Reconfigurable Fast Memory Management System Design for Application Specific Processors

S. Kagan Agun and Morris Chang
Computer Science Department, Illinois Institute of Technology, agunsal@iit.edu

Abstract

This paper presents the design and implementation of the new Active Memory Manager Unit (AMMU) designed to be embedded into System-on-Chip CPUs. The unit is implemented using VHDL in Field Programmable Gate Array (FPGA) technology. The modified buddy system is used as the hardware algorithm for memory management. The RISC compatible open-source CPU is deployed with the memory management unit to demonstrate the feasibility of implementation. The results indicate that the proposed AMMU achieves high performance in memory allocation and deallocation for software systems.

1. Introduction

Intellectual Property (IP) core-based hardware-software systems have emerged as a result of the new System-on-Chip (SoC) design paradigm. SoC is generally defined as an integration of complex functional modules or cores, where each core is complex enough to be a complete Integrated Circuit (IC) in itself. In order for these IP cores to be as reusable as possible, the cores must be soft cores in a synthesizable Hardware Description Language (HDL) form so that they can be targeted to different semiconductor process technology.

The traditional approach to system design involves combining a microprocessor and other devices on a single circuit board. Today advanced submicron technologies enable a complete design on a single chip. Increasing the density and speed of Field Programmable Gate Arrays (FPGAs) leads to adoption of IP core-based System-on-Chip (SoC) designs. Since processors are common in system design, the result of these advances is the ability to embed them with other functions into a single device.

Reduced Instruction Set Computer (RISC) instruction sets have gradually moved to Complex Instruction-Set Computer (CISC) instruction sets during the 1980s. After two decades and the invention of SoC processors, RISC machines are gaining more attention due to the fact that only 25% of the instructions of a complex instruction set are frequently used (about 95% of the time) [6]. A RISC instruction set typically contains less than 100 instructions with fixed-length format such as 32 bits. Only three to five addressing modes are used. Most instructions are register-based. Memory access is done using load/store instructions. Because of the reduction in instruction-set complexity, RISCs are implementable on a single chip. For example, Sun Microsystems has released PicoJava, a stack-based 32-bit microprocessor softcore which executes Java bytecode instructions [2]. Xilinx Inc. and IBM announced IBM's PowerPC hard core on Xilinx's Virtex-II FPGA [1]. Altera *Excalibur* solution provides embedded processor programmable logic to the design community. The Excalibur softcore solution is a configurable RISC processor called *Nios*. Nios offers high flexibility and scalability and low cost. The Excalibur hardcore solution is an ARM-based processor from Advanced RISC Machines (ARM) Limited and Million Instructions Per Second (MIPS) - based from MIPS Technologies, Inc. [3].

The SoC approach encourages design engineers to adopt existing IP cores. This promotes reuse. A modern IP core library typically includes features for specific applications such as communication ports, image processing units, and Floating Point Units (FPUs). Nowadays, object-oriented software applications are getting dynamic memory intensive[9]. This creates a need for a high-performance memory allocator and deallocator as a core extension. For example, the *Active Memory Management Unit* (AMMU) provides high performance in memory management [7]. AMMU uses hardware accelerated dynamic memory management algorithm based on the modified buddy system [4]. To study the feasibility of such a processor extension, we developed a detailed design of the Active Memory Management Unit.

In reference [7], the memory allocation scheme used in seven applications from the SPECjvm98 benchmark suite was analyzed with the results given in Table 1. This reference is also reported that over 98% of the objects were smaller than 4,096 bytes and over 90% of the allocation calls were completed within 1000 cycles. For example, 92

Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'03)
0-7695-1904-0/03 $17.00 © 2003 IEEE

percent of the memory allocations in the *Compress* application takes an average of 410 cycles to allocate less than 4 Kilobytes of memory. While memory allocation is taking 410 ns on 1 GHz system for *Compress*, it takes an average of 139 ns in RISC1000 + AMMU(256) implemented on a Xilinx FPGA (VIRTEXE V300EFG456).

