10

YEARS

07.NOV.2024

ELEVATING THE DEVELOPERS’ COMMUNITY

devday
GET INSPIRED

PLATINIUM

ONIRYX

GOLD

delaware

GAMING1

GENESIS CONSULT
IT Consultancy Services

PARTNER

birdit

DevApps.be

meet
innovate
create

SENSE
OF TECH

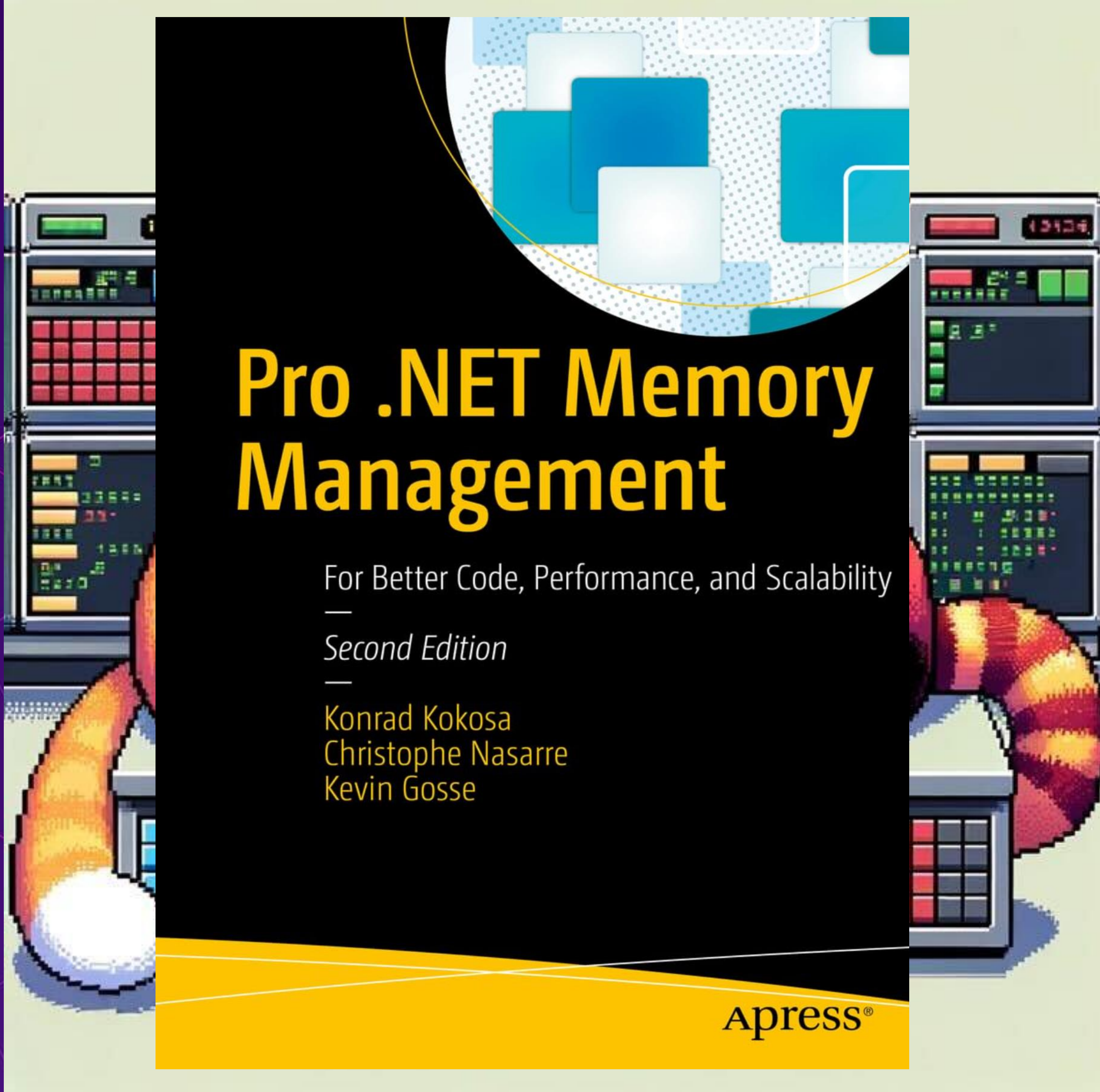
Sparkle

devday
GET INSPIRED

Quoi de neuf pour la gestion de la mémoire en .NET ?

Kevin Gosse
@KooKiz
Christophe Nasarre
@chnasarre

DATADOG



Pro .NET Memory Management

For Better Code, Performance, and Scalability

—

Second Edition

—

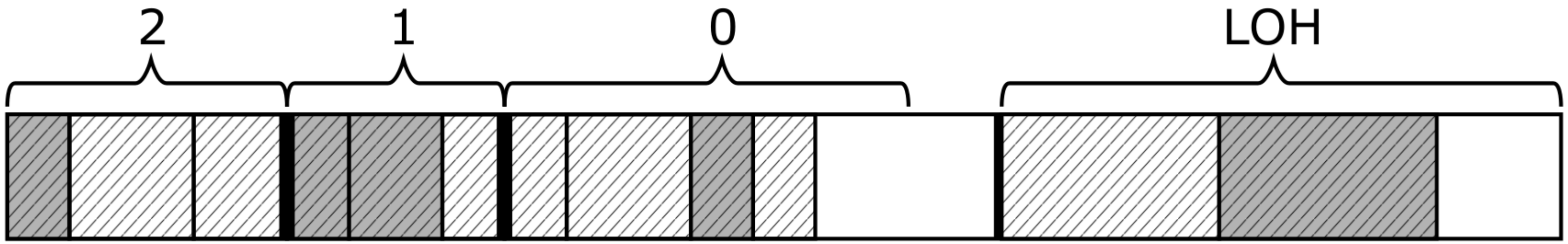
Konrad Kokosa
Christophe Nasarre
Kevin Gosse

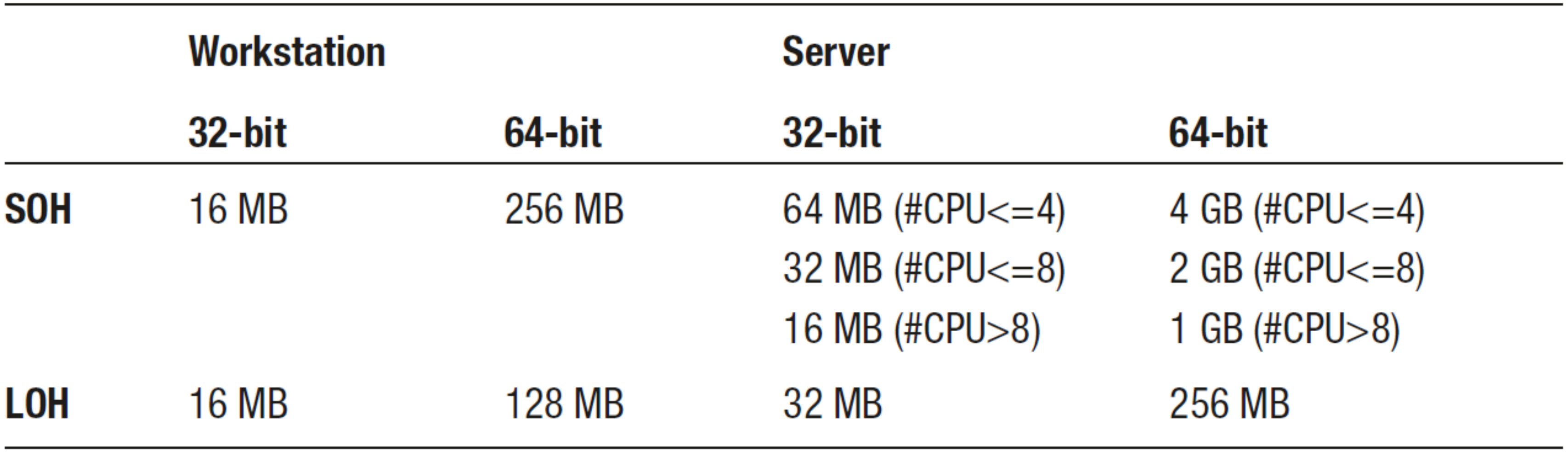
Apress®

Reminder: gen0, gen1, gen2 and LOH

- High level view of the memory layout
  - 3 generations and the LOH

Segments: Ephemeral, gen2 and LOH

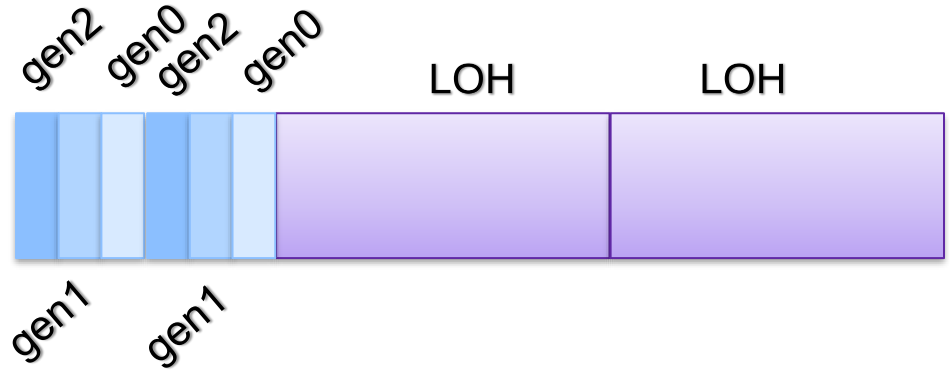




	Workstation		Server	
	32-bit	64-bit	32-bit	64-bit
SOH	16 MB	256 MB	64 MB (#CPU<=4) 32 MB (#CPU<=8) 16 MB (#CPU>8)	4 GB (#CPU<=4) 2 GB (#CPU<=8) 1 GB (#CPU>8)
LOH	16 MB	128 MB	32 MB	256 MB

Regions: LOH and generations

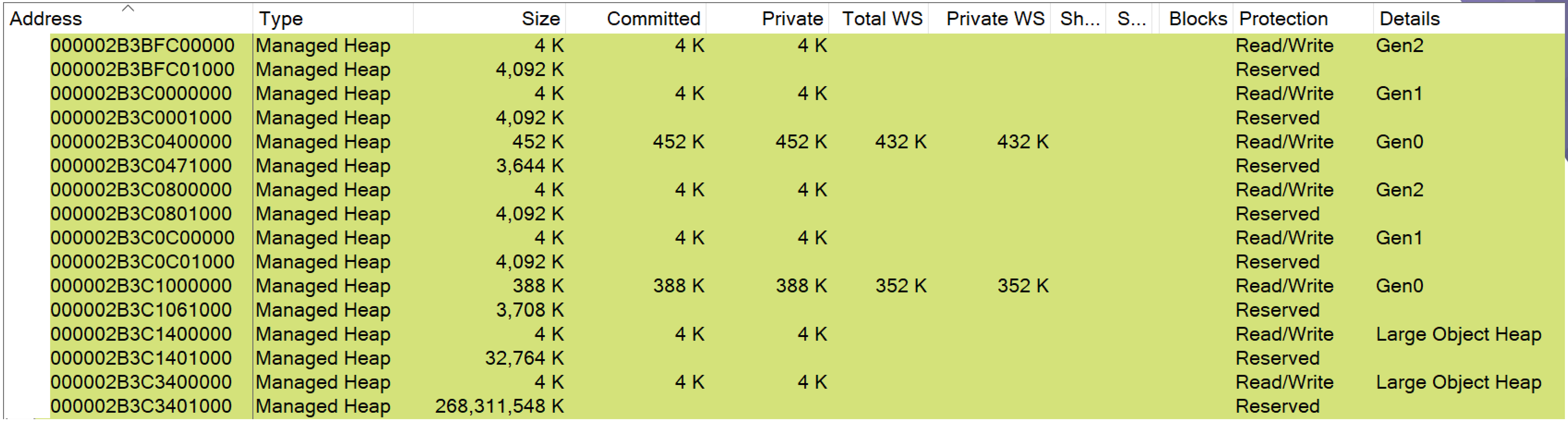
Example: Server mode initialization on a 2 cores machine



SOH : 4 MB
LOH : 8 x 4 MB = 32 MB

How to visualize the heap?

- VMMap for live applications
  - support .NET Core 5+ since v3.4



Address	Type	Size	Committed	Private	Total WS	Private WS	Sh...	S...	Blocks	Protection	Details
000002B3BFC00000	Managed Heap	4 K	4 K	4 K						Read/Write	Gen2
000002B3BFC01000	Managed Heap	4,092 K								Reserved	
000002B3C0000000	Managed Heap	4 K	4 K	4 K						Read/Write	Gen1
000002B3C0001000	Managed Heap	4,092 K								Reserved	
000002B3C0400000	Managed Heap	452 K	452 K	452 K	432 K	432 K				Read/Write	Gen0
000002B3C0471000	Managed Heap	3,644 K								Reserved	
000002B3C0800000	Managed Heap	4 K	4 K	4 K						Read/Write	Gen2
000002B3C0801000	Managed Heap	4,092 K								Reserved	
000002B3C0C00000	Managed Heap	4 K	4 K	4 K						Read/Write	Gen1
000002B3C0C01000	Managed Heap	4,092 K								Reserved	
000002B3C1000000	Managed Heap	388 K	388 K	388 K	352 K	352 K				Read/Write	Gen0
000002B3C1061000	Managed Heap	3,708 K								Reserved	
000002B3C1400000	Managed Heap	4 K	4 K	4 K						Read/Write	Large Object Heap
000002B3C1401000	Managed Heap	32,764 K								Reserved	
000002B3C3400000	Managed Heap	4 K	4 K	4 K						Read/Write	Large Object Heap
000002B3C3401000	Managed Heap	268,311,548 K								Reserved	

DPAD

DPAD

- Dynamic Promotion and Demotion
- `DOTNET_GCEnableSpecialRegions=1`

Experimental feature!

