Dynamic Memory Management for Embedded Real-Time Systems

Alfons Crespo, Ismael Ripoll and Miguel Masmano

Grupo de Informática Industrial – Sistemas de Tiempo Real
Universidad Politécnica de Valencia
Instituto de Automática e Informática Industrial

http://www.gii.upv.es

Outline

- Introduction
- Basic concepts
- DSA requirements for real-time systems
- Allocator classification
- TLSF description
- Evaluation
- Conclusions

Introduction

- Nowadays embedded systems are used in a wide range of industrial sectors requiring appropriate functionalities.
- The main advantages of embedded systems are their reduced price and size, broadening the scope of possible applications, but the main problem for their use is the limited computational capabilities they can offer.
- Optimal use of the resources is an important issue

Introduction

- **Dynamic memory management or Dynamic Storage Allocation (DSA)** is one part of the software system that influences the performance and the cost of a product the most.
- The system must be optimized due to the **limitation of memory**.
- Real-time deadlines must be respected: the dynamic memory management system must **allocate and deallocate blocks in due time**.

Introduction

- But there is a general misunderstanding of the use of Dynamic Memory Allocation
  - It is not used in real-time systems
  - Other techniques (ad-hoc) have been used
- It is mostly forgotten in QoS techniques
  - Processor, Network, Energy, ...
- Applications
  - Multimedia systems
  - Mobile phones
  - Applications using AI techniques
- Languages: Java, RTJava

Misunderstandings about DSA

1. **Long running** programs will fragment the heap more and more, consuming **unbounded memory**.
2. **Memory request** operations (malloc / free) **are** inherently **slow and unbounded**
3. It is usually better to implement your own **ad-hoc memory allocator** than use a known allocator

Consequence: It is **not used in real-time systems**

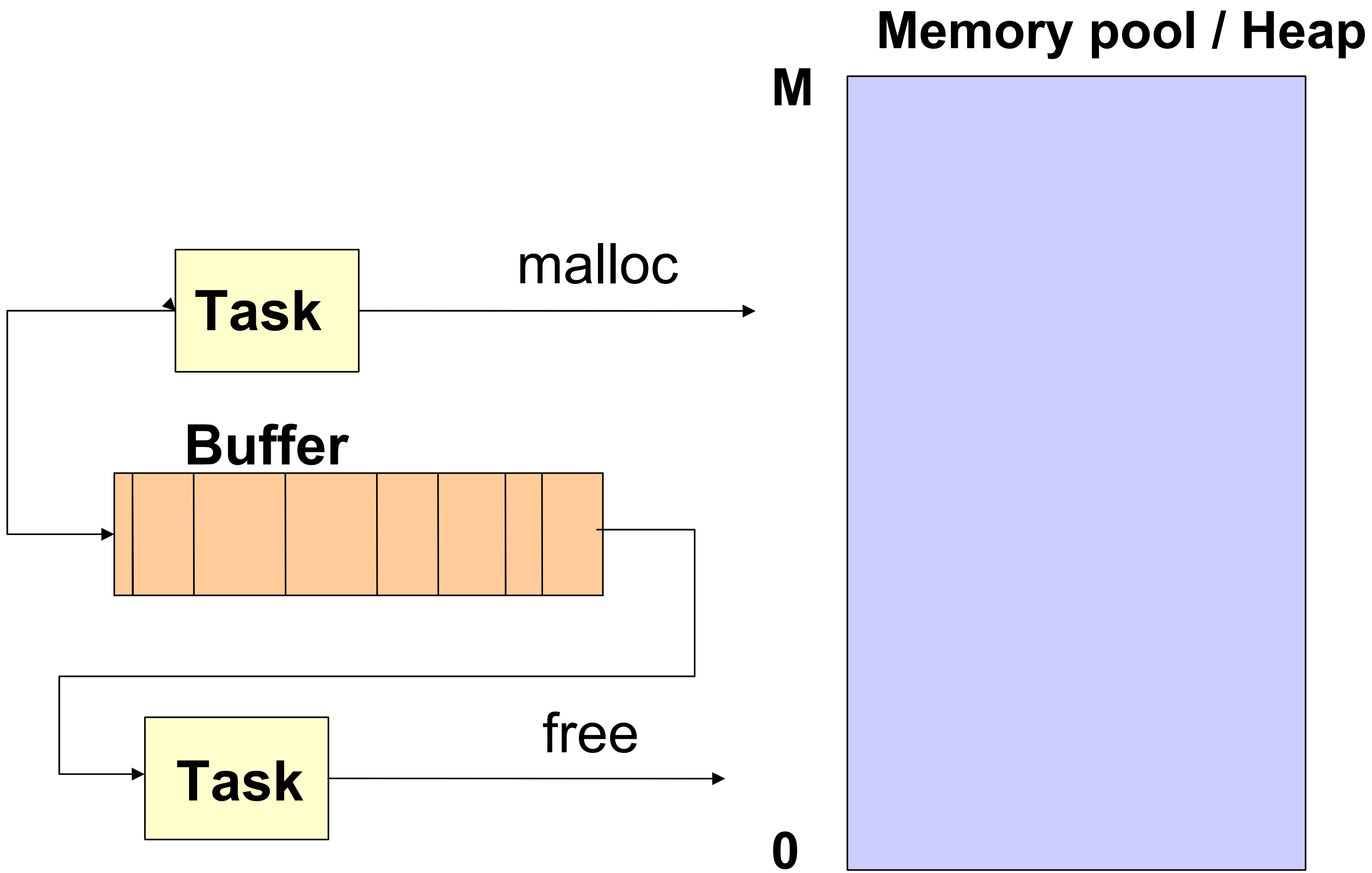
Dynamic Memory and RT Systems

- Currently, RT-Systems do not use explicit dynamic memory because
  - Allocation response time is either unbounded or very long
  - The **fragmentation** problem
- However, currently, several factors such as RTJava, the existence of more and more complex applications will force the use of dynamic memory

Explicit Dynamic Memory Management

- Dynamic memory allocation consists in managing, in execution time, a free area of memory (**heap**) to satisfy a sequence of requests (allocation/deallocation) without any knowledge of future requests
- This is an on-line optimisation problem: “space optimisation”
- Competitive analysis → it is not possible to design an optimum algorithm
- There exist many memory allocators (First-Fit, Best-Fit, Binary-Buddy, etc) (Robson, 80) The off-line version of the problem is NP-Hard

