DYNAMIC MEMORY MANAGEMENT FOR EMBEDDED REAL-TIME MULTIPROCESSOR SYSTEM ON A CHIP

A Thesis
Presented to
The Academic Faculty

by

Mohamed A. Shalan

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy

School of Electrical and Computer Engineering
Georgia Institute of Technology
November 2003

DYNAMIC MEMORY MANAGEMENT FOR EMBEDDED REAL-TIME MULTIPROCESSOR SYSTEM ON A CHIP

To my family

ACKNOWLEDGMENTS

I extend my sincere gratitude and appreciation to everyone who made this Ph.D. thesis possible. Special thanks are due to my supervisor, Professor Vincent Mooney, for his patience and guidance. Also, I would like to thank my colleagues: Pramote Kuacharoen, Tankut Akgul and Eung Shin for their support. Moreover, I have to thank Brian Faust for his help in porting the SPLASH-2 benchmarks.

Finally, I would like to give my special thanks to my wife whose patient love enabled me to complete this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

ASIC	Application Specific Integrated Circuit
CAD	Computer-Aided Design
DX-Gt	**D**ynamic memory management unit and **X**bar **G**enera**t**or
EDA	Electronic Design Automation
FFT	Fast Fourier Transform
GUI	Graphical User Interface
HDL	Hardware Description Language
HW	Hardware
IC	Integrated Circuit
IP	Intellectual Property
ITRS	International Technology Roadmap for Semiconductors
OFDM	Orthogonal Frequency Division Multiplexing

PDA	Personal Digital Assistant
PE	Processing Element
RTL	Register Transfer Level
RTOS	Real Time Operating System
SoC	System-on-a-Chip
SoCDMMU	System-on-a-Chip Dynamic Memory Management Unit
SW	Software
VLSI	Very Large Scale Integration

SUMMARY

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 System-on-a-Chip (SoC) designs. Such SoC designs typically have more than one processor and huge (tens of Mega Bytes) amount of memory, all on the same chip. However portions of the on-chip memory can be dynamically allocated to the on-chip processors during runtime using software memory allocation techniques, real-time SoC designers prefer to partition the on-chip memory statically during the design phase to achieve maximum predictability sacrificing flexibility and efficient memory utilization. Dealing with the global on-chip memory allocation/deallocation in a dynamic yet deterministic way is an important issue for 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 shift in the way designers look at on-chip dynamic memory allocation – we present the 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 management of memory allocated to a particular on-chip Processing Element, e.g., an operating system's management of memory allocated to a particular processor). 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. A new tool is introduced to generate a custom optimized version of the SoCDMMU. Also, a real-time operating system is modified support the new proposed SoCDMMU. We show an example where shared memory multiprocessor SoC that employs the Two-Level Memory Management and utilizes the SoCDMMU has an overall average speedup in application transition time as well as normal execution time.

CHAPTER I

INTRODUCTION

In a year or so an integrated circuit will appear with over one billion transistors on a single chip [2, 34]. Such a chip is no longer an individual component to a system but instead is a "silicon board." Given that current computers spend 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 billion transistor 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), reconfigurable logic, large memory, analog components and digital interfaces [2, 19, 22, 52]. An 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 [23].

Designers of a multiprocessor SoC with heterogeneous processing elements and

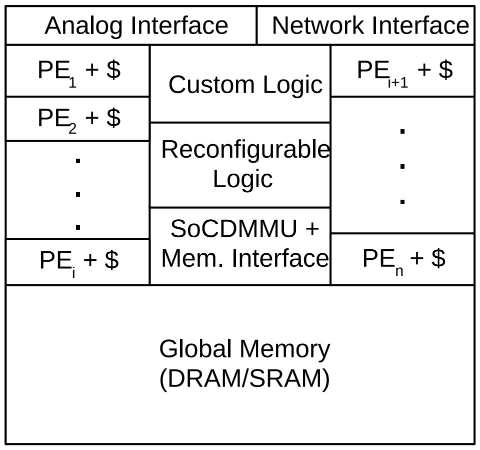


Figure 1: Example of a billion-transistor SoC.

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 memory management for embedded systems has focused on static allocation and how to synthesize memory hierarchies for an SoC [23, 31, 51]. The static allocation of memory makes the on-chip memory utilization inefficient, especially for applications whose memory requirements change significantly during run-time. Moreover, such static memory allocation makes system modification after implementation very difficult if not impossible. On the other hand, dealing with memory allocation between the PEs in a dynamic way can result in more efficient utilization of memory. 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 [1]. However, in shared memory multiprocessing, dynamic memory allocation is almost never deterministic; moreover, it typically requires hundreds or thousands of clock cycles in the worst case [1, 49], which makes satisfaction of real-time constraints on such shared memory architectures difficult if not impossible.

In this thesis, we present a memory management hierarchy, Two-Level Memory Management, for a multiprocessor SoC that combines the best of dynamic memory management techniques (flexibility and efficiency) with the best of static memory allocation techniques (determinism). The result is suitable for real-time systems as well as non real-time systems. In Two-Level Memory Management, large on-chip memory (not the L1 caches) is managed between the on-chip processors ("Level Two" management of memory); while the memory assigned to any processor is managed by the operating system running on that particular processor ("Level One" management of memory). To implement this memory management scheme we present the System-on-a-Chip Dynamic Memory Management Unit (SoCDMMU) for allocation of the global on-chip memory among the on-chip processors.

The thesis is organized as follows. First, Chapter 2 will give an overview of memory management techniques and problems focusing only on memory allocation and deallocation. Chapter 3 will describe our proposed Two-Level memory management hierarchy for multiprocessor SoCs. Chapter 4 discusses the SoCDMMU and shows the implementation of the SoCDMMU as a custom hardware and as software running on a general purpose microcontroller. Chapter 5 describes a CAD tool that can generate an optimized SoCDMMU hardware according to the user's inputs. Chapter 6 discusses the implementation of Two Level memory management in a real-time operating system. Finally in Chapter 7, we present some experimental results.

CHAPTER II

PREVIOUS WORK

2.1 Introduction

Memory management can be defined as the process of coordinating and controlling the use of memory in a computer system. Memory management has been an important topic in computer systems for a long time, and there have been many techniques developed to make it more efficient. Memory management can be static (determined at compile time) or dynamic (determined at run-time).

2.2 Static Memory Management

In static memory management, the memory is allocated (or assigned) at compile (or design) time. Static memory management can be as simple as allocating static arrays or as complex as synthesizing memory structures and/or software memory allocators suitable for certain type of applications [23, 51, 47].

Wolf et al. present an algorithm for hardware software co-synthesis of multi-rate real-time systems on heterogeneous multiprocessors that considers the impact of memory hierarchy on the system performance and cost [23]. The algorithm not only synthesizes the hardware and software parts of the applications, but also the memory hierarchy: it takes into account the impact of memory hierarchies on system performance and cost in the co-synthesis process.

MATISSE [47] is a design environment that takes care of the background memory management problem for dynamic data structure intensive applications by bridging the gap from a system specification, using a concurrent object-oriented language, to an optimized embedded single-chip hardware/software implementation. The MATISSE environment addresses all the aforementioned tasks to synthesize a custom distributed memory architecture. It permits the exploration of different architectures so that an optimal choice can be made, which is crucial as memory bandwidth often is the main performance bottleneck in this type of application.

Static memory management techniques are suitable for real-time systems as the memory allocation is deterministic (determined at design time). However, static memory management techniques do not always use memory efficiently. Moreover, static memory management techniques are tuned for a certain application (or set of applications); hence, they typically do not work efficiently with applications that are different than the applications used during the design phase.

2.3 Dynamic Memory Management

In dynamic memory management, memory is allocated at run-time. Dynamic memory management uses memory efficiently when compared to static memory management techniques. However, dynamic memory management can consume a great amount of a program's execution time [41] – especially in object-oriented applications [53]. Dynamic memory management can be classified into two categories:

- ***Manual memory management***

  In manual memory management, the programmer has direct control over when memory is allocated and when it might be recycled. Usually this is either by explicit calls to *heap* management functions (e.g., *malloc()/free()* in the C language) or by language constructs that affect the stack (such as local variables). Although manual memory management is easier for programmers to understand and use, memory management bugs are common when manual memory management is used.

- ***Automatic memory management***

  Automatic memory management is a service, either as a part of the language (e.g., Java and Lisp) or as an extension, that automatically deallocates memory that a program will not use again. Automatic memory managers (often known as garbage collectors) usually do their job by recycling blocks that are unreachable from program variables. Automatic memory management eliminates most memory management bugs. However, sometimes automatic memory managers fail to deallocate unused memory locations causing memory leaks.

Although automatic memory management seems attractive, automatic memory mangers usually consume a great amount of the CPU time and usually have non-deterministic behavior. In this thesis, we focus only on supporting manual memory management techniques suitable for real-time systems.

Many mechanisms have been proposed to manage memory manually. These mechanisms can be categorized into two groups: (1) software-based solutions and (2) hardware-based solutions. Sections 2.2, 2.3 and 2.4 explain these state-of-the-art mechanisms and discuss their drawbacks. Before going further, we introduce definitions of some memory management terms that will be used in the following sections.

- ***Heap***. The heap is the memory area managed by dynamic allocation.
- ***Coalescing***. Coalescing is the act of merging two adjacent free memory blocks. Coalescing can be done as soon as blocks are freed, can be deferred until some time later (known as deferred coalescing) or can be not done at all, ever.
- ***External fragmentation***. External fragmentation is the division of free memory into many scattered small blocks. The free blocks cannot be coalesced; hence, no large contiguous blocks can be allocated.
- ***Internal fragmentation***. Internal fragmentation occurs when the memory manager allocates more for an allocation than is actually requested.

- ***Memory Leak***. A memory leak is a condition caused by tasks/programs that allocate memory blocks without ever deallocating them. As a consequence, the dynamic memory manager usually runs out of memory, thus rendering the system unusable.

2.4 Conventional Software Memory Allocation

Memory allocation is the process of assigning blocks of memory on request. An allocator must keep track of which parts of memory are in use and which parts are free. The optimal memory allocator would spend minimal time managing memory and waste minimal (preferably zero) memory space. There are many common ways to perform memory allocation, with different strengths and weaknesses. Common memory allocation techniques can be divided into the following mechanisms: sequential fits, segregated free lists, the buddy system, indexed fits and bitmapped fits.

2.4.1 Sequential Fits

Sequential fit algorithms use a single linear list of all free memory blocks called a *free list*. Initially, the *free list* contains only one block that represents all the memory available for allocation. Typically, the *free list* is usually encoded inside the free memory blocks, eliminating the need for dedicated memory. The search for a free memory block in the list can be done in several ways which include the following:

- ***First Fit.*** The First Fit algorithm searches the *free list* from the beginning for the first free memory block large enough to satisfy the allocation request.

If the block found is too large, the allocator splits the block and the remaining part is put back into the *free list*.

- ***Next Fit.*** The Next Fit algorithm is similar to First Fit algorithm, except the *free list* is searched from the location where the last allocated block was found.

- ***Best Fit.*** The Best Fit algorithm searches the *free list* for the best (smallest free memory block that satisfies the allocation size) matched memory block. While the Best Fit algorithm has good memory utilization in the general case, it suffers from long execution times.

- ***Worst Fit.*** The Worst Fit algorithm always searches for the largest free memory block to satisfy the requested allocation. In practice, this tends to work quite poorly because it quickly eliminates large blocks, so that eventually large requests cannot be met. The Worst Fit algorithm works well only when the largest possible allocation is much smaller than heap size.

Since the sequential fit algorithms make use of a sequentially searched list (*free list*), they suffer from serious scalability problems [49]. As memory available for allocation becomes large, the search time becomes large and in the worst case can be proportional to the memory size. Also, as the *free list* becomes large, the memory used to store the *free list* becomes large, hence the memory overhead becomes large.

2.4.2 Segregated Free Lists

Segregated free lists are a class of allocation mechanisms which divide the *free list* into several subsets, according to the size of the free blocks. A freed or coalesced block is placed on the appropriate subset list. An allocation request is serviced from the appropriate subset list. Segregated free list algorithms can be categorized as follows:

- ***Simple Segregated Storage*** Simple segregated storage is a segregated free list allocation mechanism that divides storage into areas and only allocates objects of a single size, or small range of sizes, within each area. No memory splitting or coalescing is allowed. This makes allocation fast, but may lead to high external fragmentation. If the requested allocation size is not available, a larger memory block is allocated and the unused part of allocated memory block cannot be used for other allocations.

- ***Segregated Fit*** Similar to the simple segregated storage mechanism, segregated fit mechanisms make use of an array of *free list*s, each holding free blocks of a particular range of sizes. To satisfy an allocation request, the allocator identifies the appropriate *free list* and allocates from it (often using a sequential fit mechanism such as first fit). If this fails, a larger block is taken from another list and split. The remainder is put into an appropriate *free list.* When a memory block is freed, it might be coalesced with a block in the same *free list* and put into an appropriate *free list.* Although segregated fit algorithms do not suffer from severe fragmentation, they have large search times.

2.4.3 Buddy System

A buddy allocator uses an array of *free list*s, one for each allowable block size (e.g., allowable block size is a power of 2 in the case of binary buddy system). The buddy allocator rounds up the requested size to an allowable size and allocates from the corresponding *free list*. If the *free list* is empty, a larger block from another *free list* is selected and split. A block may only be split into a pair of buddies (of the same size in case of binary buddy system). A block may be coalesced only with its buddy, and this is possible only if the buddy has not been split into smaller blocks.

The advantage of a buddy system is that the buddy of a block being freed can be quickly found by a simple address computation. The disadvantage of a buddy system is that the restricted set of block sizes leads to high internal fragmentation, as does the limited ability to coalesce.

Different sorts of buddy system are distinguished by the allowable block sizes and the method of splitting. They include binary buddies (the most common), Fibonacci buddies (block sizes are Fibonacci numbers), and weighted buddies (block sizes are 2^k or $3 * 2^k$).

2.4.4 Indexed Fits

Indexed fits are a class of allocation mechanisms that use an indexing data structure, such as a tree or hash table, to identify suitable free blocks, according to the allocation policy. For instance, a tree ordered by block size may be used to implement the best fit policy. Although indexed fits allocators seem to have smaller search times than other memory allocators for the average case, in practice, they tend to have search times linearly proportional to heap size [49].

