# Hardware Support for Real-Time Embedded Multiprocessor System-on-a-Chip Memory Management

Mohamed Shalan
Georgia Institute of Technology
School of Electrical and Computer Engineering
801 Atlantic Drive
Atlanta, GA 30332-0250
(404) 894-0966
shalan@ece.gatech.edu

Vincent J. Mooney III
Georgia Institute of Technology
School of Electrical and Computer Engineering
801 Atlantic Drive
Atlanta, GA 30332-0250
(404) 385-0437
mooney@ece.gatech.edu

## ABSTRACT

The aggressive evolution of the semiconductor industry — smaller process geometries, higher densities, and greater chip complexity — has provided design engineers the means to create complex, high-performance Systems-on-a-Chip (SoC) designs. Such SoC designs typically have more than one processor and huge memory, all on the same chip. Dealing with the global on-chip memory allocation/de-allocation in a dynamic yet deterministic way is an important issue for the upcoming billion transistor multiprocessor SoC designs. To achieve this, we propose a memory management hierarchy we call Two-Level Memory Management. To implement this memory management scheme — which presents a paradigm shift in the way designers look at on-chip dynamic memory allocation — we present a System-on-a-Chip Dynamic Memory Management Unit (SoCDMMU) for allocation of the global on-chip memory, which we refer to as Level Two memory management (Level One is the operating system management of memory allocated to a particular on-chip Processing Element). In this way, processing elements (heterogeneous or non-heterogeneous hardware or software) in an SoC can request and be granted portions of the global memory in a fast and deterministic time (for an example of a four processing element SoC, the dynamic memory allocation of the global on-chip memory takes sixteen cycles per allocation/deallocation in the worst case). In this paper, we show how to modify an existing Real-Time Operating System (RTOS) to support the new proposed SoCDMMU. Our example shows a multiprocessor SoC that utilizes the SoCDMMU has 440% overall speedup of the application transition time over fully shared memory that does not utilize the SoCDMMU.

## Keywords

System-on-a-Chip, dynamic memory management, real-time systems, embedded systems, SoCDMMU, two-level memory management, Atalanta, real-time operating systems.

## 1. INTRODUCTION

In few years integrated circuits will have close to one billion transistors on a single chip [25]. Such chips will no longer be individual components to a system but "silicon boards." Given that current computers waste much time transferring data between compute and storage units, it is appealing to combine significant processing power and a large amount of memory in the same chip. A typical System-on-a-Chip (SoC), as shown in Figure 1, will consist of multiple *Processing Elements* (PEs) of various types (i.e., general purpose processors, domain-specific CPUs such as DSPs, and custom hardware), configurable logic, large memory, analog components and digital interfaces [1], [2], [20]. Architecture such as this will be suitable for embedded real-time applications. Such applications — especially multimedia — require great processing power and large volume data management [13], [19].

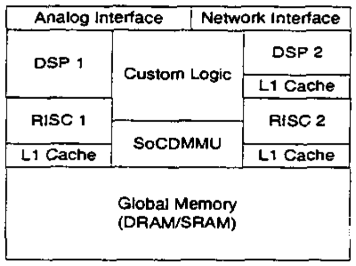

**Figure 1. Example of A Billion-Transistor SoC.**

Designers of multiprocessor SoC with heterogeneous processing elements and significant on-chip memory must pay careful attention to the management of the on-chip memory. They have to decide whether the allocation will be static (i.e., determined at compile time) or dynamic (decided at run-time and capable of being changed from one moment to another during operation)? Most previous research in the memory management for embedded systems has focused on static allocation and how to synthesize memory hierarchies for an SoC [3], [5], [17], [19]. The static allocation of memory makes the on-chip memory utilization inefficient especially for applications whose memory requirements change significantly during run-time. Moreover, it makes system modification after implementation very difficult. On the other hand, dealing with memory allocation between the PEs in a dynamic way can make the memory utilization more efficient. Also, the memory allocation will be programmable and can be changed at any moment depending on the system load. From the general-purpose end of the spectrum, there has been significant research in shared memory multiprocessing [22]. However, in shared memory multiprocessing, dynamic memory allocation is not deterministic; moreover, it typically requires hundreds or thousands of clock cycles in the worst case [14], which makes satisfaction of real-time constraints on such shared memory architectures difficult if not impossible.

