Hardware Support for Fast and Bounded-Time Storage Allocation *

Extended Abstract

Steven M. Donahue, Matthew P. Hampton, Ron K. Cytron, Mark Franklin
Washington University Box 1045
Department of Computer Science
St. Louis, MO 63130 USA

Krishna Kavi
University of North Texas

March 22, 2002

Abstract

With the advent of operating systems and programming languages that can evaluate and guarantee real-time specifications, applications with real-time requirements can be authored in higher-level languages. For example, a version of Java suitable for real-time (RTSJ) has recently reached the status of a reference implementation, and it is likely that other implementations will follow.

Analysis to show the feasibility of a given set of tasks must take into account their worst-case execution time, including any storage allocation or deallocation associated with those tasks. In this paper, we present a hardware-based solution to the problem of storage allocation and (explicit) deallocation for real-time applications, Our approach offers both predictable and low execution time: a storage-allocation request can be satisfied in the time necessary to fetch one word from memory.

We have implemented our approach in the context of IRAMs (intelligent storage) using FPGAs and our approach is based on Knuth's buddy algorithm. We present the design, implementation, and experimental results of our approach.

1 Introduction

Most modern programming languages offer some mechanism for for *dynamic storage management* [7, 6]. Figure 1 illustrates the "life" of a typical object. At some point, the application allocates storage for an object from an area called the *heap*. The program has access to the object for some period of time, during which we say the object is *live*. At some point, the object becomes inaccessible or *dead*. Subsequently, the storage associated with the object is *deallocated*—returned to the heap to help satisfy future storage requests. The action that triggers deallocation varies by language: some languages offer automatic garbage collection [6] while others require explicit action to be taken by the application.

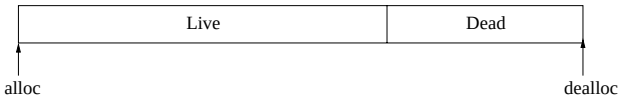

Figure 1: The life of an object.

For allocation, languages such as Java and C offer primitives such as `new` and `malloc` that cause a specified or implied number of bytes to be taken from the heap and allocated for the program's use. Although dynamic-storage usage will vary by application, there are important applications that use dynamic storage intensively. Performance of such applications, particularly in a real-time environment, can be significantly influenced by their storage-allocation facility.

Recently, standards for real-time programming languages have emerged that bring modern, high-level languages within reach of real-time applications. An example of this trend is the **Real-Time Specification for** Java (RTSJ) [1], which provides for bounded-time dynamic-storage allocation. Because Java mandates initialization of dynamically allocated storage, a block of n bytes cannot be allocated in less than $\Omega(n)$ time. For C and C++ where such intialization is unnecessary, dynamic storage could be allocated in constant time. Factoring out initialization, the common challenge for an allocator is *finding* a suitable block in constant time.

Allocators based on an unorganized free-list (illustrated

*Sponsored by NSF under grant ITR–0081214; contact author cytron@cs.wustl.edu

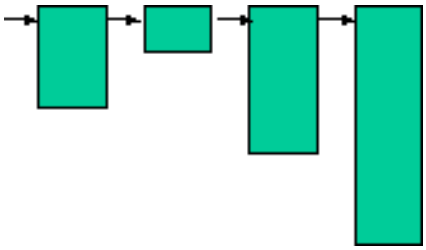

Figure 2: An unorganized free-list: the block that can satisfy a storage request may be at the end.

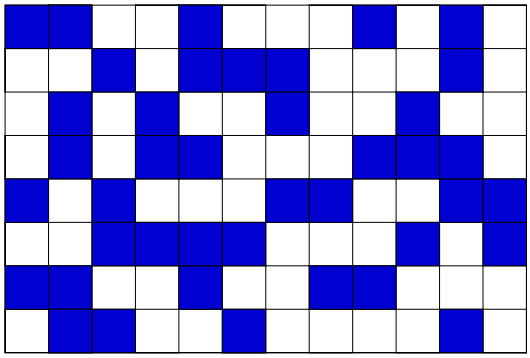

Figure 3: A customized allocator manages a set of identically sized blocks. The unshaded blocks are in use by the program and the shaded blocks are threaded in a linked list (not shown).

in Figure 2) cannot deliver on the lower bound, because the block that satisfies a given allocation request could lie at the very end of the list. In such a case, the time to satisfy a request for n bytes is $O(N)$, where N is the size of the storage heap rather than $\Omega(n)$. Because such performance is unacceptable for real-time systems, an allocator based on unorganized free-lists is unsuitable.