- Rule 1: Sweep in plan (SIP)
  - What is sweep?
  - What is plan?

DPAD

Mark

Plan

Compact

Sweep

Relocate

DPAD

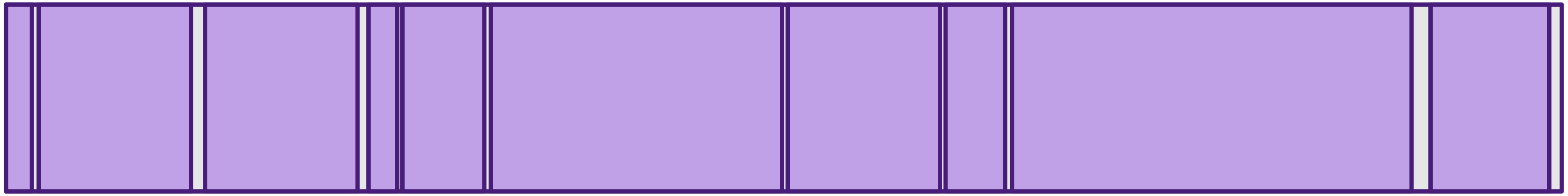
Mark

Plan

Compact

Sweep

Relocate



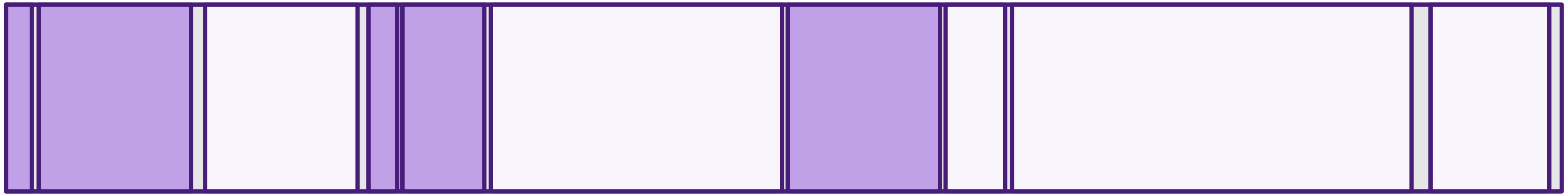
Mark

Plan

Compact

Relocate

Sweep



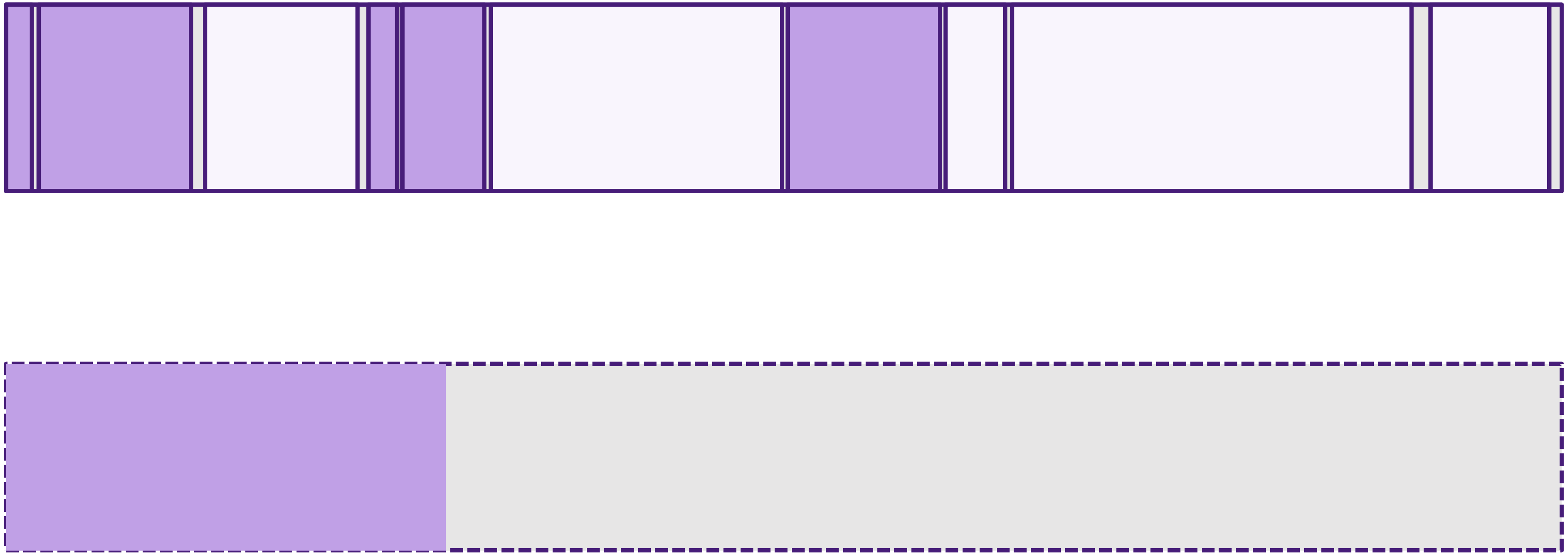
Mark

Plan

Compact

Relocate

Sweep



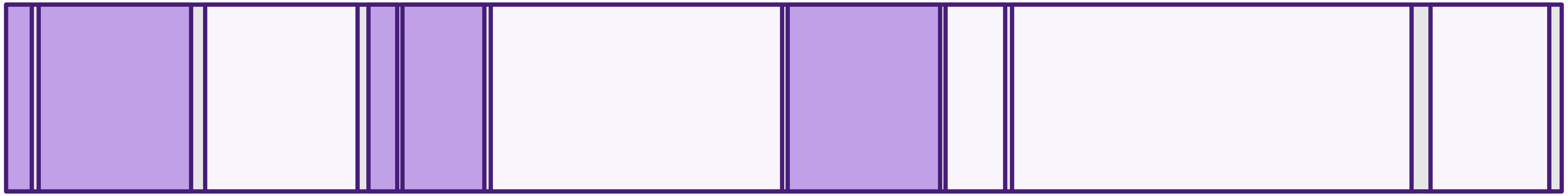
Mark

Plan

Compact

Relocate

Sweep



DPAD

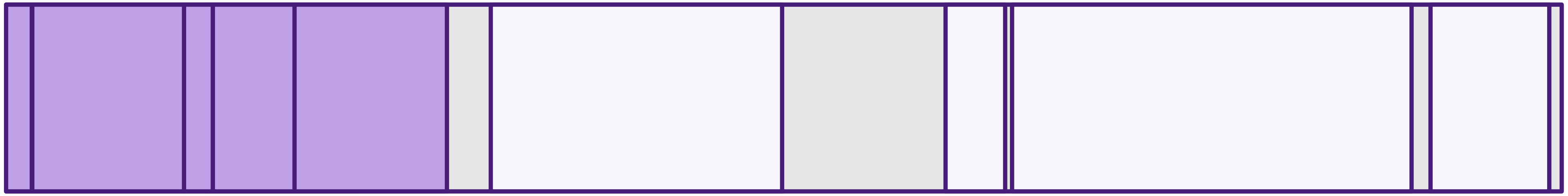
Mark

Plan

Compact

Relocate

Sweep



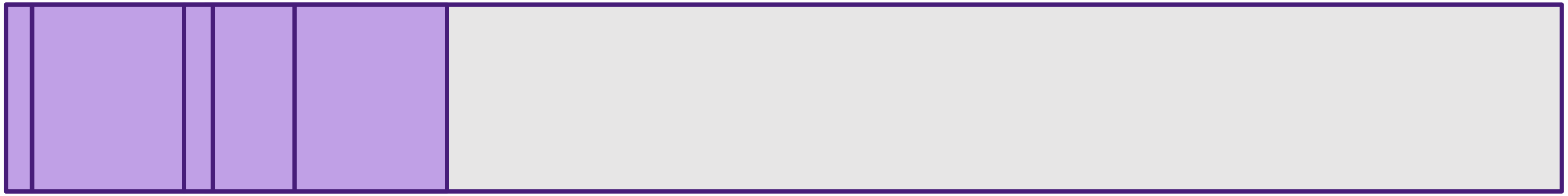
Mark

Plan

Compact

Sweep

Relocate



DPAD

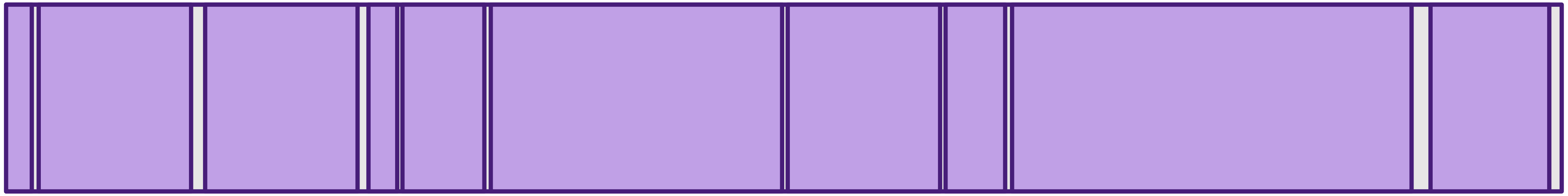
Mark

Plan

Compact

Sweep

Relocate



DPAD

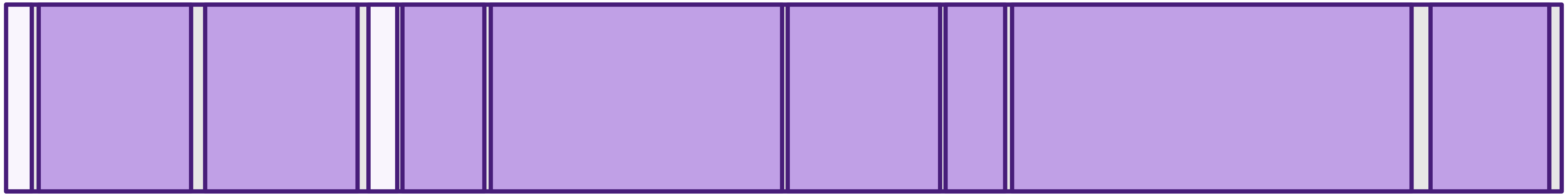
Mark

Plan

Compact

Relocate