2.4.5 Bitmapped Fits

Bitmapped fits are a class of allocation mechanisms that use a bitmap to represent the usage of the heap. Each bit in the map corresponds to a part of the heap, typically a word, and is set if that part is in use. Allocation is done by searching the bitmap for a run of clear bits. Bitmapped fit mechanisms have never found their way into software allocators [49]; however, they have been used with hardware memory allocators [8, 10, 11, 15, 29, 40, 41]. Although bitmapped fits have the advantage of having a small memory overhead, the search time is proportional to the bitmap size.

2.5 *Real-Time Software Memory Allocation Techniques*

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 Personal Digital Assistant (PDA). Another approach is to allow dynamic memory allocation but to not support dynamic memory allocation in the kernel [7]. In this case, the kernel is fast and deterministic, but any dynamic memory allocation falls outside the scope of the kernel and thus is the responsibility of the user!

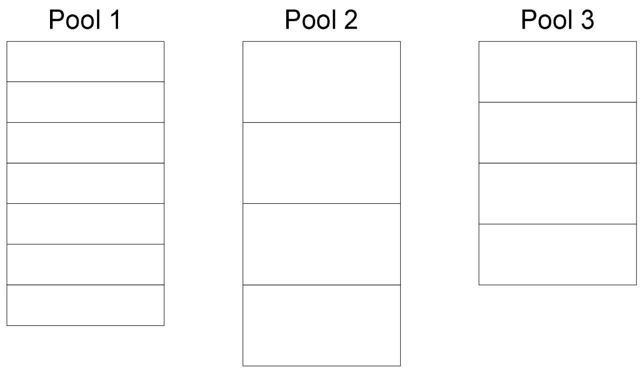


Figure 2: Memory pools of fixed-sized allocation units.

Yet another approach to "dynamic" memory allocation is to statically assign partitions (at compile time) with fixed block sizes (e.g., partitions of size 1KB with blocks of 32B) used to satisfy "dynamic" memory allocations [14, 20, 26, 30]. In this case, each request can only be for a single block, which has the advantage of short and predictable execution time because only one pointer needs to be changed [20]. However, the disadvantage occurs in allocating multiple blocks: the allocation time is linear in the number of blocks allocated! Furthermore, since the partitions are statically created, the partition should be large enough to satisfy the worst-case memory requirements during the runtime.

An RTOS usually divides the memory into pools each of fixed-sized allocation units, as shown in Figure 2, and any task can allocate only one unit at a time [14, 20, 26, 30]. In this way the RTOS memory manager does not need to use complex data structures to store the status of the allocated memory chunks and hence eliminates the non-deterministic or long search time associated with complex data structures. Thus, an RTOS can guarantee constant execution time for the memory management functions. Table 1 classifies few real-time operating systems according to which use the fixed-sized block allocation and which support heap allocation.

Table 1: Dynamic Memory Management in Some Real-Time Operating Systems.

RTOS	**Fixed-Sized Blocks Allocation**	**Heap Allocation**
uCOS-II	Y	N
VRTXsa	Y	Y (mixture of sequential first fit and simple segregated storage)*
eCos	Y	N (but supports variable size block allocation with optional coalescing)*
VxWorks	N	Y (sequential first fit)
Atalanta	Y	N

* Not recommended for critical tasks

2.6 Hardware Memory Allocators

To reduce the execution time of dynamic memory management routines (allocation, deallocation and garbage collection) and/or make their execution times deterministic, many researchers have proposed hardware accelerators for dynamic memory management.

The literature shows several hardware implementation of memory allocators [8, 10, 11, 15, 29, 40, 41]. Those hardware memory allocators implement modified versions of the buddy allocation algorithm. To store the heap allocation status, they use a bitmap vector. Each bit in that bitmap vector presents a part of the heap. Thus, the heap is divided into fixed number of allocation units called memory blocks. The hardware allocator searches this bitmap vector to find free memory blocks to satisfy the requested allocation.

2.6.1 Early Hardware Allocators

The hardware implementation of a simple buddy allocator was first proposed by Knowlton [8, 17]. It is a simple and fast buddy allocator that can allocate memory blocks each of whose size is a power of two; hence, the allocator suffers from internal and external fragmentation [8, 29].

Puttkamer introduced a hardware buddy allocator that does not suffer from internal fragmentation [10, 29]; however, the allocator does not have a constant execution time. The allocator utilizes a shift register to store information regarding the allocation status (one bit per memory block). To find free memory blocks to satisfy the requested allocation the allocator shifts the memory bitmap vector one bit at a time until it finds free memory blocks that satisfies the allocation request.

2.6.2 Chang's Hardware Buddy Allocator

Chang and Gehringer propose a modified hardware-based buddy system that eliminates internal fragmentation and has a constant execution time [10].

Chang et al. have implemented the *malloc()*, *realloc()* and *free()* C-Language functions in hardware [40]. Also, they propose a hardware extension to be a part of future microprocessors to accelerate dynamic memory management [11]. The hardware *malloc()* implements a modified binary buddy allocator algorithm. To allocate memory, the hardware allocator first finds a memory space large enough for the allocation request using an *or-gate* tree (example shown in Figure 3). Then the hardware allocator marks this memory space to indicate that it is allocated.

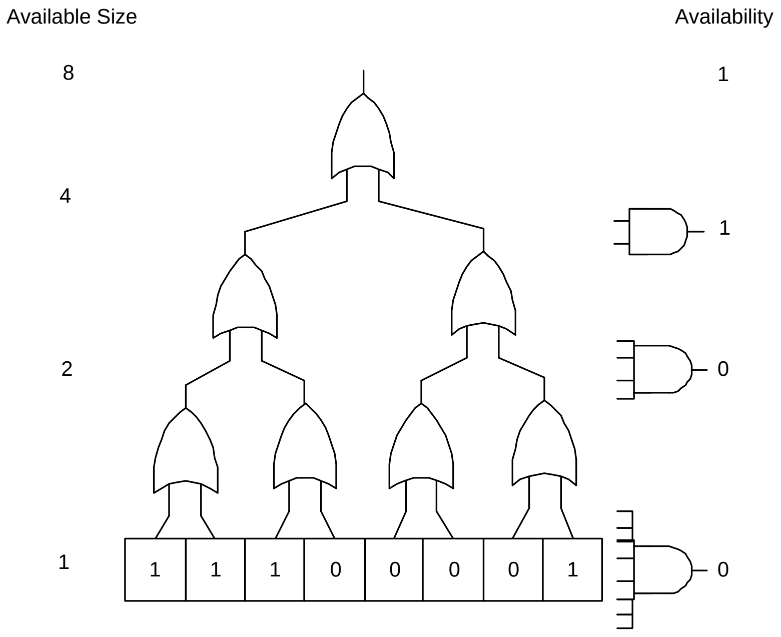


Figure 3: Chang's allocator *or-gate* tree.

Although Chang's modified binary buddy allocator eliminates internal fragmentation, the allocator can detect only a free memory block chunk if it starts at an address that is power of two. This problem is called the blind spot problem and may prevent the allocator from satisfying allocation requests even though there are free memory blocks (these free memory blocks start at an address which is not a power of two) that can satisfy the request.

Example 1 Figure 3 shows a part of Chang's allocator; specifically, the bitmap vector used to store the heap allocation status and the *or-gate* tree used to determine the size of the available free space. Each bit of the bitmap vector determines the allocation status of each allocation unit in the heap (usually a word). The bitmap vector in Figure 3 has a size of 8 (which is unrealistic but used in this example for simplicity). Assume that initially allocation units 1, 2, 3 and 4 are free indicated by zeros at bits 1,2,3 and 4 (the bits are numbered from right to left starting with 0). The *and* gates at the right side of the *or-gate* tree are used to indicate the existence of free memory blocks of certain sizes. Those sizes are marked using the numbers at the left hand side of the *or-gate* tree. When the output from the *and* gate is zero that means the corresponding size is available. For instance, the *and* gates outputs indicate that free memory areas of 1 block and 2 blocks exist. The *or-gate* tree cannot recognize the four free memory blocks 1, 2, 3 and 4 because the four free blocks start at bit position 1 which is not a power of 2. Hence, a memory allocation request of allocation size 3 or 4 cannot be satisfied by this approach; however, there are free 4 memory blocks available for allocation. □

2.6.3 Cam's Hardware Buddy Allocator

To overcome the blind spot problem, Cam et al. proposed a hardware buddy allocator that detects any available free block of requested size and eliminates internal fragmentation [8].

In order to be able to find free memory blocks that start at locations not a power of two, Cam modified Chang's allocator *or-gate* tree to the one in Figure 4. Cam's *or-gate* can detect free memory even if it does not start at memory block number that is a power of two.

Although Cam's allocator does not have the *blind spot* problem, it rejects some memory allocation requests for which there exists enough free memory to satisfy the request. Specifically, to search for available memory to satisfy a request, Cam's allocator has to round-up the requested size into a number which is a power of two; thus,

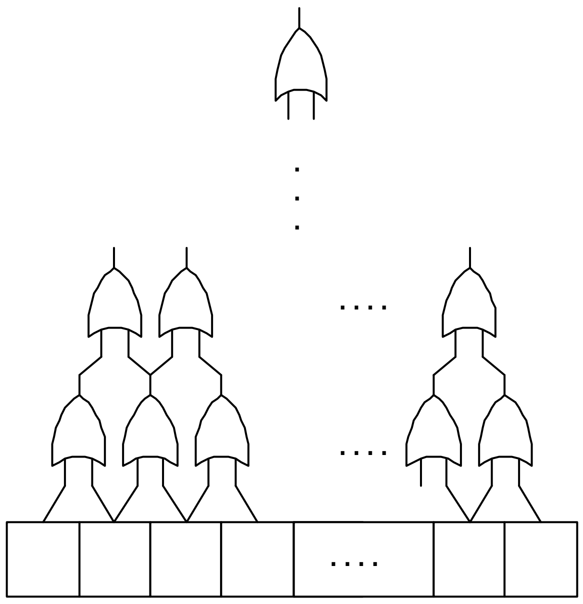


Figure 4: Cam's *or-gate* tree.

it is possible that some requests might not be satisfied even though enough memory blocks exist to satisfy the request. Also, Cam's allocator has much larger area than Chang's allocator. For example, for 8-bit bitmap vector, Chang's *or-gate* tree has 7 *or* gates, while Cam's *or-gate* tree has 28 *or* gates.

2.6.4 Summary

As explained in the previous subsections, all earlier research known to the author focuses only on the hardware implementation of a buddy allocator for uniprocessor systems. None discuss in detail how these functionalities could be integrated into a system (except Chang's work) or present any system examples. Moreover, the use of these hardware allocators for multiprocessor systems has not been addressed.

Although some of the introduced hardware implementations of memory allocators show good performance in terms of execution time (Chang's and Cam's allocators), all of them suffer from one serious problem: the allocators utilize a bitmap vector to store the memory allocation status which means that the memory should be divided into fixed-size blocks. If most of the dynamic memory allocations done by applications are for small objects, then, for the previous approaches, the granularity of the memory blocks needs to be increased (i.e., memory block size needs to be decreased) by increasing the size of the bitmap vector; however, this increases the hardware complexity and execution time of the allocator.

Example 2 A typical 32-bit RISC processor has an address space of 2^{32} bytes (4GB). If dynamic memory allocation is handled by a hardware buddy allocator that has a 1024-bit bitmap vector (which can be considered large), the size a memory block assuming equally sized blocks would by $2^{32}/2^{10} = 2^{22}$ (4MB!). Recalling that most memory allocations are for small objects ($<$ 1KBytes) for object oriented applications, then significant amounts of memory would be wasted (e.g., the request of an allocation of a 32 byte object allocates 4MBytes). Increasing the granularity of the memory blocks by decreasing the block size to 1KByte (hence reducing the memory wasted) would require a bitmap vector of size $2^{32}/2^{10}= 2^{22}$ bits! $\square$

It is obvious, from Example 2, that hardware implementations of dynamic memory allocators that are based on the bitmap vector approach described above are not suitable for use with a uni-processor system with a large (typically 4GB) address space and applications requiring mostly small (e.g., 1KB or less) dynamic memory allocations. Clearly, then, the previous work is even less applicable to typical multiprocessor SoC systems with similar characteristics (a large amount of addressable memory and many small dynamic allocations).

2.7 Summary

In this chapter we gave an overview of static and dynamic memory management. We discussed in detail different methods of dynamic memory allocation. One way to manage memory and dynamically allocate it is by using hardware allocators. Although prior hardware allocators are fast, they do not scale well.

In the following chapters we will introduce a completely new approach for hardware dynamic memory management. Chapter 3 will introduce a new hierarchy for dynamic memory management for heterogeneous multiprocessor SoC. Then, the following chapter, Chapter 4, will discuss the implementation of that new memory management hierarchy.

CHAPTER III

TWO-LEVEL MEMORY MANAGEMENT FOR MULTIPROCESSOR SOC

Multiprocessor SoCs with multiple PEs and large global on-chip memory are suitable for mobile multimedia applications and hardware (e.g., a multimedia Personal Digital Assistant). Such applications require great processing power and large data. Also, these applications can demand a lot of memory. For a mobile device that is built around a multiprocessor SoC running memory demanding applications able to be downloaded wirelessly, dynamic memory management is very desirable. For example, if quick download, installation and memory allocation primitives are available on a handheld PDA, then thousands of applications available at a wireless server could service the PDA, resulting in an appearance of thousands of available applications with only, say, ten of the applications actually downloaded and installed at any point in time. Thus, dealing with the large global on-chip memory in multiprocessor SoC in a dynamic yet deterministic way is an important issue.

In this chapter we introduce a new memory management scheme suitable for multiprocessor SoC. Although the new memory management scheme provides a way to dynamically allocate on-chip memory among the on-chip PEs, our new memory management scheme keeps dynamic memory management fast and deterministic.

3.1 Problem Statement

Although dynamic memory management of the on-chip memory in multiprocessor SoC is a desirable option, current embedded designers often avoid dynamic memory management in favor of determinism. In this thesis we try to answer the question: "How can global on-chip memory in the multiprocessor SoC be allocated among the on-chip processing elements in a dynamic yet deterministic way?" If our answer is successful, our approach could signal a significant shift in the way embedded software designers code up their real-time applications.