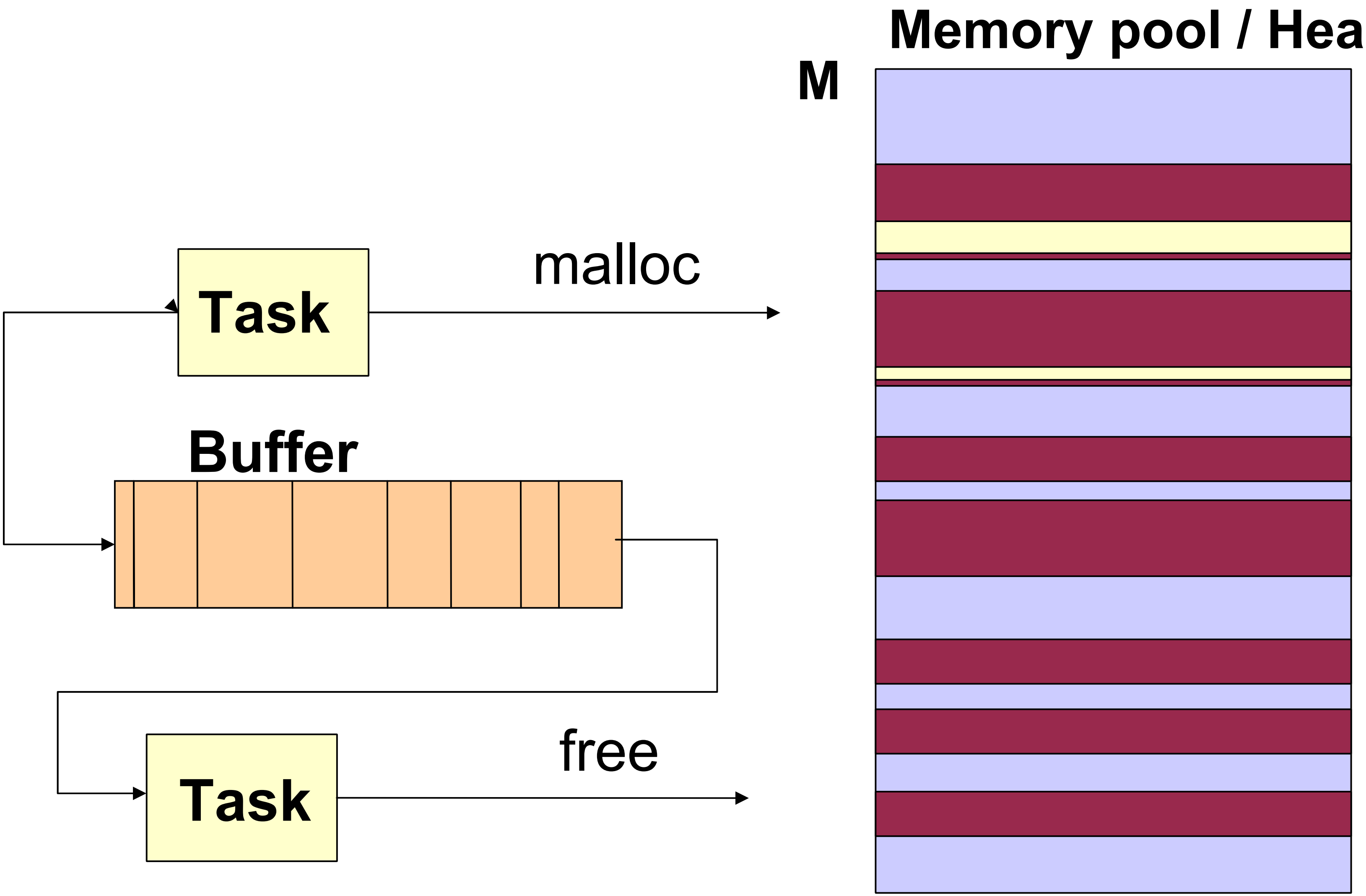
Basic concepts



Basic concepts

Allocated blocks

Free blocks

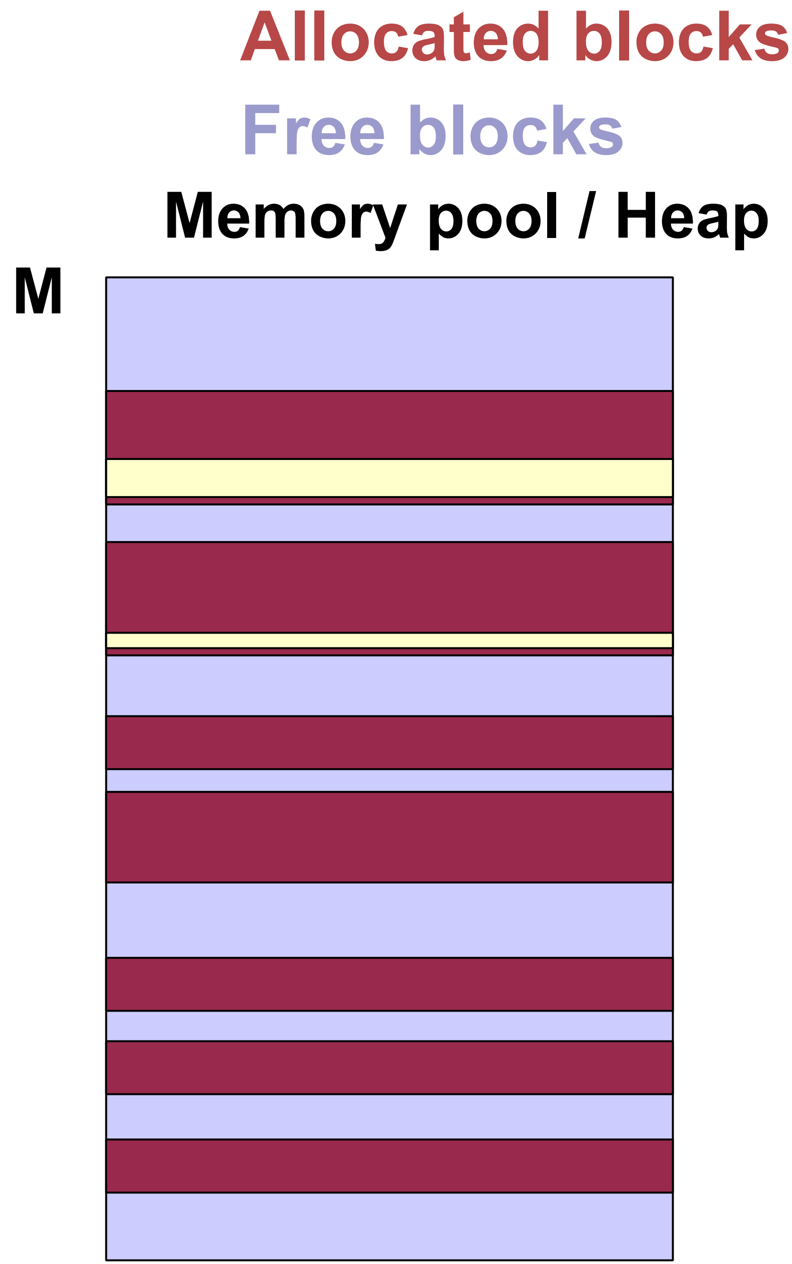


Basic concepts

Fragmentation: the inability to reuse memory that is free

or

the amount of wasted memory at a "steady state"