Sweep



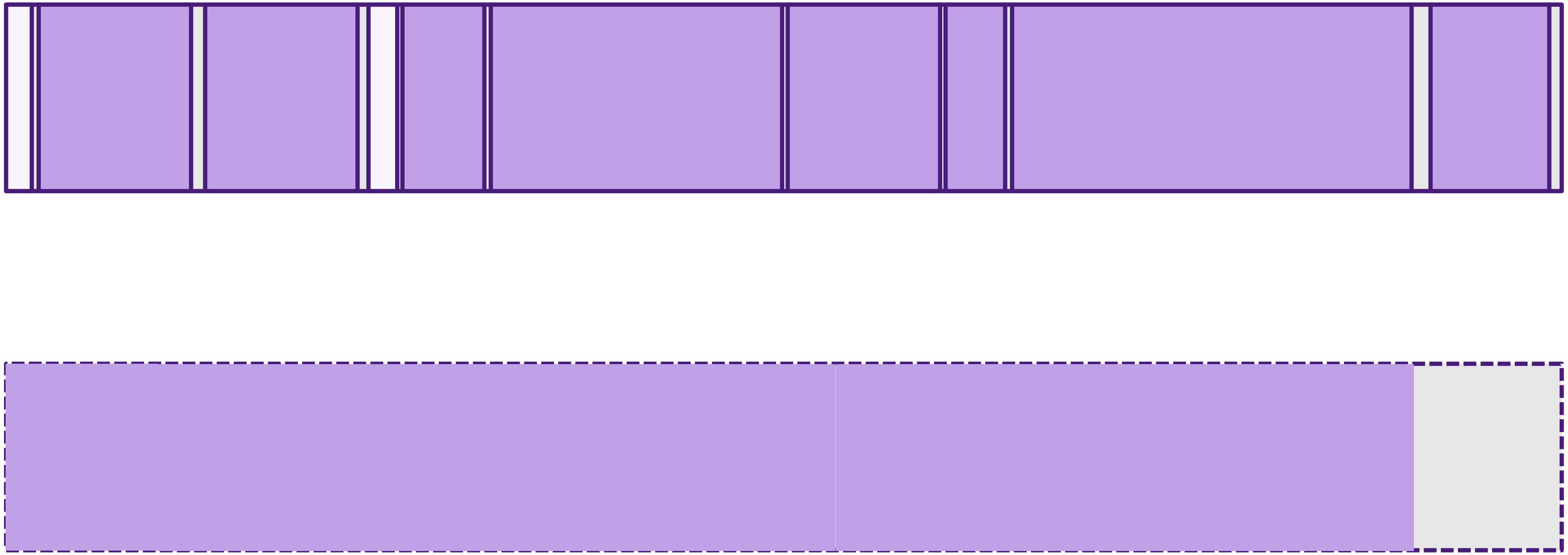
Mark

Plan

Compact

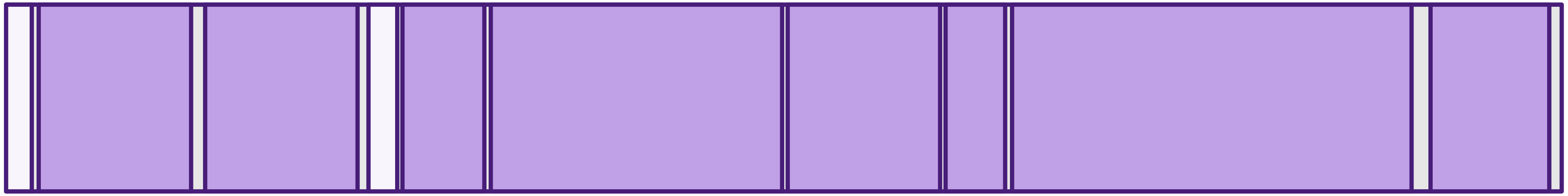
Relocate

Sweep



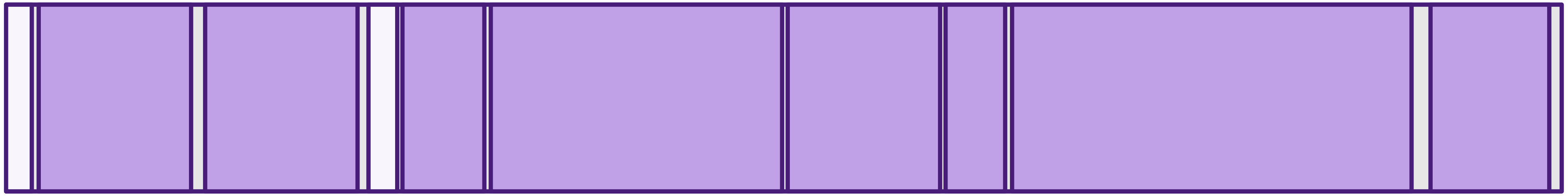
Mark Plan Compact Relocate

Sweep



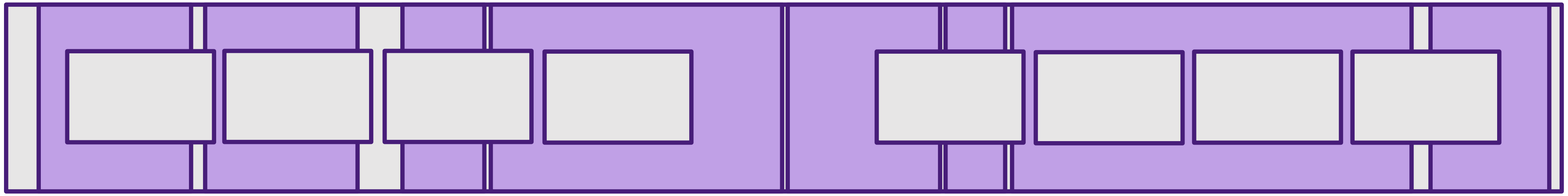
DPAD

- Rule 1: Sweep in plan (SIP) : during plan phase, if survival rate > 90%

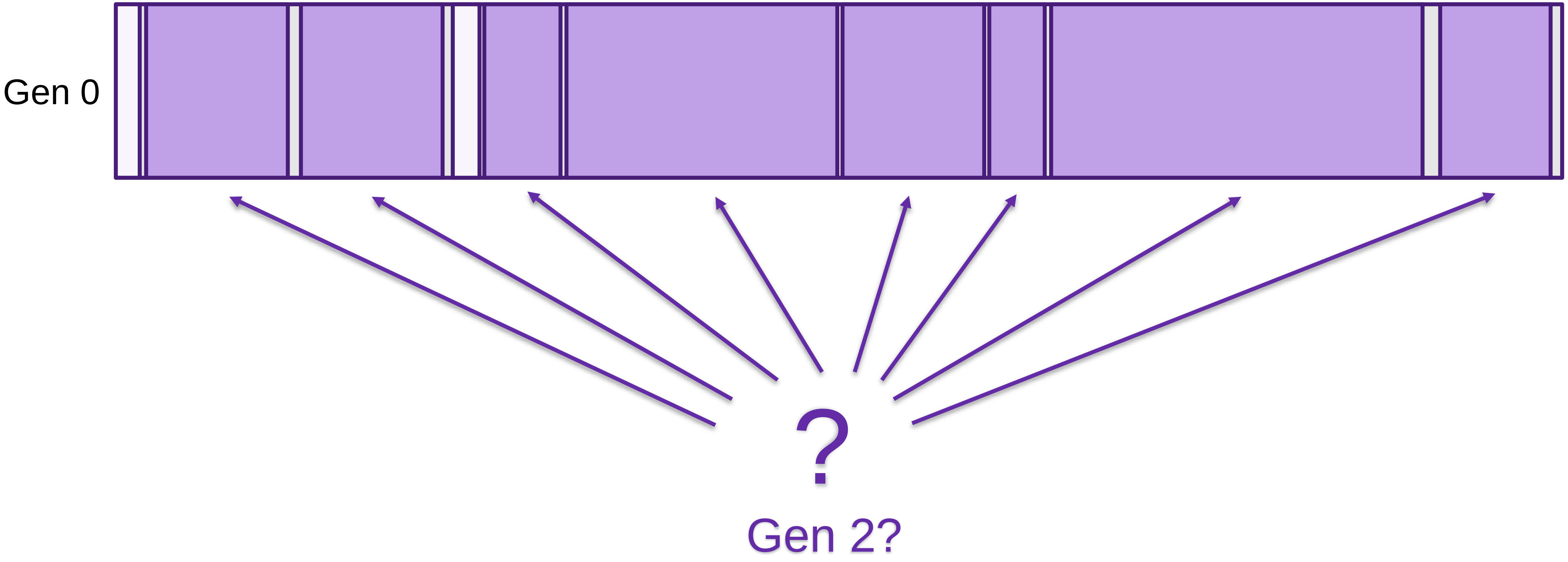


DPAD

- Rule 1: Sweep in plan (SIP) : during plan phase, if survival rate > 90%

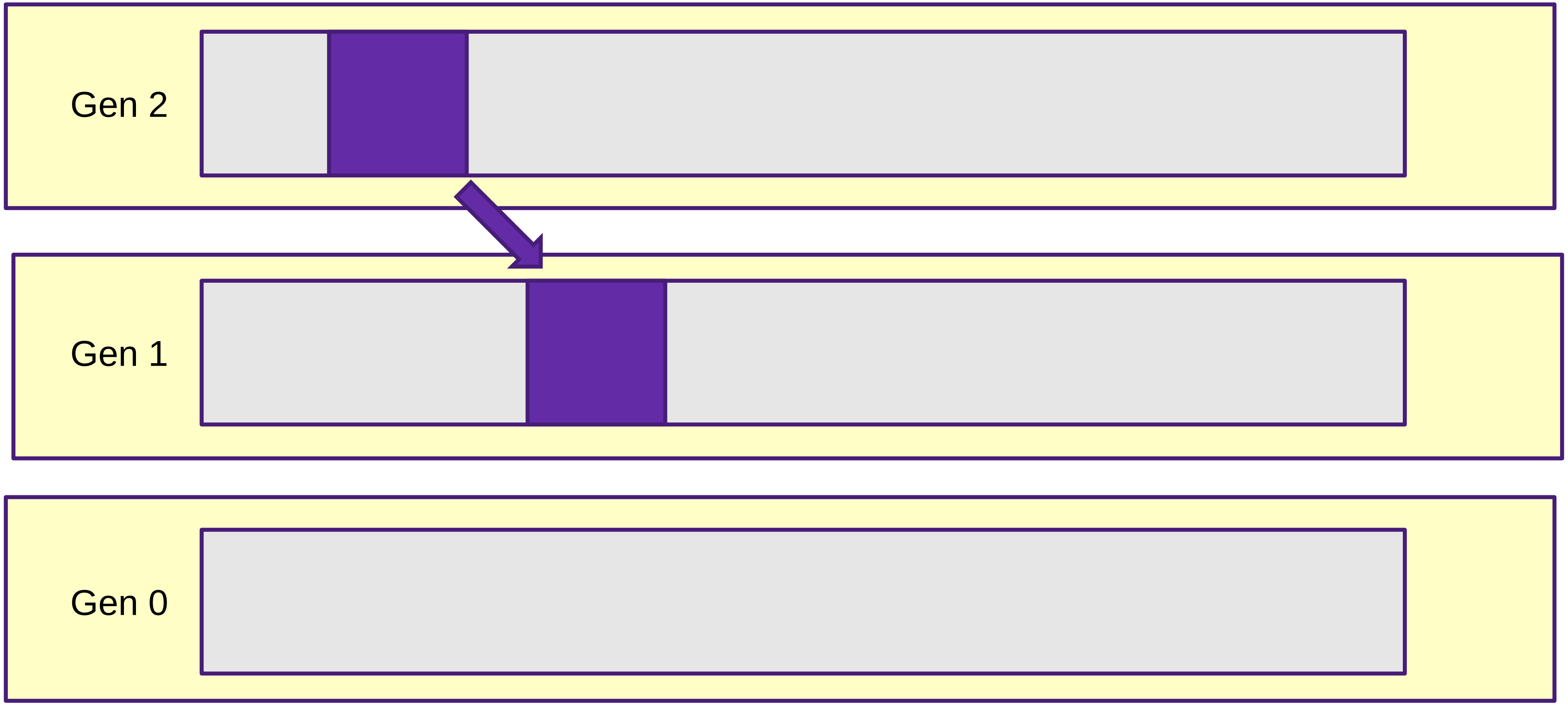


- Rule 2: Dynamic promotion



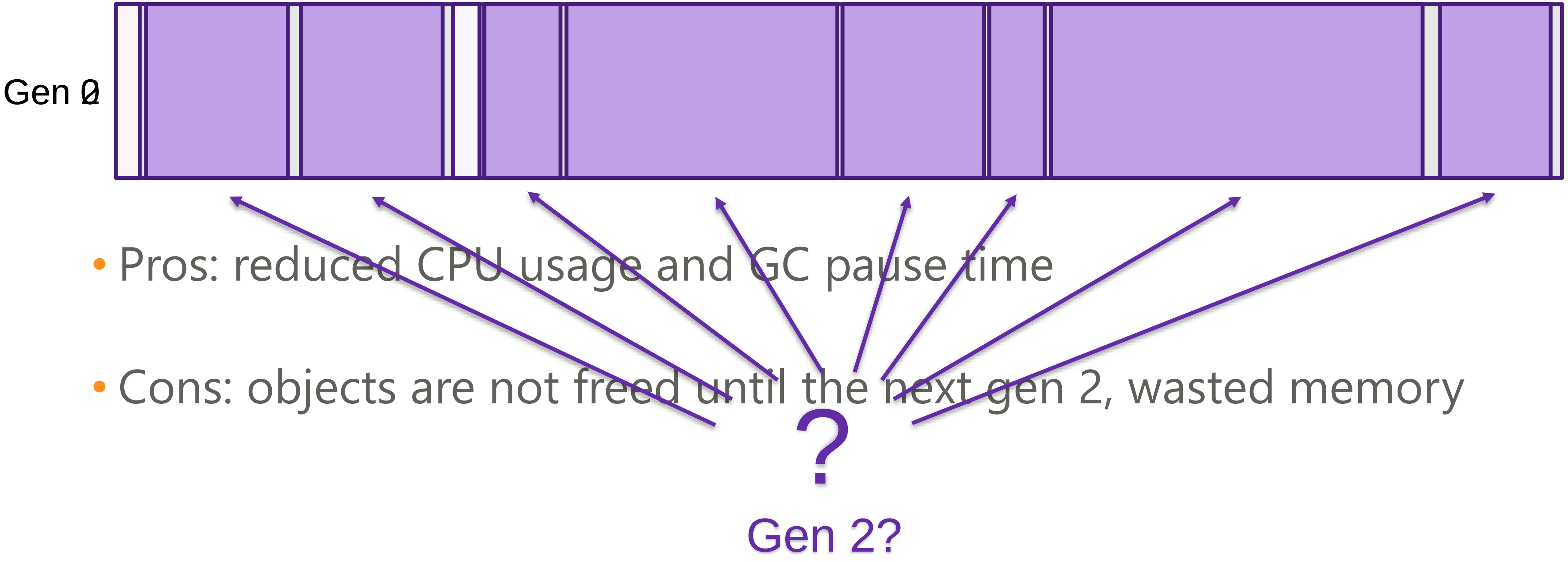
DPAD

- Rule 2: Dynamic promotion



DPAD

- Rule 2: Dynamic promotion : if gen 2 roots > 90%



- Pros: reduced CPU usage and GC pause time
- Cons: objects are not freed until the next gen 2, wasted memory

