کردن ورودی، خروجی به صورت تکرار شونده و موازی. این الگوریتم ۳ گام داره که در هر تکرار انجام می شود.

ما تکرارها رو در یک cell slot انجام میدیم.

اگر در یک cell slot بتوانیم ۲ تکرار انجام بدیم. در هر تکرار این ۳ گام انجام می شود



بعد از هر iteration، یک جواب بدست می آید. در هر تکرار جواب نسبت به جواب قبلی، بهتر می شود. s(t) بدست می آید.

Scanned with CamScanner

گفتیم ۳ تا step داریم: Request ← step1 ، Grant ← step2 ، Accept ← step3

step1: اون پورت های ورودی ای که هنوز به هیچ خروجی ای match نشدند ← در اولین تکرار، تمام ورودی ها unmatch هستند.

ولی در تکرار دوم تعدادی هیچ می‌شوند و باید کنار گذاشته شوند.

تمام پورت های ورودی ای که unmatch هستند یک Request میدهند به خروجی هایی که، برای این خروجی ها در ورودی سلول دارند. گفتیم هر ورودی N تا صف داره. هر صفی که سلول داره، پورت خروجی مدنظر Request میفرسته

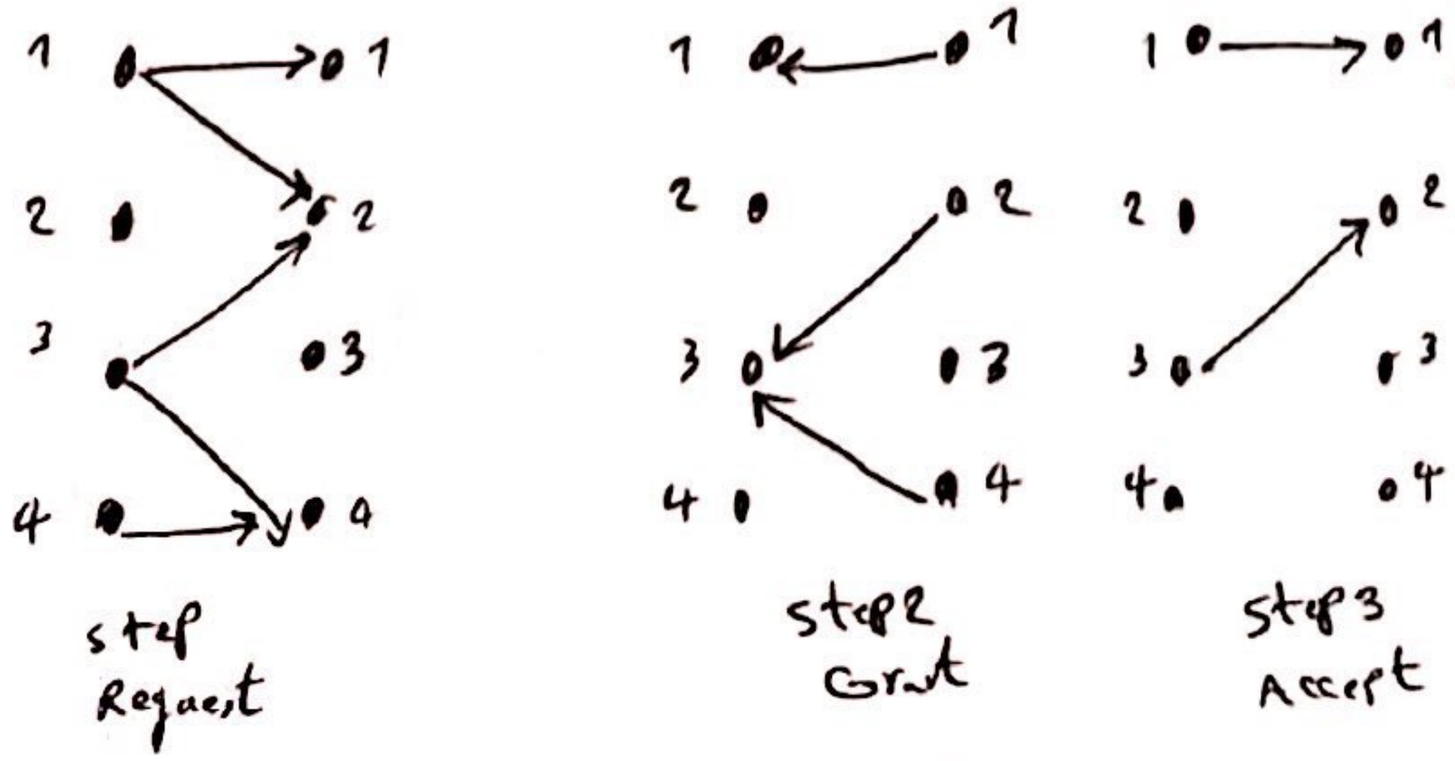
step2: Grant: یک خروجی ۳ حالت کنه داشته باشه

1. هیچ Request ای از هیچ ورودی ای بهش نیاد
2. یک Request داشته باشه
3. چند تا Request بیاد.

اون خروجی هایی که unmatch هستند (مشابه ورودی ها)، در تکرار اول همه خروجی ها unmatch هستند ولی در تکرارهای بعدی کسی که match شده باشند) از بین Request هایی که دریافت کردند، به یکی از Request ها به صورت تصادفی Grant می‌دهد. به صورت رندوم انتخاب می‌کنند.

step3: Accept: یک ورودی کسی از بین Request هایی که داشته، هیچ کدوم شون از طرف خروجی Grant دریافت نکنه، یا یکی شون Grant دریافت کنه یا چند تا شون Grant دریافت کنه. به صورت Random از بین درخواست هایی که Grant دریافت کردند یکی را Accept میکنه. و با این کار ورودی i به خروجی j، match می‌شود و تنها عمل match اتفاق پیوسته این پورت ها match می‌شوند و در تکرارهای بعدی کنار گذاشته می‌شوند.

