An Algorithm with Constant Execution Time for Dynamic Storage Allocation *

Takeshi Ogasawara
Advanced Compiler Group
Tokyo Research Laboratory, IBM Japan, Ltd.
1623-14, Shimo Tsuruma, Yamato-Shi, Kanagawa 242, Japan
takeshi@vnet.ibm.com

Abstract

The predictability of the computation time of program modules is very important for estimating an accurate worst-case execution time (WCET) of a task in real-time systems. Dynamic storage allocation (DSA) is a common programming technique. Although many DSA algorithms have been developed, they focus on the average execution time rather than the WCET, making it is very difficult to calculate their WCET accurately. In this paper, we propose a new algorithm called Half-Fit [1] whose WCET can be calculated accurately. The algorithm finds a free block without searching on a free list or tree, allowing extra unusable memory called incomplete memory use. In a simulation following a queueing model of $M/G/\infty$, Half-Fit has the advantage over the binary buddy system [2] of more efficient storage utilization. The binary buddy system is a conventional algorithm whose WCET can be calculated.

1 Introduction

In real-time computing systems, tasks are scheduled to be completed before specified deadlines. However, if the execution time of a task in a real-time system is unknown at scheduling time, no scheduling algorithm can handle that tasks, and consequently the real-time system cannot satisfy its time requirements. In such systems, therefore, it is crucial to know *the worst-case execution time (WCET)* of tasks before scheduling. The WCET can be considered at two levels: the algorithm level and the implementation level. The WCET at the algorithm level is the time order of an algorithm in the worst case. The WCET at the implementation level is time to execute an algorithm on a practical implementation. The WCET at the implementation level as well as at the algorithm level is important because some algorithms that satisfy time requirements at the algorithm level cannot be implemented to exploit their algorithmic merits on modern computer architectures. In this paper, we focus on both levels.

As real-time systems treat complex world, we need the help of high-level languages. We can find many instances where practical real-time tasks have been developed by high-level languages such as Ada [3, 4, 5, 6, 7]. Generally, a real-time task consists of more than a program module. A program module executes an algorithm created by using language constructs such as for and while. In simple real-time systems, it is not difficult to ensure that all program modules in tasks have upper bounds on the execution time by using only programming techniques whose behaviors as regards timing are well-known. This restriction gives real-time systems the WCET of tasks, but creates extra difficulty for programmers developing complex real-time systems, and reduces the usability of some features of high-level languages.

Dynamic storage allocation (DSA) is a useful programming technique for maintaining memory objects whose sizes are not decided at runtime in high-level programming languages. DSA is also useful for maintaining dynamic memory objects whose lifetimes are shorter than that of a task efficiently as regards storage utilization. Non-real-time programs often use runtime library routines such as malloc() and free() for DSA under operating systems such as Unix. Modern high-level languages are designed heavily dependent on DSA[8], and Ada9X that is an improved version of Ada83 and will be the most commonly used real-time language still supports dynamic allocation feature[9].

DSA is often believed to be undesirable in real-time systems because (1) it is possible for DSA to exhaust address space and (2) the execution time of commonly used DSA algorithms are unpredictable. However, if the size of a data object is unknown at compile time, a compiler cannot generate code that reserves an exact storage of the object. There are many such situations [3]. Even if the size of a data object is known at compile time, it is wasteful and impractical to reserve storage of all objects statically when the lifetimes of the objects are shorter than a task and the activity of the reserved storage is low. So we consider DSA is important in real-time systems. Programmers should write an exception handler for error recovery from the problem (1) or the memory exhaustion[10]. In this paper, we focus on the problem (2) or the execution time of DSA algorithms.

Previous research has focused on how fast and how efficiently DSA algorithms allocate or deallocate memory blocks on the average. Conventional DSA algo-

*This study was completed at University of Tokyo's Sakamura Laboratory.