We proposed a novel approach for memory allocation/de-allocation between PEs in an SoC that is suitable for real-time applications [11]. Such systems require behavior that is deterministic and fast in all cases, unlike general-purpose computer systems, which have memory management that may be neither fast nor deterministic. Our approach focuses on implementing a special hardware SoC Dynamic Memory Management Unit (SoCDMMU) to dynamically allocate the large global on-chip memory between the PEs. Note that after the SoCDMMU allocates a portion of the large global on-chip memory to a particular PE, the PE itself manages the use of this memory by its processes/threads. The SoCDMMU allows fast and deterministic dynamic memory allocation/de-allocation of large global on-chip memory between the PEs.

In this paper, we show how a Real-Time Operating System (RTOS) might be modified to support the SoCDMMU. Also, we present simulation results that show a multiprocessor SoC that utilizes the SoCDMMU performs far better than an equivalent system that does not utilize the SoCDMMU.

This paper is organized as follows. First, Section 2 gives an overview of the work done to implement the memory management in hardware. To make the paper self-contained, Section 3 briefly describes the SoCDMMU architecture. Section 4 shows Real-Time Operating System (RTOS) support for the SoCDMMU. Section 5 gives some experimental results, and finally Section 6 concludes the paper.

## 2. RELATED WORK

The dynamic management of memory has been an important topic in computer systems for a long time. Dynamic memory management can consume a great amount of a program's execution time — especially object-oriented applications. Moreover, memory management routines often do not have deterministic behavior. To reduce the execution time of dynamic memory management routines (allocation, de-allocation, and garbage collection) and/or make their execution times deterministic, many researchers have proposed hardware accelerators for dynamic memory management. The literature shows that a hardware implementation of a simple buddy allocator was first proposed by Puttkamer [5]. P.R. Wilson et al. mention a possible implementation of a bitmap memory allocator in hardware [14]. Chang and Gehringer propose a modified hardware-based buddy system, which eliminates internal fragmentation [8]. Chang et al. have implemented the *malloc()*, *realloc()*, and *free()* C-Language functions in hardware [18]. Also, they propose a hardware extension to be a part of the future microprocessors to accelerate the dynamic memory management [7]. In the same way, Cam et al. propose a hardware buddy allocator that detects any available free block of requested size and eliminates the internal fragmentation [6]. The previous research focuses only on the hardware implementation of specific functionality (e.g., allocation or de-allocation) but does not discuss in detail how these functionalities could be integrated into a system nor present any system examples. Moreover, they focus only on speeding up memory management rather than making it deterministic, which means some of the previous implementations are not suitable for real-time systems.

## 3. THE SoCDMMU

In this section we briefly give an overview of the SoCDMMU [11]. The SoC Dynamic Memory Management Unit (SoCDMMU) is a hardware unit, to be a part of the SoC, which deals with the global on-chip memory allocation/de-allocation between the PEs. The SoCDMMU allows a fast and deterministic dynamic way to allocate/de-allocate the global memory (see Figure 1) between the PEs.

The SoCDMMU resides between the PEs and the global on-chip memory. Each PE's memory bus is connected to the SoCDMMU to allow the SoCDMMU to control all of the global memory accesses. This enables the SoCDMMU to convert the PE-address to a physical address. The PE can map any allocated block to any memory location inside the PE's address space. This feature allows the allocation of non-contiguous memory blocks, so there is no need for memory compaction of the global memory blocks (memory compaction may be an issue within a particular block).