P.16:



در step2: پورت خروجی ۲، فقط یک Request گرفته پس از بین درخواست هایی که گرفته باید به صورت تصادفی به یکی Grant بده. پس به این یک درخواست Grant میده.

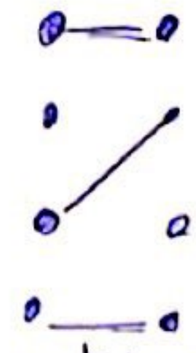
پورت خروجی ۳ به صورت تصادفی از بین ۲ تا درخواستی که داشته به درخواست پورت ورودی ۳ Grant میده.

و همین ترتیب.

Scanned with CamScanner

بعد از نداشتن ۳ تا step ، یک تکرار انجام دادیم و یک matching بدست آوردیم. اگر زمان داشته باشیم میتوانیم یکبار دیگر هم

این مراحل را تکرار کنیم.



matching 2

matching 1

در تکرار دوم پورت‌هایی که match شدن کنار نداشته و سوئیچ (پورت 1 و 3) پورت 2 هم که درخواستی ندارد

پورت ۴ ← درخواست میده به پورت ۳ پورت ۳ هم ← request دارد. مثل Grant میده و Accept

بعد از نداشتن ۳ تا step ، یک تکرار انجام دادیم و matching بدست آوردیم. اگر زمان داشته باشیم میتوانیم یکبار دیگر هم این مراحل را تکرار کنیم.

matching 2 matching 1

در تکرار دوم پورت هایی که match شدن کنار نداشته و سوئیچ (پورت 1 و 3) پورت 2 هم که درخواستی ندارد پورت 4 درخواست میده به پورت 4 هم Request دارد. بعش Grant میده و Accept

ضمیمه ۱۶ (ادامه ی input buffer)

ادامه ی روش های تکرار شونده (maximal matching) گفتیم در هر تکرار می تونن جواب رو بهتر کنند.

اگر بخوایم جواب بهینه رو بدست بیاریم باید ۱ حالت (تمام حالت های جواب رو بررسی کنیم). در روش های maximal

فضای مسئله

ابتدا یک جواب اولیه (در تکرار اول) بدست میاریم بعد در تکرارهای بعدی این جواب رو سعی می کنیم بهتر بشه (و بهتر می شود)

پس تمام روش های maximal ، روش های تکرار شونده هستند. در هر تکرار جواب بهتر می شود. تا جایی که زمان داشته باشیم، تکرار انجام میدیم تا به جواب های بهتر برسیم. طبیعیه که بعد از چند تکرار بالا فرض به جوابی میرسیم که با تکرارهای زیادتر کمتر تغییر کنه.

در روش اول (PIM) ۳ تا step داشت که توضیح دادیم.

گفتیم. در روش PIM گفتیم Grant دادن به درخواست هایی که به یک پورت خروجی میرسه به صورت Random (به صورت تصادفی) میدهد.

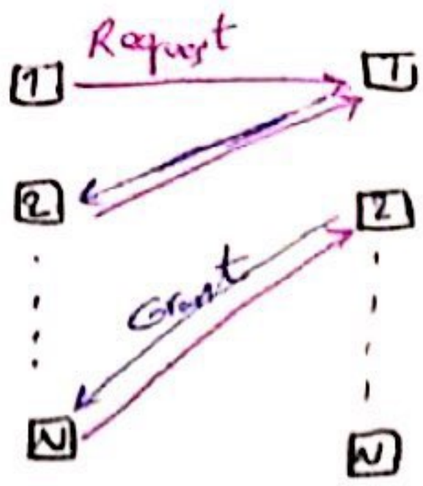
تصادفی بودن این حق رو داره که ما تفاوتی بین (اولویتی) پورت های ورودی قائل نمی شیم و این باعث میشه عدالت برقرار بشه.

چون تمایزی بین پورت های ورودی قائل نمی شویم ← پس سرویسی که ارائه میده Fairness هست.

در step 3 گفتیم ورودی هایی که Request ارسال کردند، ممکنه هیچ Grant ای نگیرن یا فقط یک Grant بگیرن یا چند Grant میدانت که ← باز در اینجا هم گفتیم به صورت تصادفی از بین اون هایی که Grant دارند یکی را انتخاب می کنه و Accept میکنه ← به صورت تصادفی چون انتخاب کرده ← پس به صورت عادلانه انتخاب کرده و S_{ij} ، پر می شود و این پورت ورودی و خروجی کنار گذاشته می شود و در تکرار بعدی بین پورت های ورودی و خروجی که هنوز پر نشدند، سعی می کنیم matching رو با این ۳ تا گام انجام بدیم و matching بهتری بدست بیاریم.

Scanned with CamScanner

در بررسی و گفتن معیارهای scheduling (یا matching) گفتیم روش ارائه شده، باید به صورت سخت‌افزاری قابل پیاده‌سازی باشد. می‌خواهیم بررسی کنیم ببینیم روش PIM به صورت سخت‌افزاری قابل پیاده‌سازی هست یا خیر.



به تعداد پورت ورودی داریم از 1 تا N

" " خروجی " " " " "

- می‌توانیم یک سیگنال Request داشته باشیم که به صورت سخت‌افزاری سیگنال بدهیم. (از ورودی به خروجی‌ها)

- در خروجی هم می‌توانیم به صورت سخت‌افزاری Grant بدهد.