3.2 Two-Level Memory Management

We propose a programming model and memory management scheme that we call Two-Level Memory Management for multiprocessor SoC on-chip memory management. Two-Level Memory Management assumes that the allocation of the global on-chip memory (Level Two) is handled separately (e.g., by the SoCDMMU), while each PE handles local dynamic memory allocation among the threads/processes running on the PE (Level One). Typically, if a process requests a memory allocation the process will request the memory from the RTOS. In our system, if the current PE has enough extra global memory to satisfy the request, the RTOS will simply allocate the memory right away. This allocation by the RTOS of memory already allocated to the PE is what we refer to as Level One memory management. Otherwise, if the PE does not have enough global memory already allocated to satisfy the request, the RTOS will request more memory from the SoCDMMU. This is referred to as Level Two memory management. So there are two levels of memory allocation and management: the process/thread level managed by the RTOS (local allocation, Level One) and the PE level (global allocation, Level Two) managed by the SoCDMMU.

3.3 Assumptions

Before going further through the rest of the thesis we will first state some assumptions on which we base our approach:

- The application running on the SoC fits (instruction and data) in the global on-chip memory. Note that data may be streaming through the global memory, e.g., as would occur in a video application.

- The global memory is divided into a fixed number of equally sized blocks (for example, 16KB or 64KB), each of which is called a global memory block or a *G_block*.

- Global memory allocation done by the SoCDMMU is referred to as *G_allocation*; global memory deallocation done by the SoCDMMU is referred to as *G_deallocation.*

- A *page* consists of one or more *G_block*s.

- Each page has a unique identifier we call a *Page ID*. The *Page ID* is assigned by the PE that allocates the page. The *Page ID* is used to reference a particular page to perform a certain operation (e.g., move or deallocation).
- Each *G_block* has one physical address and one or more virtual addresses. The base virtual address (i.e., the virtual address of the very first word in the *G_block*) a particular PE assigns to a *G_block* may differ from one PE to another.
- A *G_block* can be moved (by changing its virtual address) from one place to another in the PE address space at runtime.
- A *G_block*'s base virtual address will be referred to as a PE-address.
- Each PE may request dynamic *G_allocation* or *G_deallocation* of a page.
- Multiple PEs may issue *G_allocation* and/or *G_deallocation* commands simultaneously.
- The PEs can *G_allocate* three types of memory pages:
  - ***Exclusive Memory***: Only the owner (the PE that *G_allocates* the memory) can access the *G_allocated* memory. No other PE can access the page. We will refer to the command that *G_allocates* this type of page as *G_alloc_ex*.
  - ***Read/Write***: The PE that *G_allocates* the page can read from or write to the global memory. Another PE can read from the page if that PE

G_allocates the page as *read only*. No other PE may *G_allocate* the same memory as *Read/Write* or as *Exclusive* Memory. This limitation significantly reduces cache coherency problems. We will refer to the command that *G_allocates* this type of page as *G_alloc_rw*.

– ***Read Only***: The PE can read from (but not write to) the page. The user may choose to require that the page be previously *G_allocated* as *read/write* by a different PE. This choice is set by a control register in the SoCDMMU. We will refer to the command that *G_allocates* this type of page as *G_alloc_ro.*

We deal with the cache coherency problem that may exist in the pages that are *G_allocated* as *read/write* using a simple cache invalidation scheme. Note that since only one processor may allocate any page as *read/write* (with other processors only allowed to allocate the page as *read only*), the only time a cache line needs to be invalidated is when a PE writes *read/write* memory that other PEs – which have allocated the memory as *read only* – have cached. In this case, the write through from the PE with *read/write* will cause the cache lines in the *read only* PEs to be invalidated.

However the proposed programming model restricts the shared memory to the case where the shared memory page has only one writer and multiple readers, the SoCDMMU does not prevent writes to the same memory page by multiple processing elements. Hence, the SoCDMMU can work with systems that use snoopy caches and

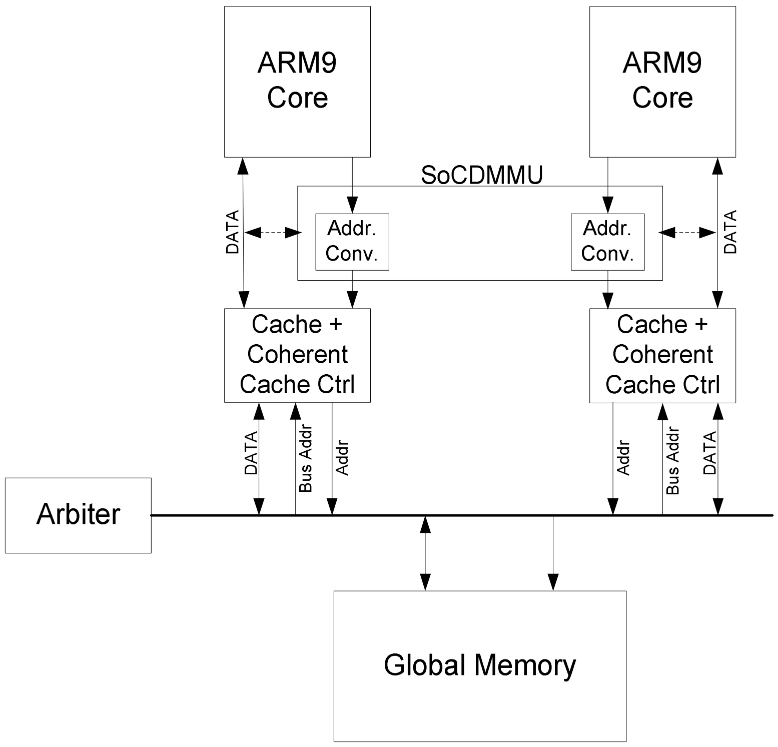


Figure 5: Example of two-PE SoC with coherent caches and the SoCDMMU.

standard cache coherency protocols (e.g., MSI and MESI).

Example 3 Figure 5 shows in detail how the SoCDMMU can be connected to a system that uses snoopy caches and a cache coherency protocol. The system shown in Figure 5 consists of two PEs, memory, caches and coherent cache controllers. The SoCDMMU is a connected in such a way that its address converters perform the virtual to physical address conversion of the address lines going to the caches. Since the address conversion takes place before the caches, the caches are considered as physical caches. Each cache controller snoops into the bus to monitor the other cache bus transactions to keep its contents coherent. □

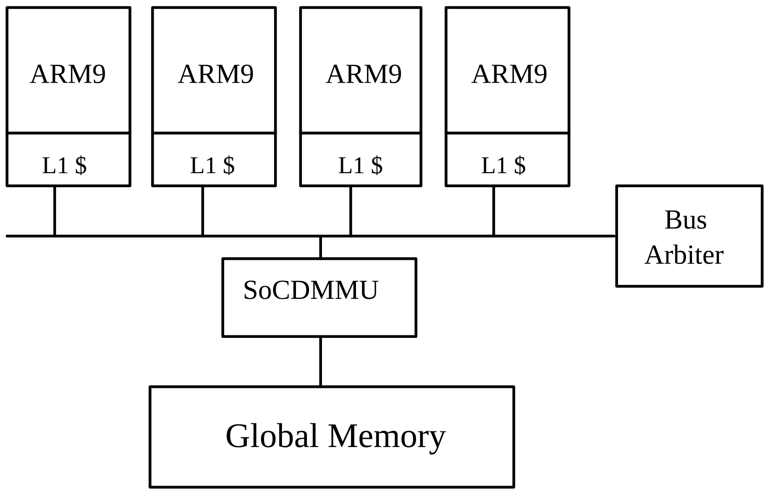


Figure 6: Example of four-PE SoC with the SoCDMMU.

3.4 The Virtual Memory Model

The memory buses of the PEs are connected to the SoCDMMU to allow the SoCDMMU to control all of the global memory accesses as shown in Figure 6. This enables the SoCDMMU to convert the PE-address (virtual address) to a physical address. The PE can map any allocated *G_block* to any memory location inside the PE's address space. This feature allows the allocation of non-contiguous physical memory *G_block*s and the mapping of them into contiguous memory locations in the PE's address space; thus, there is no need for memory compaction of the *G_block*s (memory compaction may be an issue within a particular *G_block*).

Example 4 Consider the SoC in Figure 6 that has four ARM processors, global on-chip memory of 16MBytes and an SoCDMMU. The global on-chip memory is divided into 256 blocks each of which is 64KB. Figure 7 shows the mapping of six memory blocks in PE address space

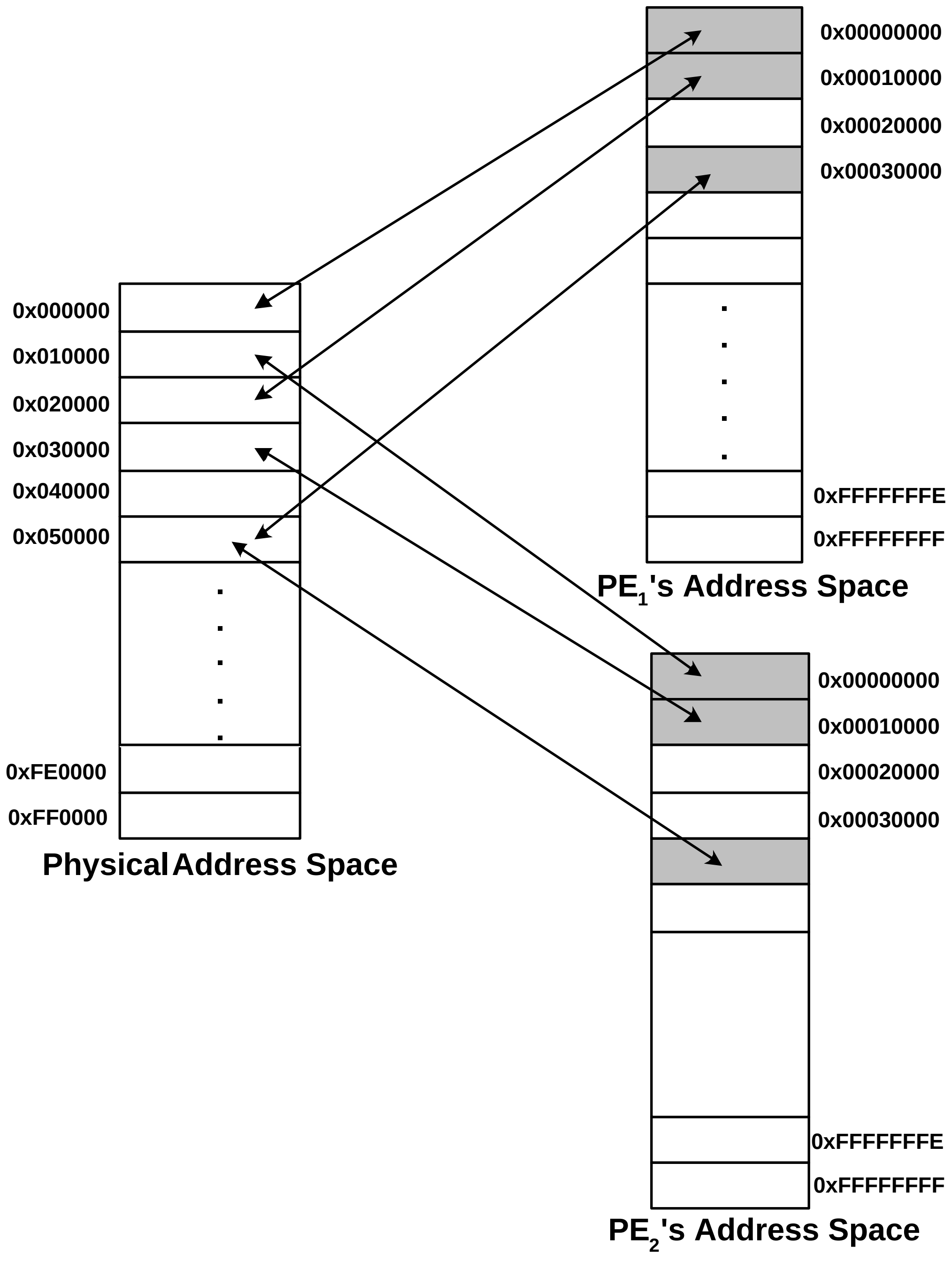


Figure 7: Mapping of physical memory *G_blocks* to PE address spaces.

into five physical memory *G_block*s. PE1 allocated one page of two *G_block*s starting at address 0x00000000 in its address space. However, this page is mapped into two non-contiguous physical memory *G_block*s starting at 0x000000 and 0x020000 respectively. Also, the page that consists of one physical memory *G_block* that starts at physical address 0x050000 is shared between PE1 and PE2; PE1 maps the *G_block* to the location 0x00030000 in PE1's address space while PE2 maps the *G_block* into a different address (0x00040000) in PE2's address space. □

Please note that the Two-Level memory management hierarchy is a way to handle the large on-chip memory. Two-Level memory management hierarchy identifies two levels on which on-chip memory can be allocated. Level Two is the global on-chip level where portions of the large on-chip memory are assigned to PEs (typically, portions of cache memory cannot be assigned to any PE). Level One is the OS level where memory assigned to any PE is assigned to the tasks that run on that PE. Our Two-Level memory management hierarchy is a completely different concept than two level caching or the three level caching hierarchies. These caches hierarchies are used mainly to speed up memory access by caching frequently accessed main memory portions at different levels. Hence, Two-Level memory management hierarchy and the SoCDMMU can work with SoCs that have any number of caching levels as it does not deal or interfere with the cache operation (other than virtual to physical address conversion).

3.5 Intuitive Advance: Centralize Multiprocessor Memory Management Decisions

Intuitively, a shared memory heterogeneous multiprocessor typically implements a distributed memory allocation paradigm. This paradigm involves software on each processor, cache coherent hardware controllers, and shared data structures. What we propose here is to have a centralized unit, the SoCDMMU, handle all requests and grants of blocks of the large on-chip memory. Thus, we have covered so far in this thesis an overview of previous work and an explanation of our proposed two-level memory management model. The way a PE handles memory allocated to it is well known; thus, how will the allocation of the large on-chip memory be handled? In the next chapter (Chapter 4), we will explain in more detail how the proposed SoCDMMU operates.