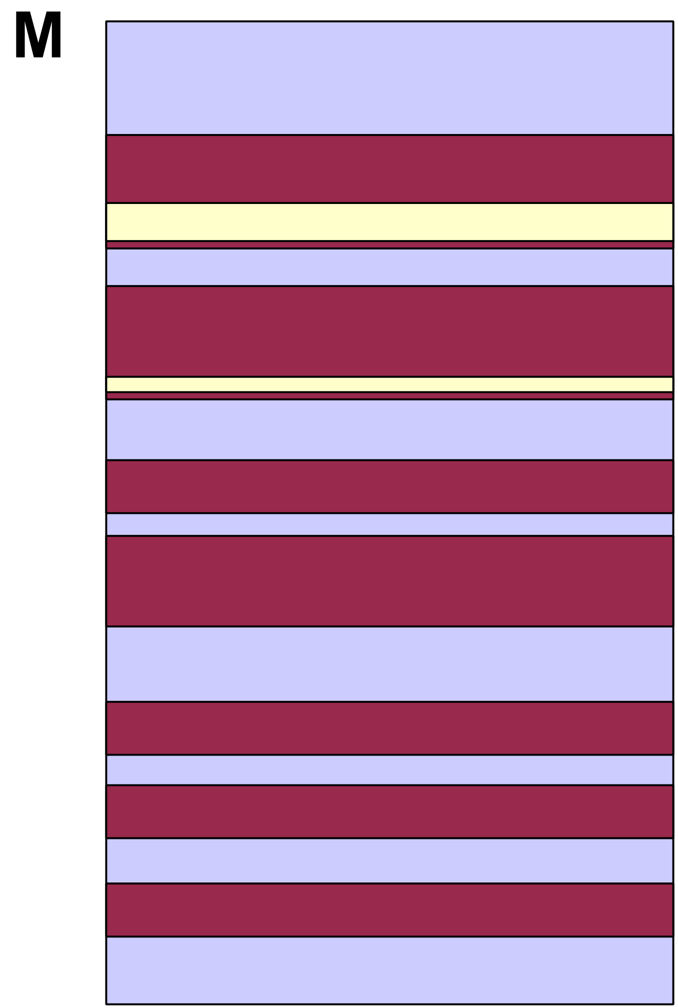


Basic concepts

Allocated blocks

Free blocks

Memory pool



•**Fragmentation:**

There are two sources of wasted memory:

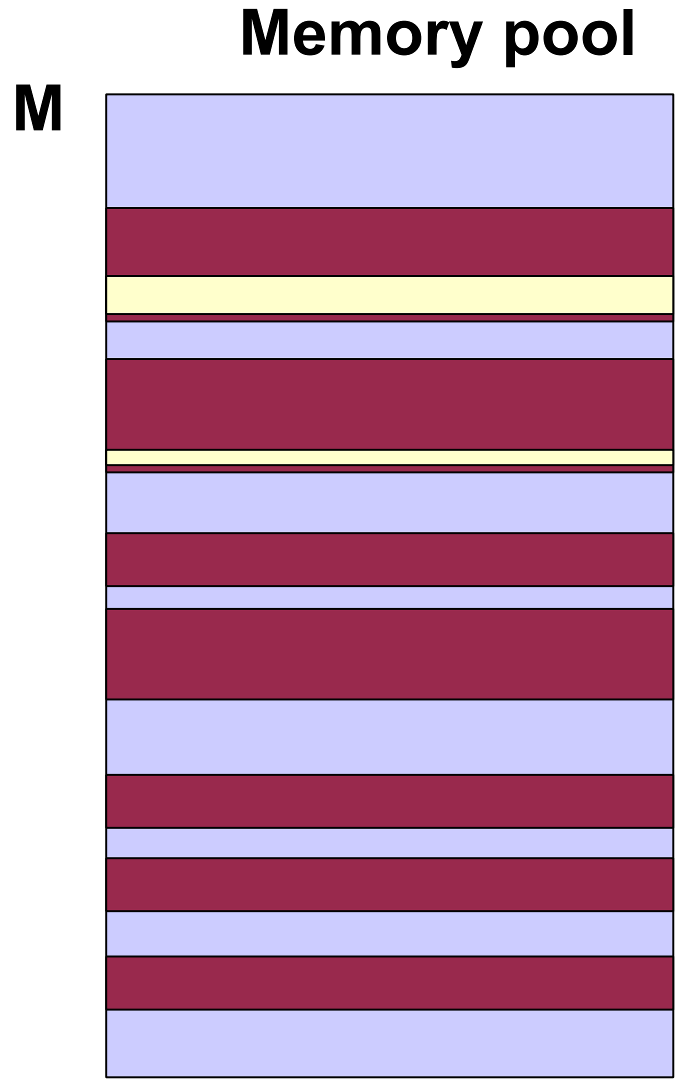
- internal fragmentation
- external fragmentation

Basic concepts

Finding free blocks:

- best fit: extensive search

- good fit: find a free block near the best.



Allocated blocks

Free blocks

Requirements for DSA

- **Bounded response time**. The worst-case execution time (WCET) of memory allocation and deallocation has to be known in advance and be independent of application data.
- **Fast response time**. Besides, having a bounded response time, the response time has to be fast enough to be usable.
- **Memory requests need to be always satisfied through an efficient use of memory**. The allocator has to handle memory efficiently, that is, the amount of wasted memory should be as small as possible.

Allocators classification

Policy

- **Allocation**
  - First-fit
  - Best-fit
  - Good-fit
  - Next-fit
  - Worst-fit
- **Deallocation**
  - Immediate coalescence
  - Deferred coalescence
  - No coalescence

Mechanism

- Sequential fits (linked lists)
- Segregated lists (set of free lists)
- Buddy systems (Segregated free lists)
- Indexed structures (AVL, Cartesian trees)
- Bitmaps

- Several mechanisms (bitmaps + segregated list)

Selected Dynamic Memory Allocators

- Reference algorithms
  - **First-Fit** and **Best-Fit**
- Labelled as RT-Systems allocators
  - **Binary Buddy**
- Widely used allocators
  - **Doug Lea's** malloc (DLmalloc)
- Designed for RT-Systems
  - **Half-Fit** and **TLSF**

Most used/known allocators

Allocator		Allocation Policy	Deallocation Policy	Mechanism
First-fit		First fit	Immediate coalescence	Linked List
Best-fit		Best fit	Immediate coalescence	Linked List
Binary-buddy		Best fit	Immediate coalescence	Buddy systems
AVL		Best fit	Immediate coalescence	Indexed Lists
DLmalloc	< 512b	Exact fit	No coalescence	Bitmaps
	≥ 512b	Best fit	Deferred coalescence	Linked List
Half-fit		Good fit	Immediate coalescence	Bitmaps + Segregated list
TLSF		Good fit	Immediate coalescence	Bitmaps + Segregated list

Worst /Bad Case Costs

Allocator	Allocation	Deallocation
First-fit/Best-fit	$O(M/(2 \cdot n))$	$O(1)$
Binary-buddy	$O(\log_2(M/n))$	$O(\log_2(M/n))$
DLmalloc	$O(M/n)$	$O(1)$
AVL	$O(2.44 \cdot \log_2(M/n))$	$O(4.32 \cdot \log_2(M/n))$
Half-fit/TLSF	$O(1)$	$O(1)$

M: Maximum memory size (Heap)