DPAD

- `DOTNET_GCEnableSpecialRegions=1`
- Takes advantage of regions to reduce the amount of work done by the GC
- Increases the working set
- Broken on .NET 7/8

In gen0 or LOH? That is the question…

In gen0 or LOH? That is the question…

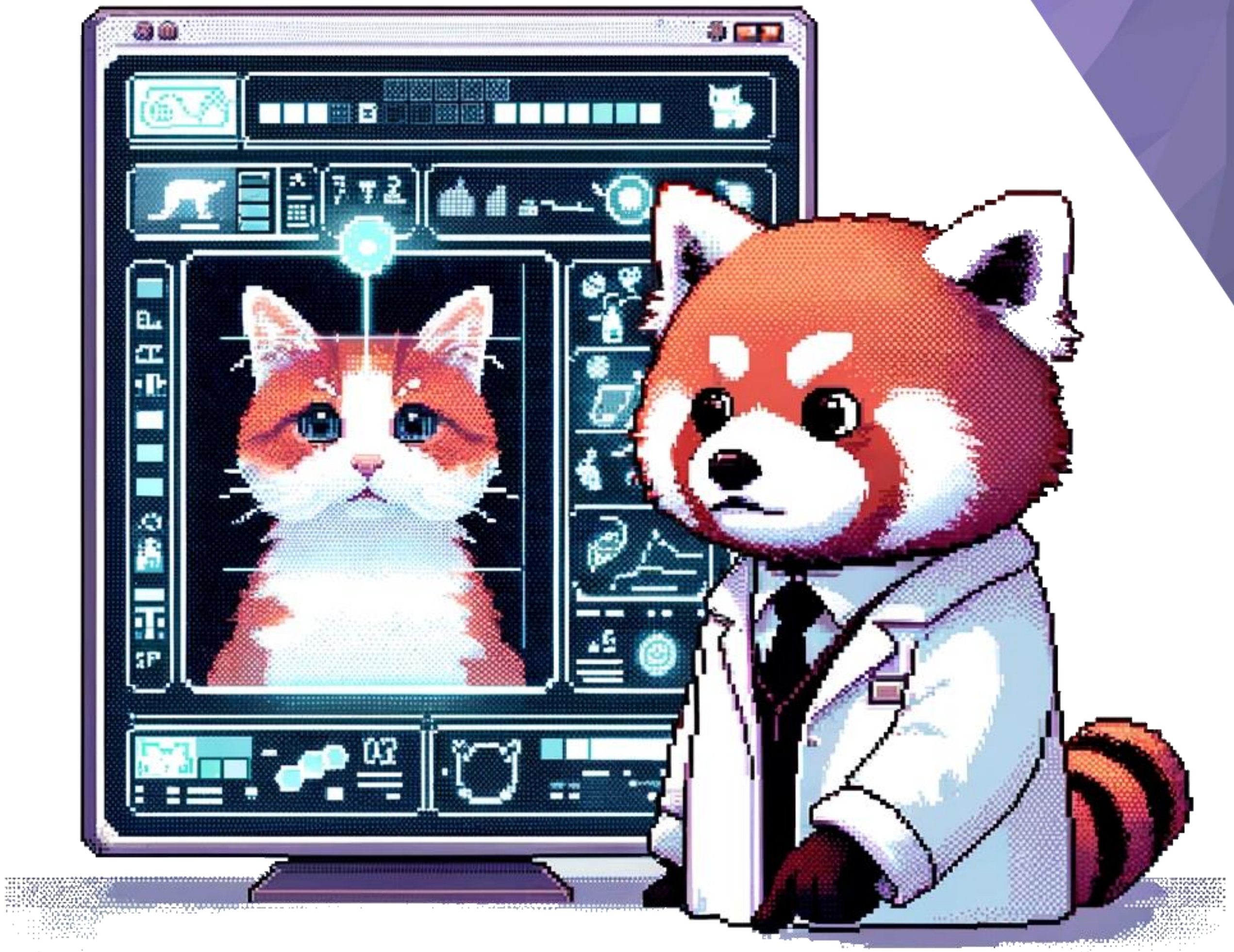
- Large Object Heap
  - possible to optimize large allocations scenario

- Example of threshold impact

```
řsîŵǻţê șţǻţîç ŵộîđ Aľľộçǻţê îŋţ çộụŋţ
    čỳţê   čụǧǧês   ŋụľľ
    ǧộs  îŋţ î   。  î   çộụŋţ   î
        čụǧǧês   ŋêx čỳţê ¨_。。。
        čụǧǧês  。    čỳţê î
```

Threshold	Duration	gen2	gen1	gen0
85000	1020 ms	2702	2702	2702
102400	424 ms	0	1	1010

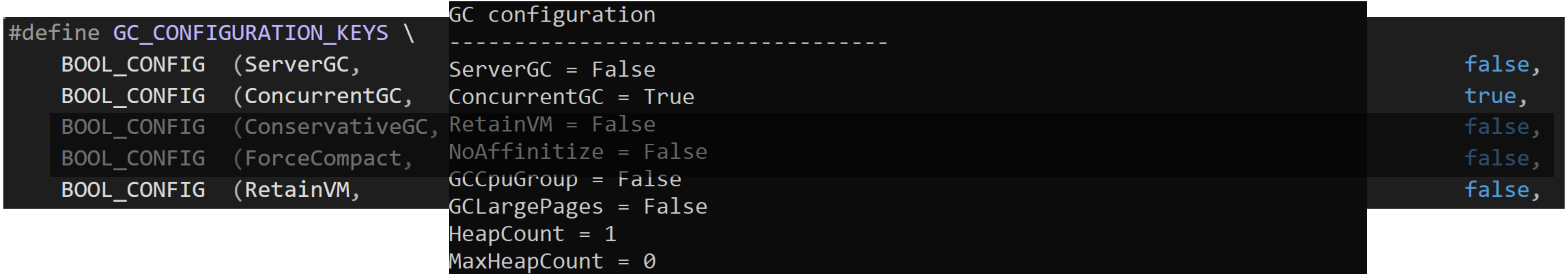
How to get GC configuration?



How to get GC configuration?

- What is the GC configuration?
  - `GC.GetConfigurationVariables()`
  - …but not `DOTNET_GCLOHThreshold`

src/coreclr/gc/gcconfig.h



```
#define GC_CONFIGURATION_KEYS \
    BOOL_CONFIG  (ServerGC,                                                          false,
    BOOL_CONFIG  (ConcurrentGC,                                                      true,
    BOOL_CONFIG  (ConservativeGC,                                                    false,
    BOOL_CONFIG  (ForceCompact,                                                      false,
    BOOL_CONFIG  (RetainVM,                                                          false,
```

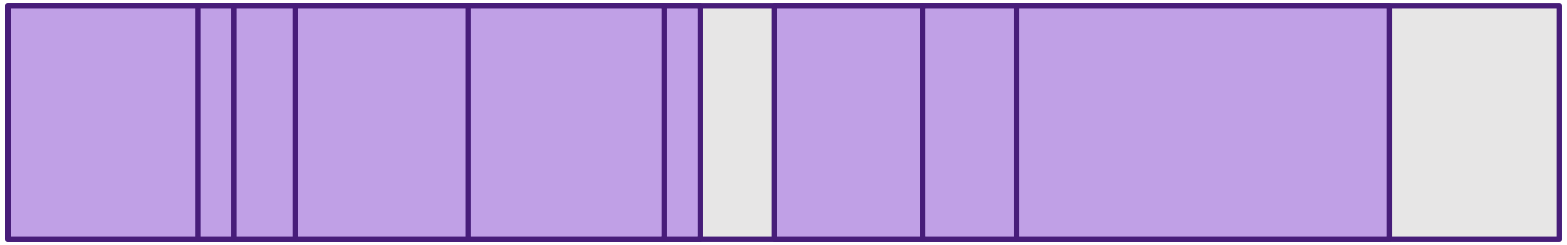
```
GC configuration
----------------------------------
ServerGC = False
ConcurrentGC = True
RetainVM = False
NoAffinitize = False
GCCpuGroup = False
GCLargePages = False
HeapCount = 1
MaxHeapCount = 0
```

DATAS

Why does my heap get so big?

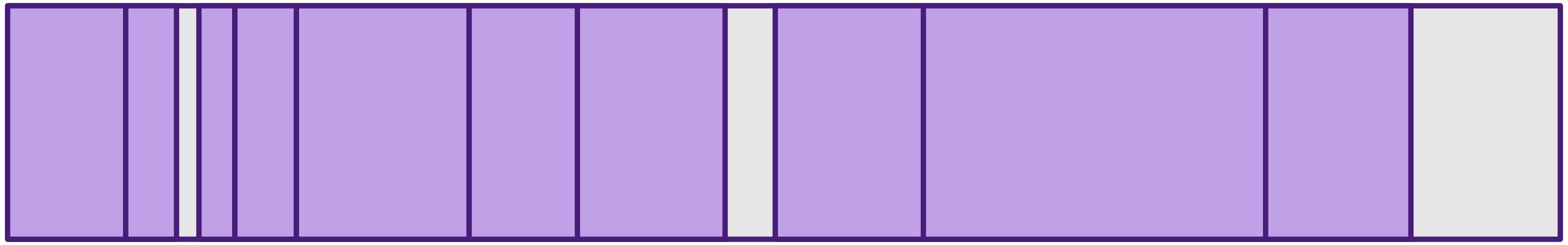
Background GC

Why does my heap get so big?



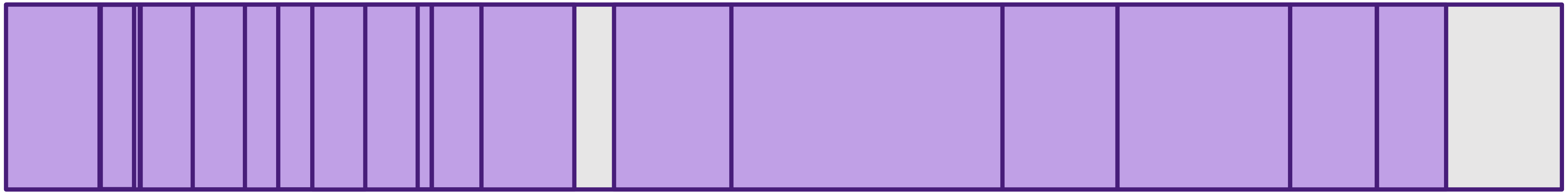
Background GC

Why does my heap get so big?



Background GC

Why does my heap get so big?



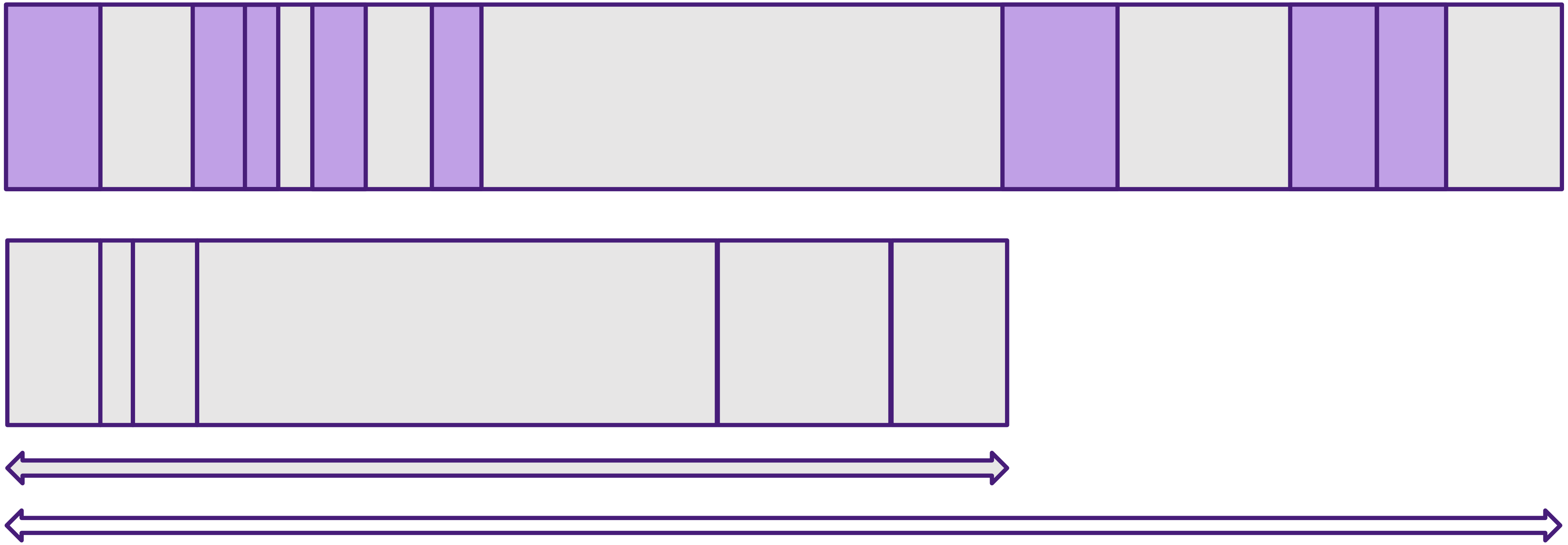
Background GC

GCConserveMemory

- `DOTNET_gcConserveMemory=[1-9]`
- Available on .NET Framework 4.8 and .NET 6+
- Defines the maximum amount of “fragmentation” (free objects) in gen 2 and LOH
- 1=10% of live data (90% of fragmentation)
  9=90% of live data (10% of fragmentation)