Table 1: Malloc latencies of objects with size less than 4,096 bytes [7]

Application	Total Allocated Objects	Ave. Minimum latency (cycles)	% of allocation less than 1000 cycles
Compress	11,624	410	92.25
DB	3,215,782	407	98.99
Jack	6,871,646	387	94.46
Javac	5,943,930	386	94.91
Jess	7,939,856	388	90.20
MpegAudio	15,182	413	93.83

This paper presents our design experience in combining the proposed memory management unit into a 32-bit RISC compatible microprocessor, OpenRISC 1000 [5]. OpenRISC 1000 is a RISC architecture written in synthesizable Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL). OpenRISC 1000, with source code available in the public domain, can be targeted to a wide range of platforms. The Xilinx Foundation 2.1i tool is used to design each component.

The remainder of this paper is organized as follows. Section 2 introduces the modified buddy system for memory management. Section 3 describes the Active Memory Management Unit implementation and Section 4 describes the OpenRISC 1000 processor. Section 5 evaluates the result of the implementation of the design. The last section presents the conclusions of this paper.

2. Memory allocation based on the buddy system

When a memory block of a given size is to be allocated, the buddy system locates a block whose size is a power of two that is large enough to fullfill the request. The block is split in half as many times as possible, until it reaches the block size needed to hold the requested size. When a block is split, its two halves are known as buddies. At the time a block is freed if its buddy is also free, the buddy system naturally coalesces the blocks, making available a block of larger size.

Many real-time programmers prefer binary buddy allocation because it is easier to implement and more predictable. A separate free list is used for each possible object size (power of two). Allocation of a new object starts from the smallest free segment list and continues until it finds segments large enough to satisfy the allocation request. If a list is empty, the algorithm allocates a larger segment from one of the upper layers and divides it into buddies and updates the lists. When an object is freed, the algorithm attempts to coalesce with its buddy and update the lists.

Binary buddy systems always allocate memory in sizes equal to a power of 2, so they may leave unused space. This is known as internal fragmentation. In the worst case, memory utilization is poor if it is necessary to allocate memory in different sizes. In practice, internal fragmentation imposes a higher memory overhead.

On the other hand, a bit-map approach can easily form the base of a binary tree and it can be efficiently maintained in hardware. Allocation and deallocation are done using a hardware-maintained binary tree through combinational logic. The hardware approach eliminates internal fragmentation by allocating/deallocating memory of the exact sizes required. This approach is known as the modified buddy system. Such bit-manipulation is quite easy in hardware realization.

3. Implementation of the AMMU

3.1. Locating free blocks

In this approach, the or-gate tree determines whether there are enough free cells of the size requested to allocate. Anding output of or gates at the same level gives the availability of the size of blocks requested. If one of the nodes that shows the availability of requested sizes at the same level has a zero value, which shows the availability of the requested size, the "and" results of this level will be zero. For example, at level 2, each node, b and c, represents 2^2 cells and anding b and c produces the availability of free cells of this size. Figure 1 demonstrates free cell availability.

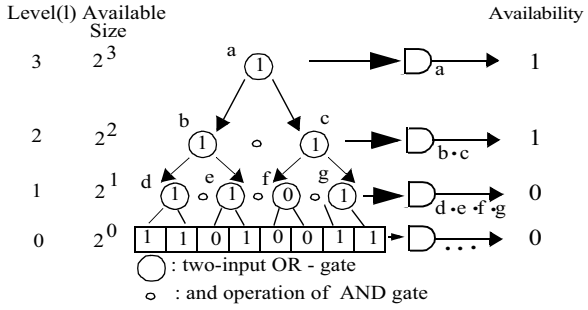


Figure 1. Building the *or-gate* tree

On the other hand, a large free memory block can be shared by the buddies without making a large enough block available to the system. This is known as a blind spot. For example, the bit array “11011001” cannot report the availability of any blocks of size 2 in the *or-gate* tree. This

blind spot limitation can be eliminated by shifting the bit array left or right at least 1 and no more than *n-1* times where *n* is the size of the bit array.