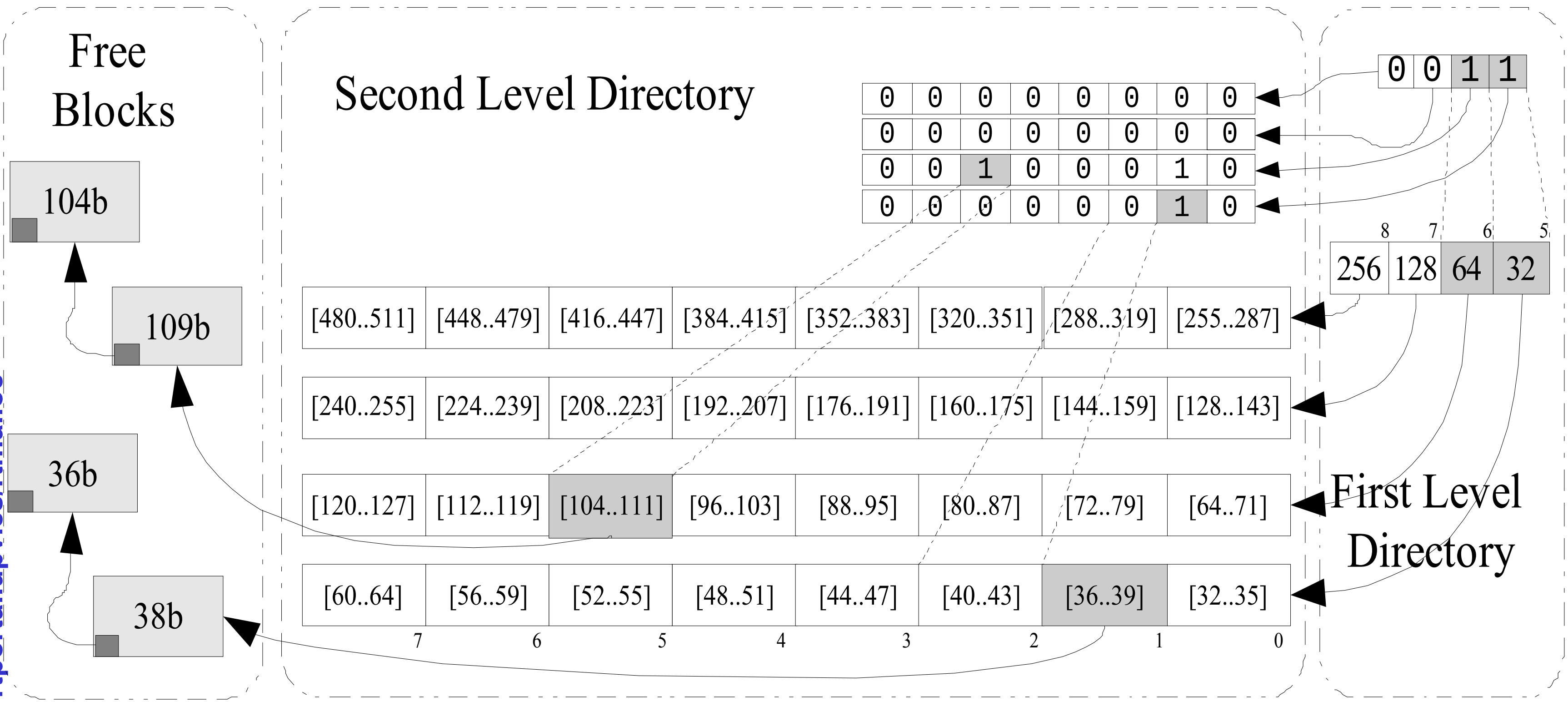
n: Largest allocated block

TLSF Design

- TLSF (**Two Level Segregated Lists**)
- TLSF was designed and implemented in the EU project OCERA (Open Components for Real-Time Embedded Applications) (http://www.ocera.org)
- It performs **inmediate coalescense** of free blocks
- Uses **Segregated list & bitmaps**
- Uses the **Good fit policy**

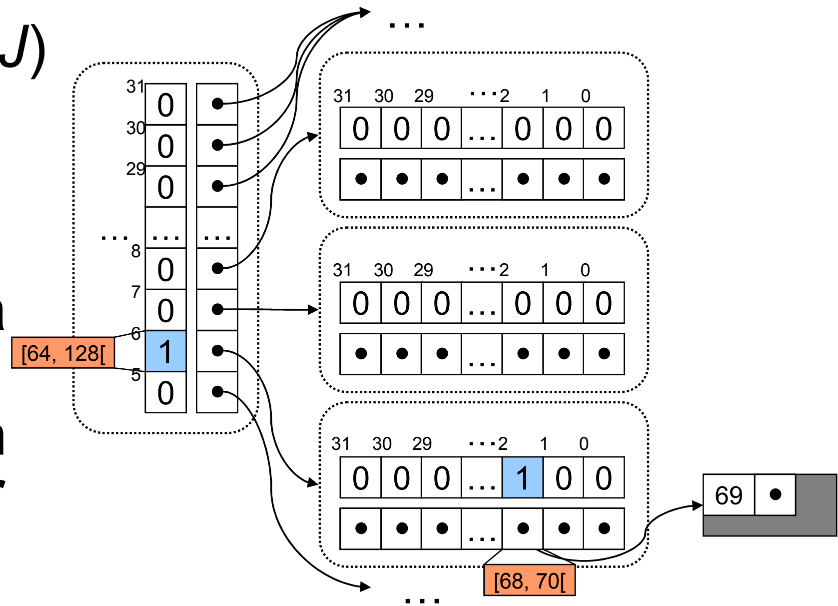
TLSF Design

- Uses **Segregated list & bitmaps**



Implementation issues

- Two configuration parameters (*I*, *J*)
  - *I*: 2^I maximum block size
  - *J*: 2^J number of second level lists
- Each set of list have associated a bitmap
  - Bitmap search using bit search forward and reverse (**bsf** y **bsr** in IA32)



- List **(i, j)** has blocks in the range

$$(i,j) = [2^i + j \cdot 2^{i-J}, 2^i + (j+1) \cdot 2^{i-J}[$$

  - Ex:. *J*=5, the list (i: 10, j: 5) has blocks of size in range [1084, 1216[

Implementation issues

- Two translation functions are provided
  - **Search** : returns the first list in the range **higher or equal to r**

$$\textit{search(r)}\quad \begin{cases} i : \left\lfloor \log_2\left(r + 2^{\lfloor \log_2(r) \rfloor - J} - 1 \right) \right\rfloor \\ j : \left\lfloor \left(r + 2^{\lfloor \log_2(r) \rfloor - J} - 1 - 2^{i} \right) / \left(2^{i-J} \right) \right\rfloor \end{cases}$$

  - **Insert**: return the list which range **includes r**

$$\textit{insert(r)}\quad \begin{cases} i : \lfloor \log_2(r) \rfloor \\ j : \left\lfloor \left(r - 2^{i} \right) / \left(2^{i-J} \right) \right\rfloor \end{cases}$$

Evaluation

- Worst / Bad Case Execution Time
  - Specific scenarios to achieve the worst or bad case and measure it (cycles and number of instructions)
- Synthetic loads (real-time loads?)
  - Measures of average, st_dev, maximum and minimum of several tests (> 20 experiences)

Worst / Bad scenario evaluation

- Identification of each worst/bad case
- Definition of a load to achieve the w/b case
- Execute the load
- Measure