0-8186-7106-8/95 $04.00 © 1995 IEEE

rithms use free lists (or trees) to maintain free blocks, and algorithms other than *buddy systems* [2] search on the free lists. It is difficult or impractical to calculate their WCET because of the unpredictability of the search time. Therefore, conventional DSA algorithms that work fast and efficient in a non-real-time sense cannot be used in real-time programming. For this, algorithms whose WCET can be calculated and whose time complexity is $O(1)$ are appropriate. Buddy systems meet the second condition; however, in the worst case, they touch memory addresses that are not on any cache lines or TLB entries at the time of allocation or deallocation, thus causing cache misses and TLB misses in a processor. The memory efficiency of buddy systems is worse than that of other conventional algorithms (the reason for this is discussed in Section 2).

Our goal is to introduce a DSA algorithm whose WCET is $O(1)$, that touches a few cache lines, and whose memory efficiency is better than that of buddy systems. This paper proposes a new algorithm called *Half-Fit* [1] that has a constant execution time and whose WCET can be easily estimated. The key idea of the algorithm is to eliminate searching for free memory blocks, thus allowing extra unusable memory called incomplete memory use.

In Section 2, we briefly survey conventional DSA algorithms and discuss their non-real-time behaviors. In Section 3, we introduce a new DSA algorithm that has a constant execution time order and is appropriate for real-time programming. The results of an experiment comparing the memory efficiency of our approach with that of the binary buddy system are given in Section 4, and some conclusions are drawn in Section 5.

2 Conventional DSA Algorithms

2.1 Classes of Free Lists

All conventional DSA algorithms use free lists (or trees) to manage free memory blocks. They can be roughly classified into two groups according to the way in which free lists are constructed. One way is to have free blocks of different sizes on a free list; This way is called *variable-sized-block list search (VLS)*. The other is to have free blocks of a fixed size on a free list; This way is called *fixed-sized-block list search (FLS)*.

There are many VLS algorithms, such as First-Fit and Best-Fit [2, 11, 12]. Free blocks of various sizes are linked on a free list in VLS, so inevitably a free block larger than or equal to a requested size will be searched for on the free list. Previous research has focused on shortening the average search time on free lists to achieve fast memory allocation while preserving memory efficiency. For example, folding of free lists in the tree structure [13, 14, 15] and caching [16, 17, 18] have been introduced in order to reduce the search overhead.

FLS does not "search" in the same sense as VLS; that is, FLS does not search for a free block on a free list but searches for a free list that has free blocks to satisfy an allocation request. FLS algorithms other than buddy systems are not used alone; instead, they are usually combined with VLS. We call such composite FLS and VLS algorithms composite FLS (CFLS) algorithms. Quick-Fit [19, 20] and Fast-Fit [13, 14] are classified as CFLS algorithms. In FLS algorithms, free blocks of a fixed size are linked on each free list, so that there is no search as long as allocation requests succeed on the lists. In CFLS algorithms, when an allocation fails in FLS, VLS is invoked to find a free block on variable-sized-block lists.

Buddy systems [21, 2, 22, 23, 24, 25] are classified as FLS algorithms that have unique characteristics. They limit allocation sizes to a set of particular integers, and each request size is rounded up to the nearest integer in the set. These systems have an advantage over CFLS algorithms in that their execution time is bounded. The time complexity of the buddy systems is the order of the size of the set. The binary buddy system [21, 2] is most popular of the buddy systems. It allocates a free block by rounding a request size up to a power of two.

2.2 Classes of Coalescing Deallocated Blocks

There are three policies for coalescing deallocated blocks. The first is to coalesce as many deallocated blocks and free blocks as possible in a memory space are coalesced whenever blocks are deallocated. For example, the binary buddy system [2] coalesces a deallocated block with all the free blocks located at the *buddy* positions. The second is to coalesce deallocated blocks during searching at allocation time. The third is to coalesce deallocated blocks by a different process. GC (garbage collection) [12] and the Recombination-Delaying Buddy System [25] are classified as following the third policy.