3.2. Finding the first zero address

A second binary tree is constructed from *and* gates to determine the first zero address. This *and-gate* tree propagates the information upward in the tree. Searching the zero nodes from root to leaves gives the first zero address e.g. nodes 1, 2, 5, and 11 in Figure 2a. Branching to the right is '0' and to the left is '1' in the address determination. The root node of the tree represents the most significant bit and the lowest leaves the least significant bits in the address. In Figure 2a, the address is "010" = '0' (from node 1 to 2), '1' (from node 2 to 5), and '0' (from node 5 to 11).

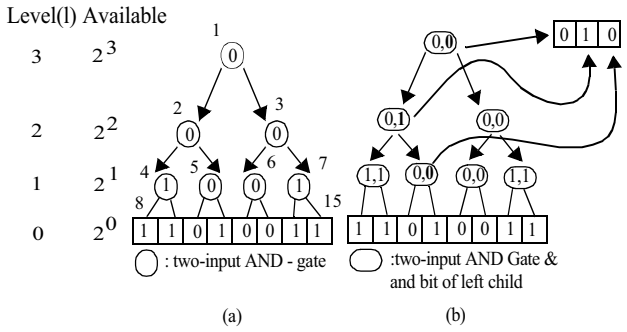


Figure 2. Building the *and-gate* tree

To find the first address of the free cell(s) in the desired level, the or-gate results of this level are transferred to the and-gate tree which passes the availability of cell(s) at the required level upward toward the root. Suppose the and-gate tree determines the address of the first zero at level l of the or- gate tree. However, in the search for free cells, the algorithm goes back to the parent node to check the next leaf node, therefore the and-gate tree needs a backtracking algorithm to calculate the exact address of the free cell(s). By propagating the left child of each and-node to the parent node, this need is avoided and the target address is formed via these propagated bits (Figure 2b). But this address is a relative address which points to the beginning of the 2^{level} size cells. Therefore this address should be multiplied by 2^{level} or simply shifted to the right by level bits.

3.3. Designing locator: *or-and-gate* tree

In the algorithm proposed above, the nodes of the desired level of the *or-gate* tree are fed into the *and-gate* tree which determines the first zero space of this level. This approach requires bit-map transfer through combinational logic. The *or-gate* values of level(l) feed the combinational logic. The output of the combinational logic is the input of the *and-gate* tree. Combining the *or-gate* and the *and-gate* into the *or-and gate* tree eliminates this overhead and reduces the complexity of the *and-gate* tree.

Each node in the *or-and-gate* tree consists of an *or* gate, an *and* gate and a *multiplexor*. The *multiplexor* transfers the *or-gate* or the *and-gate* value to the *and-gate* tree above. This transfer happens only for the requested size. The requested size determines the allocation level. The propagation bit, the *P_bit*, provides the back-tracking information to the multiplexor that finds the first available free space address. Anding of the *or-gate* values of each level shows the availability of any size in this level. Figure 3(a) demonstrates the *or-and-gate* node and (b) VHDL code. Figure 4 shows the complete *or-and-gate* tree for block size 8.

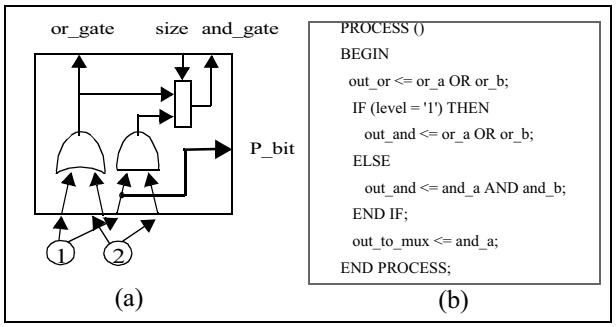


Figure 3. *or-and-gate node*: locator node.

