A Hardware Assisted High Performance PHK Memory Manager

Wentong Li Saraju P. Mohanty Krishna Kavi
Email-ID: wl@cse.unt.edu Email-ID: smohanty@cse.unt.edu Email-ID: kavi@cse.unt.edu
Dept. of Computer Science and Engineering, University of North Texas, Denton, TX 76203.

Abstract

Complex mechanisms for dynamic memory management and garbage collection are needed in modern imperative programming languages. Implementation of memory management functions efficiently both in terms of memory usage and execution performance becomes important for programs written in such languages. In this paper, we introduce a memory allocator that uses hardware assistance to improve the performance of a existing software allocator (PHK allocator). On average, our design reduces the execution time of memory management functions by 58.9%.

1 Introduction and Our Contribution

Significant amounts of execution time in modern imperative languages like C++/JAVA is spent on dynamic memory management. Memory management functions are performed by pure software in current systems. In some applications, the amount of execution time spent on memory management is as much as 42% [1]. Thus, there is a need for implementation of a low cost allocator, which has both good execution performance and memory locality in order to build efficient systems for memory intensive applications.

Software allocators search through lists of free memory chunks during allocation, and the search is in the critical path of allocator performance. Hardware allocators can perform parallel search through the lists of available memory. Moreover a hardware allocator can easily hide the execution latency of freeing objects, since freeing can run concurrently with application execution. A hardware allocator can coalesce free chunks of memory, in the background, while the application is not using that portion of the memory. Hardware units can also perform garbage collection in the background. The major disadvantages of a hardware-only allocator are the hardware complexity in implementing complex allocators and the lack of flexibility in changing allocation strategies.

In this paper, we show a new software/hardware co-design. Our design is based on the PHK [2] allocation method used in the Free-BSD system and Chang's hardware allocator [3]. We aim to balance the hardware complexity with performance by using both hardware and software. To prove our claims, we present a comparison of our design in terms of hardware complexity with a hardware-only allocator and a comparison in terms of performance with a software-only allocator. We have prototyped the hardware components using FPGA. Our proposed hardware-software allocator can find important uses in the applications written in programming languages like C++/JAVA where a significant amount of time is spent in memory management.

The rest of paper is organized as follows. We summarize the background and related research in Section 2; present the proposed software-hardware co-design dynamic allocator and its FPGA prototype in Section 3; compare our design with existing hardware only and software only allocators in Section 4; and present our conclusions in Section 5.

2 Background and Related Research

Our research deals with memory allocators, and thus we will briefly introduce both available software-only allocators and hardware-only allocators in this section.

There are two most popular open source software allocators, Doug Lea [4] used in LINUX system and PHK used in Free-BSD system. Berger et. al., [1] have shown that general purpose allocators such as the one by Doug Lea [4], or PHK [2] work well across a wide range applications. Feng et. al, [5] show that the performance difference between these two general purpose allocators for four most memory allocation intensive benchmarks in SPEC 2000 suite [12] is less than 3%. Thus if one were to implement memory allocators in hardware, one should consider one of these two general-purpose allocators.

There are a few hardware allocator designs [3] [6] [7] reported. All of these are based on the buddy system invented by Knuth [8]. None of them have found practical or commercial acceptance due to the excessive hardware complexity. All these allocators target embedded applications, where only physical addresses are used. This can be very limiting in most general-purpose systems that use virtual addresses.

To take advantage of the speed of a hardware-only allocator and the higher object localities of a general-purpose software-only allocator, we design a hybrid allocator. Our hardware portion of the design is based on Chang's hardware and our software portion of the design is based on the PHK allocator, thus making our design suitable for general-purpose computing environments.

3 The Proposed Software-Hardware Hybrid Allocator

The PHK allocator is a page based allocator, in which each page can only contain objects of one size. Allocation requests for different sized objects will be satisfied by using multiple pages. For large objects, sufficient number of pages are allocated to accommodate the object. For applications written using object-oriented languages such as Java and C++, most allocated objects are small. For small objects (less than half a page), object size is padded to the nearest power of 2, to match the size of objects in that page. The allocator keeps a page directory for all the allocated pages. At the beginning of each page, a bitmap is maintained to track allocation within that page. When allocating a small object, the PHK allocator perform a linear search on the bitmap for the first available chunk in that page. This search is performed in the following sequence: first locate the first word in the bitmap that has a free chunk, then locate the first byte in that word that represents a free chunk.