(تا اینجا خیلی پیچیده نیست)

- در ورودی هم می‌شود مداری طراحی کرد که کار Accept را انجام بدهد.

در Grant و Accept ها تا کار انجام بدهیم. (یک کار تو Grant یک کار تو Accept) باید از بین ریکوئست‌هایی که گرفتیم به صورت تصادفی یکی را در (Grant) انتخاب کنیم.

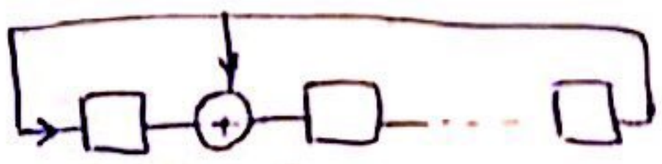
- انتخاب تصادفی ← یعنی ایده‌ی مدار Random Number Generator داشته باشیم RNG که یکی از درخواست‌ها را برای انتخاب کند (این شبیه مداری که در خروجی نیاز داریم)

در ورودی می‌خواهیم از بین Grant ها یکی را به صورت تصادفی انتخاب کنیم ← باز هم نیاز به RNG داریم.

N پورت ورودی داریم. N پورت خروجی ← $N+N=2N$. ۲N مدار تولیدکننده عدد تصادفی باید داشته باشیم

چطوری می‌توانیم یک مدار سخت‌افزاری ها ایجاد کنیم (درست کنیم) که عدد Random تولید کند؟

معمولاً ما با مدار تولید کننده CRC آشنایی داریم. یک سری شیفت رجیستر و بیت‌های XOR داریم. اگر به این مدارات ورودی بدهیم و Reset کنیم با یک تعداد بیت زیاد و تمام حالات ممکن را برود (حالت‌های ثابت سرهم نیفتد) و بازگشتی با 0 کار کند. هر لحظه یک snapshot بگیری یک عدد شبه رندوم می‌گیری.



اگر تمام حالت‌های برود و برود به مدار سخت‌افزاری Random تولید کردیم. (اگر حالت‌مند بودیم برید بخوابند)

هزینه پیاده‌سازی Random Generator ها خیلی بالاست. چون تعداد بیت‌ها باید زیاد باشد. تعداد بیت‌های XOR مناسب (تابع شناخته شده) باشد. بنابراین ۲N مدار RNG درست در عمل قابل پیاده‌سازی هست ولی هزینه بالایی دارد.

Scanned with CamScanner

مثلاً فرض کنیم سوئیچ ما ۵۰۰ تا پورت ورودی و ۵۰۰ تا پورت خروجی داشته باشد. اونوقت باید هزارتا مدار RNG داشته باشیم

هم هزینه سخت افزاری میبره بالا یعنی دارد، هم چون این مدارات باید همیشه در حال کار باشند پس ممکنه انرژی زیادی مصرف کنه.

پس روش RRM یا PIM یا Q روشی که قابل پیاده سازی هست ولی هزینه بالایی دارد و مناسب نیست. میریم سراغ روش بعدی

(iRRM) iterative Round Robin matching ←

تقریباً همون روش قبلی هست با این تفاوت که سعی کرد با یک روش ساده تری Random Generator ها رو حذف کنه.

RNG ها که حذف بشوند هزینه پیاده سازی به شدت کاهش پیدا میکنه.

سوال: چگونه می شود RNG ها را حذف کرد؟ ایده ای که مطرح شد این بود ← همه شیء شمه من قرار نیست که عدالت رو رعایت کنم؟ از دید خروجی میگه من یه تا پورت ورودی دارم.

step1: مشابه PIM ← Request

step2: ~~شیء~~ از دید خروجی شیء من N تا پورت ورودی دارم. میاد یه g_i = grant pointer i نگه میداره. پورت خروجی i ام

برای هر پورت خروجی یک grant pointer نگه میداره: $g_1, g_2, \dots, g_i, \dots, g_N$

این پوینتر شماره پورتی رو نشون میده که اگر چندتا Request گرفتم به پورت خروجی، به Request ای Grant میده که g_i نشون میده. وقتی Grant دادم، g_i رو یه دونه میبره جلو، به پورت آخر که رسیدم دوباره میاد به پورت ورودی اول. به صورت چرخشی Grant میده

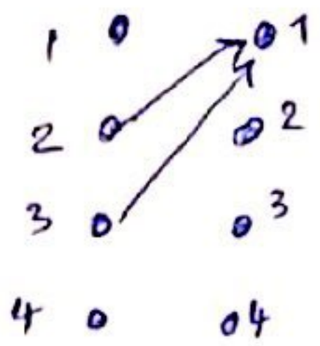
← پس در step2 شیء اگر پورت خروجی ای که هنوز match نشده، از بین Request هایی که داره، به اونی Grant میده که grant pointer داره بهش اشاره میکنه. یا فرض پورتی که g_i داره بهش اشاره میکنه یا نزدیک ترین عدد به اون. مثلاً ممکنه g_i عدد یا روشون به ورودی پورت ورودی i ام Request ای نداشته باشه. پس اون Request رو Grant میکنم که به g_i نزدیک ترین باشه.