Worst-case (WC) and Bad-case (BC) allocation

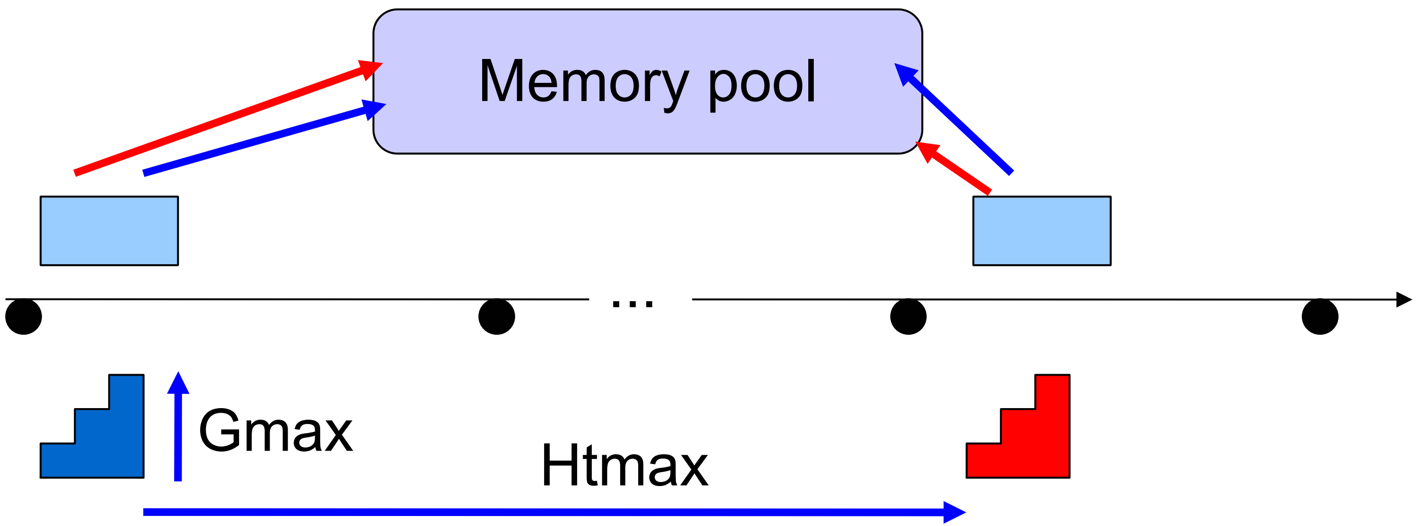
Malloc	FF	BF	BB	DL	HF	TLSF
Processor instructions	81995	98385	1403	721108	164	197

Evaluation loads

- Real load
  - There are not examples of dynamic memory use in real-time systems
  - Available loads of classical programs using dynamic memory: compilers (gcc, perl,..), applications (gs, espresso, cfrac,…)
- Synthetic loads
  - Several general purpose models
  - **Generated from periodic memory use models**

Synthetic load for periodic models

- Periodic task model extension
  - Each task is defined as $T_i=(c_i, p_i, d_i, \mathbf{g_i}, \mathbf{h_i})$
    - $\mathbf{g_i}$: Maximum amount of memory per period
    - $\mathbf{h_i}$: Holding time



Attributes:

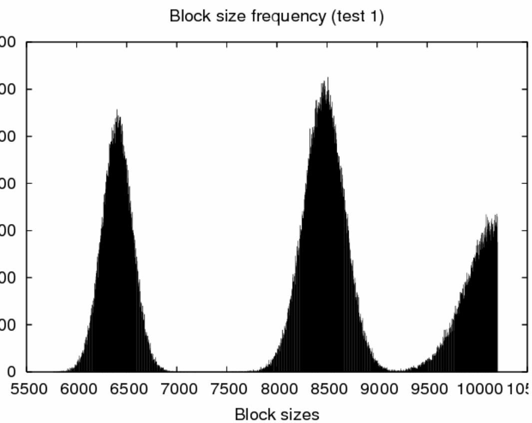
Gmax - maximum amount of memory allocated by pe

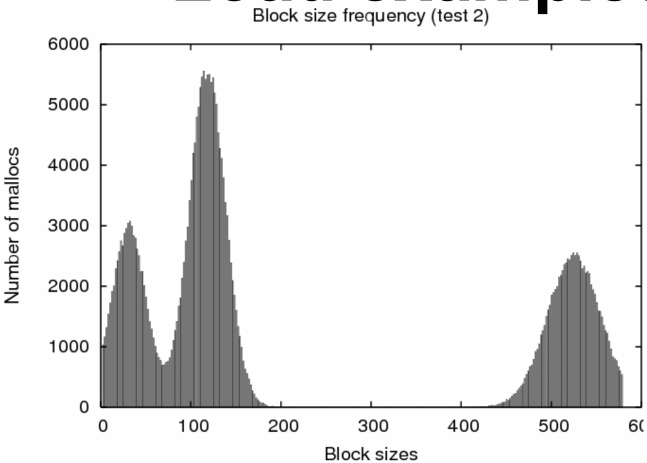
Htmax - maximum holding time

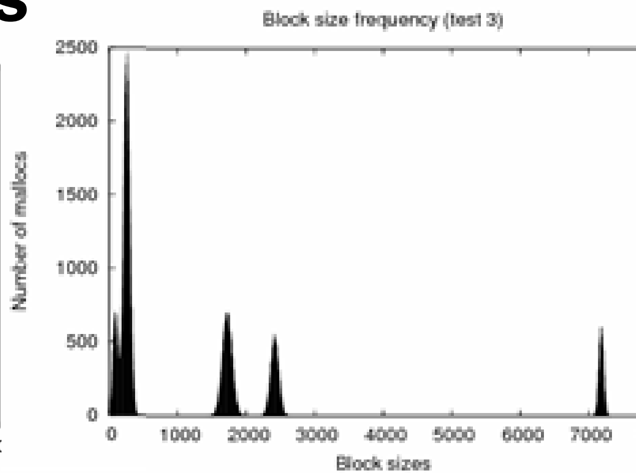
Load generation

- A load generator produces set of task with different profiles.
  - Huge blocks
  - Small blocks
  - Hybrid (small and large) blocks