Allocators with *organized* free-lists segregate available blocks by their *size*, typically using one of the following approaches.

General allocator: One example of a general approach [7] is Knuth's *buddy system* [5], illustrated in Figure 10, and discussed in Section 3. For a heap of size N, there are $\log_2 N$ lists of identically sized blocks, where list i contains free blocks of size 2^i. Thus, a suitable block can be found in $O(\log N)$ time. Such performance is considered *reasonably bounded*, although technically such allocation takes greater than constant time. Other variations on a general, segregated-by-size allocator are also suitable [7].

A disadvantage of this approach is that the *average* performance of the buddy allocator is typically worse

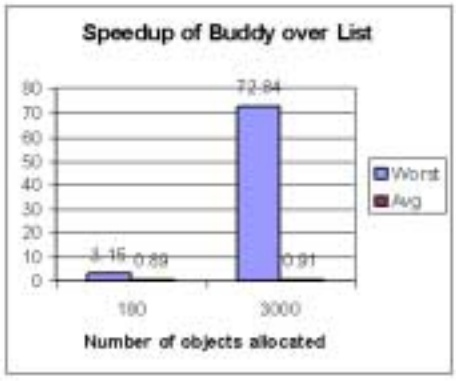

Figure 4: Contrived worst-case application for an unstructured list allocator. The worst-case time can become arbitrarily bad, but the average performance beats the buddy system.

than the free-list allocator [4]. Figure 4 shows results for a contrived example that allocates and deallocates storage such that the block to satisfy a request is at the end of the unstructured allocator's free-list. However, even for this contrived example, average performance of the free-list allocator is stronger than for the buddy system.

The large gap between worst- and average-case performance means that real-time applications must grossly overprovision for the execution time of the free-list allocator. The buddy system offers a much better ratio of worst- to average-case performance, but with an overall performance loss on average.

Application-specific allocator: A very efficient and bounded-time allocator can be written specifically for a given application as follows. Suppose it is known that the application allocates blocks of size $s_1, s_2, \ldots, s_d$ with at most $l_1, l_2, \ldots, l_d$ occurrences of those blocks concurrently live. The application is given d distinct allocators, one for each block size; allocator i is given $s_i \times l_i$ storage to satisfy requests for that block size. Figure 3 illustrates the status of one such allocator in the middle of an application's execution. All blocks are identically sized, and the unshaded blocks are considered to be live at this point. The available blocks (shaded) are simply linked together in a list. A block is allocated from the head of that list in constant time; similarly, a block is deallocated by placing at the head of the list in constant time.

The application must be rewritten to use invoke the block-specific allocator at the appropriate points in

the program.[1]

A disadvantage of this approach is that the application and allocator are highly specialized and excessively coupled, which can make development, testing, and maintenance of the application more costly. Also, the total storage required for such applications is

$$\sum_{i=1} ds_d \times l_d \geq M$$

where M is an application's *maxlive* statistic—the maximum number of bytes that are concurrently live during the application's execution.[2] When the above summation significantly exceeds M, then the amount of heap overprovisioning becomes costly for embedded applications.

In summary, the ideal storage allocator has the following characteristics:

- It can be used without modification on any application.
- It finds a suitable block in constant time.
- It does not require unreasonable overprovisioning of the heap.
- The gap between its worst-case and average-case performance is as small as possible.
- Its overall speed is as fast as possible.

In this paper, we present hardware support for storage allocation to achieve the above properties. This hardware imposes a very small incremental cost to the storage subsystem, yet it can find a block of storage in the time it takes to read a single storage location. The rest of this paper is organized as follows. Section 2 introduces some Java and C benchmarks for this paper and presents measurements concerning the dynamic storage-allocation behavior of those applications. Space limitations prohibit a detailed explanation of the buddy system [5], but the essence of the approach is presented in Section 3. Section 3.1 presents a simple VHDL implementation of the buddy system—essentially a straightfoward translation of the software algorithm into hardware. Section 3.3 presents two optimizations that signicantly improve the time needed to find a suitable block for an allocation request. Section 4 presents experiments that quantify the effects of our work and Section 5 presents conclusions and ideas for future work in this area.

[1] If the block-size at a given allocation site is unknown prior to runtime, then the requisite allocator can be looked up in $O(d)$ worst-case time. For most applications, $d \ll N$, and so can be considered constant.