**Table 1: Execution Times in Cycles**

| Command | Number of Cycles |
|---|---|
| G_alloc_ex | 4 |
| G_alloc_rw | 4 |
| G_alloc_ro | 3 |
| G_dealloc | 4 |
| Worst-Case Execution Time | 4 x (the number of the PEs in the SoC) |

The SoCDMMU is mapped into a location in the I/O space of each PE. This memory mapped address or I/O port to which the SoCDMMU is mapped is used to send commands to the SoCDMMU (write data to the port or memory-mapped location) and to receive the status of the command execution (reading from the port memory-mapped location). There are three types of commands that the SoCDMMU can execute: G_Allocate commands (exclusive, read-only, and read-write), G_deallocate command, and Move command. The move command is used to re-map allocated memory blocks to another location in the PE-address space. This is useful because it allows PE address space compaction. Table 1 summarizes the execution time of each of the SoCDMMU commands in cycles.

## 4. RTOS SUPPORT FOR THE SoCDMMU

Conventional memory allocation algorithms (e.g., buddy-heap) are not suitable for Real-Time systems because they are not deterministic and/or have a quite high Worst Case Execution Time (WCET). A deterministic execution time is a very desirable trait for real-time applications. Currently, software approaches to automatic dynamic memory management often fail to yield predictable execution time [7]. The most often used software approach in maintaining allocation status is sequential fit or segregated fit. These two approaches utilize a linked-list to keep the occupied chunks or free chunks. With a linked-list, the turnaround time often relates to the length of the list. As the linked-list becomes longer, the sequential search time grows longer as well [14]. Similarly, the software approaches to garbage collection also yield unpredictable turnaround time. Two of the most common approaches for garbage collection are mark-sweep and copying collector. In both instances, the execution time is not deterministic [7], [14].

The fastest and most deterministic approach to memory management is to disallow dynamic memory allocation and to make the programmer allocate all memory statically. However, such an approach has obvious problems dealing with dynamically changing workloads, e.g., as would be introduced by downloading new code onto a PDA. Another approach is to allow dynamic memory allocation but to not support dynamic memory allocation in the kernel [12]. In this case, the kernel is fast and deterministic, but any dynamic memory allocation falls outside of the scope of the kernel and thus is the responsibility of the user!

Yet another approach to "dynamic" memory allocation is to statically assign partitions with fixed block sizes (e.g., partitions of size 1KB with blocks of 32B) used to satisfy "dynamic" memory allocations [9], [23], [24], [26]. In this case, each request can only be for a single block which has the advantage of short and predictable execution time due to the fact that only one pointer needs to be changed [9]. However, the disadvantage occurs in allocating multiple blocks: the allocation time is linear in the number of blocks allocated!

A more fully dynamic memory allocation involves the use of *malloc()*, *free()*, and their equivalents. Our hardware/software multiprocessor RTOS implementation uses a different approach to support this fully dynamic case (allocation of memory using *malloc()*, *free()* and their equivalents). In the following section we give an overview of a way to support dynamic memory management of partitions by extending the Atalanta open-source RTOS developed at Georgia Tech [4] to support the SoCDMMU introduced previously [11].

## 4.1 Atalanta Memory Management

An RTOS usually divides the memory into fixed-sized allocation units and any task can allocate only one unit at a time [9], [23], [24], [26]. However, we present in this section a way to support real-time (see Table 6 and Table 7) dynamic allocation of partitions using Atalanta for the RTOS software and the SoCDMMU for part of the RTOS functionality in hardware. Atalanta is an open source RTOS developed at the Georgia Institute of Technology to be used for SoC [4]. We adapted Atalanta to support the SoCDMMU. As an RTOS, Atalanta manages memory in a deterministic way; tasks can dynamically allocate fixed-size blocks by using memory partitions as shown in Figure 2.