Load examples





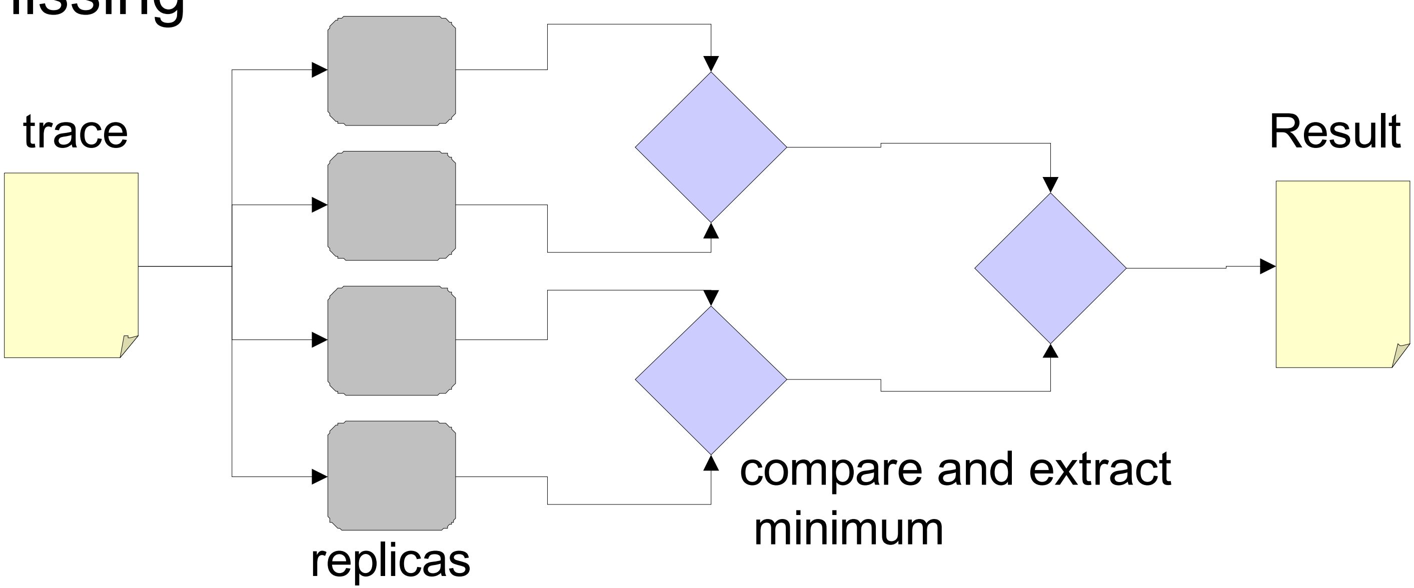


Evaluation

- Temporal measures
  - **Number of cycles:** **Highly dependent of the processor used (AMD, Intel) and of the data caches ,…**
  - **Number of instructions:** **Unaffected by cache, TLBs, processor,… but it is hard to measure (Processor switched to single-step mode)**
- Spatial measures
  - **Fragmentation:** **very huge number of operations**

Evaluation: Number of proc. cycles

- In order to reduce the system (hardware, os, interrupts, ..) interferences
  - Each test has been executed with interrupt disabled
  - A trace has been generated and used for 4 replicas in order to avoid processor interferences and cache missing



Evaluation: Number of proc. cycles

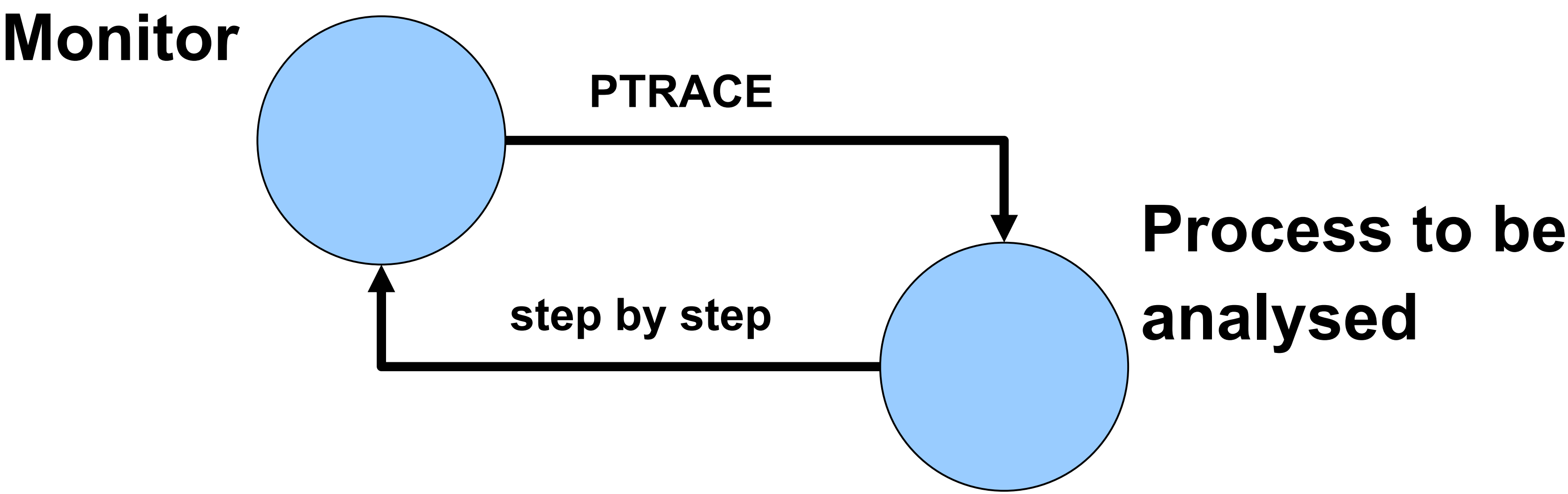
Temporal cost of the allocator operations in processor cycles

Malloc	**Test1**				**Test2**				**Test3**			
Alloc.	**Avg.**	**Stdv.**	**Max.**	**Min.**	**Avg.**	**Stdv.**	**Max.**	**Min.**	**Avg.**	**Stdv.**	**Max.**	**Min.**
First-fit	182	218	977	101	150	286	1018	95	169	282	2059	10
Best-fit	479	481	2341	114	344	348	2357	98	1115	1115	6413	10
Binary-buddy	169	465	1264	140	156	228	656	104	162	225	1294	11
DLmalloc	344	347	1314	123	268	290	2911	79	309	312	2087	8
Half-fit	189	331	592	131	148	577	657	108	157	552	626	13
TLSF	206	256	**371**	135	169	268	324	109	191	229	349	11

Free	**Test1**				**Test2**				**Test3**			
Alloc.	**Avg.**	**Stdv.**	**Max.**	**Min.**	**Avg.**	**Stdv.**	**Max.**	**Min.**	**Avg.**	**Stdv.**	**Max.**	**Min.**
First-fit	162	188	1432	87	148	182	1412	86	174	195	1512	9
Best-fit	152	188	1419	88	121	235	1287	87	145	179	1450	9
Binary-buddy	152	302	1127	126	155	300	760	126	150	306	824	12
DLmalloc	122	211	342	87	101	328	335	75	127	186	335	7
Half-fit	181	212	518	104	171	233	870	103	182	210	1025	10
TLSF	192	215	624	109	161	211	523	106	187	214	552	10

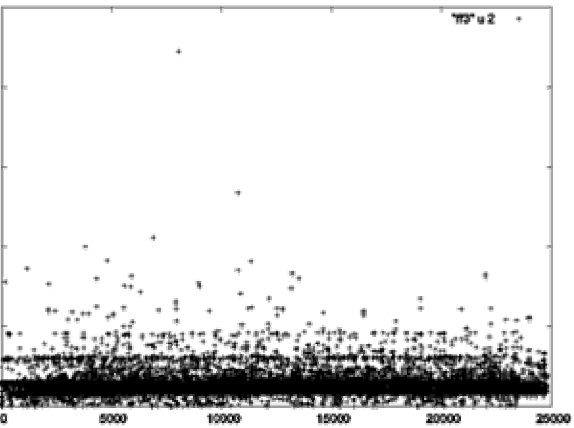
Evaluation: Number of instructions

- **A process (parent) counts (PTRACE) the executed instructions of another process (process that performs the mallocs and frees)**