We compare our design with Chang's hardware allocator design [3] as we do not have access to detailed hardware designs for the other allocators cited above. Chang uses a first-fit allocation policy based on a binary OR-tree and a binary AND-tree. Each leaf node of the OR-tree represents the size of the smallest unit of memory that can be allocated, and other nodes provide information if such a unit is available. All allocated objects are multiples of the base size. The leaves of the OR-tree together represent the entire memory that is managed. The input of the AND-tree is generated by a complex interconnection network of the OR-tree. The AND-tree has the same number of leaves as the OR-tree. The AND-tree is used to generate the address of the first available chunk for a particular sized object. The interconnection between the OR-tree and the AND-tree is the most complex part of Chang's design. The interconnection has the same critical path delay as the OR-tree and the AND-tree. The final allocation result is produced by the output of the AND-tree through a set of multiplexers. The critical path (CP) delay, in term of gate delays, of this algorithm can be expressed as follows: $D_{CP} = D_{OR-tree} + D_{Interconnection} + D_{AND-tree}$. The hardware complexity, in terms of the number of gates, is $O(n \lg n)$, where n is the number of memory chunks, which depends on the size of the memory managed, and $O(\ln n)$ is the critical path delay.

We note that pure hardware allocators based on buddy system are not scalable since the complexity of the hardware increases with the size of the memory managed. Also buddy system is known for its poor object locality [9]. The PHK algorithm provides better object localities than the buddy system, but software allocators have the problem of poor execution performance. Hence our design that uses hardware assistance to improve the performance of PHK allocator.

The software in our allocator is responsible for creating page indexes and for initializing the page header as in a software implementation of PHK. For large objects (larger than half a page), the software takes full responsibility without any hardware assistance. When an application requests allocation for a small sized object, the software portion of our hybrid system will locate the bitmap and issues a search request to the hardware. The hardware portion in our allocator will search the page index (or bitmap) in parallel to find a free chunk, and mark the bitmap to indicate an allocation. This co-design allocator can be fully compatible with current multitasking operating system.

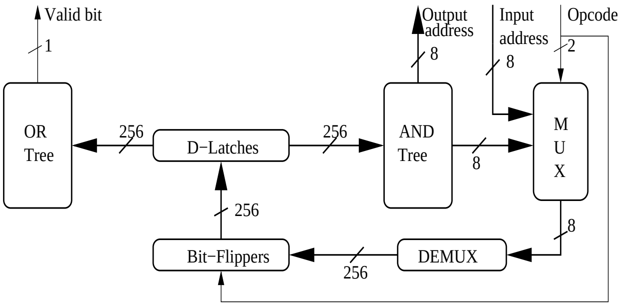

Figure 1: Block Diagram of Our Proposed Hardware Component (For Page Size 4096 bytes and Object Size 16 bytes)

Figure 1 shows the block diagram of the hardware we use for parallel searching. We have an OR-tree and an AND-tree similar to Chang's design [3]. The OR-tree is responsible for determining if there is a free chunk in a page. The AND-tree will locate the position of the first free chunk in the page. Because an OR-tree and an AND-tree are dedicated to one object size, the complex interconnections between the OR-tree and the AND-tree are not needed (unlike Chang's [3]). The individual implementation of the OR-tree and the AND-tree are identical to those of Chang's designs. The multiplexer (MUX) uses the opcode to select the address of the bit needed to be flipped. If the opcode is "alloc", the address from the AND-tree will be chosen. If the opcode is "free", the address from the request will be selected. D-latches in our design are used as storage devices, where the bitmap will be loaded from the page being searched for allocation. The de-multiplexer (DEMUX) is used to decode the address from the MUX. Bit-flippers use the decoded address and the opcode to determine how to flip a desired bit. Because of the page limits, we will not show the details of the flipper logic here. It may be noted that the critical path in this design is only the AND-tree for the "allocate" operation. The "free" doesn't generate any output, and the processor can immediately continue execution of application code.