[2] In other words, a heap of less than M bytes would cause the application to fail regardless of the allocation approach.

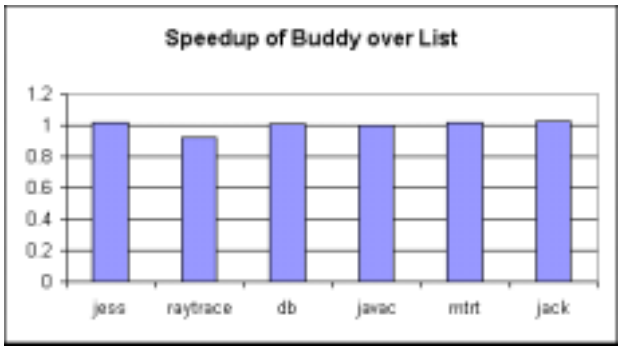

Figure 5: Speedup of the software buddy system over the unstructured list allocator, size 100 SPEC benchmarks.

2 Benchmarks and their Storage-Allocation Times

In this section we characterize the Java and C benchmarks used in our study. After presenting some simple statistics about the benchmarks, we examine the overhead of Java's and C's standard allocator, which uses an unstructured free-list as illustrated in Figure 2.

In Figure 8 we see the SPEC [3] benchmarks, and note that two of them (`mpegaudio` and `compress`) are computational in nature and thus do not allocate many objects.

The C benchmarks are shown in Figure 9. These applications were part of a suite of malloc benchmarks [10]. Two of the C benchmarks generate a significant number of mallocs, while the others are more computational.

2.1 Time spent in the storage allocator

Figure 6 shows the fraction of time spent in storage allocation for the large runs of the SPEC benchmarks shown in Figure 8. The data in Figure 6 was obtained by running applications in Sun's JDK 1.1.8 interpreter. The overall execution times were captured as well as the times spent in the storage allocator (exclusive of any garbage collection necessary to satisfy a request). In those runs, up to 8.5% of the execution time was spent on storage allocation. Figure 5 illustrates the performance of a software implementation of the buddy system, as implemented by us in Sun's JDK 1.1.8. The resulting performance is close to the unstructured-list allocator. Even in its software implementation, the buddy system offers reasonable bounds on allocation time, yet it does not dramatically improve allocation time overall nor does it always outperform the unstructured list allocator.

Allocation costs for the C benchmarks described in Figure 9 are shown in Figure 7. In these benchmarks,

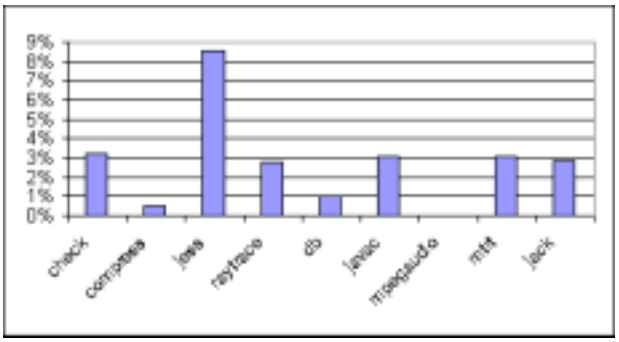

Figure 6: Percent of runtime spent in the storage allocator (unstructured list) for the SPEC benchmarks. Note these were the large (size 100) runs of these benchmarks. The size-1 runs (used in the latter part of this paper) generally spent less time in allocation.

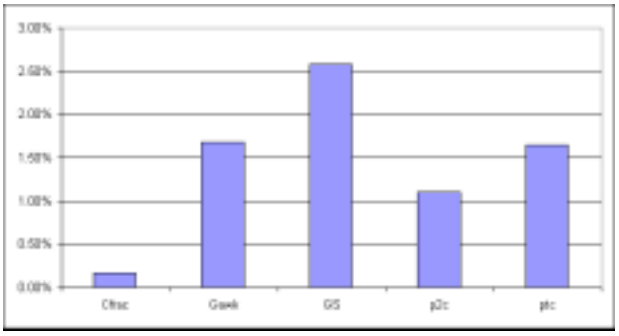

Figure 7: Percent of runtime spent in the storage allocator (`malloc`) for the C benchmarks.

up to 2.5% of the applications' execution times was consumed by the storage allocator.