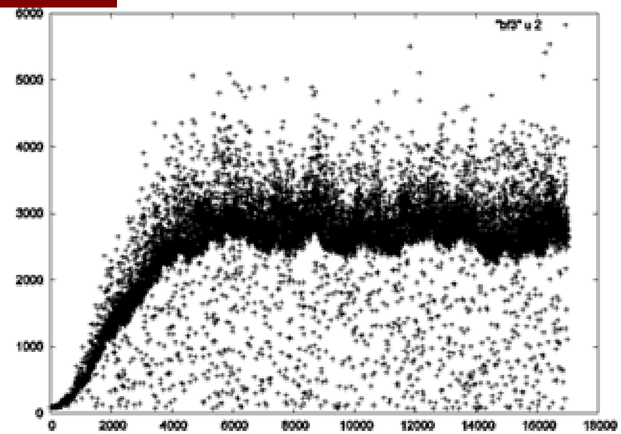


Evaluation: Number of instructions

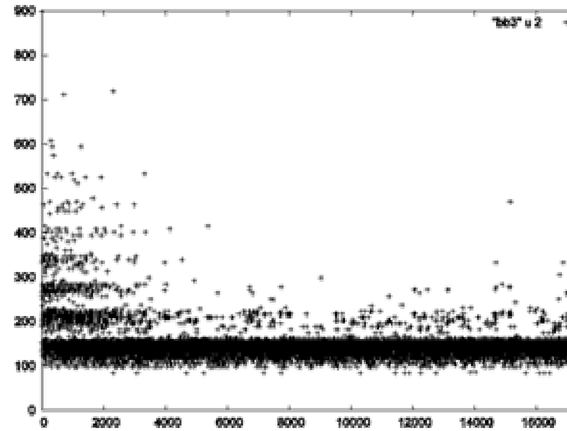
500 **First-fit**



6000 **Best-fit**

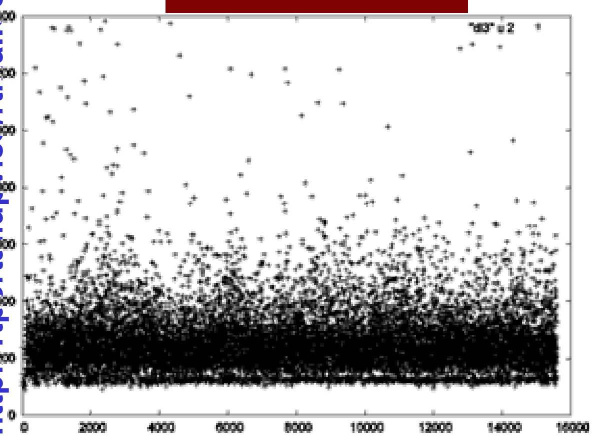


900 **Binary-Buddy**

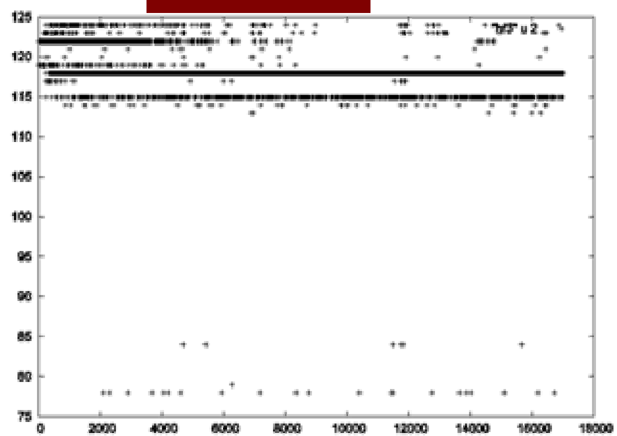


using real load: cfrac, gawk, perl

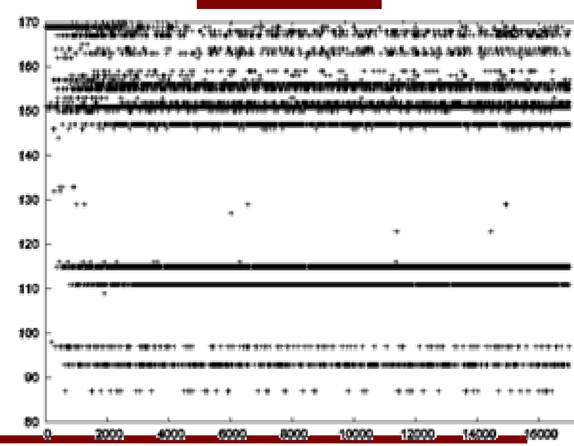
400 **DLmalloc**



125 **Half-fit**



170 **TLSF**



Evaluation: Number of instructions

Temporal cost of the allocator operations in processor instructions

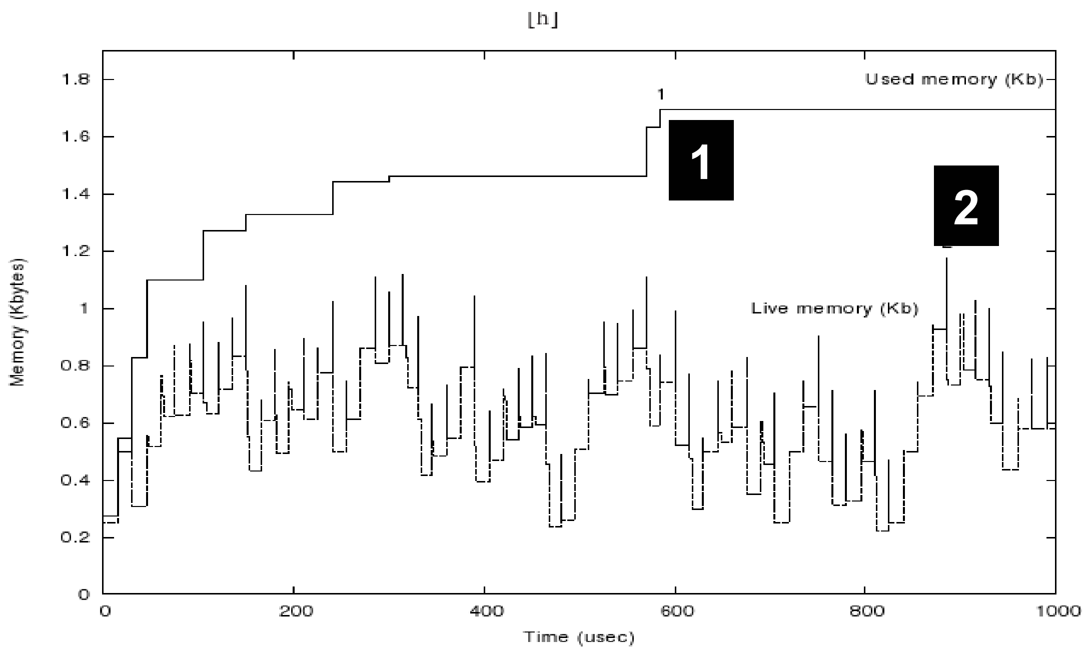
Malloc	**Test1**				**Test2**				**Test3**			
Alloc.	**Avg.**	**Stdv.**	**Max.**	**Min.**	**Avg.**	**Stdv.**	**Max.**	**Min.**	**Avg.**	**Stdv.**	**Max.**	**Min.**
First-fit	204	21	478	71	201	17	818	70	203	23	957	70
Best-fit	582	69	798	76	442	130	1006	76	805	179	1539	76
Binary-buddy	169	17	843	157	136	22	1113	95	153	24	1113	95
DLmalloc	279	107	921	64	161	126	933	49	232	152	1277	57
Half-fit	118	1	123	115	116	7	123	76	118	1	123	82
TLSF	147	13	164	104	118	25	164	84	133	22	164	84