CHAPTER IV

THE SOCDMMU

In this chapter, we will first give a detailed overview of the hardware implementation of the SoCDMMU. The SoCDMMU can be implemented as hardwired logic or as software that runs on a general purpose processor. The SoCDMMU implementation that involves a general purpose processor and software implementation of the allocation algorithm is flexible; however, it is over 10X slower when compared to the SoCDMMU implementated as hardwired logic. In Chapter 7, we show a speed comparison of both implementations. Although this chapter focuses on a hardwired implementation of the SoCDMMU, we will show how parts of the SoCDMMU can be implemented as software that runs on general purpose processors.

4.1 SoCDMMU Interface

The SoCDMMU command interface is mapped into *one* location in the I/O space of the PE (typically, a single address accessing 32 bits of data when an SoCDMMU command fits into 32 bits – see Section 4.1.1 for details about the SoCDMMU commands).

This memory mapped address or I/O port to which the SoCDMMU command register and status register are 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).

4.1.1 The SoCDMMU Commands

There are three types of commands that the SoCDMMU can execute:

- ***G_Allocate* Commands**

  Figure 8 shows the command word format for the *G_allocate* commands. There are three types of *G_allocate* commands: *G_alloc_ex*, *G_alloc_rw* and *G_alloc_ro*. The right-most field of the command word specifies the command type. The next field specifies the PE address to which the PE wants to map the requested *G_block*s. The third field specifies the allocation size in *G_blocks*. Finally, the fourth field specifies the software assigned ID of that page (*Page ID* – see Section 3.2).

The memory *page*'s *G_blocks* are contiguous in the PE address space; hence, only the address of the first *G_block* is required to be specified to identify a memory *page* that consists of multiple *G_block*s. Note that, if the user desires the allocation of two non-contiguous memory *G_block*s, two *G_allocation* commands must be issued, one for each *G_block*.

- ***G_deallocate* Command**

G_alloc_ex	**Page ID**	**Size**	**Virtual Block No.**	**000**
G_alloc_rw	**Page ID**	**Size**	**Virtual Block No.**	**001**
G_alloc_ro	**Page ID**	**n/a**	**Virtual Block No.**	**010**
G_dealloc	**Page ID**	**n/a**	**n/a**	**011**
G_Move	**Page ID**	**n/a**	**Virtual Block No.**	**100**

Figure 8: The SoCDMMU commands.

Figure 8 shows the format of the *G_deallocate* command word. The *G_deallocate* command needs to know the *Page ID* to be deallocated.

- ***G_move* Command**

Figure 8 shows the format of the *G_move* command. The *G_move* command is used to re-map an allocated memory page to another location in the PE-address space (virtual address). The command needs two parameters: the first parameter specifies the new PE address to be assigned to the page (virtual block number); the second parameter specifies the *Page ID* of the page to be moved. Thus, the *G_move* command allows PE address space compaction (see Example 5).

Compaction of the PE's address space is required when the PE address space is highly fragmented and there is no available contiguous set of virtual memory

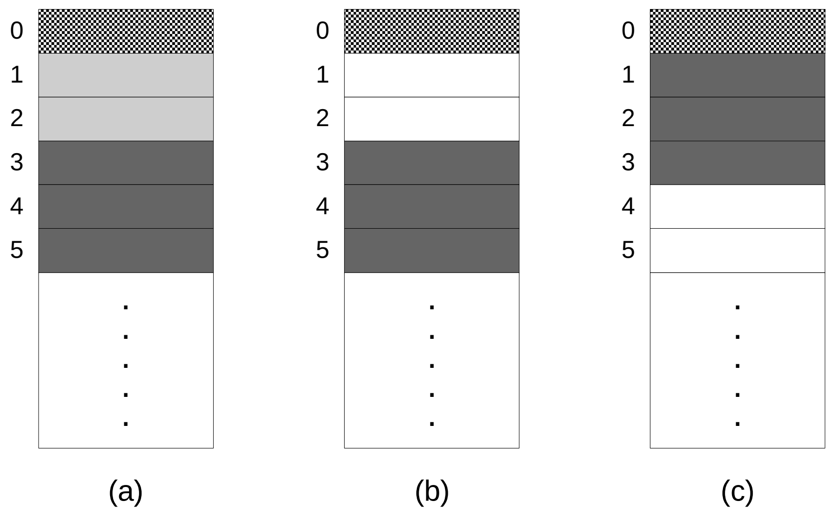
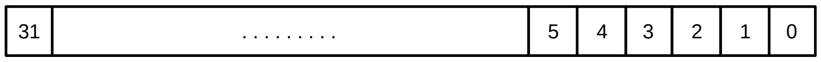


Figure 9: An example use of the *G_move* command.

blocks to which requested – and allocated – physical memory *G_block*s can be mapped.

Example 5 To explain the idea of PE-address space compaction, consider the first three allocated pages as shown in Figure 9 (a). Figure 9 (a) shows the locations of three physical memory pages in the PE-Address space. For example the first page consists of one memory block and is mapped to block number 0 in the PE-Address space. If the second page is deallocated as in Figure 9 (b), we end up with a hole in the PE-Address space that may not be suitable for future allocations. We can use the move command to move the third page so that it starts at block number 1, as shown in Figure 9 (c), instead of block number 3 which is where the start of the third page was located in Figure 9 (a) and Figure 9 (b). Please note that the actual physical memory has not been moved or changed in any way; only the PE address (which is virtual and thus which the SoCDMMU translates into a physical address upon access) has been moved in



Bit 0: 1-Not enough memory. 0-No error.
Bit 1: 1-Page was not allocated. 0-No error.
Bit 2: 1-Not an owner. 0-No error.
Bit 3: 1-Page not found (move). 0-No error.
Bit 4: 1-Page not found (deallocation). 0-No error.
Bit 31: 1-Invalid Status Word.

Figure 10: The SoCDMMU status register error codes.

the virtual address space. □

The SoCDMMU *Control Unit* has a configuration register that affects the SoC-DMMU operation. One of the configuration register bits affects the execution of the *G_alloc_ro* coomand. When this bit is cleared to 0, a PE can allocate *read only* memory without being previously allocated as *read/write* by another PE. If this particular bit is set to 1, a PE cannot allocate memory as *read only* unless it is allocated as *read/write* by another PE and an error will be generated. This bit is set to 1 on reset.

4.1.2 The SoCDMMU Error Codes

The PE has to read the status word to determine the status of the last command sent to the SoCDMMU. Figure 10 shows the status register format and the possible error codes. Please note that a status register of a value '0' indicates a successful execution of the previously issued command. Bit 31 in the status register indicates whether the status register was prematurely read. The possible errors that can be detected at the SoCDMMU level in a command received are as follows:

- Not enough memory to *G_allocate* (*G_allocation* error).

- Trying to *G_allocate_ro* a memory page that was not allocated previously as *read only* (if the user decides to enable this error) (*G_allocation* error).
- Trying to *G_deallocate* a non-owned memory page (*G_deallocation* error).
- Trying to move a non-existing memory *G_block* (*G_move* error).
- Trying to *G_deallocate* a non-existing memory page (*G_deallocation* error).

4.2 The SoCDMMU Hardware

In this section, we will describe the implementation of the SoCDMMU as a hardwired unit. In Section 4.4, we will show a different way to implement the SoCDMMU to provide programmability and flexibility.

Figure 11 shows the structure of the SoCDMMU. The block labeled "BASIC SoCDMMU" can handle only one request (allocation/deallocation) at a time. Multiple requests are handled by having multiple command and status registers. Each PE has its own command register and status register.

Each PE writes its command into its associated command register. When multiple commands are received simultaneously, the *Request Scheduler* determines which command will be executed on the BASIC SoCDMMU according to a priority-scheduling algorithm implemented in the *Request Scheduler*, where priorities are dynamically assigned to ensure that, among commands received in the same clock cycle, the *G_deallocate* commands are always executed first (note that commands received in

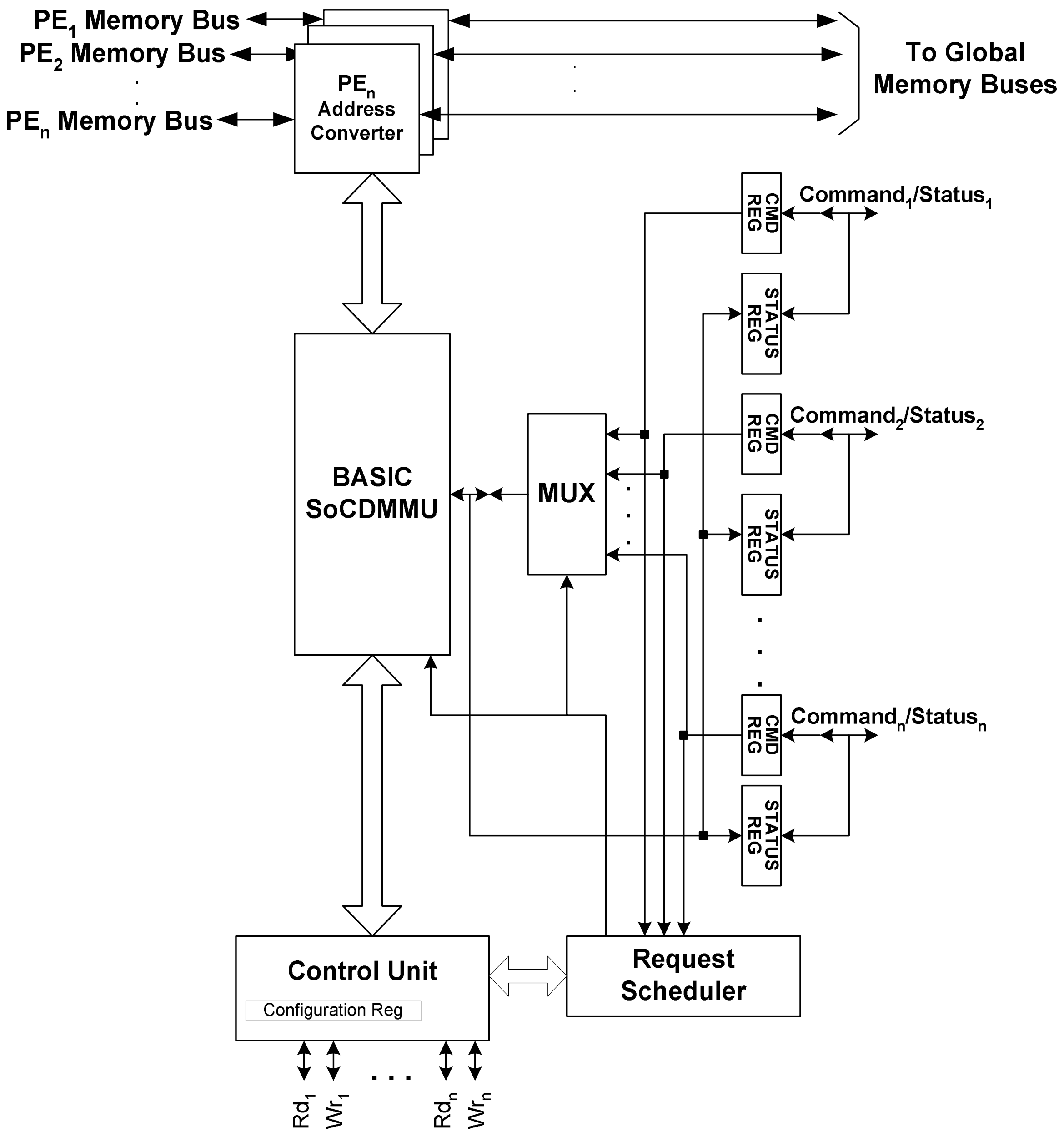


Figure 11: The SoCDMMU architecture.

earlier clock cycles are always executed before commands received in subsequent clock cycles).

4.2.1 The Address Converter

The *Address Converter* is used to convert a PE address (virtual address) into a physical memory address. Each PE has its own *Address Converter*. The *Address Converter* is an associative memory array (similar to the memory used to store the tags in set-associative cache) where every entry corresponds to a particular *G_block*. For each memory *G_block*, the *Address Converter* stores the *G_block* base address in the PE address space. Specifically, the first entry stores the *G_block* base address in PE address space of the first physical *G_block*, the second entry stores the *G_block* base address in PE address space of the second physical *G_block* and so on. Also, there is a valid bit associated with every *Address Converter* entry to indicate whether the entry contains a valid *G_block* base address in PE address space or not.

Example 6 Consider the SoC of Example 4 that utilizes the SoCDMMU with n=256 *G_block*s each of which is 64KB. In this example, "Offset" is 16 bits long, "Virtual Block no." is 16 bits long and "Physical *G_block* no." is 8 bits long. As we can see in Figure 12, the PE base address of the first *G_block* (*G_block* zero) is 0x00FA and that of the third *G_block* (*G_block* two) is 0x0A3F. Thus, for example, virtual address 0x00FA9864 would be translated to physical address 0x00009864, and virtual address 0x0A3FFE68 would be translated to 0x0002FE68. □

The information stored in the *Address Converter* is updated by the Basic SoC-DMMU during the execution of the *G_allocate* and *G_move* commands. A new record

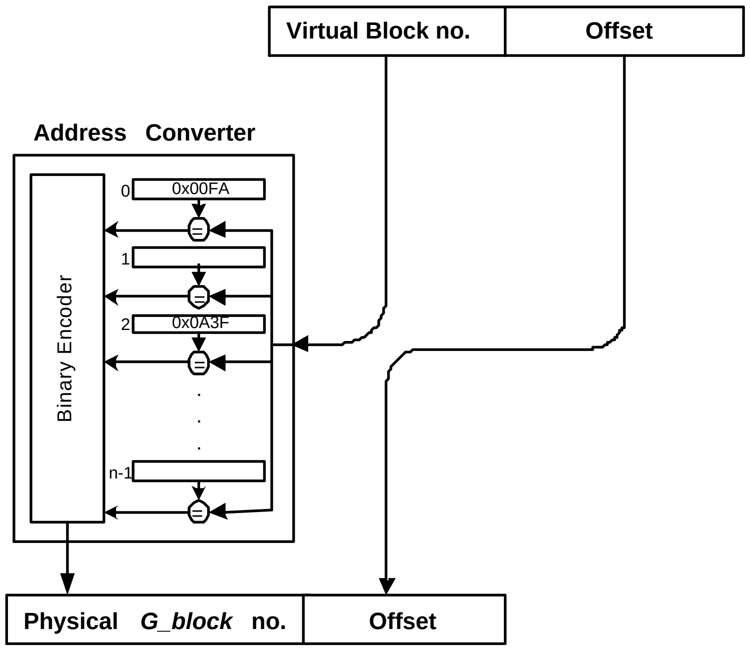


Figure 12: PE address to physical address conversion.