The timing of coalescing is a trade-off between throughput and response time. The first coalescing policy is intended to improve the response time while reducing the throughput because of extra coalescing. The third policy is intended to improve the throughput while increasing the WCET of algorithms.

2.3 Classes of Interference in Memory Efficiency

There are two well-known types of impediment to memory efficiency: *internal fragmentation* and *external fragmentation* [2, 26, 27]. Once memory fragmentation occurs, allocation requests cannot be satisfied even if the total size of unused memory exceeds the requested sizes. Internal fragmentation is a situation in which unused memory spaces exist in allocated blocks; that is, in which DSA algorithms allocate memory blocks larger than the requested sizes. The "dead" spaces are never reallocated for other requests. External fragmentation is a situation in which allocation requests fail even though small separate free blocks exist in a memory space and have a total size sufficient to meet the requests. The small blocks result from splitting a block into a block of a requested size and other blocks.

2.4 Applicability to Real-Time Programming

VLS and CFLS algorithms cannot be applied to real-time programming, because their WCETs are unpredictable, owing to their searching on free lists. The

GC approach is not practical, because the WCET must be calculated if GC occurs every time allocation or deallocation.

On the other hand, FLS algorithms or buddy systems can be applied, because the WCET of buddy systems is bounded and $O(1)$. But, as explained in the previous section, in the worst case buddy systems access many memory addresses at both allocation and deallocation. There is a high possibility that the accesses will result in data cache misses and TLB entry misses, since *buddies* are distant from each other when free blocks are large. In addition, the memory efficiency of the buddy systems is the worst of all the conventional algorithms [28, 29, 30, 31, 32, 33].

3 Half-Fit

This section introduces a new DSA algorithm called *Half-Fit* [1]. Half-Fit has free lists on which free blocks of variable sizes are linked. However, it never searches on the free lists and does not belong to VLS. It takes a free block of a requested size from a free list on which blocks always satisfy the request. Half-Fit's behavior in taking blocks is similar to that of FLS. This section describes how free blocks are maintained and how allocation requests are satisfied.

3.1 Free Blocks Maintenance

In Half-Fit, free blocks whose sizes are in the range $[2^i, 2^{i+1})$ are grouped into a free list indexed by i. Deallocated blocks are immediately coalesced with neighboring free blocks. When a free block is created after coalescing and the size of the block is r (where the unit is bytes, words, etc.), we can use the following equation to compute the index i of the free list on which the block is linked:

$$i = \lfloor \log_2 r \rfloor. \tag{1}$$

i takes values from 0 to $wordlength - 1$. Half-Fit has a word-length bit vector to keep track of which free lists are empty and which are not. For example, i is 7 when r is 255.

Many CPUs have bit search instructions that are executed during a machine cycle. These instructions are useful for calculating the equation $\lfloor \log_2 r \rfloor$. Assuming a bit search instruction FFS that searches for the first set bit on a register from MSB to LSB, $\lfloor \log_2 r \rfloor$ can be calculated by executing

```
FFS r1=r1
```

where r1 is a register. FFS r1=r1, searches the first set bit on the register r1 and assigns the result to the register r1. FFS can be always executed in a machine cycle, because it needs no memory accesses. In the previous example on a 32-bit CPU, r1 is at first 00000000.00000000.00000000.11111111. After FFS, r1 is 00000000.00000000.00000000.00000111.

3.2 Allocation Algorithm

Half-Fit uses a special allocation technique in order to eliminate searching on free lists. Assuming that the size of a memory request is r, an index i that we will visit is calculated by the following equation:

$$i = \begin{cases} 0 & \text{if } r \text{ is } 1 \\ \lfloor \log_2(r-1) \rfloor + 1 & \text{otherwise} \end{cases} \tag{2}$$