3 Design and Implementation

In this section, we present two designs for a hardware-based buddy-system allocator. The first design, the Reference Buddy System (RBS) is a straightforward translation of the software algorithm into hardware. The second design, the Optimized Buddy System (OBS), leverages hardware characteristics to perform optimizations that significantly improve the worst-case time bound for storage allocation. In Section 4 we assess the performance of our hardware system compared with the standard software (list-based) allocator as well as with a software implementation of the buddy system.

Buddy System Allocation

We begin with a quick summary of Knuth's buddy system [5]. Each storage request is resolved to a block of size 2^k for some positive, integral value of k. Figure 10(a) shows the buddy system's structure in its initial state, assuming the heap is 256 bytes.

The buddy system operates as follows:

1. When the program requests memory, the allocator first calculates the smallest power of 2 that is larger than or equal to the size requested. More specifically, a request of size s is translated into a request of size $2^k, k = \lceil \log_2 s \rceil$.

2. The free-list at index k is consulted for an available block.

3. If a block of size 2^k is not available, then two such blocks can be obtained through bisection of a block of size 2^{k+1}. Figure 10(b) shows the result of subdividing the initial heap into two sub-blocks.

4. Applying this strategy recursively, increasingly larger blocks can be subdivided until a block of size 2^k can be obtained.

For example, the heap in Figure 10(c) has blocks available of size 16. Thus, a request for a block of size 10 can be satisfied immediately, with the resulting block returned in the time it takes to unlink a block from the size-16 free-list.

Further, consider a request for a block of size 8. Because the list of blocks of size 8 is empty, the buddy system hunts upwards for a larger block that can be subdivided to obtain the desired size. The time necessary for that search is $O(\log M)$ where M is the size of the storage heap. For embedded and real-time systems, we assume that the heap size is fixed and that $O(\log M)$ time is considered efficient.

Buddy System Deallocation

When blocks are deallocated, a common problem for most storage-management algorithms is the coalescing of free blocks that happen to lie consecutively into larger blocks. The buddy system greatly simplifies this task. When a block of size 2^{k+1} is bisected into two blocks of size 2^k, the resulting blocks are said to be *buddies* of each other. A buddy of size 2^k can compute its buddy's address by flipping a predetermined bit of its own address—typically bit k where bit 0 is the rightmost bit.[3]

Figure 10(c) shows the result of requesting a block of size 16 given the initial condition shown in Figure 10(a). The initial block is recursively subdivided until two

[3]Without loss of generality, we assume the heap's origin is address 0.

Name	Description	Lines of Source	Objects Created	Execution Time (sec)
compress	Modified Lempel-Ziv	6,396	10129	7463
jess	Expert System	570	7,923,782	1802
raytrace	Ray Tracer	3750	6,346,487	2101
db	Database Manager	1020	3,210,520	3766
javac	Java Compiler	9485	5,902,305	1969
mpegaudio	MPEG-3 decompressor	N/A	7,555	8519
mtrt	Ray Tracer, threaded	3750	6,586,584	2223
jack	PCCTS tool	N/A	6,579,042	2336

Figure 8: SPEC benchmark properties.

Name	Description	Lines of Source	Number of Mallocs	Execution Time (sec)
cfrac	Continued Fraction Algorithm	3,644	1,528	2.05
gawk	GNU's AWK interpreter	10,737	723,498	120.72
gs	Aladdin Ghostscript	25,388	108,541	17.71
p2c	Pascal to C Converter	35,581	5,479	1.01
ptc	Pascal to C Converter	9,974	2,885	0.35

Figure 9: C benchmark statistics.