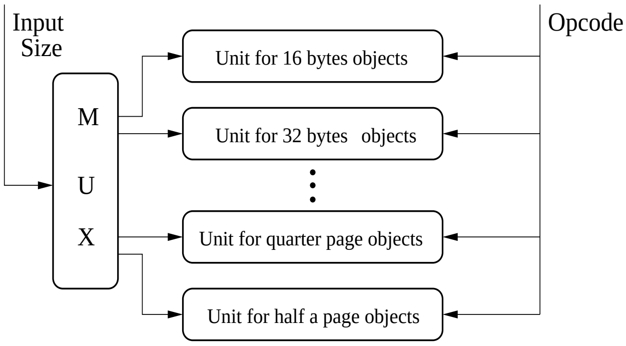
Input Size
Opcode
M U X
Unit for 16 bytes objects
Unit for 32 bytes objects
Unit for quarter page objects
Unit for half a page objects

Figure 2: The Block Diagram of Overall Hardware

Figure 2 shows the overall design of our system with 4096-byte pages. We have shown one unit for one page in Figure 1. For different object sizes, the hardware needed to support the bit-map will be different. In our design, we preselect object sizes from 16-bytes to 2048 bytes and include hardware to support pages for these objects. It should be noted that the larger the object size the smaller the amount of hardware needed to support the bit-maps indicating the availability of chunks in that page. For example, we need only 2 bits for a page that allocates 2048-byte objects (in a 4096 byte page). The MUX here is used to select the hardware unit that will be responsible for supporting objects of a given size. With 4096-byte pages, we have 8 different sized objects ranging from 16-bytes to 2048-bytes (and 8 hardware units in our design). For allocating 16-byte objects, we need trees with 256 leaves. Each tree only needs 255 AND/OR gates. For the overall system including all object sizes, we need 502 AND gates and 502 OR gates. This is very small amount of hardware compared with billions of transistors available on modern processor chips.

4 Experimental Results

Our hybrid allocator presented in the previous section has much lower hardware complexity than hardware-only allocators. At the same time our hybrid allocator improves execution performance of software-only allocators. In this section, we compare the hardware complexity of our design with Chang's allocator, and the performance with software-only PHK allocator.

4.1 Complexity Comparison

Existing hardware allocator designs implement the buddy system of allocations. The amount of hardware that is used to implement a buddy system based allocator is proportional to the size of the total memory [3] [7]. Thus, the buddy system based allocators are not scalable. Our design has much lower hardware complexity than Chang's allocator. In order to compare hardware complexity, the following notations are used: M is the total memory size, P is the page size, and S is the smallest allocated object size. Table 1 shows details of the comparison with Chang's algorithm.

Table 1: Comparison of Chang's Allocator and Our Design

Attributes	**Chang's Allocator**	**Our Allocator**
Algorithm Design	Total Memory	Page Based
Interconnection Complexity	$O\left(\frac{M}{S}\lg\frac{M}{S}\right)$	Interconnection Not needed
Overall Hardware complexity	$O\left(\frac{M}{S}\lg\frac{M}{S}\right)$	$O\left(\frac{P}{S}\right)$
Scalability	No	Yes
Need for Software Assistance	No	yes
Critical Path Delay	$O\left(\lg\frac{M}{S}\right)$	$O\left(\lg\frac{P}{S}\right)$
Clock Frequency	Slow	Fast
Allocation Locality	Poor	Better
POSIX Compatible	No	Yes
Compatible with Virtual Address	No	Yes

The complex interconnection determines the hardware complexity of Chang's allocator and it grows as $O\left(\frac{M}{S}\lg\frac{M}{S}\right)$. The hardware complexity of our design is $O\left(\frac{P}{S}\right)$. Normally, the page size is small and in most cases pages are of fixed size. For example, in a 2GByte memory system where the smallest object allocated is 16-bytes, Chang's allocator needs several hundred million gates, while our design only needs twenty thousand gates when 4096-byte pages are used.

The critical path delay of our design is much less than that of Chang's design. For Chang's allocator, the critical path delay is $O\left(\lg\frac{M}{S}\right)$ which grows with the size of the memory managed. For our design, the critical path delay is $O\left(\lg\frac{P}{S}\right)$. For a system as previously described, the height of the trees in Chang's algorithm 27. The total critical path delay will be 108 logic gate delays. For our approach, the critical path incurs only 16 gate delays. Moreover, our proposed allocator can be run at much higher clock frequency than Chang's allocator, although it needs software assistance.

When freeing an object, Chang's algorithm needs the size of the object to correctly manipulate the AND and OR trees. In POSIX standard, "free" command does not provide object sizes; only the starting address of the object to be freed. This incompatibility makes Chang's approach impractical. Since the software part in our design will locate the bitmap on free, our design is fully POSIX compatible. Another incompatibility results from the use of physical memory addresses by Chang's design, while most general-purpose systems use virtual memory addresses. Since we rely on the software portion of a PHK allocator to interface with applications, our design is compatible with virtual memory based systems.