The free list indexed by i keeps blocks whose sizes are between 2^i and $2^{i+1} - 1$. The sizes of requests to the free list i are between $2^{i-1} + 1$ and 2^i, so that requests are always satisfied by any blocks on the list. As previously noted, an index i can be computed by using bit search instructions. In addition to the previous assumption, assume that AI is an instruction to add a constant value to a value in a register, and SLA is a shift left instruction. The following codes compute an index i in a register r1 where r1 initially holds the value of r.

```
AI  r1=r1,-1
SLA r1=r1,1
FFS r1=r1
```

Once an index i has been calculated, a free block is taken from that free list indexed by i if the free list is not empty. If it is empty, Half-Fit takes a block from the subsequent non-empty free list whose index is closest to i. Bit search instructions are also useful for searching for the nearest non-empty free list.

If allocated blocks are larger than the request sizes, each is split into two parts: one that is returned to the requester and one that is relinked on a free list.

The allocation algorithm causes extra unusable free blocks, because the algorithm cannot find free blocks whose sizes are larger than a requested size r if i that is computed for r by the algorithm is larger than the index of the free list on which the free blocks are linked. The situation is called *incomplete memory use*.

3.3 Deallocation Algorithm

A memory block that is deallocated in Half-Fit is coalesced with adjacent free blocks in the first policy described in 2.2. The number of coalesced free blocks of Half-Fit is at most three, where that of the binary buddy system in the worst case is the number of bits in a word. DSA routines should touch more memory addresses as the number of coalesced blocks increases.

Free blocks adjacent to a deallocated block, of which the number is at most two, can be obtained in the time complexity of $O(1)$ in Half-Fit, because adjacent memory blocks are doubly linked with each other and the links are always maintained.

4 Experimental Simulation Results

To compare the memory efficiency of Half-Fit and that of the binary buddy system, we performed a set of simulations to test the failure ratios of allocation while limiting the total amount of memory space [19, 34, 12, 13] by using synthetic allocation/deallocation patterns. The failure ratios are deeply related to the frequency with which the algorithms issue system memory requests to operation systems by means of system functions such as **sbrk**, since the limited total amount of memory space is equivalent to a **break value** of **sbrk**.

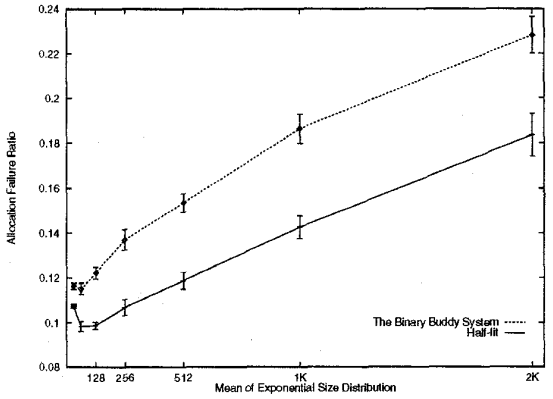

Figure 1: The ratio of allocation failure of Half-Fit vs. the binary buddy system varying means of the exponential size distribution

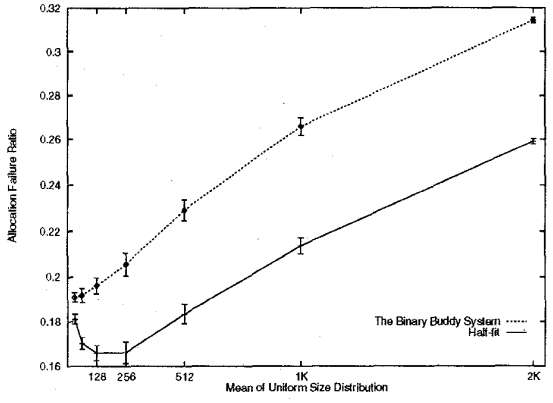

Figure 2: The ratio of allocation failure of Half-Fit vs. the binary buddy system varying means of the uniform size distribution

4.1 Simulation Methodology

According to previous studies, simulations are strongly affected by three distributions: the sizes of allocation requests, the lifetimes of allocated blocks, and the intervals at which allocation requests arrive [34]. It is necessary to choose these distributions carefully in order to obtain results that reflect the situations in actual usage as accurately as possible.