Free	**Test1**				**Test2**				**Test3**			
Alloc.	**Avg.**	**Stdv.**	**Max.**	**Min.**	**Avg.**	**Stdv.**	**Max.**	**Min.**	**Avg.**	**Stdv.**	**Max.**	**Min.**
First-fit	93	96	128	59	90	92	128	57	92	95	128	57
Best-fit	91	115	126	57	69	148	126	57	79	198	128	57
Binary-buddy	68	70	225	65	68	72	277	65	69	73	228	65
DLmalloc	70	128	77	53	59	177	77	39	67	168	77	39
Half-fit	117	117	167	73	115	116	165	76	117	117	167	76
TLSF	140	140	217	91	107	110	216	87	120	122	217	87

Evaluation: Fragmentation

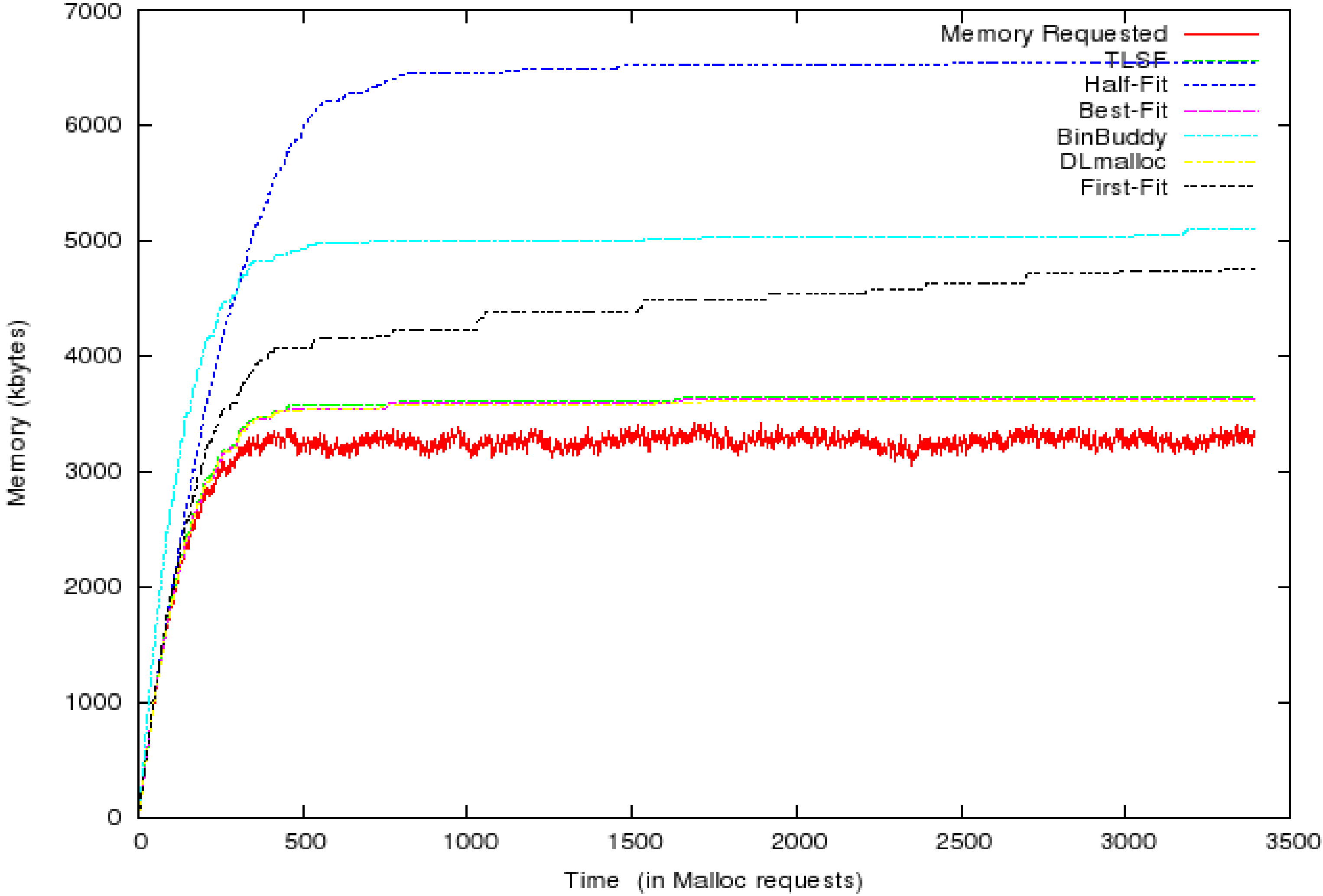
- It is measured to a factor F which is computed as the point of the maximum memory used by the allocator relative to the point of the maximum amount of memory used by the load (live memory).

$$F = \frac{1 - 2}{2}$$



- Metric proposed by Johnstone et al. **(Johnstone et al., 98)**

Evaluation: Fragmentation



Evaluation: Fragmentation

Fragmentation results Factor F

	Test1				Test2				Test3			
oc.	**Avg.**	**Stdv.**	**Max.**	**Min.**	**Avg.**	**Stdv.**	**Max.**	**Min.**	**Avg.**	**Stdv.**	**Max.**	**Min.**
st-fit	93,25	3,99	99,58	87,57	83,21	9,04	98,17	70,67	87,63	4,41	94,82	70,7
st-fit	10,26	1,25	14,23	7,2	21,51	2,73	26,77	17,17	11,76	1,32	14,14	9,7
nary-buddy	73,56	6,36	85,25	66,61	61,97	1,97	65,06	58,79	77,58	5,39	84,34	64,8
malloc	10,11	1,55	12,9	7,39	17,13	2,07	21,75	14,71	11,79	1,39	13,72	9
lf-fit	84,67	3,02	90,07	80,4	71,5	3,44	75,45	65,02	98,14	3,12	104,67	94,2
SF	10,49	1,66	11,79	6,51	14,86	2,15	18,56	9,86	11,15	1,1	13,91	7,4

Conclusions

- TLSF ia an allocator that performs operations
  - with predictable cost
  - very efficient (fast)
  - low fragmentation
- It permits:
  - To consider dynamic storage allocation in real-time systems (predictable operations)
  - The analysis of a resource as memory for systems with memory constraints (embedded systems)