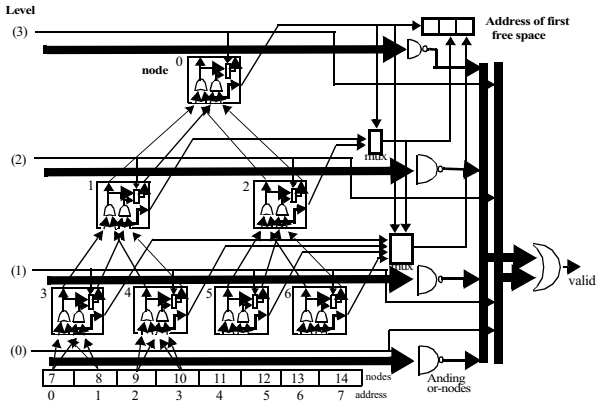


Figure 4. Complete *or-and-gate* tree for block size 8

3.4. Marking Free Space

When the blocks are located by the *or-gate* tree and the *and-gate* tree, the number of bits requested for allocation need to be marked as occupied. Thus we need an algorithm that it is feasible to implement in hardware to mark the corresponding bits. The free cell locator provides the starting address of these cells. We need to mark logic blocks of the requested size from the starting address. The basic implementation of marking the bit array is described in Figure 5. The size value is transferred into the

consequent bit array, then shifted by the starting address. This shifted bit array is "or"ed with the actual bit array to mark the requested bits.

The marking algorithm above is very simple and easy to implement in terms of a behavioral design. On the other hand, it is not efficient due to nondeterministic design size. This approach takes more space in the Configurable Computing Machine (Table 4). Therefore, a much smaller version of the marking algorithm is needed. Considering the *or-gate* tree and the *and-gate* tree, marking can be achieved through signal propagation. Marking signals will be directed by size and starting address inputs, so only corresponding bits will receive the marking signal. Unlike the *and-gate* tree, the bit-marker propagates signals from the root of the tree to the leaves.

Initially we need a route signal, which is set to 1 by default. This signal travels from the top of the tree to the bottom and routes starting address bits at each level. A node that receives a route signal, considers whether to mark one of its subnodes. The starting address bits at this level determine which nodes will receive the mark signal. Other nodes will receive a route signal in the case of potential partial allocation in this subtree. In this example node 4 receives mark signal and propagates it to left and right sub nodes and so on. Once a mark signal is activated in one direction, a route signal is routed in the other direction. Figure 5 shows the marking operations. The routing signals of the marking nodes in the last row are not shown in order to simplify the diagram.

The advantage of using the buddy system is fast cell allocation. This approach is using significantly more cells than other strategies at the space locating stage. However the final marking can mark the exact size. The modified buddy system does not always yield better cell utilization. Due to external fragmentation and blind spots, there would be cell wastes among the allocated cells.

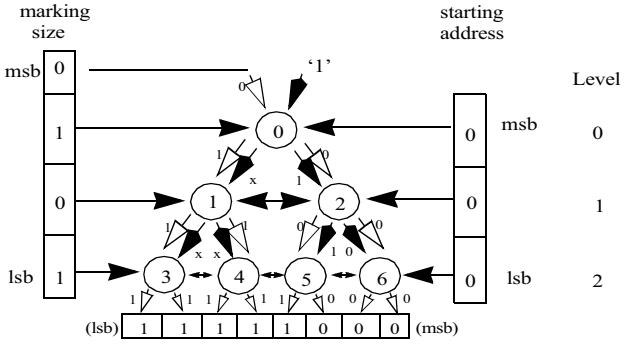


Figure 5. Marking Tree

The implementation of the marking tree is based on a one-bit marking node. Managing the signal distribution among the nodes is the most serious design challenge. This is achieved through careful node and signal labelling. In each node the sending signal is labelled with the target node label. For example, if node 1 is sending signals to nodes 3 and 4, the respective outgoing signals are labelled 3 and 4. The Out signals, mark and route, consist of two bits, the most significant bit is for the right node, the least significant bit is for the left node. In signals are labelled with the receiving node label, so node 1 receives mark signal [1] and route signal [1]. Labelling nodes starting from 0 and from left to right gives an advantage in determining starting and ending node numbers in each row.