We chose two size distributions: exponential and uniform. The averages of the size distributions are determined, by referring to the total size of the memory managed by DSA algorithms, to be 32, 64, 128, 256, 512, 1024, and 2048 words, where the total amount of memory is 32 K words. The distribution of the lifetime of memory blocks is specified as uniform, and ranges from 5 to 15 time units. The distribution of intervals is specified as exponential, so that simulations are considered as a queueing model of $M/G/\infty$ and the number of allocated blocks follows a Poisson distribution of the mean λ/μ, where $1/\lambda$ is the mean of the arrival interval distribution and $1/\mu$ is the mean of the lifetime distribution.

4.2 Simulation Results

Figures 1 and 2 show the simulation results for exponential request size distribution and uniform request size distribution, respectively, with 95 percent confidence limit. Half-Fit has the advantage over the binary buddy system of greater memory efficiency.

5 Conclusions

This paper has proposed Half-Fit, a new DSA algorithm whose WCET is $O(1)$. This WCET can be accurately estimated by counting a small number of machine instructions. Half-Fit also guarantees a short WCET, because it accesses only a few memory addresses, whereas the binary buddy system whose WCET is also $O(1)$ but which accesses many memory addresses in the worst case, is unpredictable because of data cache misses and TLB entry misses. The simulation results show that Half-Fit has the advantage over the binary buddy system of greater memory efficiency.

Acknowledgments

We would like to thank Ken Sakamura and Hiroaki Takada at University of Tokyo for discussion.

References

[1] T. Ogasawara, "Half-Fit: A Real-Time Dynamic Storage Allocation Technique," Master's thesis, Dept. Information Science, University of Tokyo, 1991.

[2] D. E. Knuth, *The Art of Computer Programming, Vol. I: fundamental Algorithms.* Reading, Massachusetts: Addison-Wesley, 1968.

[3] A. Burns and A. Wellings, *Real-Time Systems and Their Programming Languages.* International Computer Science Series, Wokingham, England: Addison-Wesley, 1989.

[4] D. Christodoulakis, ed., *Ada-Europe International Conference*, (Athens, Greece), Springer-Verlag, May 1991.

[5] J. van Katwijk, ed., *11th Ada-Europe International Conference*, (Zaandvoort, The Netherlands), Springer-Verlag, June 1992.

[6] M. Gauthier, ed., *12th Ada-Europe International Conference*, (Paris, France), Springer-Verlag, June 1993.

[7] M. Toussaint, ed., *First International Eurospace-Ada-Europe Symposium*, (Copenhagen, Denmark), Springer-Verlag, September 1994.

[8] S. Kamin *et al.*, "Report of a Workshop on Future Directions in Programming Languages and Compilers," *ACM SIGPLAN Notices*, vol. 30, pp. 9–28, July 1995.

[9] Ada 9X Mapping/Revision Team, "Ada 9X Mapping, Volume II, Mapping Specification and Rationale (Annexes) Abridged, Version 4.1," Tech. Rep. IR-MA-1250-3, Intermetrics, Inc., Cambridge, Massachusetts, March 1992.

[10] Doug Smith, "Ada Quality and Style: Guidelines for Professional Programmers, Version 02.01.01," Tech. Rep. SPC-91061-CMC, Software Productivity Consortium, Inc., Herndon, Virginia, December 1992.

[11] C. Bays, "A Comparison of Next-fit, First-fit, and Best-fit," *Communications ACM*, vol. 20, no. 3, pp. 191–192, 1977.

[12] N. Nielsen, "Dynamic Memory Allocation in Computer Simulation," *Communications ACM*, vol. 20, pp. 864–873, November 1977.

[13] M. J. Tadman, "Fast-Fit: A New Dynamic Storage Allocation Technique," Master's thesis, Dept. Computer Science, University of California at Irvine, 1978.