(the PE base address of a *G_block* plus a valid bit indicating if the entry is valid or not) is inserted into the address converter when a *G_allocate* command is successfully executed. One or more records might be updated when a *G_move* command is successfully executed.

PE address to physical address conversion is required and can make allocated non-contiguous (in physical address space) memory *G_block*s appear as contiguous memory blocks to the PE address space. This avoids, at the *G_block* level, memory fragmentation in the physical address space; hence, no *G_block* level physical memory compaction is needed. Virtual to physical address conversion is done as shown in Figure 12. The virtual block number part of the PE address is compared to all of the valid (with valid bit set to one) *Address Converter* entries in parallel. The output

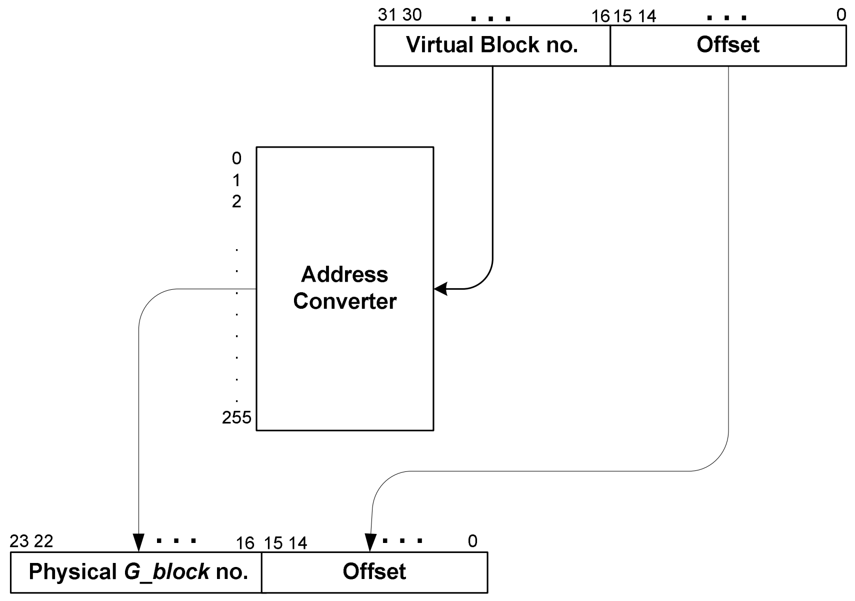


Figure 13: The *Address Converter* for Example 7.

of the comparator that corresponds to the matched *Address Converter* entry is set to one while the outputs of the other comparators are cleared to zero. The outputs from all of the comparators go to the binary encoder. The binary encoder converts the outputs of the comparators into a binary number that equals to the index of the matched entry (the physical *G_block* number). The physical *G_block* number is used along with the block-offset part of the PE address to construct the physical address.

Example 7 Consider the SoC of Example 4. Since each *G_block* is 64KB, each "Offset" field shown in Figure 12 is sixteen bits long; thus, since a 32-bit address (the ARM9TDMI has a 4GB address space) is used, the "Virtual Block no." at the top of Figure 13 is (32-16)=(log_2*(4GB)*-log_2*(64KB))*=16 bits long. Furthermore, since there are 256 (=2^8) *G_block*s, the "Physical *G_block* no." field in the bottom left of Figure 13 is eight bits long. Now, the SoCDMMU of Examples 2 and 4 uses four *Address Converter*s each of which is an associative memory of

256 (=2^8) words. The first entry in the associative memory – entry zero – holds the 16-bit Virtual Block number in PE address space which is mapped to *G_block* zero, i.e., the *G_block* with Physical *G_block* number zero. The second entry in the *Address Converter* – entry '1' in Figure 13 as indicated on the left-hand side of the *Address Converter* block in the center of Figure 11 – holds the 16-bit Virtual Block number in PE address space which is mapped to *G_block* 1, i.e., the *G_block* with Physical *G_block* number '1'. This continues up to the last entry – entry '255' in Figure 13 – which holds the 16-bit Virtual Block number in PE address space which is mapped to *G_block* 255. As shown in Figure 13, the virtual block number (the *G_block* base address in PE address space) is used to lookup the *G_block* physical base address (which is of size log_2*(number of G_blocks)*=8 bits). Finally, the physical address is formed by concatenating the offset part of the PE address with the *G_block* physical base address obtained from the *Address Converter*.

It may seem to the reader that using associative memory for the *Address Converter* is expensive in terms of the area of the *Address Converter* (the *Address Converter* has the same area as 1.22KB 6T-SRAM – see Chapter 5, Section 5.3); however, the alternative implementations are more expensive either in area or speed. The *Address* Converter could be implemented as a lookup table using SRAM where the virtual block number part of the address is used as an index; such implementation would require 64KB SRAM which is more than 50 times larger in area than our implementation. Also, the *Address Converter* could be implemented as a sequential search lookup table which has less area but needs many more cycles to find a match. □

Although the *Address Converter* (in our current implementation) performs a virtual to physical address translation, it does not currently replace all the functionality of a Memory Management Unit (MMU) that can be found in modern processors. While the SoCDMMU's *Address Converter* does provide an address translation mechanism, an SoCDMMU does not provide memory protection or page-level access control provided by an MMU (however, those functionalities can be implemented into the SoCDMMU). Also, note that if the SoCDMMU is used with a microprocessor that utilizes an MMU, the Translation Look-aside Buffer (TLB) found in the processor MMU cannot replace the Address Converter (the SoCDMMU needs full control over the TLB – if we want to use the TLB instead of the *Address Converter* – but such full control is not usually provided by microprocessor core vendors). Please note that the SoC systems we simulated that utilize the SoCDMMU have microprocessors that either do not have MMUs at all (e.g., ARM9TDMI) or provide the ability to disable the MMU (which we have done for the MPC750 and MPC755).

4.2.2 The Basic SoCDMMU Architecture

Figure 13 shows the structure of the BASIC SoCDMMU. This unit performs *G_allocation* and *G_deallocation* commands. The *Allocation Vector* is a bit vector where the number of bits equals the total number of *G_block*s in the global memory. Each bit of the *Allocation Vector* shows if the corresponding *G_block* in the global memory is *G_allocated* or not. Thus, the SoCDMMU uses the *Allocation Vector* to keep track of the allocation status of the physical memory *G_block*s.

The BASIC SoCDMMU also stores information about the *G_allocated* *G_block*s

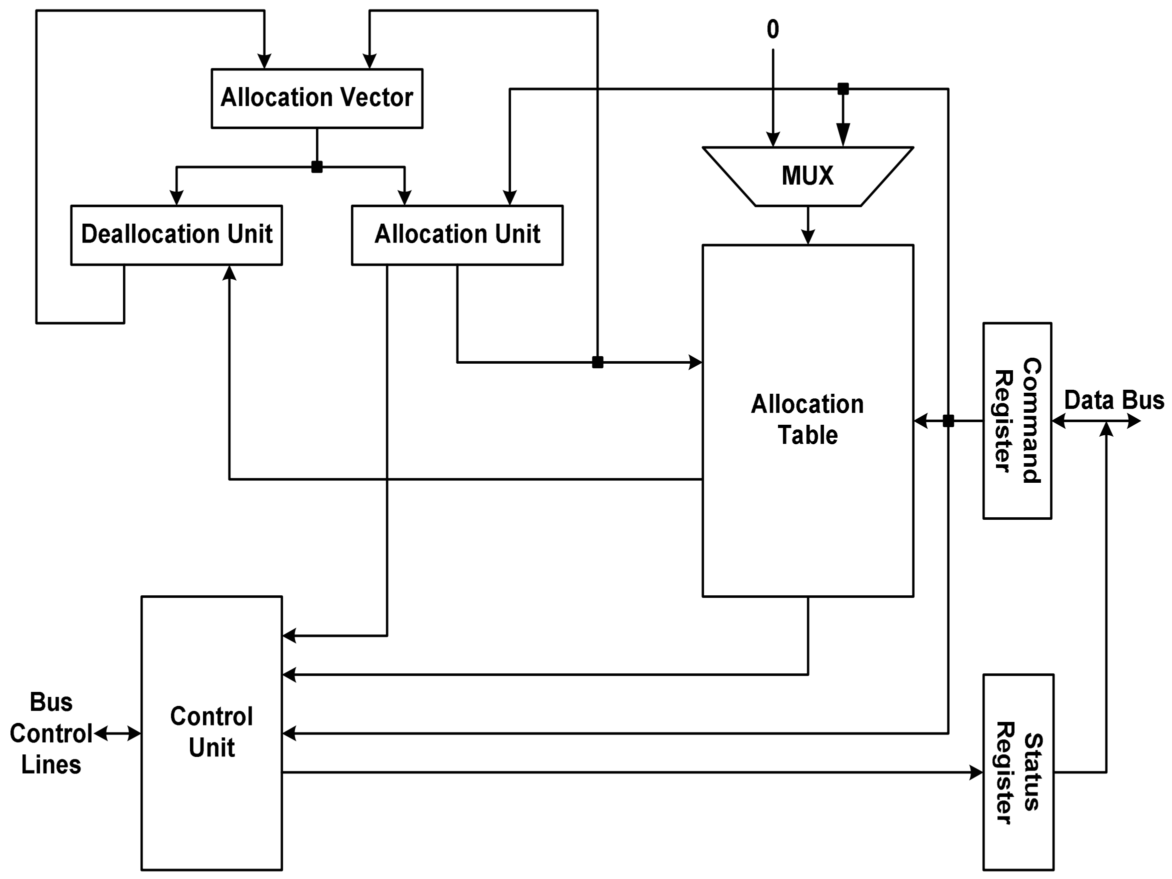


Figure 14: The Basic SoCDMMU architecture.

using the *Allocation Table*. The *Allocation Table* has a register file with number of entries equal to the total number of *G_block*s. Each entry corresponds to a particular *G_block* and contains the following information about that particular *G_block* (as shown in Figure 14): the allocation mode (exclusive, read only or read/write), the PE that allocated the block and the *Page ID* given to the *G_block*. Please note that the *Page ID* field width depends on the number of Global on-chip *G_block*s: specifically, *Page ID* field width is equal to log_2*(number of G_blocks)*.

If a wider range of *ID*s is required at the programming language level, the compiler can automatically generate an *ID* that fits into the *Page ID* field from the programmer specified *Page ID*.

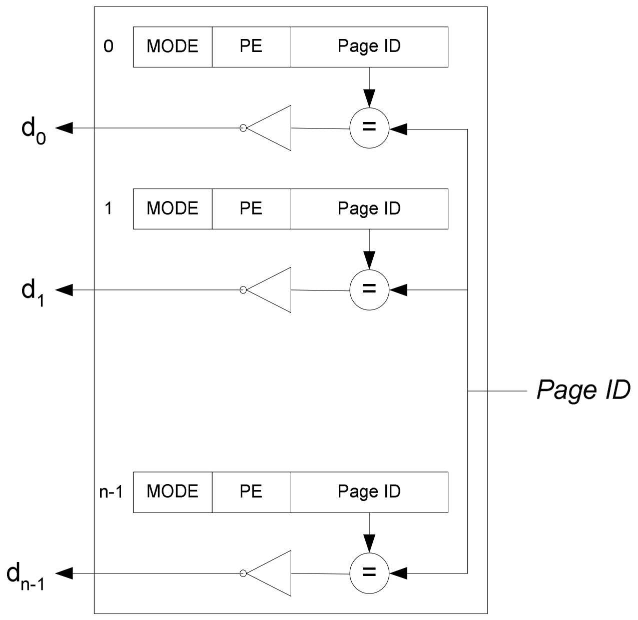


Figure 15: The *Allocation Table* bit vector output.

The *Allocation Table* has a bit vector output to identify the *G_block*s of any

MODE	PE	Page ID

Figure 16: The *Allocation Table* entry format.

memory *page* given the *Page ID* of that *page* as shown in Figure 15. This output has 0s in the bit positions that correspond to the *page G_block*s and 1s in other bits. This bit vector output is used during the *G_deallocation* process as explained later.

Example 8 Consider the SoC of Figure 1 that utilizes the SoCDMMU. The system consists of four PEs and a global on-chip memory of 1MB. The global memory is divided into 8 *G_block*s (this is an unrealistic number but we use it to simplify the example; a typical number is 256) each of which has a size of 128KB. Assume that initially no memory *G_block* was allocated to any PE. Figure 15 shows the *Allocation Table* after allocating the first three memory *G_block*s to PE1 for exclusive use (three records are inserted into the *Allocation Table*; each record corresponds to one of the three *G_block*s that form the page). This three *G_block* page is given the *Page ID* 5 by the PE that allocated it (PE1). Please note that the *G_block*s that form a page are given the same *Page ID*. Also note that the PE field width is 2 bits (the SoC has only four PEs); while the *Page ID* field width is 3 bits (the physical memory is divided into eight memory *G_block*s). □

The *G_allocation* (*G_alloc_ex*, *G_alloc_ro* and *G_alloc_rw*) process is performed using the *Allocation Unit*. The *Allocation Unit* inputs are the page size (number of *G_block*s) and the information stored in the *Allocation Vector*. Using this information, the *Allocation Unit* allocates the requested page using a first fit algorithm. The output

MODE (2 bits)	PE (2 bits)	Page ID (3 bits)
EX	PE1	5
EX	PE1	5
EX	PE1	5
	⋮	

Figure 17: *Allocation Table* example.

from the *Allocation Unit* is used to update the *Allocation Vector* and insert records in the *Allocation Table*. Also, output from the *Allocation Unit* is used to update the information stored in the *Address Converter* look up table. If there are any errors, then the appropriate error code will be written to the status register and no update will be performed to the *Allocation Table*, *Allocation Vector* or the *Address Converter*.

Please note that if the user selects the option that a PE needs to allocate a *page* as *read/write* before other PE can allocate it as *read only*, the *Allocation Unit* is not used during the execution of *G_alloc_ro*. In this case, the execution of the *G_alloc_ro* command involves only the retrieval of the information of a previously *G_alloc_rw*'ed page; thus, the execution of the *G_alloc_ro* command requires one cycle less than that required for *G_alloc_rw* or *G_alloc_ex*.