3.5. Round-up System

This system rounds up a given n-bit binary data item to the closest higher value of the form 2^N. The highest '1' of the input sets the feedback to '1'. The shift signal indicates whether there are ones in lower significant bits of the input after the highest significant one. The shift signal will set the next highest bit to one to round up the input to 2^N. The decision slice is determined by the first bit equal to '1' in the imput with feedback = '0'. Once the condition is met, this slice generates the feedback and the rest of the slices pass this feedback signal to the next ones. Figure 6. shows the n-bit *round-up* system.

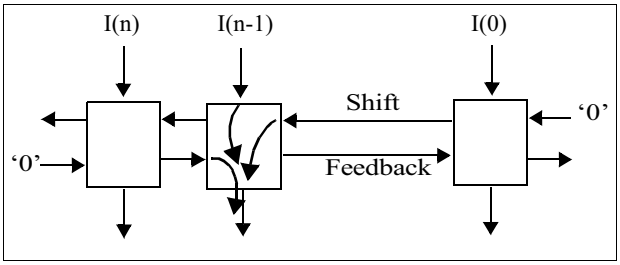


Figure 6. n-bit *round-up* system

3.6. Final Design: AMMU

The final step is forming the Active Memory Management Unit (AMMU). Every component of AMMU is the result of a parameterized design that can be instantiated to any precision. These components can be reused regardless of the precision. The VHDL **generate** and **generic** statements can instantiate small components (e.g. 1-bit round-up) to a *n*-bit round-up system in an *iterative* fashion. Such an approach results in a reusable design.

A configuration file is used to keep precision parameters in order to instantiate the AMMU. A clock is also added to the final design to store the allocation bit map in a local register. Figure 7 shows the AMMU final design. The designs of 256 and 512 block allocators are synthesized with speed optimization by the XILINX Foundation series 2.1i tool. Table 2 shows the synthesize results of AMMU for two different sizes. The 512 block allocator can work at the speed of 7.1 MHz but it takes 21% of the total chip area.

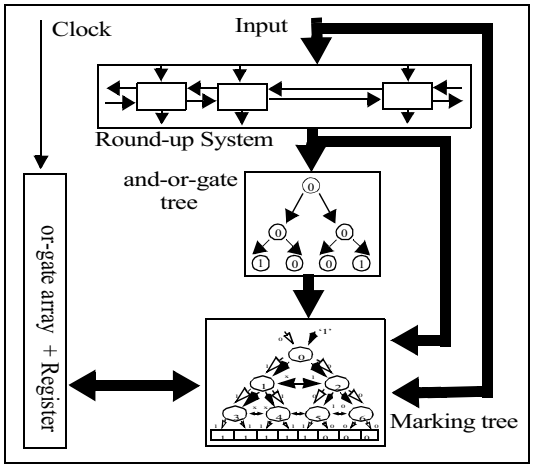


Figure 7. The design of the n-bit AMMU

Table 2: AMMU area and speed requirements*

	AMMU(256)	AMMU(512)
Number of Slices	987 (32%)	1,929 (62%)
Slice Flip Flops	256	512
LUT(4 input)	1,940	3,795
Gate Count	13,688	26,866
Max clock frequency (MHz)	14.308	11.286
Max net delay(ns)	7.944	11.092

*VIRTEXE V300EFG456 speed -8 (optimized for speed)

4. Description of the OpenRISC Processor

OpenRISC 1000 architecture is the architecture of a family of open source, synthesizable RISC architectures. It is a 32-bit load and store RISC architecture designed for speed and scalability. It also provides native bus interface to communicate with compatible devices. Figure 8 shows the OpenRISC architecture.