[14] T. A. Standish, *Data Structure Techniques*. Reading, Massachusetts: Addison-Wesley, 1980.

[15] R. P. Brent, "Efficient Implementation of A First-Fit Strategy for Dynamic Storage Allocation," *ACM Trans. Programming Languages and Systems*, vol. 11, no. 3, pp. 388–403, 1989.

[16] G. Bozman, "The Software Lookaside Buffer Reduces Search Overhead with Linked Lists," *Communications ACM*, vol. 27, no. 3, pp. 222–227, 1984.

[17] R. R. Oldehoeft and S. J. Allan, "Adaptive Exact-Fit Storage Management," *Communications ACM*, vol. 28, no. 5, pp. 506–511, 1985.

[18] D. Grunwald and B. Zorn, "CustoMalloc: Efficient Synthesized Memory Allocators," *Software – Practice and Experience*, vol. 23, no. 8, pp. 851–869, 1993.

[19] C. B. Weinstock, *Dynamic Storage Allocation Techniques*. PhD thesis, Dept. Computer Science, Carnegie-Mellon University, Pittsburgh, April 1976.

[20] C. B. Weinstock and W. A. Wulf, "Quick Fit: An Efficient Algorithm for Heap Storage Allocation," *SIGPLAN Notices*, vol. 23, no. 10, pp. 141–148, 1988.

[21] K. C. Knowlton, "A Fast Storage Allocator," *Communications ACM*, vol. 8, pp. 623–625, October 1965.

[22] D. S. Hirschberg, "A Class of Dynamic Memory Allocation Algorithm," *Communications ACM*, vol. 16, pp. 615–618, October 1973.

[23] K. K. Shen and J. L. Peterson, "A Weighted Buddy Method for Dynamic Storage Allocation," *Communications ACM*, vol. 17, no. 10, pp. 558–562, 1974.

[24] J. L. Peterson and T. A. Norman, "Buddy Systems," *Communications ACM*, vol. 20, pp. 421–431, June 1977.

[25] A. Kaufman, "Tailored-List and Recombination-Delaying Buddy Systems," *ACM Trans. Programming Languages and Systems*, vol. 6, no. 1, pp. 118–125, 1984.

[26] B. Randell, "A Note on Storage Fragmentation and Program Segmentation," *Communications ACM*, vol. 12, pp. 365–372, July 1969.

[27] J. E. Shore, "On the External Storage Fragmentation Produced by First-fit and Best-fit Allocation Strategies," *Communications ACM*, vol. 18, no. 8, pp. 433–440, 1975.

[28] J. Campbell, "A Note on Optimal-fit Method for Dynamic Allocation of Storage," *Computer Journal*, vol. 14, no. 1, pp. 7–9, 1971.

[29] J. Fenton and D. Payne, "Dynamic Storage Allocation of Arbitrary Sized Segments," in *Proceedings IFIP '74*, (Amsterdam), pp. 344–348, 1974.

[30] D. G. Korn and K.-P. Vo, "In Search of a Better Malloc," in *Proceedings of the Summer 1985 USENIX Conference*, pp. 489–506, 1985.

[31] B. Z. Dirk Grunwald and R. Henderson, "Improving the Cache Locality of Memory Allocation," in *SIGPLAN '93 Conference on Programming Language Design and Implementation*, (Albuquerque), pp. 177–186, 1993.

[32] B. Zorn and D. Grunwald, "Evaluating Models of Memory Allocation," *ACM Trans. Modeling and Computer Simulations*, vol. 4, no. 1, pp. 388–403, 1994.

[33] D. Detlefs and A. Dosser, "Memory Allocation Costs in Large C and C++ Programs," *Software – Practice and Experience*, vol. 24, no. 6, pp. 527–542, 1994.

[34] J. E. Shore, "Anomalous Behavior of the Fifty-percent Rule in Dynamic Memory Allocation," *Communications ACM*, vol. 20, no. 11, pp. 812–820, 1977.