بعد از اینکه grant داریم ← $g_i = g_i + 1$ اما چون باید به صورت چرخشی تکرار شود باید mod N باشد.

$$g_i = (g_i + 1) \bmod N$$

Scanned with CamScanner

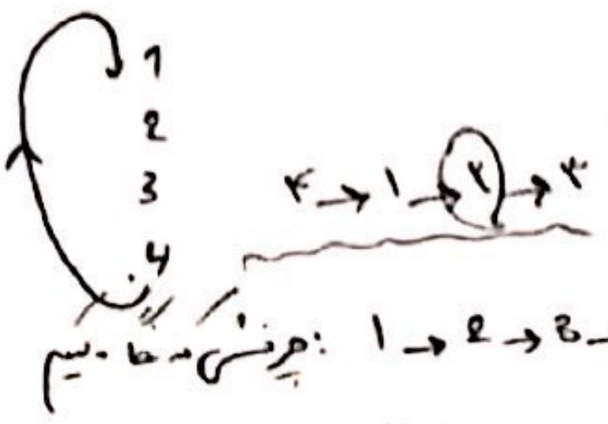
مثال: در پورت خروجی 1، g_1 به پورت ورودی 1 اشاره میکند (مثلا) $g_1=1$



در این حالت چون پورت ورودی 1، درخواستی ندارد پس اون درخواستی انتخاب میشه که به این پورت نزدیک تر باشه ← درخواست پورت 2 بش Grant داده میشود.

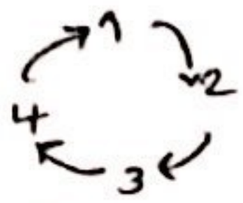
مثال: اگر در پورت خروجی 1، g_1 به پورت ورودی 4 اشاره کنه ← $g_1=4$ باشد (در ساختار برگشت باه)

← صورت چرخشی حرکت میکنیم. میبینیم بین 1، 2، 3، 4 کدوم به 1 نزدیکتره.



پس به درخواست پورت 2، گرنت داده میشود

چرخشی ما: $1 \to 2 \to 3 \to 4 \to 1 \to 2 \to 3 \to 4$



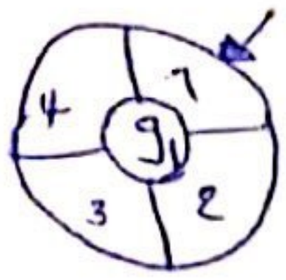
ما به 1 فرصت دادیم $g_1=1$ بود ولی درخواستی نداشت، و به درخواست 2، Grant دادیم، بعد از گرفت دادن $g_1=2+1$ ← $g_1=3$ میشود و از 1 رد میشود. و اگر درخواستی داشته باشه باید بگردیم تا در دور بعدی نوبتش شود.

پس RNG رو حذف کردیم و از grant Pointer استفاده کردیم. و g_i آپدیت میشود. increase میشود به درون بعد از اون پورت ورودی ای که بش گرنت داده شده.

بازی Accept هم هست. اگر یک ورودی بخواد Grant بگیره، یک پوینتر a_i داریم. اولین ورودی رو Accept میکنه که a_i نزدیک تره. هر کدوم رو Accept کنه به درون بعد از اون update میشود.

P.16 مثال از iRRM:

فرض کنیم که در اول کار همه پوینترهای ورودی 1 قرار دارند. در عمل اگر بخوام N mod N بگیرم باید شماره اندیس پورت همون از 0 باشه.



طبق شکل: خروجی 1، به رکوئست گرفته از پورت ورودی 1، g_{b1} هم روی 1 رو نشون میده پس به این درخواست Grant داده میشود

چون به 1، گرنت داد به درون میره جلو

پورت خروجی 2: 2 تا رکوئست گرفته، از پورت ورودی 3 و 4 1، $g_2=1$ هست. پوینتر داره به پورت ورودی 1 اشاره میکنه و درخواست هم داریم از پورت 1. پس بش گرنت میدیم.

چون به 1 گرنت داد:

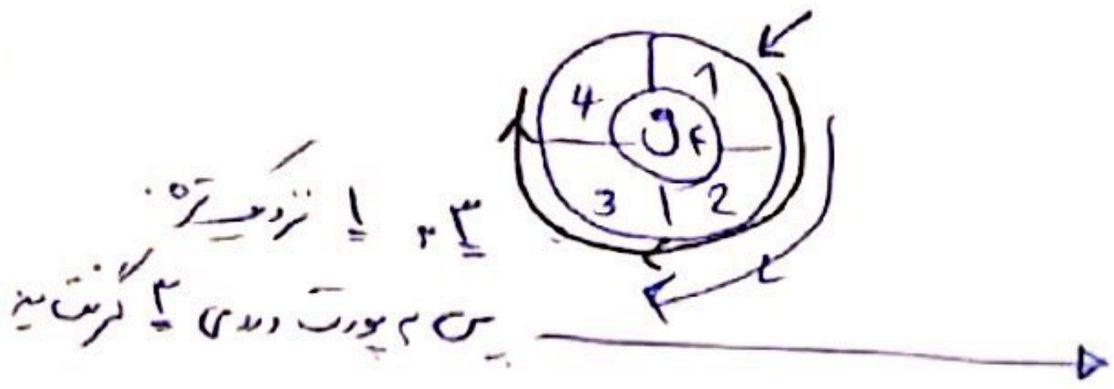
$g_{2_{new}} = 1+1=2$

Scanned with CamScanner

پورت خروجی ۲: هیچ Request ای نداره پس g_2 هیچ تغییری نمی‌کند و هیچ Grant ای داده نمی‌شود.

پورت خروجی ۴: هم از ورودی ۳ و هم از ورودی ۴ درخواست گرفته. $g_4 = 1$ - باید ببینیم کدوم درخواست نزدیک‌تر نسبت به

پورت خروجی ۴



۳ از ۱ نزدیک‌تره.