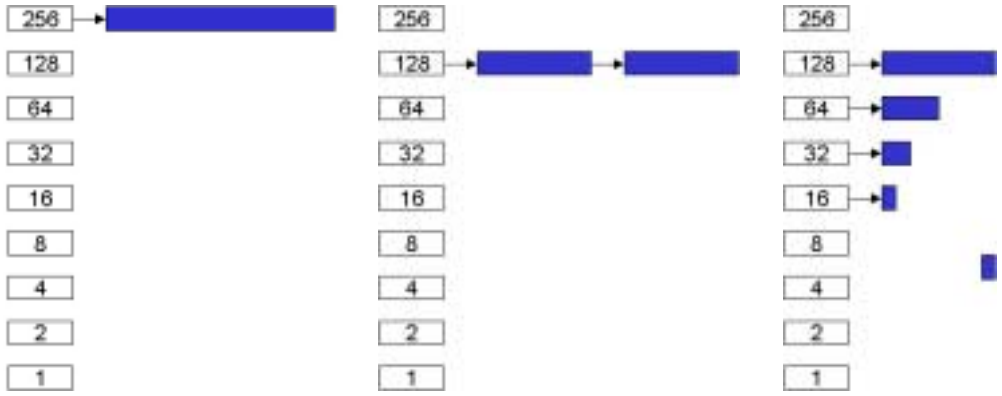

Figure 10: A block (a) can be divided into two sub-blocks (b) and this proceeds recursively to satisfy an allocation request. In the end (c), two blocks of 16 bytes are obtained; one remains free and other is returned to satisfy the request.

blocks of size 16 are obtained. One of those blocks is returned to satisfy the allocation request, and the other block remains on the free-list for blocks of size 16.

When storage is returned, the buddy system eagerly joins buddies to create ever larger blocks. Thus, if the block allocated in Figure 10(c) is immediately deallocated, buddies are joined together repeatedly until the heap is returned to the state shown in Figure 10(a).

3.1 Hardware Design of Buddy Algorithm

Our hardware design of Knuth's buddy algorithm supports *alloc* and *dealloc* as described above for software as well as an *init* operation.

The *init* operation initializes the heap to a given size. As with the software implementation, the heap is segregated by size into lists for the relevant powers of two. The free blocks are maintained in a doubly-linked list to facilitate coalescing (described below), which can remove blocks from the middle of a list. In addition to the list pointers, each free block contains the size of itself and a free/busy bit to indicate whether the block is currently allocated.

We store this *header* information at the front of each block in the heap. Because blocks are powers of two, we can reduce the header to three (32-bit) words—the busy/free bit can be the least-significant bit (LSB) of the size field. When a block is unallocated, we require 3 words of header space. Thus, the minimum size that can be allocated by our design is 16 bytes (16 is the smallest power of two larger than 12). Allocated blocks need only remember their size, so the overhead there is a single word of storage. Thus, when a program requests a block of size s bytes, a block of size $s + 4$ is required. Because all blocks are powers of two, a block of size $2^k, k = \lceil \log_2 s + 4 \rceil$ is requisitioned to satisfy the request.

The *alloc* operation allocates a block of size 2^k by searching on the list at k or above. The time to find the smallest block equal to or larger than the requisite size is called the *Find Time*. If found on a larger list, the block must be repeatedly subdivided until the requisite size is obtained (as in Figure 10). For example, suppose block B of size y was found, and $y > 2^k$. First B would be removed from its free list. Second, B's buddy at size $y/2$ is calculated, and inserted into the list at level $y/2$. The buddy calculation and list insertion is executed until $2^k = y$. The time to break down the block from y to 2^k is called the *Block Time*.

The *dealloc* operation eagerly coalesces a returned block with its buddies on subsequently larger levels until it encounters a buddy that is not free. That is, suppose block R of size 2^k is returned. First R's buddy at size 2^k is inspected. If it is not free, then R is inserted in the list for free blocks of size 2^k; if it is free, the buddy is removed from its free list, joined with R, and this process continues until a unfree buddy is found.

3.2 Design of RBS

The RBS was designed to be a simple hardware implementation of Knuth's buddy algorithm. The system is a computational resource whose complete environment includes an off-chip memory as well as a memory controller. The design of the core RBS logic is shown in Figure 11, ignoring the shaded box.

We require the following hardware components to implement the computational functions required by the buddy algorithm.

- Two shift registers keep track of the block sizes, thus eliminating a potentially expensive multiplication unit from the system.
- Size registers allow the base-2 multiplication and division necessary for the *alloc* and *dealloc* operations.
- One register file provides general-purpose storage for the algorithm. A second register file contains the head pointers of each free-list in the heap.
- Two registers store addresses and data that are sent to/from the off-chip memory.
- A simple, specialized ALU provides operations to calculate buddies (XOR) and pointer offsets (plus/minus).
- A controller handles the execution of the algorithm.

Most of the complexity of the RBS is located in the controller, which is comprised of three logical components. Each component maps to one of the three operations described in Section 3.1. The controller was implemented using a 135-state Finite State Machine. We implemented a separate memory system controller to handle the low-level details of the memory interface and reduce the complexity of the RBS controller.