Compared Chang's design which is based on Buddy systems that is known for very poor memory usage and poor object localities, our design which is based on PHK, provides better object localities and better memory usage. It should be noted that there is a buddy allocator called Address-Ordered buddy system [10] that results in better object localities than conventional Buddy systems used by Chang.

4.2 Allocator Performance Analysis

For the purpose of analyzing the performance gains from our design, we simulated the existence of a hardware-assisted PHK allocator within a conventional CPU using SimpleScalar simulation tool set [11]. Because this hardware is very simple, we assume the hardware portion of our allocator presented in Section 3 runs at the same clock frequency as the CPU. For the purpose of analysis this hardware is implemented as a special functional unit in a superscalar processor. This unit is activated by operations, find_chunk and free_chunk. The page size of the system is assumed to be 4096 bytes, and the smallest object allocated is set to 16 bytes. The detailed processor parameters used in our simulations are shown in Table 2.

We have used ten benchmarks (with varying number of memory management overheads) to study the performance gains using our design: parser and perlbmk are from SPEC CPU2000 suite; cfrac, espresso and boxed-sim are memory intensive benchmarks that are widely used by researchers; the other benchmarks are from Olden suite, which are also memory allocation intensive. The inputs to these benchmarks, average object sizes, and percentage of execution time spent in memory management by software allocators are shown in Table 3. The simulation results are shown in Table 4.

The speedup of each application is proportional to the execution time spent on memory management and the average object size. In Figure 3, we show the reduced memory management execution cycles normalized to the original execution cycles spent on memory management functions by software only allocator. This figure shows the relative performance improvements for memory management functions. The cfrac application shows the most performance improvement. The average object size in cfrac is 8 bytes, which means that most pages allocated contain 256 objects, since the smallest size of allocation is 16 bytes. The linear search of the bitmaps using software for that many objects will be slow. The hardware in our design speeds up the search, leading to 76.2% normalized performance improvement over the software-only allocation. The benchmark espresso shows the least amount of improvement using hardware-assistance. The average object size in espresso is 250 bytes. Thus, pages allocated for this benchmark contain fewer than 20 objects. Linear search of 20 objects is not as significant, and the use of hardware in our design only shows 48.0% normalized performance improvement. The other benchmarks have average object sizes of 16 bytes to 48 bytes, and thus the performance gains are

Table 2: Simulation Parameters

Pipeline Parameters	
Issue Width	4
Functional Units	Int: 4 ALU, 1 Mult/Div, Float: 4 ALU, 1 Mult/Div, 2 Memory Ports
Register Update Units	8
Load/Store Queue Size	4
Branch Predictor	Bimodal
Memory Parameters	
L1 Data Cache	4–way Set Associative, 16K Bytes
L1 Instruction Cache	Direct-mapped, 16K Bytes
L2 Unified Cache	4–way Set Associative, 256K Bytes
Cache Line Size	32 Bytes
L1 Hit Time	1 cycles
L1 Miss Penalty	6 cycles
Mem Latency/Delay	18/2 cycles

Table 3: Percentage time spent in Selected Benchmarks for different (Average) Object Sizes

Benchmarks	**Input**	**Object Size**	$(\%)$ **Time Spent on Alloction**
cfrac	22-digits number	8 bytes	29.7
espresso	largest.espresso	250 bytes	4.7
boxed-sim	-n 10 -s 1	24 bytes	2.4
parser	ref.in (first 300 lines)	16 bytes	35.6
perlbmk	perfect.pl b 2	38 bytes	10.7
treeadd	20 1	24 bytes	48.2
voronoi	20000 1	40 bytes	10.4
bisort	250000 1	24 bytes	2.3
perimeter	12 1	48 bytes	16.3
health	5 500 1	24 bytes	4.9

Table 4: Performance Comparison with PHK Allocator

Benchmarks	**PHK Execution Cycles** (in million)	**Our Design Execution Cycles** (in million)	**Speedup**
cfrac	189.7	148.1	1.28
espresso	5,241	5,129	1.02
boxed-sim	9,043	8,922	1.01
parser	27,111	21,363	1.27
perlbmk	135.5	127.3	1.06
treeadd	160.4	112.4	1.43
voronoi	128.8	122.3	1.05
bisort	424.1	418.1	1.01
perimeter	42.11	37.97	1.11
health	383.0	372.2	1.03

not as significant as that for cfrac, but better than that for espresso. On average, our co-design reduces the memory management time by 58.9%. The average overall execution speedup of our design when compared to a software only allocator implementation is 1.127 (or 12.7%) for these ten memory management intensive benchmarks.