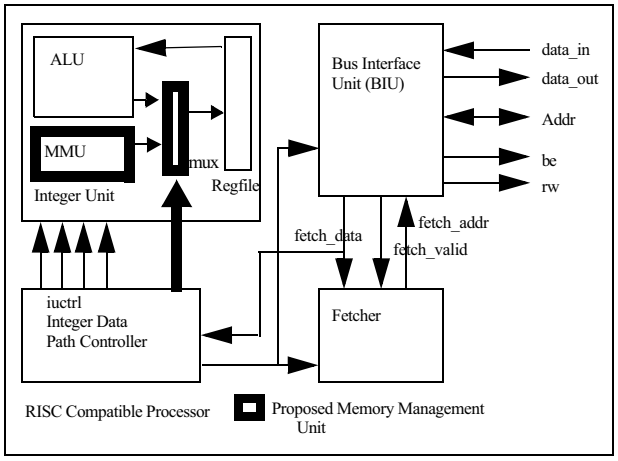


Figure 8. OpenRISC 1000 Architecture

The RISC design is synthesized for the Xilinx Virtex target device for speed optimization. Table 1 shows the area distribution of the design components. Due to its reduced instruction set architecture, the processor takes only 7% of the target device. This makes RISC based processors valuable for SoC systems.

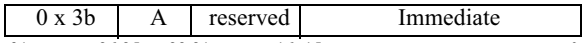
Table 3: AMMU components

Components	Number of Func. Generators	Number of Registers
Fetcher	56	85
ALU	430	61
Regfile	0	64 + (64 DRAM)
iu	586	64 + (64 DRAM)
iuctrl	130	125
biu	65	66
RISC1000	837	649 + (64 DRAM)

*Xilinx Virtex V800HQ240-6 target device

Two instructions were added to the processor instruction set to implement memory allocation and de-allocation. These instructions are the Arithmetic Logic Unit (ALU) like instructions. A mux is also added to Integer Unit (IU) to select either the ALU or AMMU result. An integer unit controller (iuctrl) generates the AMMU select signal. Figure 9 demonstrates the ALU memory instructions.

```
constant ALLOCI : std_logic_vector (5 downto 0) := "111010";
```



alloc rA, I : rA ← alloc I

```
constant ALLOCR : std_logic_vector (5 downto 0) := "101111";
```

allocr rA, rB : rA ← alloc rB

Figure 9. ALU memory allocation instructions

5. Results of the AMMU Design Implementation

The design of the locator and the behavioral and structural designs of the marking tree were intended to implement free memory space allocation algorithm on a Reconfigurable Computing Machine. The foundation series 2.1i tool was used in the design process for target device, XC4000E. Table 1 shows the result of three designs in terms of the number of Configurable Logic Blocks (CLBs), delays and Line of Code (LOC) of the VHDL source code. Structural hardware implementation of the

optimized buddy system demonstrates high performance for allocation of free memory space.

Table 4: Locator and Marker implementation results on XC4000E*

Component	LOC	Size (bits)	Number of Logic Cell	Gate Count	Max Net Delay(ns)	Max Delay(ns)
Locator (Structural)	160	16	28 (%07)	310	6.5	41.2
		32	58 (%14)	654	7.5	56.5
		64	118 (%29)	1336	14.1	83.1
		128	244 (%61)	2715	24.6	155.7
Marking (Structural)	64	16	22 (%05)	264	5.8	22.5
		32	46 (%11)	552	11.5	26.6
		64	94 (%23)	1128	14.3	29.3
		128	190 (%49)	2280	11.9	41.8
Marking (Behavioral)	30	16	88 (%22)	1301	13.6	34.6
		32	194 (%48)	3136	22.9	44.1
		64	417 (%104)	7238	--	--
		128	--	--	--	--

* Xilinx 4000E-1BG225 (400 CLBs, 10K max logic gates)

The tree-search algorithm of the active memory manager unit results in a slow clock rate. This will be the main factor in determining the speed of the RISC compatible processor. The 512-bit block addressable memory manager unit is 6 times slower than the processor (Table 5). Adding the AMMU to the processor core will slow down all the pipeline instructions. The space requirement of the AMMU is also high. The 512 block memory allocation unit takes 3.2 times more space than the RISC 1000 processor.