To *G_deallocate* a page, first the *Page ID* is used to read the *page G_block*s from the *Allocation Table* using the *Allocation Table* bit vector (shown in Figure 15). The bit vector has 0s in the bit positions that correspond to the page to be deallocated.

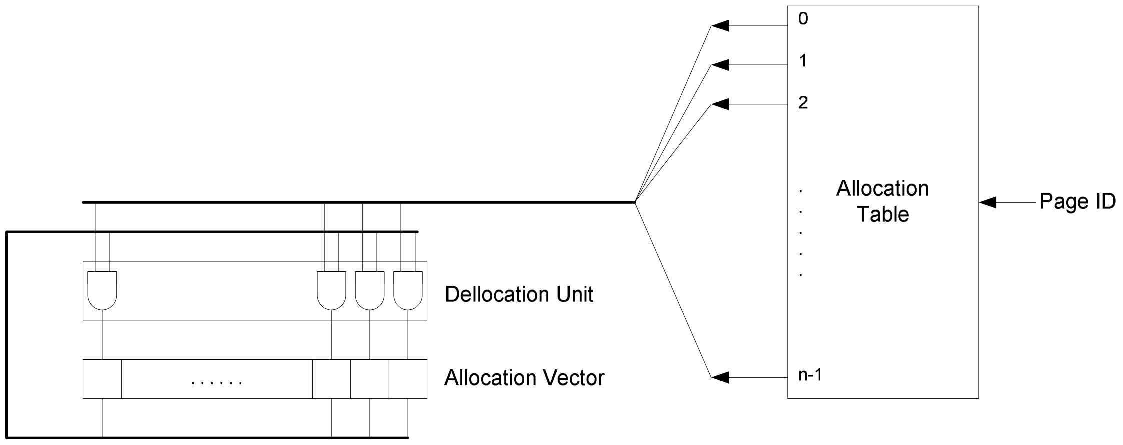


Figure 18: The deallocation process.

The page information along with the information stored in the *Allocation Vector* are fed to the *Deallocation Unit.* The *Deallocation Unit*, as shown in Figure 18, performs a logical *and* operation between the *Allocation Vector* contents and the inversion of the output from the *Allocation Table.* The output from the *Deallocation Unit* is written to the *Allocation Vector.* Also, the appropriate page entries are deleted from the *Allocation Table.*

4.2.3 The Allocation Unit

As shown in Figure 14, the *Allocation Unit* accepts the *Allocation Vector*'s output and the allocation size as inputs and generates a bit-pattern that corresponds to the allocated *G_blocks* (has 1s at the locations of the allocated *G_blocks*). The output is used to update the contents of the *Allocation Vector* (using a logical OR operation). Thus, the *Allocation Unit* is a combinational circuit that allocates any given number

```
1     allocate(size,in[0:n-1]) {
2       for (i:=0 to n-1) {
3         if (in[i]==0 and size>0) {
4           out[i]:=1;
5           size:=size-1;
6         } else out[i]:=0;
7       }
8       if (size>0) return NOT_ENOUGH_MEMORY;
9       else return out[n-1,0];
10    }
```

Figure 19: The allocation algorithm.

of *G_blocks*.

The operation of the *Allocation Unit* can be described by the algorithm shown in Figure 19. Please note that n is the size of the *Allocation Vector* in bits, size is the requested allocation size, *in[n-1:0]* is the output of the *Allocation Vector* and *out[n-1:0]* is the output from the *Allocation Unit*.

Example 9 To explain the allocation algorithm, consider an example where n=8 (this number is unrealistic but is used for simplicity; a typical value of n would be 256). Also, assume that initially the *Allocation Vector* has the value 11001011 and the requested size is 2 *G_block*s. The allocation algorithm iterates through the input bit vector (output of the Allocation Vector) searching for free *G_block*s (indicated by a '0' in the corresponding bit position) to allocate (lines 2 and 3 of Figure 19) until the requested allocation is satisfied or the last bit of the input vector is reached. When a free *G_block* is found the corresponding bit at the output is set to '1' and the size is reduced by 1 (lines 4 and 5 of Figure 19); otherwise, the corresponding bit at the output is cleared. The execution of the algorithm on our input generates the output 00010100.

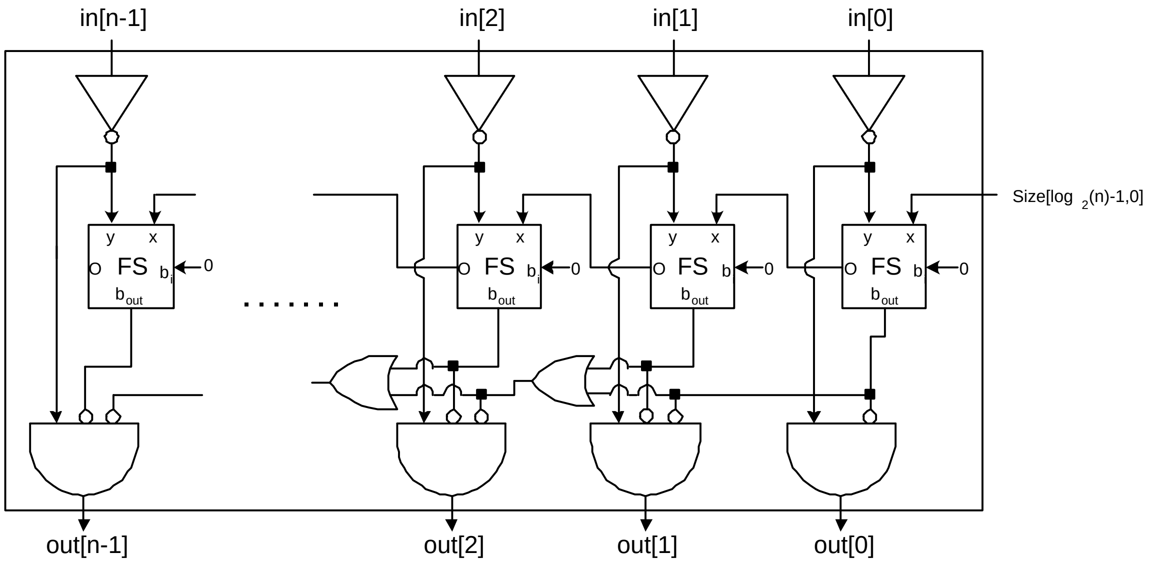


Figure 20: Unoptimized *Allocation Unit* hardware.

This output is used to update the *Allocation Vector* contents through a bitwise OR operation. The updated *Allocation Vector* would be 11011111. □

An unoptimized fully-parallel hardware implementation of the allocation algorithm is shown in Figure 20 and requires a Full Subtractor (FS) for each bit in the *Allocation Vector*. The FS block in Figure 20 performs *O=x-y* operation. The *Allocation Unit* shown in Figure 20 has two inputs: the output from the *Allocation Vector* (*in[n-1,0]*) and the allocation size (*Size*). Each bit of the bit vector *in[n-1,0]* is inverted then subtracted from the *Size* input serially (there is a FS per each bit of *in[n-1,0]*) as shown in Figure 20.

If the output from a Full Subtractor is greater than or equal to zero (indicated by cleared borrow out: b_{out}=0) and the corresponding bit of *in* is zero (which indicates an available *G_block*) then the corresponding output bit from the *Allocation Unit* is

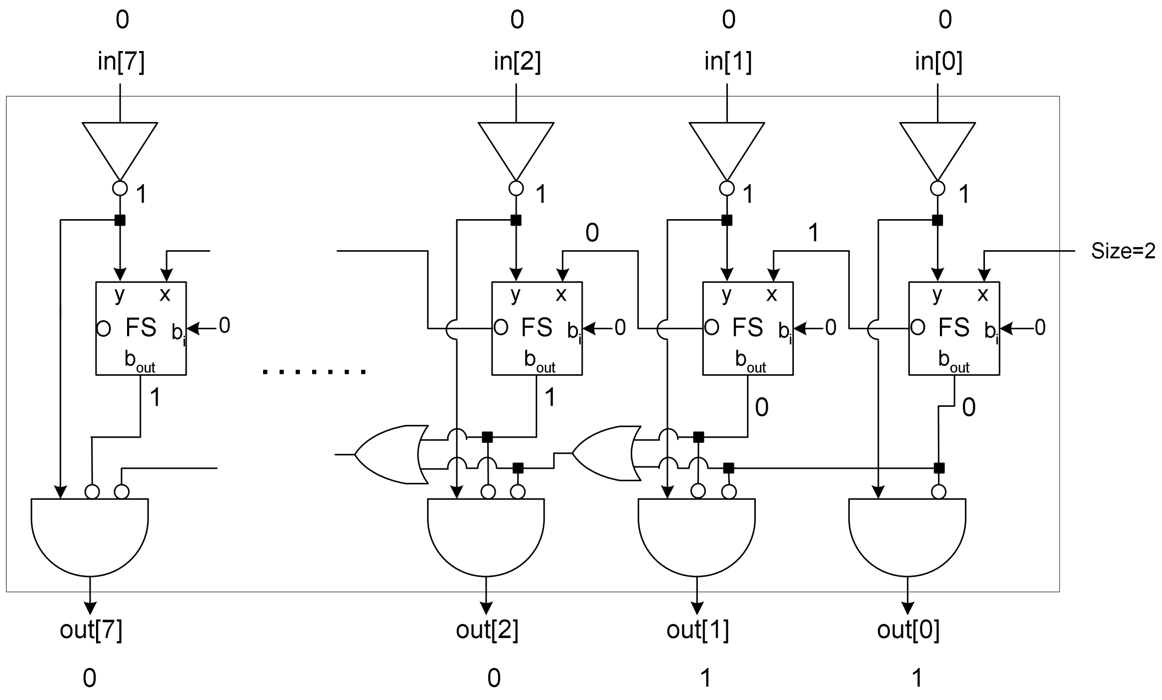


Figure 21: Unoptimized *Allocation Unit* for Example 10.

set to one. Once the output of one of the subtractors is less than zero (which means that the allocation is satisfied) then all output bits from that bit position onward are set to zero.

If the output from the last Full Subtractor (leftmost) is greater than zero then the allocation request cannot be satisfied and an error flag is set. Note that the hardware implementation of Figure 20 is expensive in terms of silicon area (gate count) and propagation delay (the longest path goes through n stages); hence, the allocator described in Figure 20 does not scale up well as n increases.

Example 10 To explain the operation of the unoptimized *Allocation Unit*, consider an example where n=8 (this number is unrealistic but is used for simplicity; a typical value of n

would be 256). Also, assume that initially the *Allocation Vector* has the value 00000000 and the requested size is 2 *G_block*s as shown in Figure 21. First, the input *in* is inverted then subtracted from the input *size* one bit at a time. After subtraction, the result from the first subtractor will be 1 and the result from the second subtractor will be 0. The borrow outs (b_{out}) from the first and the second subtractors will be 0 which cause the first and the second bits of the output *out* to be the inversion of the corresponding input *in* bits, The borrow out (b_{out}) from the third subtractor is 1 which will cause all the bits of *out* starting from the third bit to be 0. Finally, the output will be 00000011. □

Figure 22 shows a more efficient hardware implementation where the n-bit input *in[n-1,0]* is segmented into k segments each of which is m bits (e.g., a 256 bit input can be divided into 16 segments each of 16 bits). The operation of the optimized *Allocation Unit* hardware can be described by the algorithm shown in Figure 23.

The 0s in each m-bit segment are counted using a 0s counter (lines 5 and 6 of Figure 23) and the output of the 0s counter is subtracted from the remaining allocation size (the allocation size request remaining unfilled at time that the segment is considered – segments are considered sequentially from right to left in Figure 22) using n-bit unsigned full subtractor as shown in Figure 22.

The unsigned subtractor has two inputs, the output from the 0s counter and the remaining allocation size (the top arrow and the left arrow entering the full subtractor from the right respectively as shown in Figure 22). Also, it has two outputs, the subtraction result (remaining allocation size = requested size – the output of the 0s

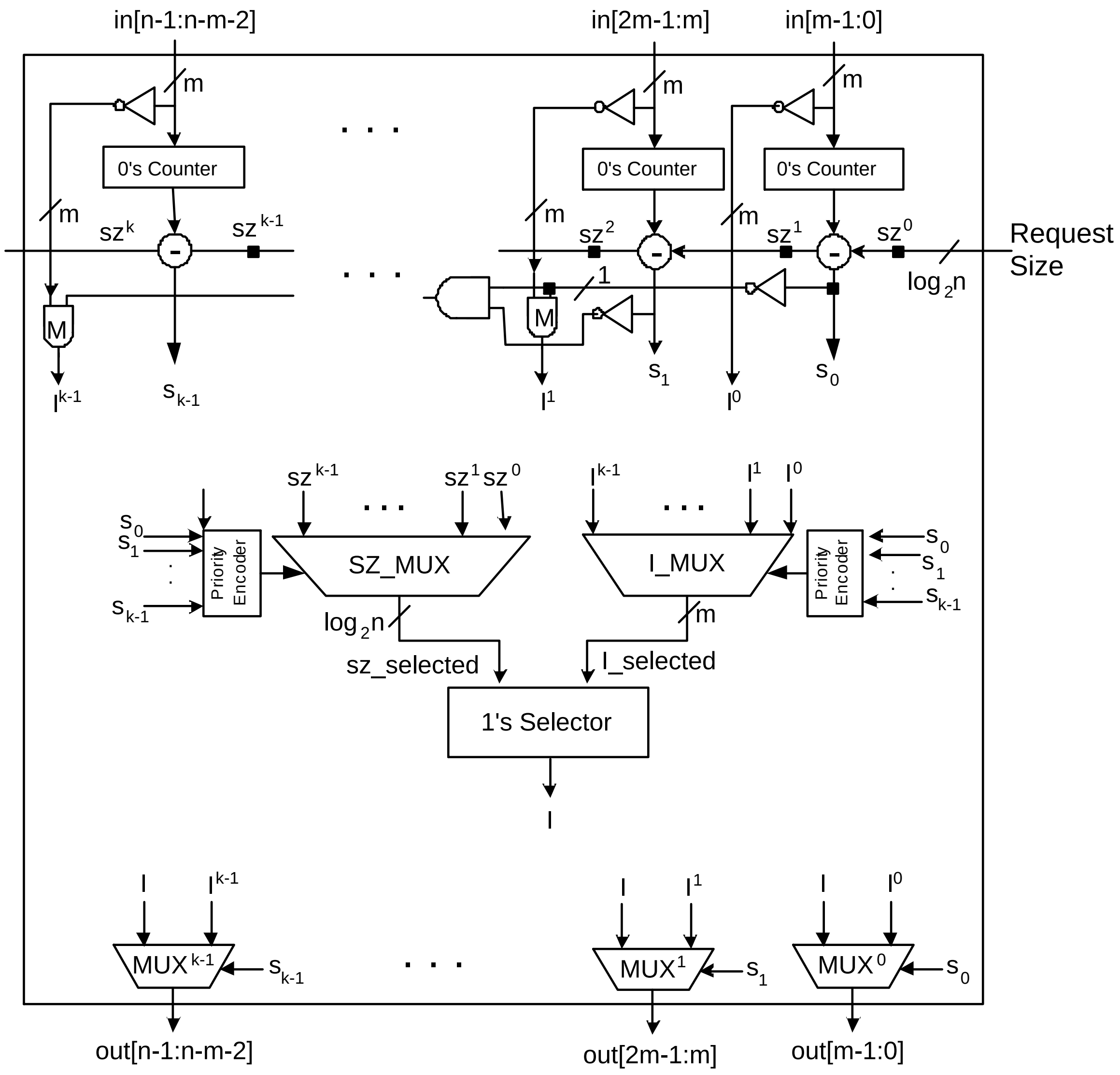


Figure 22: Efficient *Allocation Unit* hardware.