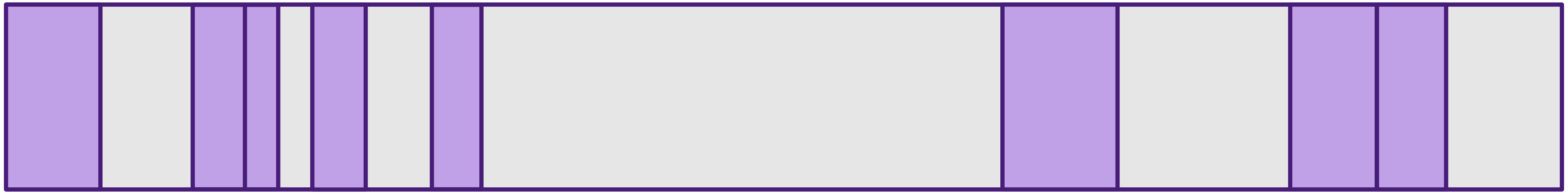
GCConserveMemory

```
DOTNET_gcConserveMemory=4  => 60% of fragmentation
```



GCConserveMemory

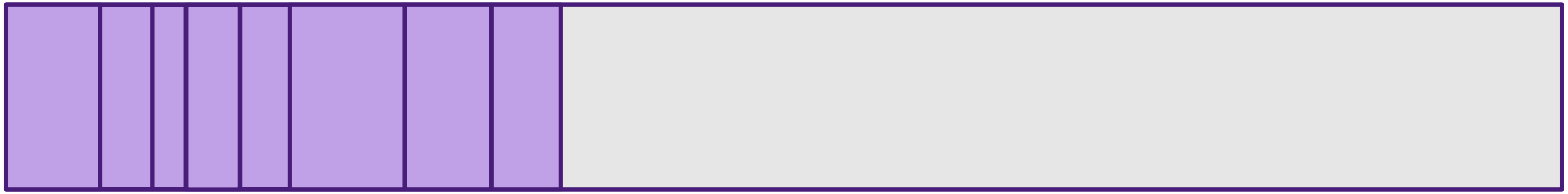
`DOTNET_gcConserveMemory=4  => 60% of fragmentation`



Compacting GC

GCConserveMemory

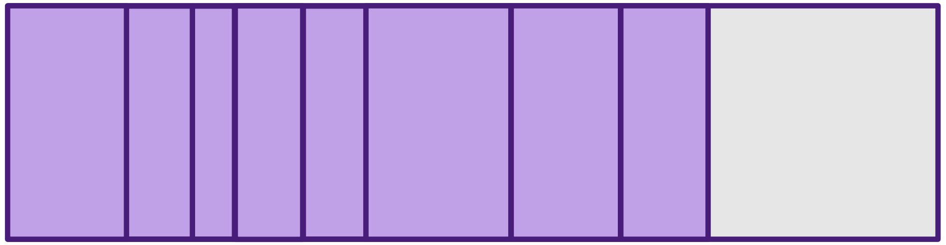
```
DOTNET_gcConserveMemory=4  => 60% of fragmentation
```



Compacting GC

GCConserveMemory

```
DOTNET_gcConserveMemory=4  => 60% of fragmentation
```



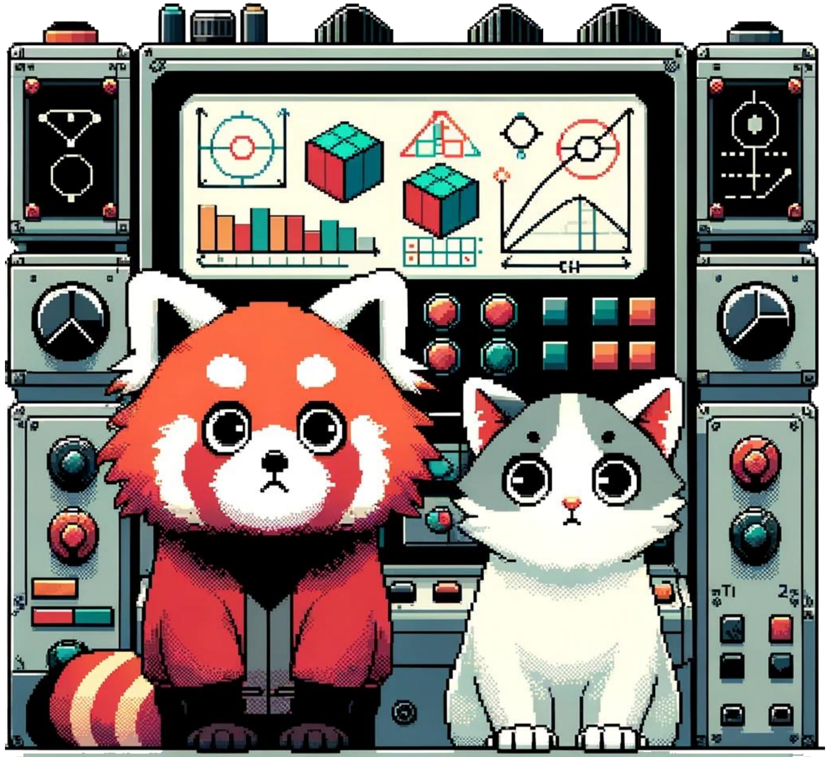
Compacting GC

Can we do better?

- Reduce gen 0 budget: `DOTNET_GCGen0MaxBudget`

- Reduce the number of heaps: `DOTNET_GCHeapCount`

- There has to be a better way…



Can we do better?

- Dynamically Adapting To Application Sizes (DATAS) : `DOTNET_GCDynamicAdaptationMode=1`
- Enabled by default in .NET 9
- Implies `DOTNET_GCConserveMemory=5` but can be tuned separately
- Automatically disabled if `DOTNET_GCHeapCount` is set

Can we do better?

- Gen 0 budget is adjusted depending on the size of other generations
- The number of heaps is adjusted depending on the workload
- Heuristics based on, over the last 3 GCs:

  Time spent in GC
  Time spent in allocation lock
  Time spent in the last gen 2 collection
  Minimum heap size
  Total heap size

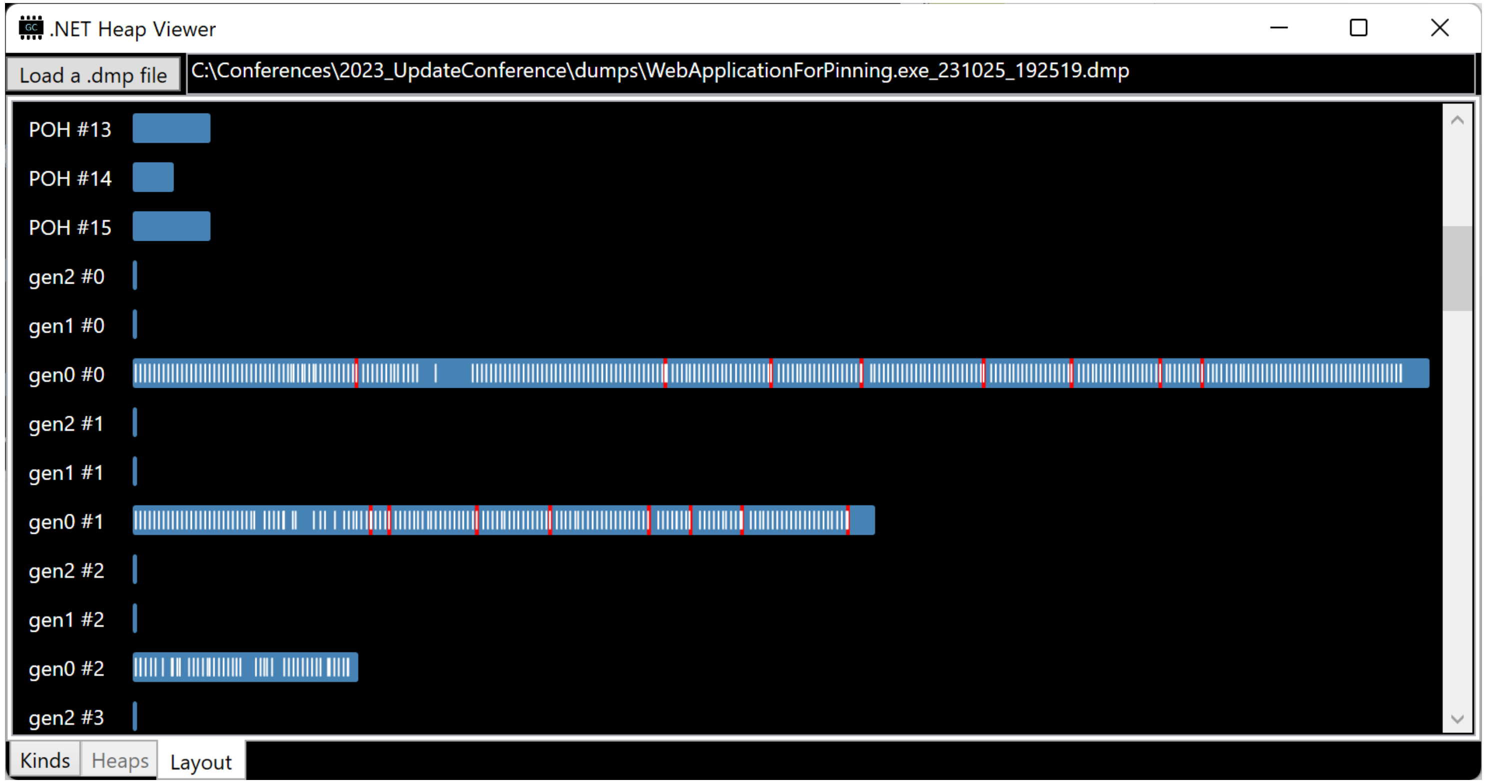
Welcome to the POH!

Welcome to the POH!

- The problem with pinned objects
  - increased memory usage (and more)

- Pinned Object Heap
  - `GC.AllocateArray` to pre-allocate pinned arrays as buffer for P/Invoke calls

- Use ClrMD to look into it
  - GCHandle for existing pinned objects (especially for asynchronous code)
  - all objects in the new POH


.NET Heap Viewer

Load a .dmp file C:\Conferences\2023_UpdateConference\dumps\WebApplicationForPinning.exe_231025_192519.dmp

String Literals… and the NonGC Heap

- String literals are eternal
  - no need to be managed by the GC
  - also for System.Type and simple readonly static fields (object, arrays of basic types)

- Allocation size heuristic

```
if < 64 KB then in NGCH
if < 85000 bytes then in gen0
else in LOH
```

Usages of the NonGC Heap

- Better JITted code
  - less pointer indirections
  - but more - https://github.com/dotnet/runtime/issues/76151

- Use ClrMD to look into it
  - but only for instanciated ones
  - System.Reflection.PortableExecutable.PEReader for the compiled ones

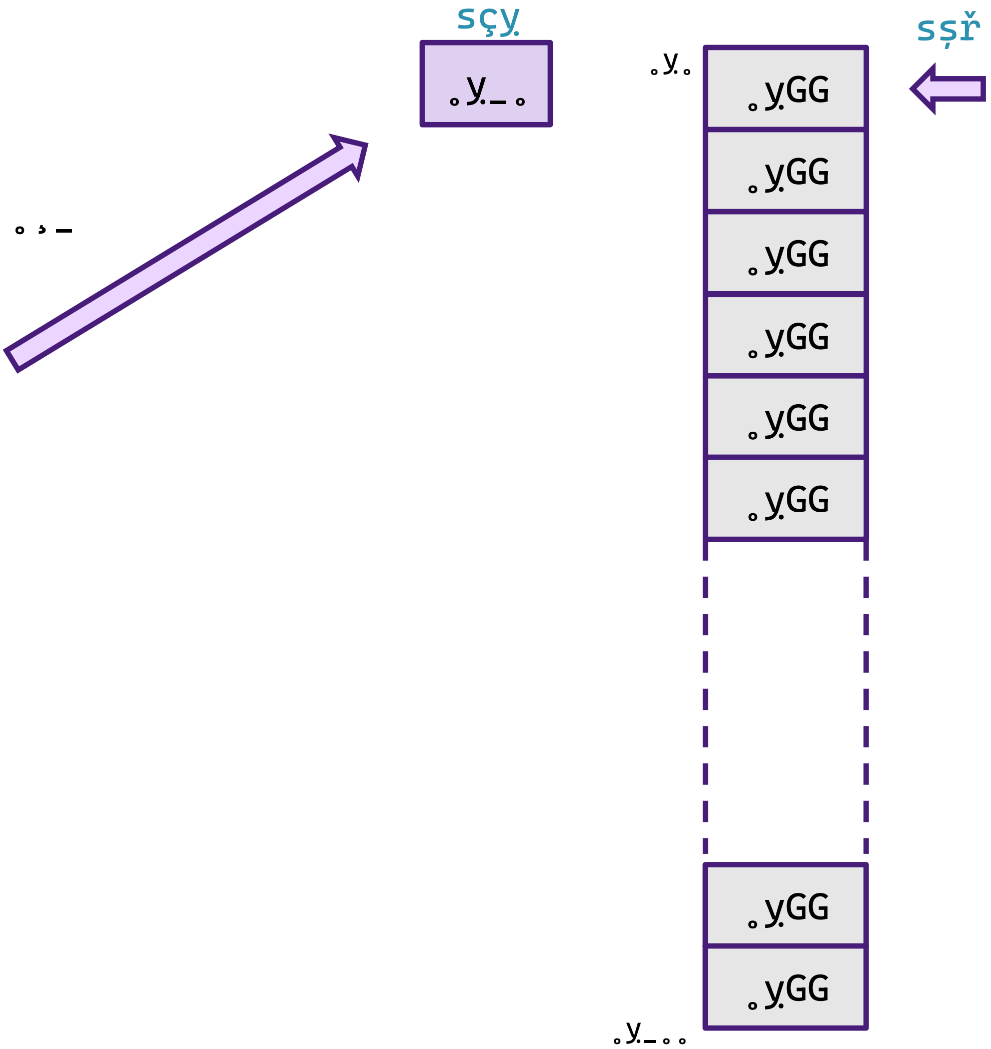
- No API

SkipLocalsInit

SkipLocalsInit

```
ŗųčľîç ŵộîđ Şţắçlắľľộç
    Şřắŋ čỳţê  șţs   șţắçlắľľộç čỳţê ,˳˒_
```