3.3 Improved design

The nature of a hardware computation makes it possible to add two optimizations to the RBS. We call the resulting system the Optimized Buddy System (OBS), whose block diagram is shown in Figure 11, including the shaded box.

The first of these optimizations, *Fast Find*, is a module that reduces the time taken to find an allocatable block. The second optimization, *Fast Return*, leverages a property of the buddy algorithm to return an answer for an *alloc* operation before the *alloc* operation has completed.

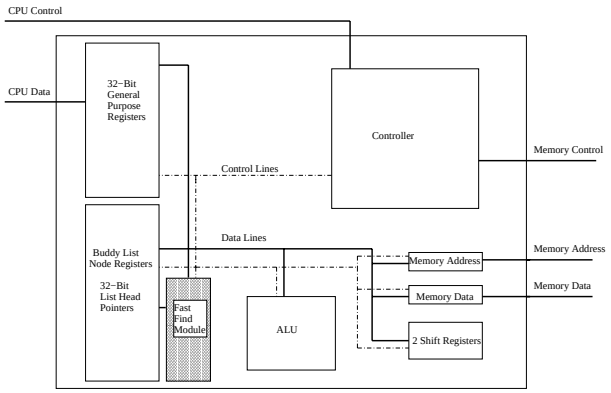

Figure 11: General architecture of our hardware allocator. The shaded box represents optimizations described in Section 3.3.

3.3.1 Fast Find Module

The Fast-Find module uses hardware to search the head pointers of the free list in parallel for a block that satisfies a memory request. Since blocks must be powers of two, all blocks are aligned on an even-byte boundary. Thus, any available block will have an address whose Least Significant Bit (LSB) is 0. We organized our free-list structure such that a head pointer for a list which is empty has an LSB of 1. The OBS then can use a parallel search, as described below, to find a suitable list whose LSB is 0.

The Fast-Find module first masks out the head pointer LSBs of any lists whose corresponding block sizes are smaller than the requested size. In parallel, it passes the remainder of the LSBs through a leading-zero detector. The first zero that is found by the detector will be the LSB of the pointer to the block we want to allocate. The leading-zero detector in the OBS implementation uses Sklansky's parallel prefix algorithm [9]. In contrast, the RBS uses a linear search on the LSBs with a worst-case complexity of $O(\log N)$ where N is the heap size in bytes. The OBS reduces this to constant time with a parallel search of complexity $O(\log \log N)$, which is effectively bounded in practice by a small constant.

3.3.2 Fast Return

One property of Knuth's buddy algorithm that we can take advantage of is that on an allocation request, the address of the block that will be returned is known early in the allocation operation. The remainder of the time is spent breaking down blocks to update the state of the buddy structure. An obvious optimization would be to have the allocator return the block as soon as the Fast-Find module locates it, and restore the buddy structure as the requesting program continues to execute. The success of this approach depends on the assumption that allocations do not occur at an interval smaller than the amount of time required for the allocator to recover. We address this issue further in the next section.

3.4 Optimized Synthesis

The OBS was synthesized to a Xilinx Virtex IIe XCV1000E FPGA to determine its resource consumption. Overall, the core buddy system, excluding the off-chip memory and memory controller, consumed 3,988 Look-Up Tables (LUT) out of the provided 24,576 (16%). The equivalent gate count for the design was 40,348. These numbers show that the buddy system consumes very little hardware real-estate.

4 Experiments

In this section we present the results of experiments to evaluate the performance of our hardware-based buddy system implementations.

4.1 Speedup of RBS compared to Software

We begin by comparing the performance of the RBS allocator with our software buddy implementation. As shown in Figure 12, the RBS provided significant savings in `malloc` time compared to the software allocator. As expected, this savings does not translate into overall performance gains for these benchmarks, because the contribution of the `malloc` calls to the overall execution time is so small (see Figure 7). GS is the only C benchmark for which the effect on overall execution time is significant. The corresponding data for the Java SPEC benchmarks shown in Figure 13 are similar to the results seen for the C benchmarks.

4.2 Evaluation of OBS compared to RBS

In Section 3.3, we presented two optimizations for improving the performance of our allocator. We evaluate the implementation of the OBS in this section.

First, we evaluate the *average-case* performance, which is of interest to non-real-time applications. The comparisons between the RBS and OBS showing the average-case performance for the Java SPEC benchmarks is shown in Figure 14. The complexity added by the Fast-Find Module, as explained in Section 3.3.1, does affect the *average-case* performance, which suffers by about 4–6 clock cycles for each benchmark.