پس به پورت ورودی ۳ گرنت میده

چون به پورت ورودی ۳ گرنت داد. $g_4 = 3+1$

$g_4 = 4$

در گام بعدی (step 3 : Accept) باید انجام بشه. مشابه عملکرد g، a_j هم عمل میکنه. از بین a_j نزدیک‌تر از بین Grant هایی که داره. (اسلایدهای صفحه‌ی ۲۰)

از همون پورتی که Accept کنه، از اون update میکنه (به اندازه ۱)

این روش از این جهت که پورت‌ها به صورت چرخشی می‌چرخند بنابراین ما نمی‌تونیم بگیم مثلاً ۳ به ۲ اولویت داره و همیشه.

هیچ کدوم اولویتی نسبت به هم ندارند ← بنابراین عدالت در این روش رعایت می‌شود. (اینطور به نظر میاد)

بعد از هر Time slot تمام پورت‌ها سنکرون نیستند (این حالت مشکل که همه به یه پورت (پورت ۱) اشاره نمی‌کنند. همه پورت‌ها یک خروجی رو انتخاب نمی‌کنند یا به یک ورودی Grant نمی‌دهند از حالت سنکرون خارج می‌شود و این کار باعث نمی‌شه که Throughput بزاریم به

اشکال روش iRRM :

در پورت خروجی ۲: به پورت ورودی ۱، گرنت داده شد (اسلاید ۱۸) ولی گرنتی که داد Accept نشد (در پورت ورودی ۱، ۲ تا درخواست گرنت گرفته بودند.) در حالی که پورت ورودی ۱ به ۲، Accept نشد و ولی چون بهش در خروجی Grant داده شد این فرصت ازش گرفته میشه $g_2 = 1+1=2$ چون گرنت بهش داده شد ولی در ورودی Accept نشد حالا باید به صورت چرخشی (سبک هایی که ۵ از پورت ورودی ۱ به پورت خروجی ۲ میخواهند بروند) باید صبر کنند تا دوباره نوبتشان شود

چون Grant، Accept نشد، سرویسی هم داده نشد. ولی ما g_2 را update کردیم و حق ورودی پورت ۱ به خروجی ۲ از بین بردیم. وقتی حق یک پورت رو از بین می‌بریم یعنی عدالت رو از بین می‌بریم. به پورت ۲ نیستم بت Grant دادیم ولی بهش سرویسی داده نشد (بهش نرسیم برد دوباره تو صف داشته.

اشکال روش iSLIP : بدون توجه به اینکه Grant، Accept می‌شود یا خیر، g_i را update می‌کند و این باعث بی عدالتی می‌شود. پورت‌هایی که سلول دارند سرویس نمی‌شند.

Scanned with CamScanner

برای از بین بردن مشکل iRRM روش iterative Round Robin with (iSLIP) پیشنهاد شد.

این روش همون روش iRRM هست با این تفاوت که شیء در step2 : Grant سمتی برای خروجی ای که به g_i نزدیک‌ترین هست Grant میدیم و پوینتر Grant Pointer را موقعی انتزایش میدیم که Grant ما در step3 Accept شود.

و این کار update کردن g_i را فقط و فقط برای تکرار اول انجام میدیم. در تکرارهای بعدی متغیر Grant، Accept هم نمی‌شود اگر Accept شد.

باز هم g_i را update نمی‌کنیم.

در تکرارهای بعدی گفتیم متغیر Grant، Accept شود هم g_i را update نمی‌کنیم. چرا؟ چون وضعیت ما در تکرار اول یک پورت‌هایی انتخاب شدند که سلول داشتند برای یک خروجی ای، ولی در matching قرار نگرفتند. در تکرار دوم چون خبر نداریم بنابراین update نمی‌کنیم.

update g_i فقط در اولین تکرار در صورتی که Grant، Accept شود انجام می‌شود.

step3 مثل قبل. g_j از نزدیک‌ترین ها انتخاب می‌شود، update می‌شود.

P.23 مثال باشکل (همون مثال قبلی که در iRRM بود رو با این روش بررسی کنیم)

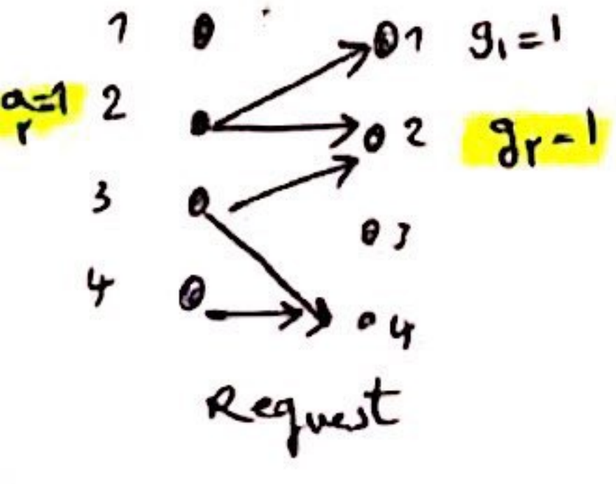
شاه توضیحات قبلی، grant ها رو میدیم. اما update رو در فاز Grant دادن انجام نمی‌دیم. صبر می‌کنیم تا ببینیم تکلیف Grant هایی که داریم چی میشه. اونایی که Accept شدند g_i update می‌شود. بنابراین موقعی که سرویس دادیم g_i را آپدیت می‌کنیم. در حالتی که سرویس ندادیم، اولویت اون پورت ورودی رو از بین نمی‌بریم.

این روش عدالت را نسبت به روش قبلی بهتر برقرار می‌کند بین پورت‌های ورودی و خروجی، بهتر تامین می‌کند. حتی میشه Throughput رو هم بهتر کنه.

روش بعدی: FIRM : FCFS First Come First Service In Round-Robin Matching

اومدن روش ISLIP رو یه ذره بهتر کرد. روش قبلی سلیقه‌ی گرفتن که دادم اگر Accept نشدن g_i رو آپدیت نمی‌کنم.