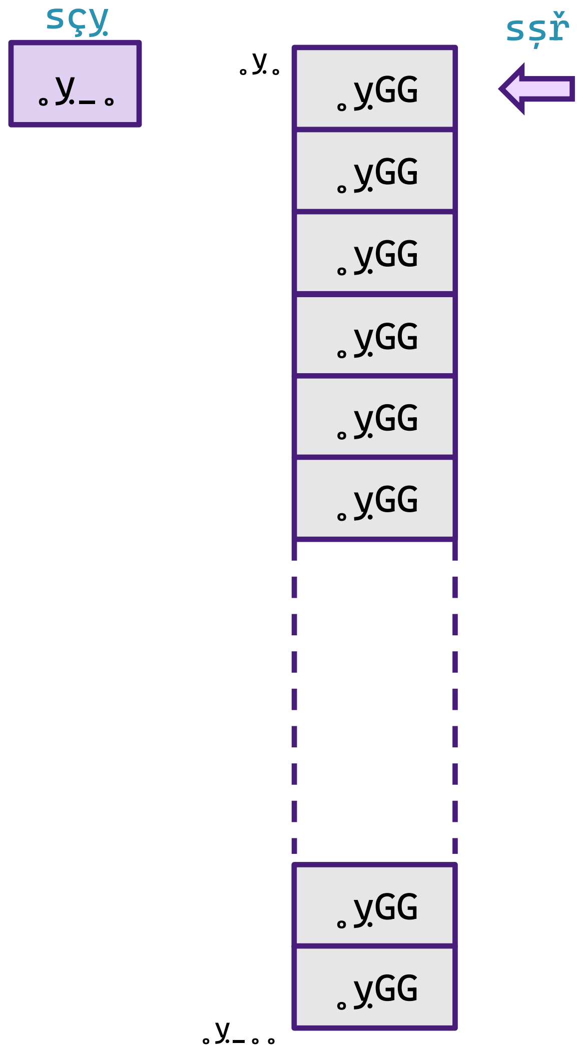
```
Ľ˳˳˒đ   ņộŵ êçy   ˳y_˳    ˋ_
Ľ˳˳˓˒   řụșḥ ˳
Ľ˳˳˓_   řụșḥ ˳
Ľ˳˳˓ˋ   đêç sçy
Ľ˳˳˓˙   kŋê șḥộsţ Ľ˳˳˓˒
```



SkipLocalsInit

```
řučľîç ŵôîđ Şţắçlắľľôç

    Şřắŋ čỳţê  ṣţs   ṣţắçlắľľôç čỳţê ,˳˒_
```

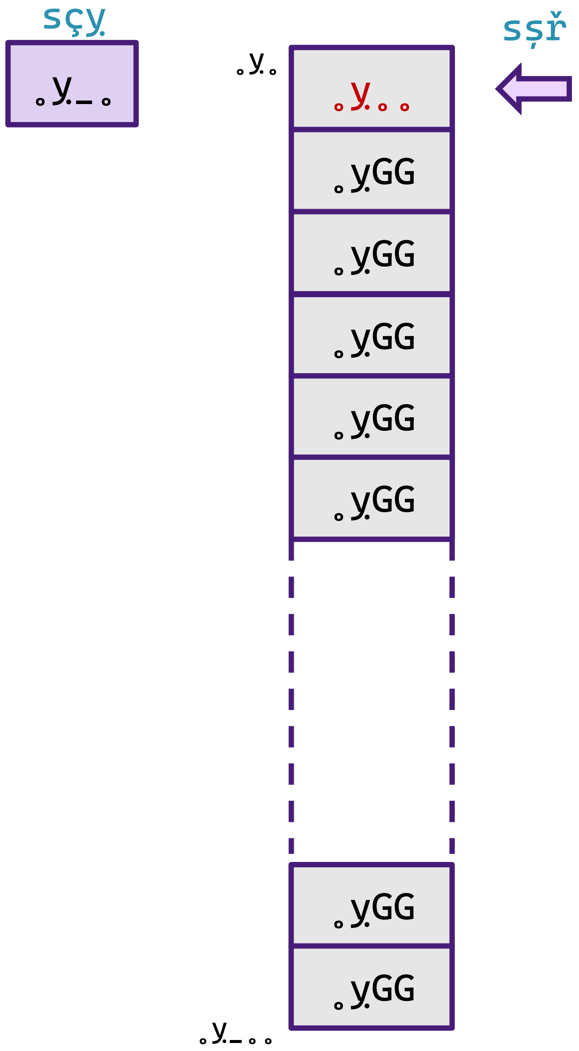
```
Ľ˳˳˒đ   ṇộŵ êçỵ   ˳ỵ_˳   `_
Ľ˳˳˓˒   řụṣḥ ˳
Ľ˳˳˓_   řụṣḥ ˳
Ľ˳˳˓`   đêç sçỵ
Ľ˳˳˓˙   kŋê ṣḥộsţ Ľ˳˳˓˒
```



SkipLocalsInit

```
ṛụčľîç ŵộîđ Şţắçlắľľộç

    Şřắŋ čỳţê  ṣţs   ṣţắçlắľľộç čỳţê ,˳˒_
```

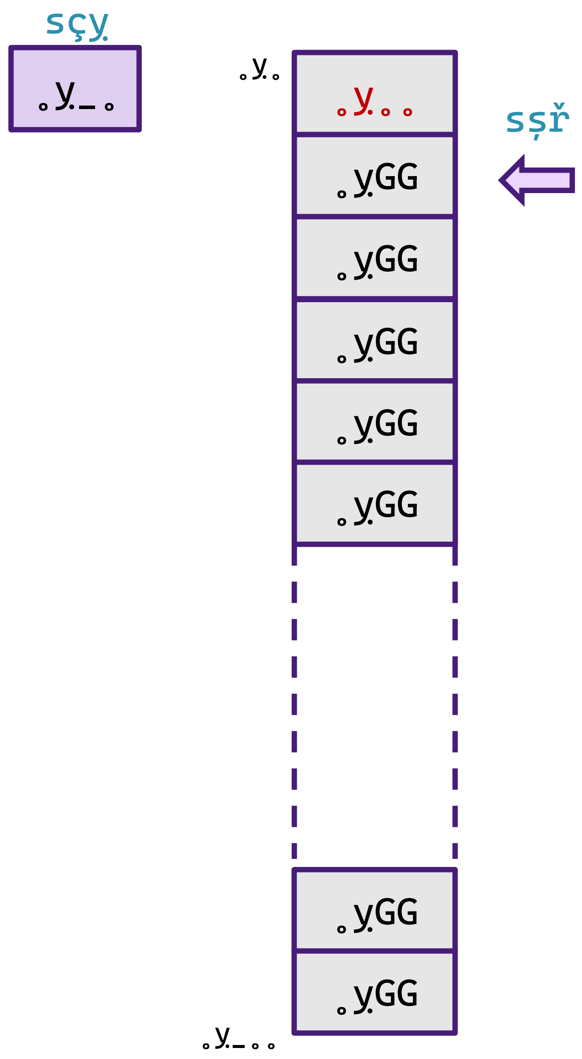
```
Ľ˳˳˒đ   ņộŵ êçỵ   ˳ỵ_˳    `_
Ľ˳˳˓˒   řụṣḥ ˳
Ľ˳˳˓_   řụṣḥ ˳
Ľ˳˳˓`   đêç sçỵ
Ľ˳˳˓˙   kŋê ṣḥộsţ Ľ˳˳˓˒
```



SkipLocalsInit

```
řųčľîç ŵộîđ Şţắçlắľľộç

    Şřắŋ čỳţê  ş̦ţs   ş̦ţắçlắľľộç čỳţê ˏ˳˒_
```

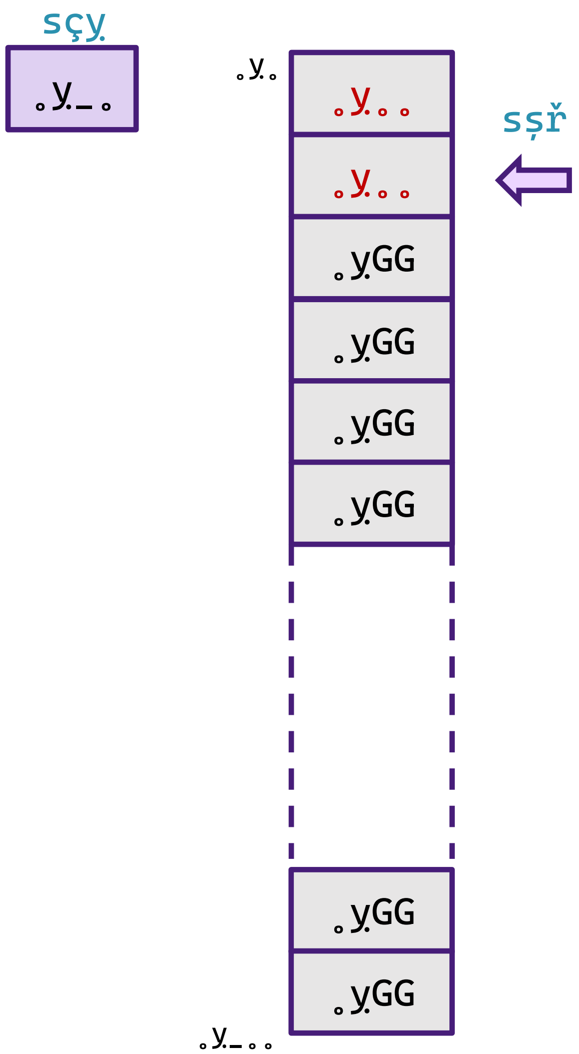
```
Ľ˳˳˒đ   ŋ̦ộŵ êçy̩  ˳y̩_˳  ˋ_
Ľ˳˳˓˒   řụş̦ḥ ˳
Ľ˳˳˓_   řụş̦ḥ ˳
Ľ˳˳˓ˋ   đêç sçy̩
Ľ˳˳˓·   kŋ̦ê ş̦ḥộsţ Ľ˳˳˓˒
```



SkipLocalsInit

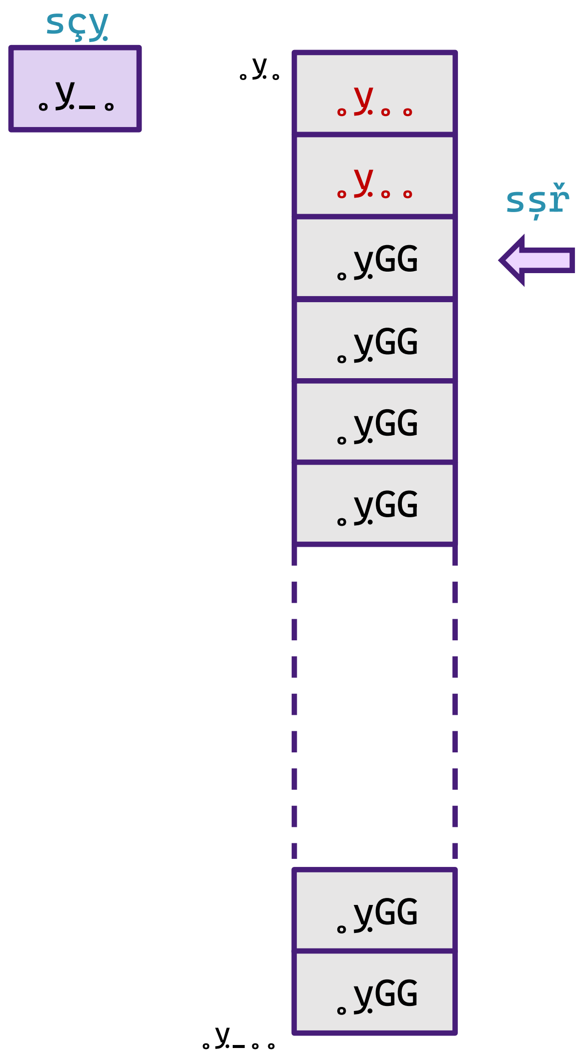
```
řučľîç ŵôîđ Şţắçlắľľôç
    Şřắŋ čỳţê  șţs   șţắçlắľľôç čỳţê ,˳ˀ_
```

```
Ľ˳˳ˀđ   ŋộŵ êçỵ   ˳ỵ_˳   `_
Ľ˳˳˓ˀ   řuṣḥ  ˳
Ľ˳˳˓_   řuṣḥ  ˳
Ľ˳˳˓    đêç sçỵ
Ľ˳˳˓    kŋê ṣḥộsţ Ľ˳˳˓ˀ
```