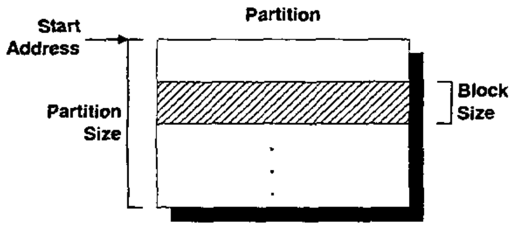

**Figure 2: Memory partition in Atalanta.**

Partitioned memory is allocated and returned in fixed-size blocks. One advantage of partitioned memory is that allocation and de-allocation of blocks can be done in constant time. Since all memory blocks in a partition are the same size, the external fragmentation that sometimes results from dynamic memory allocation does not occur. Consequently, memory compaction is not required. During RTOS initialization the partitions should be created as static arrays. Atalanta provides only four Application Programming Interface (API) functions to manage the memory. These functions are summarized in Table 2.

**Table 2. Atalanta Memory Management System Calls.**

| Function Name | Description |
|---|---|
| *asc_partition_gain* | Get free memory block from a partition (non-blocking) |
| *asc_partition_seek* | Get free memory block from a partition (blocking) |
| *asc_partition_free* | Free a memory block. |
| *asc_partition_reference* | Get partition information. |

## 4.2 Atalanta Support for SoCDMMU

We modified the Atalanta RTOS memory management to support the SoCDMMU and to allow the Atalanta RTOS to work in a multiprocessor SoC environment. While modifying the Atalanta memory management system we kept on mind the following issues. First, we add dynamic memory management for the global on chip memory. Second, we use the same memory management API functions of Atalanta. Third, we keep the memory management deterministic. Also, the following facts governed our modifications:

- The SoCDMMU needs to know where the allocated physical memory will be placed in the PE address space. This is required by the SoCDMMU allocation commands [11].
- The PE address space is much larger than the available on chip memory (a typical figure would be 64 MB of global on chip memory vs. 4GB address space for a typical 32-bit processor). This fact can be used to develop an alternative solution for the PE address space fragmentation explained earlier in Section 3.

**Table 3. New API memory management Functions introduced to the Atalanta RTOS.**

| Function Name | Description |
|---|---|
| *asc_partition_create* | Create a partition by requesting memory allocation from the SoCDMMU if necessary. |
| *asc_partition_delete* | Delete a partition and de-allocate memory block if required. |

We added new API functions to Atalanta both to create partitions at run-time when required and to delete the partitions later when no longer required (as opposed to creating the partitions as static arrays not modifiable at run-time). Table 3 explains these two new functions. The *asc_partition_create* function creates a partition in the memory allocated to the PE. If there is not enough memory or the available memory has a different mode (read only, exclusive) than that of the requested partition, *asc_partition_create* requests memory from the SoCDMMU. The *asc_partition_delete* function deletes a partition when it is not required anymore; *asc_partition_delete* will request memory de-allocation from the SoCDMMU if the entire physical block corresponding to the partition is not in use anymore.

**Table 4. *asc_memory_find* Function.**

| Function Name | Description |
|---|---|
| *asc_memory_find* | Find a place in the PE address space to which to map the allocated memory. |

Recalling that the SoCDMMU G_allocate commands require a place in the PE address space into which the physical memory blocks can be mapped, we need a function that finds an empty space in the PE address space into which to map the physical blocks. This function is called *asc_memory_find*. Table 4 gives a description of this function. The *asc_memory_find* function works in a way that minimizes the PE address space fragmentation; to achieve this, the PE address space is divided into pools (a pool is an address range in the PE's address space). Each pool has the same size of the total on-chip memory. Each pool can be used to map pages of the same size (1 block, 2 blocks, etc.,). The page size of each pool is selected to be one of the commonly used page sizes. If the commonly used page sizes are large in number, a pool can be used to allocate pages of any arbitrary size; and the SoCDMMU move command is utilized to perform address space compaction. For example, if the total on-chip memory is 64MB and the PE address space is 4GB then we have 64 pools each of 64MB. The first pool may be used to place 1-block pages, the second pool for 2-block pages, etc., as illustrated in Figure 3.