با توضیح این روش متوجه بهبود شیم.



FIRM: هر سلولی که زودتر وارد سوئیچ شد زودتر هم خارج بشه.

فرض کنیم grant pointer در 1 را نشان میدهد در پورت ورودی 2

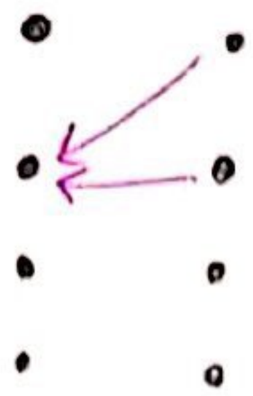
Scanned with CamScanner

در ISLIP دو پورت خروجی ۲ که $g_2=1$ و از پورت ورودی ۲ و ۳ درخواست دارد، کدوم رو برای Grant دادن انتخاب میکرد؟ ۲ رو

چون به $g_2=1$ نزدیک‌تره

پورت 1، هم که از پورت ورودی 2 درخواست دارد و $g_1=1$ هست پس در پورت خروجی 1 هم به پورت ورودی 2 گرنت

داده شده چون به $g_2=1$ نزدیک‌تره



پورت ورودی ۱، ۲ تا Grant میگیره. کدوم Accept میشه؟ 1. چون $a_2=1$

و سلول از پورت ۲ به خروجی 1 میره.

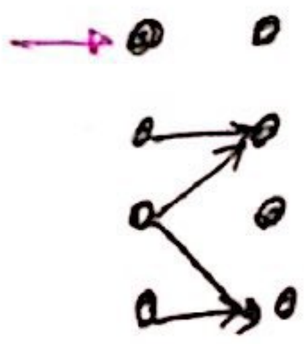
خروجی ۲ گرنت داده به ورودی ۲ ولی گرنتش Accept نشد. در روش ISLIP،

g_2، update نمی‌شود و $g_2=1$ باقی می‌ماند. پس اولویت را برای پورت 1 نگه میدارد درحالیکه

پورت 1 اصلاً سلولی برای پورت خروجی ۲ نداره. اما اگر در مثال این Time slot، سلولی به پورت ورودی 1 بیاد به

سمت خروجی ۲، $g_2=1$ هست و پس سرویس داده می‌شود درحالیکه در پورت ۲ و ۳ از قبل برای پورت خروجی ۲ سلول

منتظر Grant گرفتن بودن. یعنی مشکله FCFS رعایت نمی‌شود.



در step2:

روش FIRM مشکله که اگر یک Grant هم Accept شد که update میشه پس به پورت دیگه

بعد از اونی که سرویس داده شد.

اگر Grant دادیم و Accept نشد، g_i رو میبریم روی اونی که بهش Grant داده بودیم، چون پورت قبل از اینی که بهش گرنت

بدیم سلولی نداشته پس باید اولویت براش نگه داریم

مثل روش قبلی هست فقط در update کردن g_i متفاوته. درحالتی که Accept نشد و گرنت داده بودیم، g_i میره جلو جایی

که Grant دادیم و Accept نشده بود. این کار رو میکنه تا FCFS رو رعایت کنه.

در step3: a_j مثل حالت‌های قبلیه. نزدیکترین به a_j رو Accept میکنه ولی فقط در گام اول a_j رو update میکنه

در گام‌های بعدی update نمی‌کنه که FCFS رو رعایت کرده باشه. چون در iteration های بعدی نوبت هم میریزه و برای همین

update نمی‌کنیم.

P.27 ← مثال از FIRM

step1: گراف رکوئست را طبق درخواست‌های موجود در صف‌ها رسم می‌کنیم.

step2: g_i ها را مقداردهی می‌کنیم. ابتدا همه روی 1 هستند. $g_i=1$

پورت خروجی 1، فقط یک درخواست داره اونم از پورت ورودی ۳ و $g_1=1$ به کی Grant میده؟ به پورت ورودی ۳

چون تقاضای رکوئست رو داره

Scanned with CamScanner

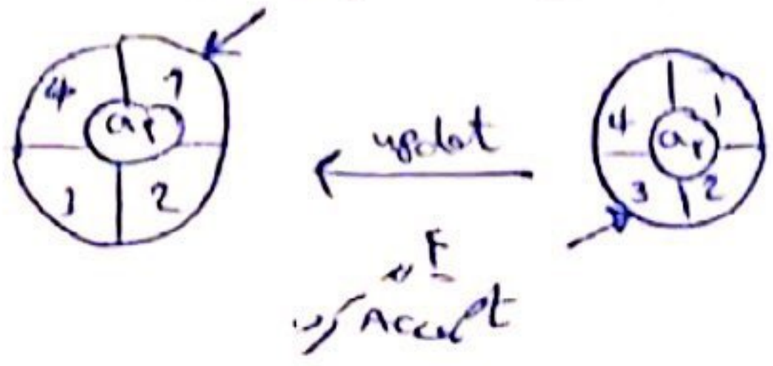
به همین ترتیب پورت خروجی ۱ و ۳ و ۴ هم گرنت ها شون طبق قاعده مشخص میشن اما به g_i ها دست نمیزنن فقط update رو میزنن

میزان بعد از فاز Accept. ← اگر Grant، Accept شد، یه دونه پیشنهاد جدید از اونی که بهش سرویس دادیم.

نکته: g_i رو update می‌کنیم و پیشنهاد می‌دیم به اونی که بهش Grant داریم.