In summary, our design presented in the previous section has much lower hardware complexity than the existing hardware designs; and our design improves execution performance of software allocators.

4.3 FPGA Prototyping

In order to demonstrate the feasibility of the proposed hybrid allocator we performed its FPGA prototyping. The hardware component of our co-designed allocator is prototyped using VHDL on the Xilinx ISE 7.1i and the functional and timing simulations are carried out using Modelsim XE III tools. The target device in the simulation is the VIRTEX II xc2vp2 FPGA processor. Figure 4 shows the RTL schematic generated by the ISE tool. Figure 5 shows the functional simulation waveforms generated by the Modelsim. In addition, the summary of the synthesis results are show in Table 5.

5 Conclusions and Future Works

In this paper, we described a hybrid software-hardware memory allocator based on the PHK and Chang's designs. We have included a FPGA prototype of the hardware portion of our design. Compared to hardware-only allocators that are typically based on buddy system, our design has significantly lower hardware complexity and lower critical path delays. Our hardware design has a fixed hardware complexity, complexity being dependent on the size of a memory page, and not the total (user) memory being managed. In addition, our design is fully compatible with the modern operating systems. Since our design is based on PHK algorithm, we are likely to achieve better object localities than buddy system based designs.

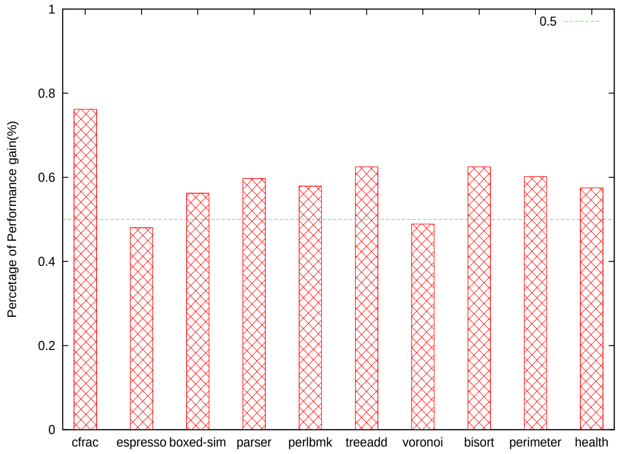

Figure 3: Normalized Memory Management Performance Improvement

Table 5: Synthesis Summary

Minimum period	20.13ns
Maximum frequency	49.677 MHZ
Cell usage (BELS)	2214
No. of IOs	21
LUTs usage	6%

We also have shown that our hardware-software allocator achieves 12.7% improvement in the overall execution performance over software-only allocator implementation for memory intensive benchmarks and improves the memory management efficiency by 58.9% (that is the execution performance improvement for memory management functions). The performance gains depend on how often an application invokes “malloc” or “free” functions, and the average size of objects allocated. Our design shows higher performance gains for applications that use small objects.

In the future, we will explore variable sized pages such that the number of allocated objects are the same in each page. In this case, all the bitmaps will have the same number of bits. Thus, we need only one pair of AND-tree and OR-tree in our design. This will further reduce the hardware complexity. We expect that this will also improve the memory management efficiency of allocators for large objects. We also plan to investigate hybrid designs for other