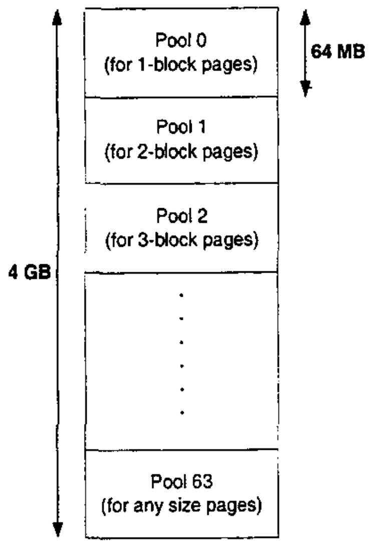

**Figure 3: The PE's address space divided into pools.**

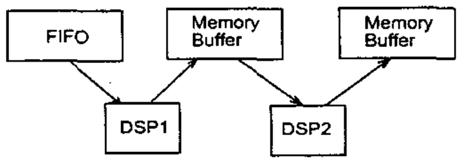

**Figure 4: OFDM Sub-System**

**Example 1** Consider a multiprocessor SoC whose functionality is dynamically changed to include OFDM communication. The SoC has two DSP processors and a global on-chip memory. The two DSP processors utilize Atalanta as the RTOS. The first DSP reads the incoming data from the FIFO buffer and performs a 1024-point FFT for each received symbol to find the original transmitted spectrum and then stores the results into a memory buffer that is shared with the second DSP. The phase angle of each transmission carrier is then evaluated and converted back to data words by demodulating the received phase. The demodulation is performed by DSP2. The operation is outlined in Figure 4. DSP1 allocates the shared memory buffer as read/write and DSP1 allocates it as read only. Figure 5 shows the code snippets for DSP1 and DSP2 that performs the dynamic memory allocations.

**DSP1**

```
#define BUF1   0x10
.
.
SYS_ERROR e;
SYS_PARTITION p1;
SYS_MEM m1;
.
.
p1=asc_partition_create(2,1,DMMU_RW,BUF1,&e);
m1= asc_partition_gain(p1,&e);
.
asc_partition_free(p1,m1,&e);
```

**DSP2**

```
#define BUF1   0x10
#define BUF2   0x20
.
.
SYS_ERROR e;
SYS_PARTITION p1;
SYS_MEM m1;
SYS_PARTITION p2;
SYS_MEM m2;
.
p1=asc_partition_create(2,1,DMMU_RO,BUF1,&e);
m1= asc_partition_gain(p1,&e);
p2=asc_partition_create(3,1,DMMU_EX,BUF2,&e);
m1= asc_partition_gain(p2,&e);
.
asc_partition_free(p2,m2,&e);
```

**Figure 5: Code snippets for the OFDM System in Example 1.**

## 5. EXPERIMENTS

### 5.1 Comparison to a Fully Shared-Memory Multiprocessor System

In this experiment we compare (i) a system that utilizes the SoCDMMU and uses the memory sharing scheme implied by using the SoCDMMU [11] to (ii) a fully shared-memory multiprocessor system. The simulation is carried out using the Mentor Graphics Seamless Co-simulation environment and the ARM Software Development Tools (SDT) v2.5 [21]. The simulated system shown in Figure 6 consists of four ARM9TDMI cores each of which has a Level one (L1) cache of 64Kbytes. All four PEs share a global bus. A shared memory of 16 Mbytes of RAM is connected to the same bus. We assume it takes five cycles to get the first word from the global memory in Figure 6. A bus arbiter controls the access of the cache controllers to the memory. The system (including the SoCDMMU) is clocked at 100MHz. The SoCDMMU has the size of 41,500 equivalent gates using the AMI 0.5-micron Logic library [11].