P.27

step3: در ورودی ۲، $a_2=2$ و پورت‌های خروجی ۱ و ۴ بهش Grant دادند. ← پورت خروجی ۴ Accept شد. ← پوینتر a_2 ← $a_2=1$



پورت ۴: g_4، Grant ای که دارد Accept شد ← پس $g_4=1$

پورت ۲: Grant دارای Accept نشد. $g_2=2$ ← چرا؟ Grant ندارم در حالی که $g_2=1$ بود؟ چون پورت ورودی ۱ سلوی نداشت به ۲، Grant دادیم.

* به همین ترتیب برای بقیه پورت‌ها هم تحلیل می‌کنیم *

در این روش یکبار که step 1, 2, 3 را انجام دادیم. اگر زمان داشتیم می‌توانیم یک iteration دیگه هم داشته باشیم.

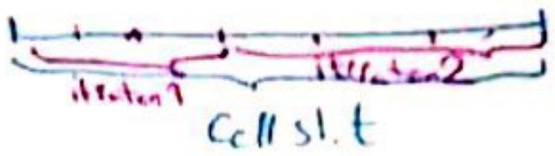
پورت‌هایی که میچ شدند را کنار می‌گذاریم. و ....

Time slot بعدی رو داره شبیه:

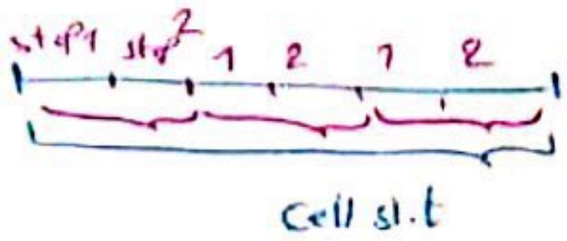
اسلایدی ۳۰ ۲ → ۴ ، ۳ → ۳ اینها متصل شدند

روش Dual Round Robin matching (DRRM):

الگوریتم‌هایی که تا اینجا توضیح دادیم تو روش maximal در هر تکرار ۳ تا step داشتند. الگوریتم DRRM، در هر تکرار ۲ تا step داره.



نمیتوان روش‌ها هر چقدر که زمان بیشتر اجازه بده الگوریتم را تکرار می‌کنیم و جواب بهتر پیدا می‌کنیم.



در DRRM چون تعداد step ها کمتر است بنابراین تعداد تکرارها هزینه بیشتر باشد و matching بهتری خواهیم داشت

۲ گامی که در DRRM داریم ← Request، Grant

step3 نیز Accept رو حذف کرده: در تصویر Accept رو حذف کنیم که نخواهیم از بین چند تا Grant، Accept کنیم. یعنی هر Grant ای رو بخواهیم Accept کنیم.

Scanned with CamScanner

برای اینکه هر Grant ای رو تبدیل به Accept کنیم ← هر ورودی بیشتر از یه Grant نباید بگیره ← پس هر ورودی باید بیشتر از یک Request بده. بنابراین اگر فقط یک Request بده، یا Grant میگیره یا نمیگیره. اگر Grant بگیره، حتماً Accept میشه.

step1: Request

برای ورودی هایی که match نشدند. هر ورودی به یک خروجی که صف غیرخالی داره، درخواست میده. با ترتیب RR یک پوینتر میذاریم به اسم Request ptr: روی همه درخواست هایی که برای خروجی ها داره ← یکی رو به خروجی درخواست میده.

step 2: Grant

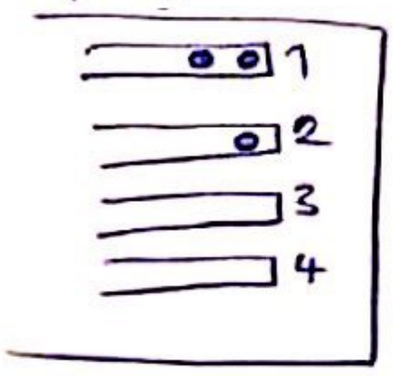
یک خروجی میاد از پوینت های ورودی مختلفش Request بگیره. سه یکی از ورودی ها رو انتخاب میکنه طبق g_i به هر کسی که Grant بده حتماً Accept میشه. و g_i رو هم update میکنه.

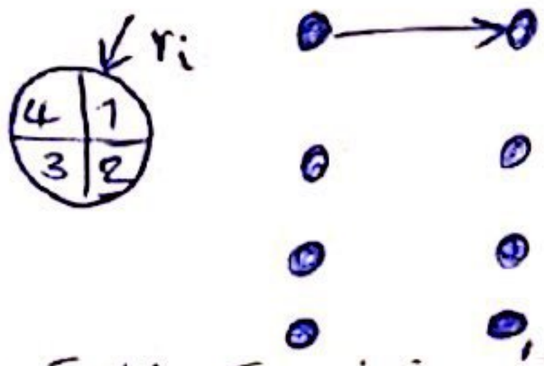
توضیح درهمی بلد

(*) ورودی دیگه Request میده امتیاز اینکه Request اش Grant بشه پایین هست. اگر چند تا Request میدا امتیاز Grant گرفتنش بیشتر میشه. (این فکر به ظاهر درسته اما باطناً اینطوری نیست. ابتدا ممکنه اینطوری باشه اما بعد چند گام دارد slot Time، g_i و R_i از بابت شکستن خارج می‌شوند و تقریباً رکورد ها حتماً Grant می‌شوند.

صفحه بعدی توضیح داده می‌شه

input 1





P.34 : Grant , Request آماده داریم.