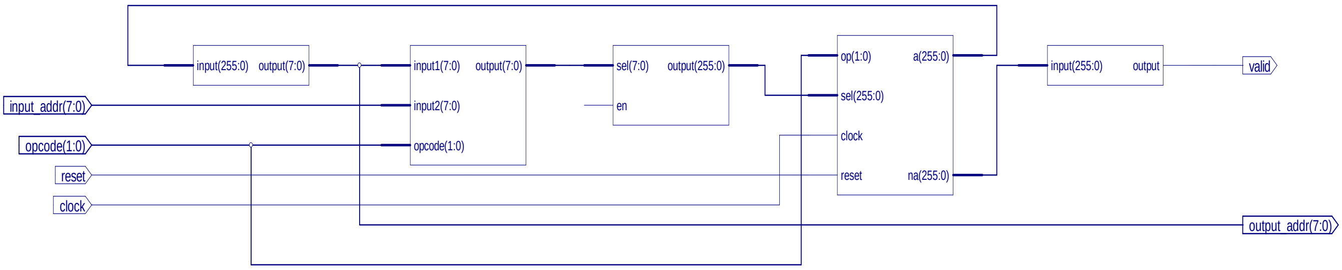

Figure 4: RTL Synthesized by ISE

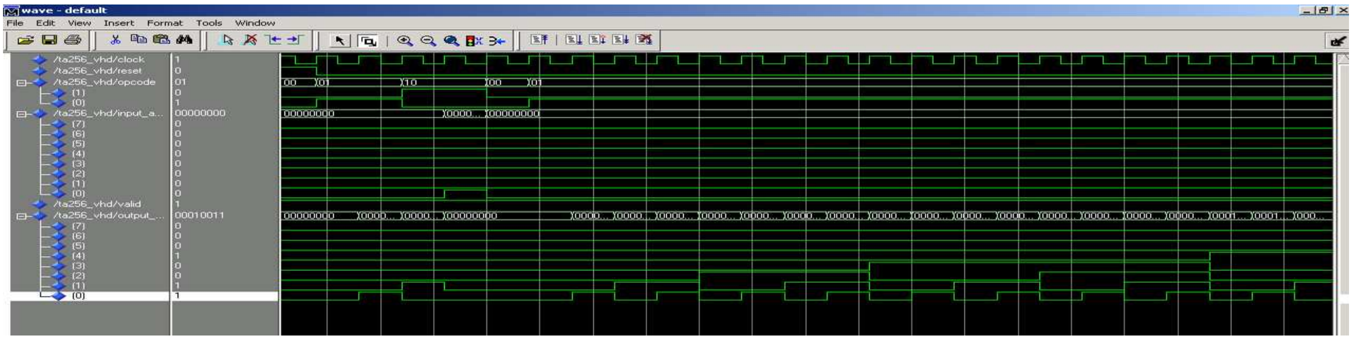

Figure 5: Modelsim Simulated Waveform

memory management algorithm's like that of Doug Lea. For example, hardware may be used to assist in tracking quick-lists used in Doug Lea's method.

References

[1] E. D. Berger, B. G. Zorn and K. S. McKinley, "Reconsidering custom memory allocation", in *Proc. of the Conference on Object-Oriented Programming Systems, Languages and Applications*, 2002, pp. 1-12.

[2] P. H. Kamp. "Malloc(3) revisited", `http://phk.freebsd.dk/pubs/malloc.pdf`.

[3] J. M. Chang and E. F. Gehringer, "A High-Performance Memory Allocator for Object-oriented Systems", *IEEE Transactions on Computers*, Vol. 45, No. 3, 1996, pp. 357-366.

[4] D. Lea, "A Memory Allocator", `http://gee.cs.oswego.edu/dl/html/malloc.html`

[5] Y. Feng and E. D. Berger, "A locality-improving dynamic memory allocator", in *Proceedings of the 2005 workshop on Memory System Performance (MSP 2005)*, Chicago, USA, 2005, pp. 68-77.

[6] H. Cam, et. al., "A High Performance Hardware Efficient Memory Allocator Technique and Design", in *Proceedings of the International Conference on Computer Design*, Austin, USA, 1999, pp. 274-276.

[7] S. Donahue, M. Hanpton, R. Cytron, M. Franklin and K. Kavi, "Hardware support for fast and bounded-time storage allocation", in *Proceedings of the Second Workshop on Memory Performance Issues (WMPI 2002)*, Anchorage, USA, 2002.

[8] D. E. Knuth, *The Art of Computer Programming Vol. I: Fundamental Algorithms*, Addison-Wesley, 1968.

[9] M. S. Johnstone and P. R. Wilson, "The memory fragmentation problem: Solved", in *Proceedings of the First International Symposium on Memory Management (ISMM'98)*, volume 34(3) of *ACM SIGPLAN Notices*, Vancouver, Canada, 1998, 26-36.

[10] D. C. Defoe, S. R. Cholleti, and R. K. Cytron, "Upper bound for defragmenting buddy heaps", in *Proc. of the Conference on Languages, Compilers, and Tools for Embedded Systems*, 2005, 222-229.

[11] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, version 2.0", *Technical Report CS-1342*, University of Wisconsin-Madison, June, 1997.

[12] "SPEC CPU2000 V1.3", `http://www.spec.org/osg/cpu2000`.