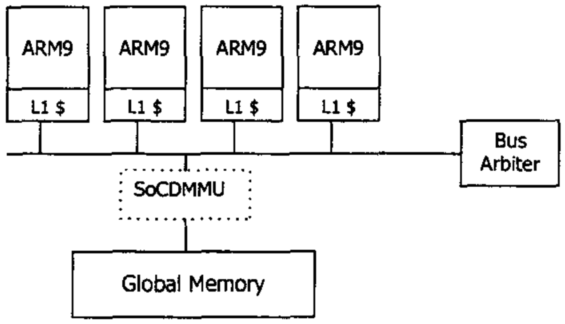

**Figure 6: Four-PE SoC with SoCDMMU.**

This SoC is used for a handheld device that can be used for communication as well as other personal applications (e.g., Video Player) like the example that is described in [16]. The device also uses OFDM. Table 5 shows the memory requirements for the MPEG-2 video player [15] and the OFDM receiver [10]. We assume that other applications take up 13.9 Mbytes leaving 2.1 Mbytes available for the OFDM receiver or the MPEG player (depending on which is running). Table 6 compares the execution time of the *malloc()* function in cycles to that of the *asc_partition_create()* and the *asc_partition_gain()* functions that utilize the SoCDMMU. Table 7 compares the execution time of the *free()* function in cycles to that of the *asc_partition_free()* that utilizes the SoCDMMU.Table 8 shows the number of cycles required to free the memory used by the MPEG-2 player and allocate the memory required by the OFDM receiver when the switching takes place. From the results, we can see that using the SoCDMMU yields more than 440% speedup (4.4x as shown in Table 6). Note that in the fully-shared memory multiprocessor, the *malloc()* and *free()* functions used for the comparison are optimized for speed for embedded applications; normal *malloc()* and *free()* implementations (e.g., *gclib*) may have larger execution times. Further, note that the execution time given for *malloc()* and *free()* was done assuming the other applications are not also requesting memory; if they were, the execution time for *malloc()* and *free()* would be longer. Finally, also note that the fully shared memory system is exactly the same as Figure 6 without the SoCDMMU (note that the SoCDMMU area is roughly equivalent to 64KB of DRAM area or 8KB of SRAM area).

**Table 5: Required Memory Allocations**

| MPEG-2 Player | OFDM Receiver |
|---|---|
| 2 Kbytes | 34 Kbytes |
| 500 Kbytes | 32 Kbytes |
| 5 Kbytes | 1 Kbytes |
| 1500 Kbytes | 1.5 Kbytes |
| 1.5 Kbytes | 32 Kbytes |
| 0.5 Kbytes | 8 Kbytes |
| | 32 Kbytes |

**Table 6: Execution Times of *malloc()* and the SoCDMMU allocation.**

| | Execution Time |
|---|---|
| *malloc()* | 106 cycles |
| SoCDMMU allocation | 28 cycles |
| Speed up | 3.78x |

**Table 7: Execution Times of *free()* and the SoCDMMU de-allocation.**

| | Execution Time |
|---|---|
| *free()* | 83 cycles |
| SocDMMU de-allocation | 14 cycles |
| Speed up | 5.9x |

**Table 8: Execution Times**

| Using the SOCDMMU | Using SDT *malloc()* and *free()* |
|---|---|
| 280 cycles | 1240 cycles |
| Speedup | 4.4x |

## 6. CONCLUSION

In this paper, we described an approach to handle on-chip memory allocation between PEs in an SoC. Our approach is based on hardware SoCDMMU that provides a dynamic, fast way to allocate/de-allocate the global on-chip memory. Moreover, the SoCDMMU allocation/de-allocation of the memory blocks is completely deterministic, which makes it suitable for real-time SoC applications. We showed how an RTOS might be adapted to support the SoCDMMU. Also, we showed an example where our approach gives a 440% overall speedup in application transition time when compared to a fully shared memory system with the same memory organization and number of processors.

## 7. ACKNOLWLEDGMENTS

This research is funded by NSF under INT-9973120, CCR-9984808 and CCR-0082164. We also acknowledge donations received from Denali, Hewlett-Packard, Intel, LEDA, Mentor Graphics, Sun and Synopsys.

## 8. REFERENCES

[1] B. Ackland et al., "A Single-Chip 1.6 Billion 16-b MAC/s Multiprocessor DSP," *IEEE Journal of Solid-State Circuits*, vol. 35, no. 3, pp. 412 –424, March 2000.

[2] C. E. Kozyrakis et al., "Scalable Processors in the Billion-Transistor Era: IRAM," *IEEE Computer*, vol. 30, no. 9, pp. 75-78, September 1997.