↓

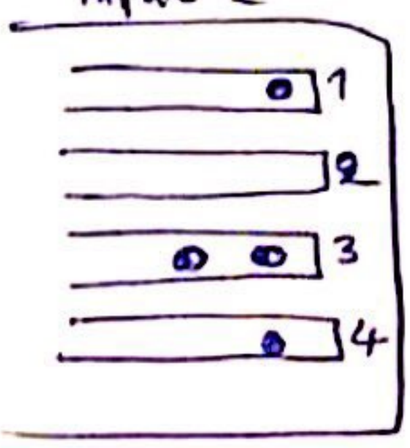
Request ptr

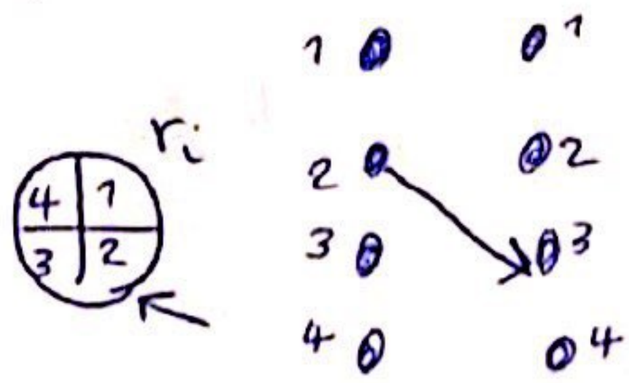
↓

شبیه یکی باید رکورد بدم. فقط به یکی از خروجی ها که صفش غیرخالی هست درخواست میده

در ورودی 1، $R_1=1$ و صف خروجی 1 غیرخالی هست پس Request میدم

input 2





در ورودی 2، $r_2=2$ هست ولی صف 2 خالیه پس سراغ نزدیک ترین رینگ 3 به رکورد ما 3 میرسه خروجی

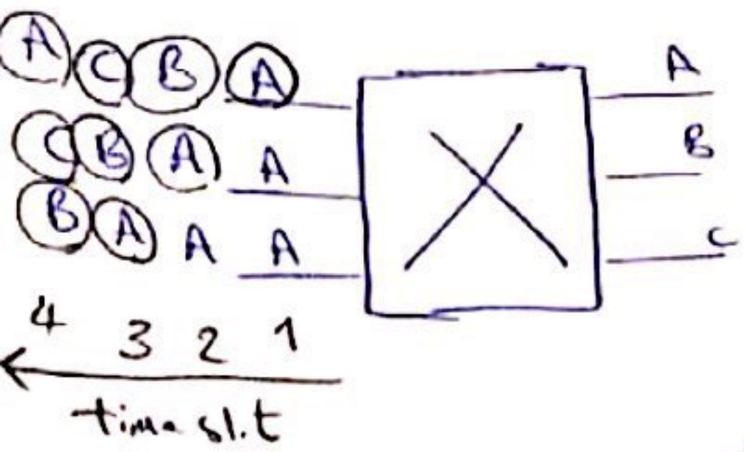
میشه که Request میدیم. لازم نیست فقط یک Request از R_i ها رو update نمیکنیم تا تکلیف Request ها مشخص بشه

بعد از مرحله ی Grant، اگر Request به دادش Grant گرفت، update میکنیم R_i رو

Grant که بگیریم، اتوماتیک Accept میشه و R_i رو update میکنیم.

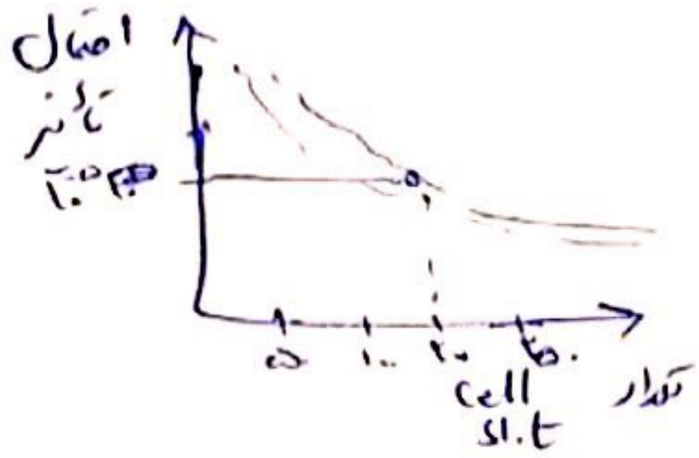
Scanned with CamScanner

Desynchronization effect



بعد از نوشت ۴ تا Time slot ها هستیم Request های پورت های فرودی مختلف می روند و می توانیم بیشترین matching و تداخل خوبی را با هم برسیم

برای نشان دادن اینکه DRRM عملکرد بهتری دارد برابر iSLIP و FIRM شبیه سازی انجام شد که نتایج در اسلایدهای 35 روی نمودار آورده.



احتمال تأخیر بیشتر به سلول که وارد صف شده باید انتظار ۲۰۰ تا Time slot (بلند) تو سوئیچ میمونه رو حساب میشه.

هرچی احتمال تأخیر متوسط کمتر باشه ← Throughput بیشتر.

از تعداد صف های فرودی داریم با کمک لیتل Little

Little → $\lambda . T$ = تعداد متوسط × Response Time فرودی

اگر تعداد متوسط را ثابت بگیریم پس هرچی λ ↑ پس T ↓

بررسی نمودار:

2(iSLIP) بیشتر iSLIP ای که speedup = ۲ داره.

در نمودار برای 2-(iSLIP) : تأخیر ۲۰۰ تا Time slot ← احتمالش 10^{-5} است.

برای DRRM این احتمال کمتر از 10^{-5} است. احتمال کوچکتر پس DRRM بهتر.

برای RR FIRM 10^{-4} است و بدتر

نتیجه: بیشتر عملکرد DRRM به دلیل اثر غیرهمگرن سازی از نظر تأخیر و Throughput بهتر از روش های دیگه عمل کرده.

Scanned with CamScanner