counter) represented by the left arrow leaving the full subtractor from the left and the borrow out (bottom arrow) which indicates if the result is less than zero (borrow out = 1) or the result is greater than or equal to zero (borrow out = 0). If the output from a subtractor is greater than or equal to zero (indicated by a “0” on the s_j signal, *0* $\leq$ *j* $\leq$ *k-1*, which is connected to the “borrow out” output of the jth subtractor) then the input bits segment (e.g., *in[2m-1:m]* as shown in Figure 22) is inverted (please note that the free *G_block*s are indicated by 0s at the input, while the allocated *G_block*s are indicated by 1s at the output) and routed to the output through the corresponding multiplexer shown at the bottom of Figure 22 (lines 8-11 of Figure 23). Thus, in effect, all unallocated *G_block*s available, as indicated by the input bits segment (e.g., *in[2m-1:m]*), are selected for allocation. Otherwise, the borrow out output of the subtractor is one, indicating that not all available *G*_blocks are needed: in this case, the inverted input bits segment (e.g., *in[2m-1:m]*) and the remaining allocation size (e.g., in Figure 22, sz^1 which is the number of *G_block*s requested from *in[2m-1:m]*) are routed to the *1s selector* block through the multiplexers *I_MUX* and *SZ_MUX* respectively as shown in Figure 22 (lines 13-18 of Figure 23).

The *1s selector* selects the number of ones equivalent to the remaining allocation size (*sz_selected* in Figure 22). The output from the *1s Selector* is routed to the output bits that correspond to the input bits segment (lines 13-18 of Figure 23). Once the allocation size is satisfied all the segment output bits that follow the segment that goes through the *1s Selector* are cleared to zero. Please note that in Figure 22 the

```
allocate(size,in[0:n-1]) {
  done:=0;
  for (i:=0 to k-1) {
    zc:=0;
    for(e:=i*m to (((i+1)*m)-1))
      if (in[e]==0) zc:=zc+1;
    if(done==0) {
      if(size-zc>0) {
        for (e:=i*m to (((i+1)*m)-1))
          out[e]:=!in[e];
        size:=size-zc;
      }else {
        for(e:=i*m to (((i+1)*m)-1)) {
          if(in[e]==0 and size>0) {
            size:=size-1;
            out[e]:=1;
          }
        }
        if (size==0) done:=1;
      }
    }else
       for (e:=i*m to (((i+1)*m)-1))
    out[e]:=0;
  }
  if (done==0) return NOT_ENOUGH_MEMORY;
  else return out[n-1,0];
}
```

Figure 23: The optimized allocation algorithm.

and gate with an "M" in the middle represents m 2-input *AND* gates. Also, note that we assume that the maximum allocation size requested by any PE is $n-1$ *G_block*s (which is a reasonable assumption for an SoC with multiple PEs and a global memory of n *G_block*s). If this restriction is not desirable, the size of the *Request Size* input must be $log_2(n)+1$; also, the size of the full subtractors should be increased to $n+1$.

This implementation uses typical parallel logic techniques to reduce the propagation delay dramatically (the longest path goes through k stages instead of n). Unlike typical parallel logic design in this specific case the area was reduced too (when compared to the area of the hardware in Figure 20).

Example 11 To explain how the optimized *Allocation Unit* works, consider an *Allocation Unit* in Figure 24 that has n=16 (this number is unrealistic but is used for simplicity; a typical value of n would be 256) and k=4 (the input vector is divided into four groups each of which has $log_2(16)$=4 bits). Also, assume that initially the *Allocation Vector* has the value **0000000011001011** and the requested size is 2 *G_block*s. These inputs cause the outputs from the four 0s counters to be **4**, **4**, **2** and **1** (please note that we use hexadecimal digits to denote the outputs from the 0s counters for the ease of reading). Also, the internal signal sz^1 (sz^0 is connected to the *Request Size* which is an input) will be **1** (the signals sz^1, sz^2, ..., sz^k represents the propagated sizes; each of which is 4 bits). Please note that in this example we are concerned only with the sz^1 signal since the remaining size is satisfied by the second segment. In the same way, the signals s_0 and s_1(which are connected to the borrow out outputs of the first and the second subtractors, from the left in Figure 24, respectively) will be **0** and

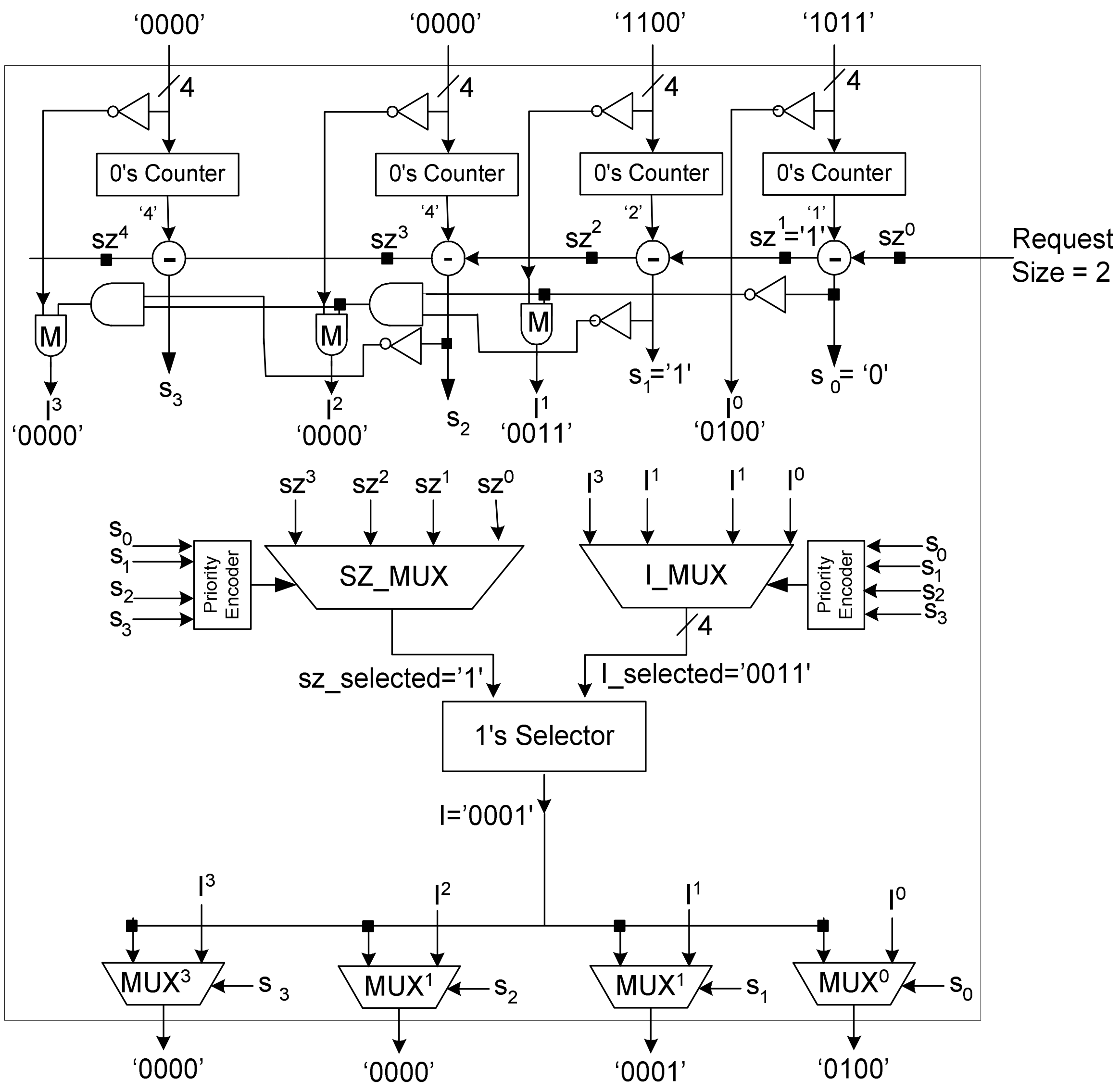


Figure 24: Efficient *Allocation Unit* hardware for Example 11.

1 respectively and I^3, I^2, I^1, I^0 will be **0000**, **0000**, **0011** and **0100** respectively. The signals s_0, s_1, s_2 *and* s_3 are encoded using a priority encoder whose output goes to the multiplexers *I_MUX* and *SZ_MUX* as shown in Figure 22. This will cause the inputs to the *1s Selector* to be *sz_selected*= sz^1=**0001** and *I_selected*=I^1=**0011**. The output from the *1s Selector* will be **0001**. Finally, the output from the *Allocation Unit* will be **0000000000010100**. Please note that the *Allocation Unit* is a pure combinational circuit thus it takes less than one clock cycle to find the free *G_block*s satisfying the requested allocation size. □

Table 2: Optimized Allocator and Unoptimized Allocator Comparisons

	Area (NAND gates)	**Worst Delay (ns)**	**Max Clk Speed (MHz)**
Optimized Allocator	5364	6.6 ns	150 MHz
Unoptimized Allocator	17930	56.3 ns	17.5 MHz
Comparison	3.3X	8.5X	

Table 2 compares the area and the speed of the optimized and unoptimized allocators. For the comparison, we developed Verilog RTL models for the optimized and unoptimized allocators each of which can perform allocation of an on-chip memory of 256 *G_block*s. We synthesized both allocators using the Synopsys Design Compiler^{TM} and a TSMC 0.25u library from LEDA [21]. The optimized allocator reported a better performance in terms of speed and area. As shown in Table 2, the area of the optimized allocator is less than one third the unoptimized allocator. Moreover, the optimized allocator is much faster (8.5X) than the unoptimized allocator. Obviously, the SoCDMMU uses the optimized *Allocation Unit*.

4.3 Implementing the SoCDMMU using General Purpose Processors

Although the hardwire implementation of the SoCDMMU shows good results in terms of area and clock speed, it does not allow any modifications of the functionalities of the SoCDMMU after implementation in custom silicon. One solution is replacing parts of the SoCDMMU with programmable elements that can be re-programmed to provide enhancements to existing functions and/or to add new features to the SoC-DMMU after implementation. These programmable elements can be in the form of general purpose microprocessors/microcontrollers that run programs that implement the same functions as the replaced hardware components.

The BASIC SoCDMMU is the most important part of the SoCDMMU as it executes the allocation algorithm by use of the *Allocation Unit* and stores the allocation information in the *Allocation Table* and the *Allocation Vector*. Once the SoCDMMU is implemented as hardwired logic, the allocation algorithm cannot be changed. If the allocation algorithm were to require upgrading or modification, then the BASIC SoCDMMU needs to be replaced by a programmable element.

We built a version of the BASIC SoCDMMU using a general purpose microcontroller – Microchip's PIC. The microcontroller runs software that implements the allocation algorithm (described earlier in Figure 16) and stores the memory allocation status. The software is stored in a flash memory and can be modified after implementation if required.

Although the SoCDMMU that uses the BASIC SoCDMMU that is implemented in software that runs on the general purpose microcontroller is flexible and extensible, the performance, in terms of clock cycles, is more than 10X slower than the hardwired implementation of the SoCDMMU as will be described in Chapter 7.

We believe that the hardwired logic implementation is the optimal implementation for the SoCDMMU. If upgradability is an issue then the hardwired logic can be implemented using an FPGA instead of an ASIC.

4.4 Summary

In this chapter we showed a hardware implementation of the SoCDMMU as hardwired logic and discussed implementing part of the SoCDMMU using software that runs a on general purpose processor to add flexibility and upgradability to the SoCDMMU. In Chapter 5, we will discuss how a custom hardwired implementation of the SoCDMMU can be automatically generated. A CAD tool for this purpose will be presented in the next chapter.

CHAPTER V

AUTOMATIC CUSTOMIZATION OF THE SOCDMMU

5.1 Introduction

The ability to design complex integrated circuits is found to be lagging the technologys capability. According to the 2001 International Technology Roadmap for Semiconductor (ITRS), while the number of available raw transistors in a single chip increases by 58% every year, designers' capability to design them grows by only 21% every year as shown in Figure 25 [34]. As the number of transistors on a single chip increases rapidly, the productivity gap (indicated by the arrows in Figure 25) between the increasing number of available logic transistors on single and the productivity (number of transistors designed/staff-month) and this gap increases year after year.