Second, we evaluate the *worst-case* allocation time, which is important for real-time applications. We also

Name	Number of Mallocs	Sum of Malloc Times in C(ns)	Sum of Alloc Times in RBS(ns)	RBS Time Savings(ns)	Savings % of Exec Time	Savings % of Malloc Time
Cfrac	1,528	3,435,085	2,691,726	743,358	0.036	21.64
Gawk	500,000	1,402,012,204	530,971,970	871,040,233	N/A	62.13
GS	108,541	457,844,619	102,653,089	355,191,529	2.06	77.58
P2C	5,479	11,209,591	6,409,678	4,799,912	0.48	42.82
PTC	2,885	5,635,971	5,078,744	557,226	0.17	9.89

Figure 12: C `malloc` times in software and VHDL.

Name	Number of Mallocs	Sum of Malloc Times in SW(ns)	Sum of Alloc Times in RBS(ns)	RBS Time Savings(ns)	Savings % of Exec Time	Savings % of Malloc Time
Compress	5,124	10,346,341	7,197,345	3,148,995	0.0005	30.44
Jess	45,868	90,520,454	53,334,023	37,186,430	0.4038	41.08
Raytrace	276,961	525,332,397	293,451,559	231,880,837	0.4994	44.14
DB	7,609	15,908,353	9,935,946	5,972,406	0.2252	37.54
Javac	26,114	51,651,837	30,952,678	20,669,158	0.3024	40.07
MpegAudio	7,551	15,781,356	9,671,071	6,110,284	0.0091	38.72
MTRT	276,085	542,677,803	297,405,369	245,272,433	0.5282	45.20
Jack	393,745	743,994,511	449,201,238	294,793,272	0.2537	39.62

Figure 13: Java Alloc Times in Software and VHDL

Name	RBS	OBS
Compress	8.91	12.0
Jess	6.47	12.0
Raytrace	5.66	12.0
DB	7.92	12.0
Javac	7.0	12.0
MpegAudio	7.69	12.0
MTRT	5.65	12.0
Jack	6.43	12.0

Figure 14: Average-case performance of hardware *Find* in clock cycles.

Name	RBS	OBS
Compress	125	13.0
Jess	125	13.0
Raytrace	125	13.0
DB	125	13.0
Javac	125	13.0
MpegAudio	125	13.0
MTRT	125	13.0
Jack	125	13.0

Figure 15: Worst-case performance for hardware *Find* in clock cycles.

examine the ratio of worst-case to average-case performance, to study the extent to which real-time applications must *overprovision* the cost of running the allocator.

As shown in Figure 15, the optimized version finds an allocatable block effectively in $\Theta(1)$ time. Thus, the OBS drastically improves the bound for worst-case *Find* times. In contrast, the complexity of RBS Find is $O(\log N)$ where N is the size of the heap. Comparing the OBS to a software-based allocator, we see that the worst-case bounds are even more distinct. Figure 16 shows that the software implementation is approximately 5 times slower than the OBS.

The OBS also does very well in terms of tight provisioning for real-time systems. A real-time system must provision for worst-case behavior. When worst- and average-case performance is similar, such provisioning is efficient. Figure 17 shows the ratio of average- to worst-case for OBS and RBS *Find* times.[4] Clearly, real-time systems should favor OBS because it leads to markedly less overprovisioning.

Finally, we analyze the effect of fast-return on actual benchmarks. Recall from Section 3.3.2, that fast-return overlaps *Block Time* with the application. If two allocation requests come too soon in succession, then OBS may not be ready to satisfy the second request, because a *Block* operation is still in progress.

Figure 18 examines the interarrival times of the `malloc` requests. For each C benchmark, the minimum

[4]The *Block* times are identical.

Name	Ratio Software/OBS
Cfrac	4.60
Gawk	9.79
GS	5.25
P2C	5.49
PTC	5.24

Figure 16: Worst-case C `malloc` times.

Name	RBS	OBS
Compress	0.0713	0.9231
Jess	0.0518	0.9231
Raytrace	0.0453	0.9231
DB	0.0634	0.9231
Javac	0.0536	0.9231
MpegAudio	0.0615	0.9231
MTRT	0.0452	0.9231
Jack	0.0515	0.9231

Figure 17: Ratio of average-case to worst-case for hardware *Find*.

alloc interarrival time is greater than the worst-case *Block Time*. Thus, each *Block* operation completes before another *alloc* request arrives.