[3] D. Verkest et al., "CoWare — A Design Environment for Heterogeneous Hardware/Software Systems," in *Proceedings of the EURO-DAC '96 European Design Automation Conference with EURO-VHDL '96*, October 1996, pp. 357-86.

[4] D. Sun, D. M. Blough, and V. J. Mooney, "Atalanta: A New Multiprocessor RTOS Kernel for System-on-a-Chip Applications," Georgia Institute of Technology, Atlanta, Georgia, Technical Report GIT-CC-02-19, 2002, http://www.cc.gatech.edu/tech_reports/

[5] E. V. Puttkamer, "A simple hardware buddy system memory allocator," *IEEE Transaction on Computers*, vol. 24, no. 10, pp. 953-957, October 1975.

[6] H. Cam et al., "A high-performance hardware-efficient memory allocation technique and design," *International Conference on Computer Design*, October 1999, pp. 274-276.

[7] J. M. Chang et al., "Introduction to DMMX (Dynamic Memory Management Extension)", *ICCD Workshop on Hardware Support for Objects and Micro architectures for Java*, October 1999, pp. 11-14.

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

[9] J. Labrosse, *MicroC/OS-II, the Real-Time Kernel*. Lawrence, KS: R&D Books, 1998.

[10] K. Ryu, E. Shin and V. Mooney, "A Comparison of Five Different Multiprocessor SoC Bus Architectures," in *Proceedings of the EUROMICRO Symposium on Digital Systems Design*, Sept. 2001, pp. 202-209.

[11] M. A. Shalan and V. Mooney, "A Dynamic Memory Management Unit for Embedded Real-Time System-on-a-Chip," *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, November 2000, pp. 180-186.

[12] M. F. Komarinski and J. Godoy, "Real-Time and Embedded HOWTO," http://www.mech.kuleuven.ac.be /~bruyninc/rthowto/rtHOWTO/memory-management.html.

[13] P. R. Panda et al., "Memory Data Organization for Improved Cache Performance In Embedded Processor Applications," *ACM Transactions on Design Automation of Electronic Systems*, vol. 2, no. 4, pp. 384–409, October, 1997.

[14] P. R. Wilson et al., "Dynamic Storage Allocation: A Survey and Critical Review," *International Workshop on Memory Management*, September 1995, pp. 1-78.

[15] P. Soderquist, "Memory Traffic and Data Cache Behavior of an MPEG-2 Software Decoder, "*Proceedings of the International Conference on Computer Design (ICCD '97)*, October 1997.

[16] S. Morgan, "Jini to the rescue," *IEEE Spectrum*, vol. 37, no. 4, pp. 44-49, April 2000.

[17] S. Wuytack et al., "Memory Management for Embedded Network Applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 5, pp. 533, May 1999.

[18] W. Srisa-an, C. D. Lo, and J. M. Chang, "A Hardware Implementation of Realloc Function," *Proceedings of WVLSI'99 IEEE Annual Workshop on VLSI*, April 1999, pp. 106-111.

[19] Y. Li and W.H. Wolf, "Hardware/Software Co-Synthesis with Memory Hierarchies," *IEEE Transaction Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 10, pp. 1405-1417, October 1999.

[20] Virtex-II Pro Platform FPGAs, http://www.xilinx.com/ virtex2pro/.

[21] ARM Software Development Products, http://www.arm. com/devtools/soft_dev_tools?OpenDocument.

[22] *Proceedings of IEEE, Special Issue on Distributed Shared Memory Systems*, vol. 87, no. 3, pp. 397-532, March 1999.

[23] *eCos Reference Manual*, redhat, September 2000.

[24] *pSOS System Concept*, Integrated Systems, 1996.

[25] Semiconductor Industry Association, "*International Technology Roadmap for Semiconductors (ITRS)*," November 2001, http://www.semichips.org.

[26] *VRTXsa Real-Time Kernel [Programmer's Guide and Reference]*, Microtec Research, 1996.