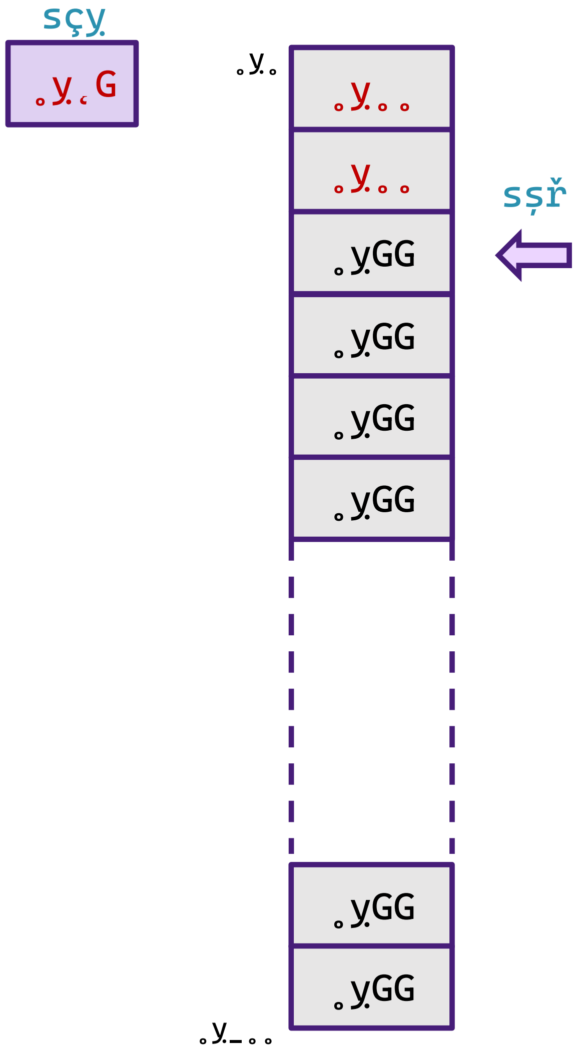
SkipLocalsInit

```
řučľîç ŵộîđ Şţắçlắľľộç

    Şřắŋ čỳţê  şţs   şţắçlắľľộç čỳţê ,˳ˀ_



      Ľ˳˳ˀđ  ŋộŵ êçy  ˳y_˳  `_
      Ľ˳˳˓ˀ  řuşh ˳
      Ľ˳˳˓_  řuşh ˳
      Ľ˳˳˓`  đêç sçy
      Ľ˳˳˓˙  kŋê şhộsţ Ľ˳˳˓ˀ
```



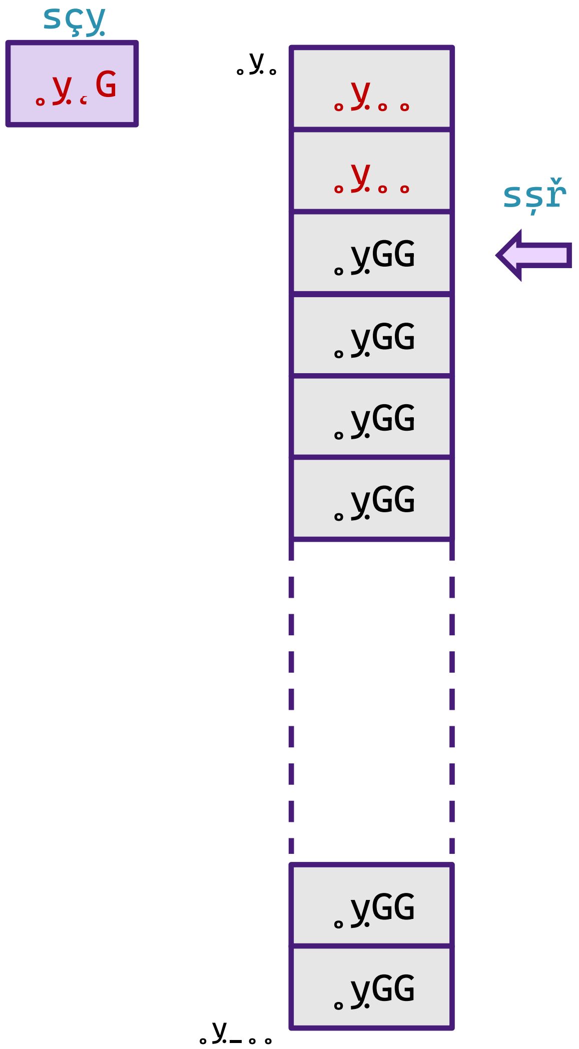
SkipLocalsInit

```
řųčľîç ŵộîđ Şţắçlắľľộç
    Şřắŋ čỳţê  şţs   şţắçlắľľộç čỳţê ,˳˒_


        Ľ˳˳˒đ  ņộŵ êçỵ   ˳ỵ_˳   `_
        Ľ˳˳˓˒  řụşḥ ˳
        Ľ˳˳˓_  řụşḥ ˳
        Ľ˳˳˓`  đêç sçỵ
        Ľ˳˳˓˙  kŋê şḥộsţ Ľ˳˳˓˒
```



SkipLocalsInit

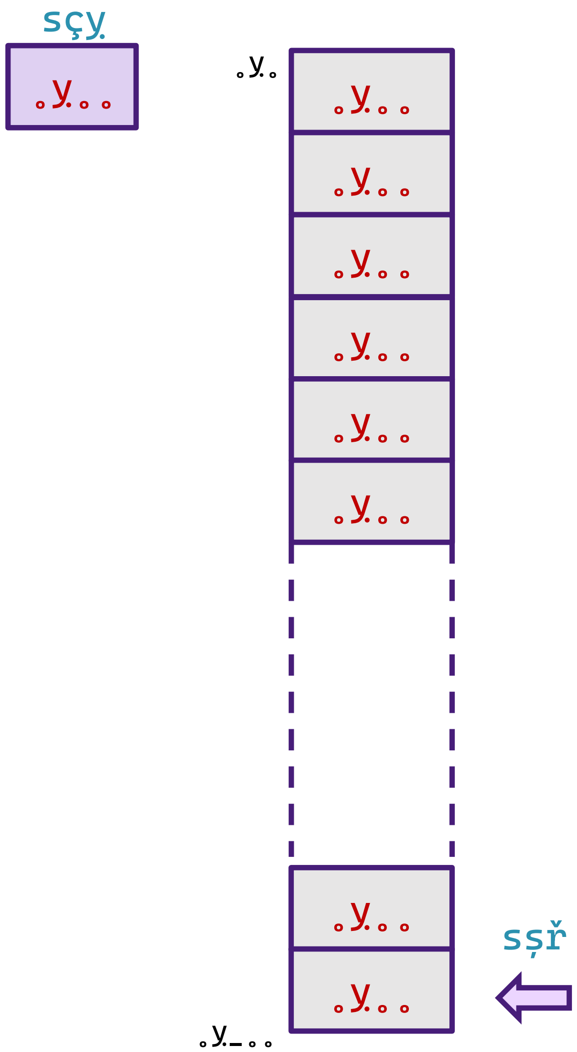
```
řučľîç ŵộîđ Şţắçlắľľộç

    Şřắŋ čỳţê  ṣţs   ṣţắçlắľľộç čỳţê ,˳˒_




        Ľ˳˳˒đ  ŋộŵ êçy̤   ˳y̤_˳  `_
        Ľ˳˳˓˒  řuṣḥ ˳
        Ľ˳˳˓_  řuṣḥ ˳
        Ľ˳˳˓`  đêç sçy̤
        Ľ˳˳˓˙  kŋê ṣḥộsţ Ľ˳˳˓˒
```



SkipLocalsInit

```
řučľîç ŵộîđ Şţăçlăľľộç

    Şřăŋ čỳţê  ṣţs   ṣţăçlăľľộç čỳţê ˏ˳˛_



        Ľ˳˳˛đ  ṇộŵ êçỵ  ˳ỵ_˳  ˋ_
        Ľ˳˳˓˛  řụṣḥ ˳
        Ľ˳˳˓_  řụṣḥ ˳
        Ľ˳˳˓ˋ  đêç sçỵ
        Ľ˳˳˓˙  kŋê ṣḥộsţ Ľ˳˳˓˛
```

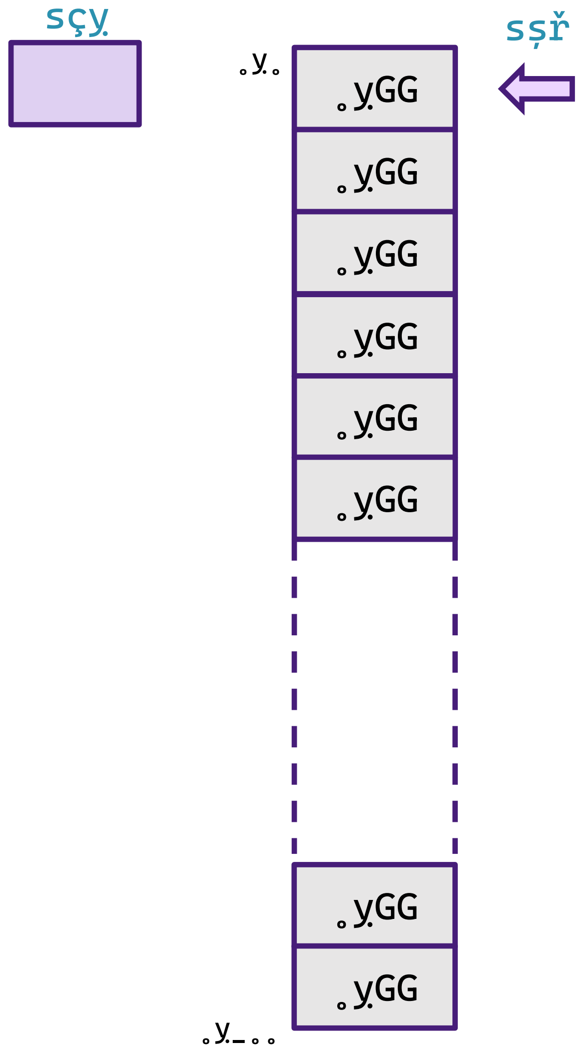
Total number of instructions: 1 + 4 x 64 = 257



SkipLocalsInit

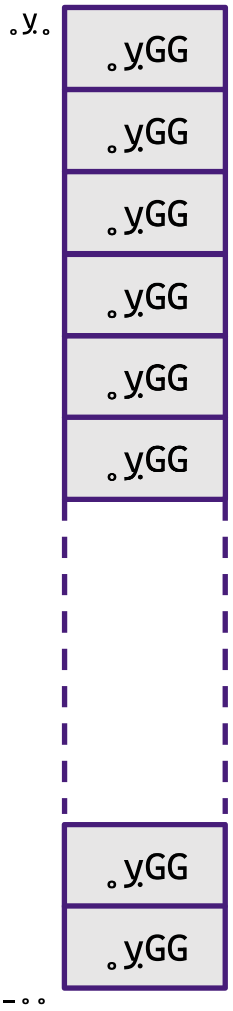
```
 ŞlîřĽộçắľşÍŋîţſ
řụčľîç ŵộîđ Şţſắçlắľľộç

    Şřắŋ čỳţſê  ṣţſs   ṣţſắçlắľľộç čỳţſê ‚˳˒_


  Ľ˳˳˒đ   ṣộŏ êṣỳ  ˳y̤_˳˳ `+˳˒_
  Ľ˳˳˓˒   řụṣḥ ˳
  Ľ˳˳˓_   řụṣḥ ˳
  Ľ˳˳˓`   đêç ṣçỵ
  Ľ˳˳˓˙   kŋê ṣḥộsţſ Ľ˳˳˓˒
```