Figure 19 shows that the performance savings due to fast return could be greater than 96–98% compared to the C software allocator. The effect on overall application performance varies from %0.16 to %2.54.

Figure 20 shows the interarrival allocation times for our Java benchmarks. Unlike the C benchmarks, allocation requests do come closely spaced—so much so in some cases that the later allocation must wait for the earlier one's *Block* operation to finish. However, in examining how often this occurs, we found that less than 15% of the requests were delayed for the `jess` benchmark. We are completing a study of the SPEC benchmarks and their actual waiting times for our allocator. Even if delayed, performance is still far stronger with the hardware-based allocator than with software.

Name	Interarrival Time(ns)	OBS Block Time(ns)
Cfrac	26,963	20,565
Gawk	26,455	19,583
GS	26,550	12,708
P2C	26,774	14,672
PTC	26,550	14,672

Figure 18: Interarrival times vs. *Block* times of C benchmarks.

Name	% Savings on Malloc	% Savings on Application
Cfrac	96.56	0.16
Gawk	97.24	1.65
GS	98.17	2.54
P2C	96.22	1.06
PTC	96.04	1.58

Figure 19: Fast-Return Effect On Performance.

Name	Interarrival Time(ns)	OBS Block Time(ns)
Compress	13,036	16,619
Jess	7,566	16,619
Raytrace	8,756	16,619
DB	12,619	16,619
Javac	9,536	16,619
MpegAudio	7,571	16,619
MTRT	9,976	16,619
Jack	10,512	16,619

Figure 20: Interarrival times vs. *Block* times of Java benchmarks.

5 Conclusions

The ideal storage allocator would be fast, general-purpose, have constant *Find* time, and as small a gap between average- and worst-case as possible. We have implemented and evaluated a hardware storage-allocation system based on Knuth's buddy algorithm and have contributed two optimizations that are due to hardware implementation. As shown in Section 4, we nearly achieved the goals of an ideal allocator. In terms of *Find*, our system is as fast as a storage read.

Future work will stress improving the worst-case bound on *Block* time. We are currently exploring several ideas as follows:

- Conceptually, the *Block* operation could be parallelized: each buddy level can try to obtain its portion of a subdivided block concurrently.
- We currently store an object's header information with the object in the heap space, which resides in (relatively slow) RAM. We would like to evaluate the performance effects of moving the header to a smaller, faster memory, to improve the efficiency of storage operations within the allocator.
- We would like to evaluate the architectural and performance issues of supporting garbage-collection operations (mark/sweep, reference counting [8], and other algorithms [2]) in the memory system.

Acknowledgements

We thank Morgan Deters and Dante Cannarozzi for help with obtaining trace data for this paper. We thank Ben Zorn for publishing the C storage-allocating benchmarks.

References

[1] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[2] Dante J. Cannarozzi, Michael P. Plezbert, and Ron K. Cytron. Contaminated garbage collection. *Programming Language Design and Implementation*, 2000.

[3] SPEC Corporation. Java SPEC benchmarks. Technical report, SPEC, 1999. Available by purchase from SPEC.

[4] Steven M. Donahue, Matthew P. Hampton, Morgan Deters, Jonathan M. Nye, Ron K. Cytron, and Krishna M. Kavi. Storage allocation for real-time, embedded systems. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software: Proceedings of the First International Workshop*, pages 131–147. Springer Verlag, 2001.

[5] Donald E. Knuth. *Fundamental Algorithms, Volume 1, The Art of Computer Programming, Second Edition*. Addison-Wesley, 1973.

[6] Paul R. Wilson. Uniprocessor garbage collection techniques (Long Version). Submitted to ACM Computing Surveys, 1994.

[7] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Henry Baker, editor, *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, September 1995. Springer-Verlag.

[8] David S. Wise, Brian Heck, Caleb Hess, Willie Hunt, and Eric Ost. Research demonstration of a hardware reference-counting heap. *Lisp Symb. Comput.*, (2):159–181, July 1997.

[9] Reto Zimmermann. VHDL library of arithmetic units. Technical report, Integrated Systems Laboratory, ETH Zürich, 1998.

[10] Benjamin Zorn and Dirk Grunwald. Empirical measurements of six allocation-intensive c programs. *SIGPLAN Notices*, 27(12):71–80, 1992. Available by `ftp://ftp.cs.colorado.edu/pub/misc/malloc-benchmarks`.