Table 5: RISC and AMMU area and speed requirements*

	RISC1000	RISC +AMMU(256)	RISC+AMMU(512)
Number of Slices	657 (21%)	1746 (56%)	2,790 (90%)
Slice Flip Flop	616	873	1129
LUT (4 Input)	904	3,063	5,080
Gate Count	19,699	34,709	48,859
Max clock frequency (MHz)	68.395MHz	13.245MHz	10.076MHz
Max net delay(ns)	7.212	8.0885	11.210

*VIRTEXE V300EFG456 speed -8 (optimized for speed)

The 32-bit processor have 2^{32} address space, a process needs only a small portion of this address space for its memory allocation and deallocation requests. Considering 4Kbyte memory blocks, the 1024 bit memory manager maps 4 Mbyte address space per process. As a result, memory intensive applications have the benefit of a memory management unit despite the slow clock rate on FPGAs. Implementation of memory management unit on ASIC will eliminate the speed factor. So, the memory allocation/deallocation instructions of the applications in Table 1 will gain an average performance of 399 with AMMU.

One solution is to increase the overall performance is embedding the smaller memory management unit by dividing the main tree into sub-trees. For example the 512-bit tree can be divided into four 128-bit sub-trees and one 4-bit top-tree. To complete the memory allocation and deallocation these instructions may require a maximum 4 clock cycles. This also requires less space on the chip.

Another approach is instead of having the AMMU inside the core, it can be placed outside of the processor so that it acts like an auxiliary memory. Allocation and deallocation can be performed by load/store like instructions without affecting processor performance.

6. Conclusion

Recent advances in software engineering, such as Graphical User Interfaces (GUIs) and object-oriented programming have caused today's applications to become dynamic memory intensive. These applications consume up to one-third of program execution time in dynamic storage management. This illustrates the need for a high-performance memory allocator and deallocator.

This paper presented a simple hardware design to allocate and free memory blocks, saving considerable time over the software approaches. Allocation and deallocation can be preformed in constant time. The node-based tree design approach provides an efficient and reusable design. The Active Memory Management unit can be integrated into System-on-Chip designs such as the processor and coprocessor. Operating Systems may benefit from this approach to the management of storage related operations.

7. References

[1] C. Matsumoto, R. Merritt, "FPGAs muscle in on ASICs' embedded turf", *EE Times*, July 31, 2000.

[2] S. Dey, C. N. Taylor, D. Panigrahi, K. Sekar, L. Chen, and P. Sanchez, "Using a soft core in a Soc design: Experiences with picoJava", *IEEE Design and Test of Computers*, July-September 2000, pp. 60-71.

[3] www.altera.com, "Altera Excalibur Backgrounder", white paper, July 2000.

[4] J. M. Chang, E. F. Gehringer, "A high-performance memory allocator for object-oriented systems", *IEEE Transaction on Computers*, Vol. 45, No. 3, March 1996, pp. 357-366.

[5] www.opencores.org, "OpenRISC 1000".

[6] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, and Programmability*, McGraw-Hill, 1993

[7] W. Srisa-an, C. D. Lo, and J. M. Chang "A Performance Analysis of the Active Memory Module (AMM)", *Proceedings of IEEE International Conference on Computer Design,* Austin, Texas, Sep. 23-26, 2001, pp.493-496.

[8] J. M. Chang and K. Agun, "Designing Reusable Components in VHDL", *Proceedings of 13th IEEE International ASIC/SOC Conference,* Washington, D.C., Sept. 13-16, 2000. pp.165-169

[9] J. M. Chang, W. H. Lee, "A study on memory allocations in C++", *Proceedings of 14th International Conference on Advanced Science and Technology*, Naperville, Illinois, April 4-5, 1998. pp. 53-62.