Total number of instructions: 1 + 4 x 64 = 257



SkipLocalsInit

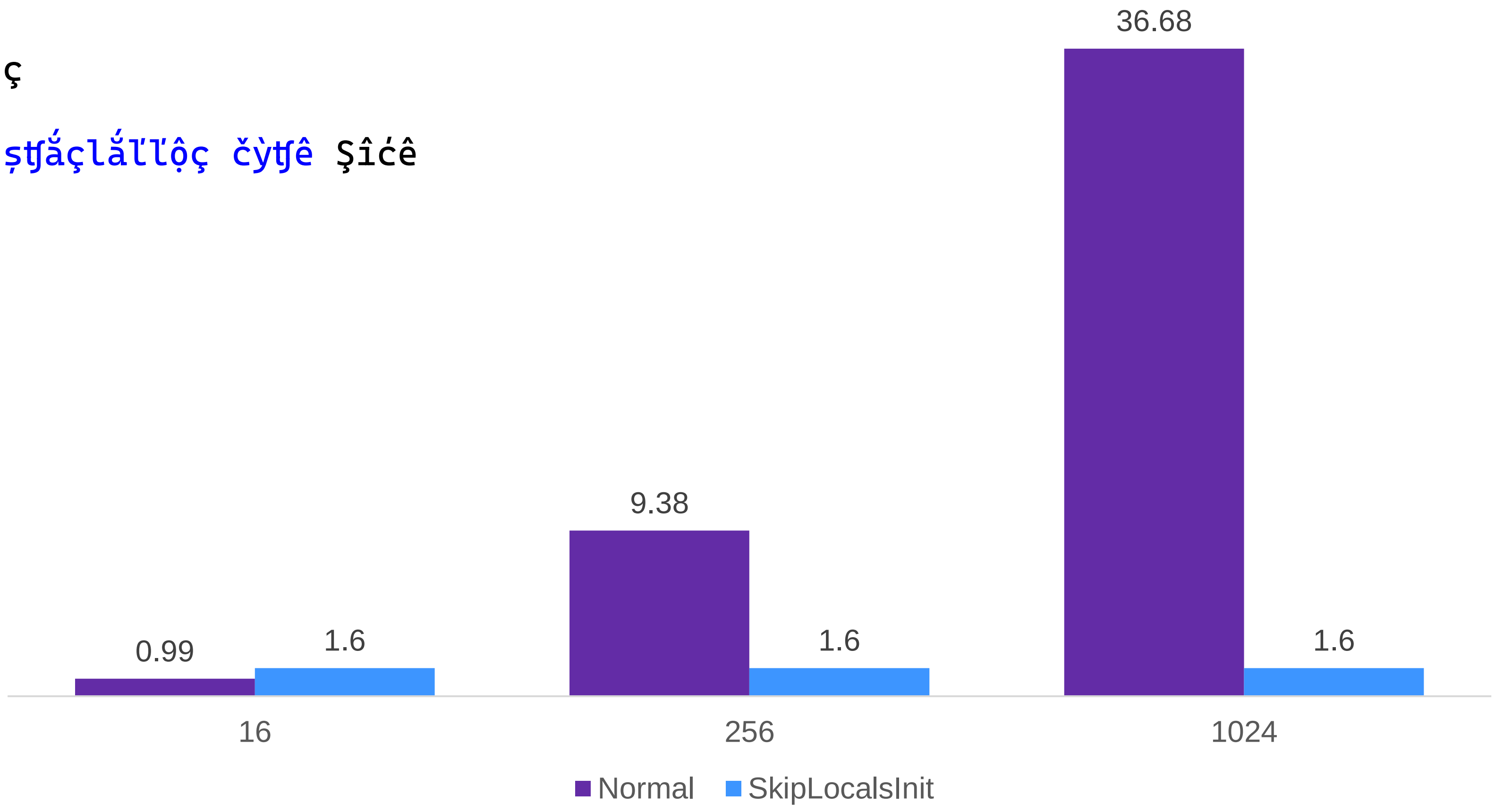
```
 ŞlîřĽộçǎľşÍŋîţ
řųčľîç ŵộîđ Şţǎçlǎľľộç

    Şřǎŋ čỳţê  şţş   şţǎçlǎľľộç čỳţê ,˳¸_

    ğîľê ŖêǎđAţľêǎşţ şţş  şţş Ľêŋĝţḥ
```



SkipLocalsInit

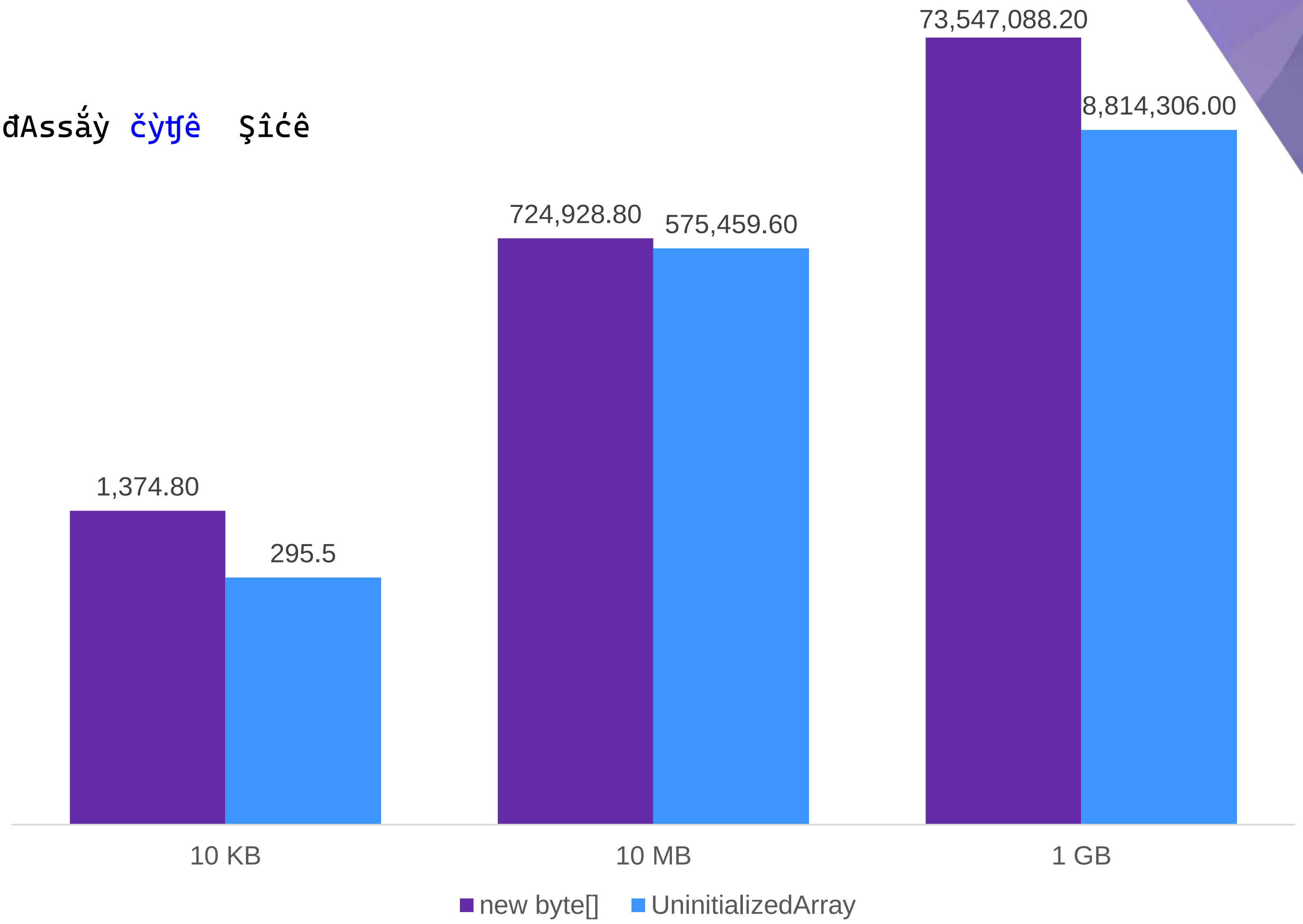
- Is it worth it?

```
 ŞlîřĽộçắľşÍŋîʧ
řụčľîç ŵộîđ Şʧắçlắľľộç

    Şřắŋ čỳʧê  şʧs   şʧắçlắľľộç čỳʧê Şîćê
```



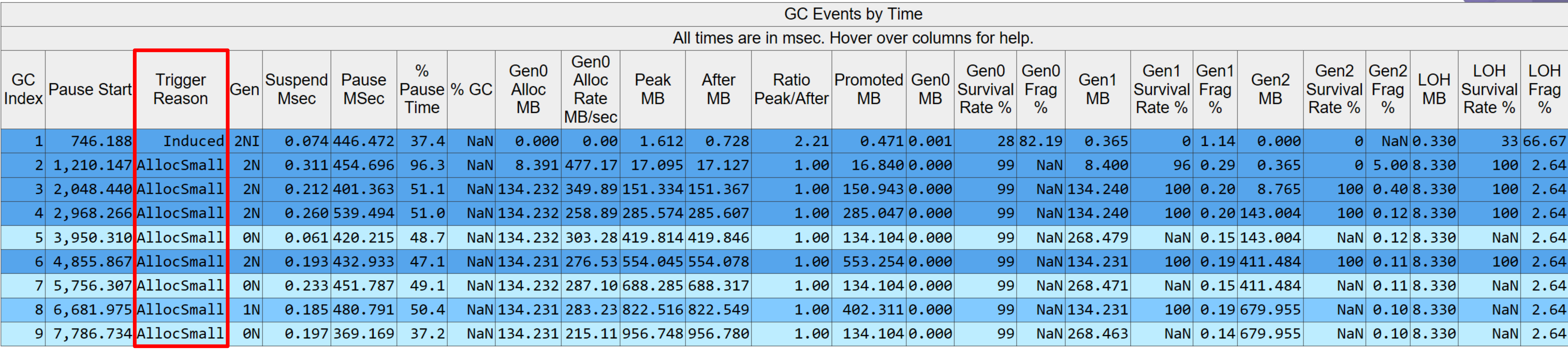
GC.AllocateUninitializedArray

```
řụčľîç čỳʧê   AľľộçắʧêAssắỳ
    sêʧụsŋ ĞC AľľộçắʧêÛŋîŋîʧîắľîćêđAssắỳ čỳʧê  Şîćê
```



Monitoring the garbage collections

- Perfview
  - GCStats view
  - nothing really new



GC Events by Time

All times are in msec. Hover over columns for help.

GC Index	Pause Start	Trigger Reason	Gen	Suspend Msec	Pause MSec	% Pause Time	% GC	Gen0 Alloc MB	Gen0 Alloc Rate MB/sec	Peak MB	After MB	Ratio Peak/After	Promoted MB	Gen0 MB	Gen0 Survival Rate %	Gen0 Frag %	Gen1 MB	Gen1 Survival Rate %	Gen1 Frag %	Gen2 MB	Gen2 Survival Rate %	Gen2 Frag %	LOH MB	LOH Survival Rate %	LOH Frag %
1	746.188	Induced	2NI	0.074	446.472	37.4	NaN	0.000	0.00	1.612	0.728	2.21	0.471	0.001	28	82.19	0.365	0	1.14	0.000	0	NaN	0.330	33	66.67
2	1,210.147	AllocSmall	2N	0.311	454.696	96.3	NaN	8.391	477.17	17.095	17.127	1.00	16.840	0.000	99	NaN	8.400	96	0.29	0.365	0	5.00	8.330	100	2.64
3	2,048.440	AllocSmall	2N	0.212	401.363	51.1	NaN	134.232	349.89	151.334	151.367	1.00	150.943	0.000	99	NaN	134.240	100	0.20	8.765	100	0.40	8.330	100	2.64
4	2,968.266	AllocSmall	2N	0.260	539.494	51.0	NaN	134.232	258.89	285.574	285.607	1.00	285.047	0.000	99	NaN	134.240	100	0.20	143.004	100	0.12	8.330	100	2.64
5	3,950.310	AllocSmall	0N	0.061	420.215	48.7	NaN	134.232	303.28	419.814	419.846	1.00	134.104	0.000	99	NaN	268.479	NaN	0.15	143.004	NaN	0.12	8.330	NaN	2.64
6	4,855.867	AllocSmall	2N	0.193	432.933	47.1	NaN	134.231	276.53	554.045	554.078	1.00	553.254	0.000	99	NaN	134.231	100	0.19	411.484	100	0.11	8.330	100	2.64
7	5,756.307	AllocSmall	0N	0.233	451.787	49.1	NaN	134.232	287.10	688.285	688.317	1.00	134.104	0.000	99	NaN	268.471	NaN	0.15	411.484	NaN	0.11	8.330	NaN	2.64
8	6,681.975	AllocSmall	1N	0.185	480.791	50.4	NaN	134.231	283.23	822.516	822.549	1.00	402.311	0.000	99	NaN	134.231	100	0.19	679.955	NaN	0.10	8.330	NaN	2.64
9	7,786.734	AllocSmall	0N	0.197	369.169	37.2	NaN	134.231	215.11	956.748	956.780	1.00	134.104	0.000	99	NaN	268.463	NaN	0.14	679.955	NaN	0.10	8.330	NaN	2.64

Resources

Documentation & source code

- CLR repository - https://github.com/dotnet/runtime
- Maoni Stephens blog, youtube channel
  and memory analysis
- Konrad Kokosa .NET GC Internals video series
- Kevin’s blog - https://minidump.net
- Christophe’s blog - https://chnasarre.medium.com

Tools

- SysInternals toolbox

DATADOG

https://prodotnetmemory.com/