One solution to reduce this productivity gap is to increase the use and reusability of Intellectual Property (IP) cores. However, an IP core should be customized/configured before being used in a system different than the one for which it was designed. Thus,

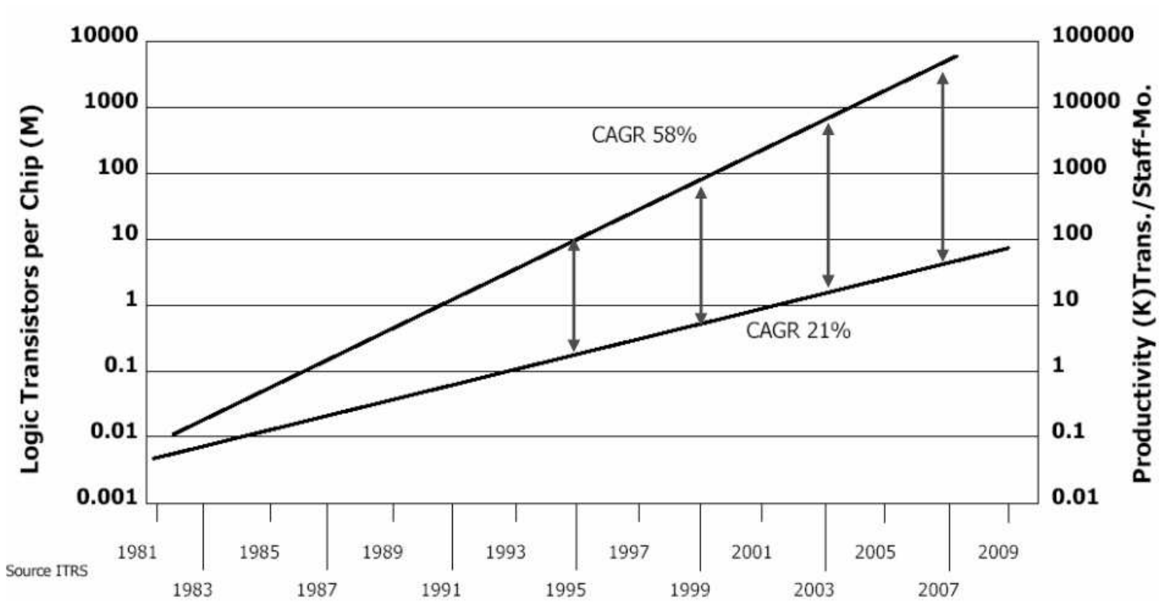


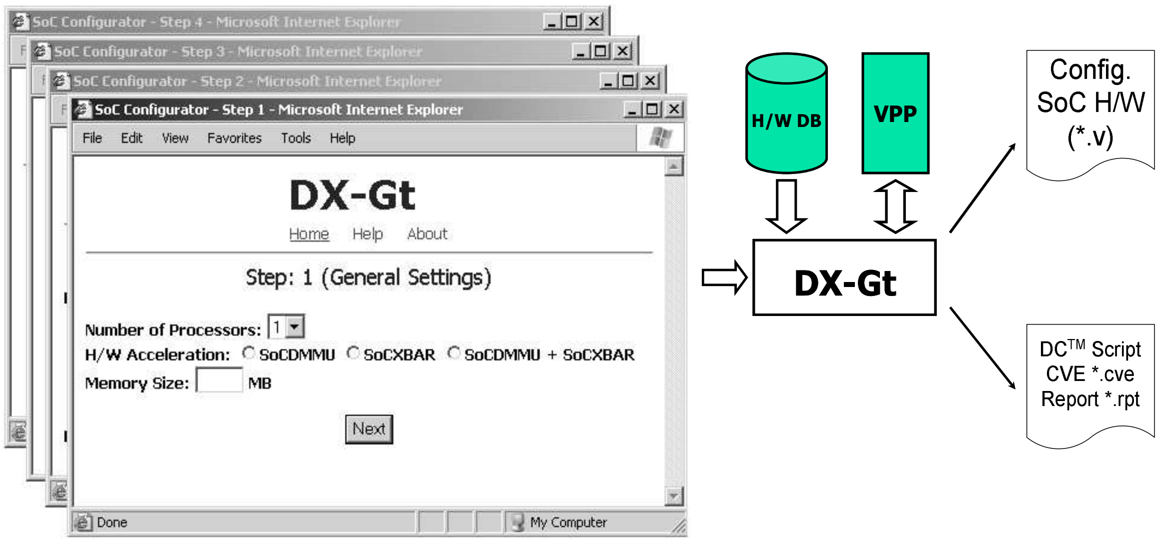
Figure 25: The Productivity Gap (source: ITRS).

to reconfigure the IP core, either an engineer must spend significant effort altering the core by hand or else an enhanced CAD tool (IP generator) can automatically configure and customize the core according to the customer specifications. For example, memory and I/O generators by Artisan [6] and processor generators by Tensilica [45] and ARC [3] supply application specific IP cores that can be highly tuned for specific applications.

The SoCDMMU, as an IP core, needs to be configured before being used in a particular SoC. As Chapter 4 showed, the SoCDMMU has many parameters that can be changed from one design to another (e.g., the number of memory blocks, the memory block size and the number of PEs). A tool that captures the user's input and generates an optimized version of the SoCDMMU IP is a must to bridge the time to market gap.

5.2 The SoCDMMU Hardware Configuration

We have developed an IP generation tool that enables an SoC designer to generate a custom version of the SoCDMMU according to the user's input; we call this tool SoCDMMU XBar Generator (DX-Gt). DX-Gt is a CAD tool that allows the designers to generate a custom SoCDMMU and configure an Xbar switch [36] for a multiprocessor SoC. In this thesis we focus only on the SoCDMMU generation part of the Dx-Gt tool.



SoC Configurator - Step 4 - Microsoft Internet Explorer
SoC Configurator - Step 3 - Microsoft Internet Explorer
SoC Configurator - Step 2 - Microsoft Internet Explorer
SoC Configurator - Step 1 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
DX-Gt
Home Help About
Next
Done
My Computer
H/W DB
VPP
DX-Gt
Config. SoC H/W (*.v)
DC™ Script CVE *.cve Report *.rpt

Step: 1 (General Settings)

Number of Processors: 1
H/W Acceleration: ○ SoCDMMU ○ SoCXBAR ○ SoCDMMU + SoCXBAR
Memory Size: MB

Figure 26: The SoC configuration tool flow.

As shown in Figure 26, a Graphical User Interface (GUI), which consists of a set of HTML forms, captures the user's inputs and passes them to a Common Gate Interface (CGI) application (developed in C-Language). The CGI application processes the user inputs and generates the configured RTOS files (C and assembly source files) and the

SoCDMMU and XBar hardware files (Verilog format). Moreover, DX-Gt generates Synopsys DC^{TM} [44] synthesis scripts and a Mentor Graphics Seamless CVE^{TM} [24] configuration file for simulating of the resulting SoC design.

Figure 27 shows the flow of DX-Gt. Once the user configurations and settings are captured using a set of HTML forms, the CGI application first checks the parameters input by the user to make sure that they are valid (the number of *G_block*s, the size of the *G_block* and the data bus width each should be a power of 2) and then selects from the database the hardware components that satisfy the user specified configurations. Next, the CGI application sets the parameters of each hardware component to reflect the user input. Then, the hardware components (Verilog files) are passed to the Verilog PreProcessor (VPP) [48] which processes them and generates new customized Verilog files for the SoCDMMU.

If the user elected to use an XBar in the SoC design and entered the required parameters for the the XBar, DX-Gt calls the application *gen_xbar* [36] to generate the necessary hardware files for XBar. Details about the Xbar generation are contained in a separate thesis [38]. Finally, DX-Gt generates the hardware top Verilog files and the scripts that are necessary to synthesize the SoCDMMU and the Xbar switch logic using the Synopsys Design Compiler. Also, DX-Gt generates configuration files required to co-simulate the SoC using Mentor Graphics Seamless CVE.

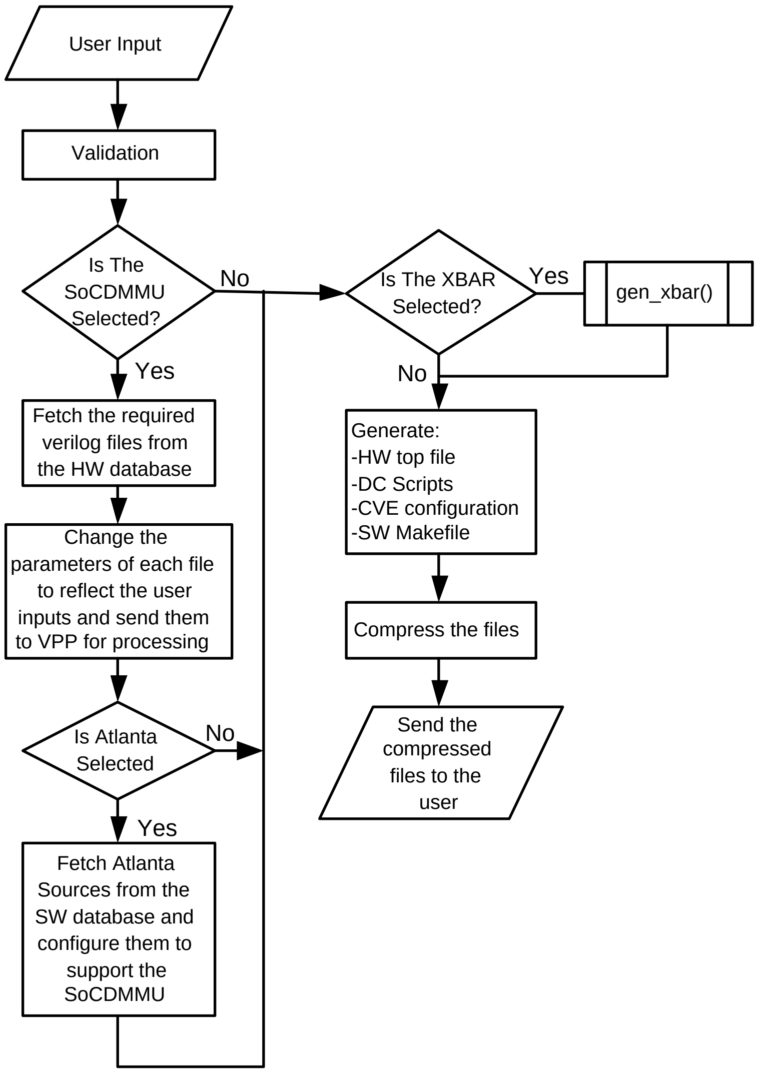

User Input
Validation
Is The SoCDMMU Selected?
No
Yes
Fetch the required verilog files from the HW database
Change the parameters of each file to reflect the user inputs and send them to VPP for processing
Is Atlanta Selected
No
Yes
Fetch Atlanta Sources from the SW database and configure them to support the SoCDMMU
Is The XBAR Selected?
Yes
gen_xbar()
No
Generate:
-HW top file
-DC Scripts
-CVE configuration
-SW Makefile
Compress the files
Send the compressed files to the user

Figure 27: Flowchart of the CGI application.

5.2.1 The GUI

The Dx-Gt GUI is made of set of HTML forms (as shown in Figure 20) that can be easily published on the web. This makes the GUI universal and platform independent. The HTML forms captures the user's inputs required to customize the SoCDMMU. The following is a list of the user specified parameters:

- System wide parameters
  - Number of PEs
  - Number of the global on-chip memory *G_block*s which determines the size of the SoCDMMU.
  - Sizes of the global on-chip memory *G_block*s which determines the address bus widths between the switches and memory blocks
  - Number of memory ports
  - Memory type which determines the memory controller chosen from a hardware database
  - PE types which determines processor interfaces to SoCDMMU chosen from a hardware database
  - Choice of use of SoCDMMU, Xbar, both or none
- SoCDMMU related parameters

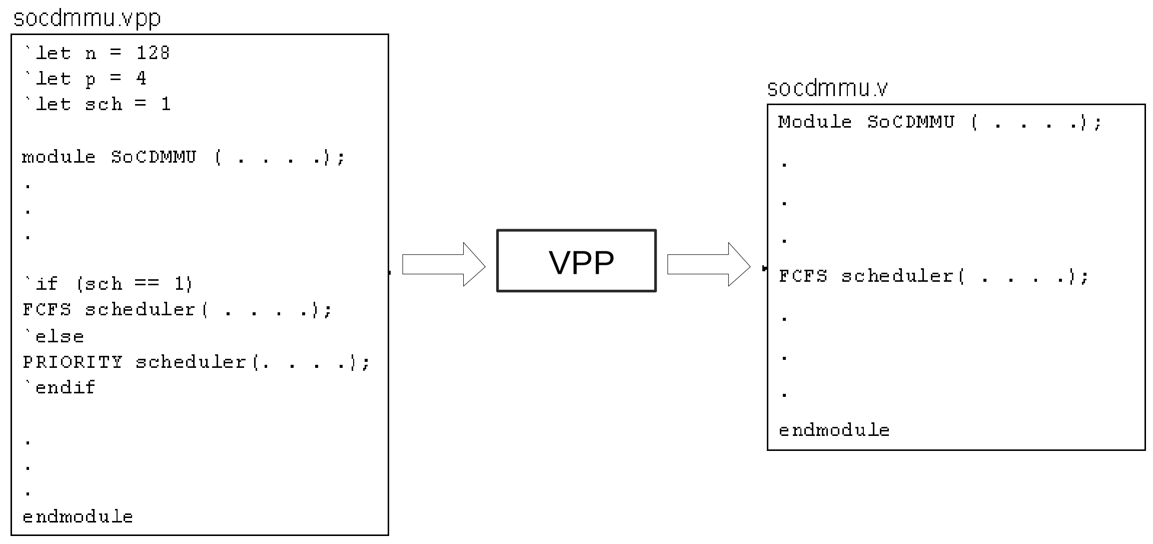
    - The scheduling scheme to resolve concurrent memory requests from different PEs (first come first served scheme or priority scheme)
    - Memory *G_block*s initially assigned to the PEs (initial memory assignment for the PEs)

- Xbar related parameters
    - The data bus width of each PE (determined by the PE type)
    - The address bus width connected to each PE (determined by the global memory size)

5.2.2 Hardware Database

In order to generate the hardware files, a "hardware database (HW DB)" of parameterized Verilog files of each system component – SoCDMMU sub-modules, processor bus wrappers, and memory controller – is being used. The Verilog files in the database are written in such a way that a custom version of the file can be generated using a Verilog preprocessor (VPP).

Example 12 Figure 28 shows the generation of one of the Verilog files of the customized SoCDMMU. The file "socdmmu.vpp" was fetched from the HW DB and modified by the CGI application (as described earlier) to set some parameters (the first three lines) to reflect the user's input. Specifically, the first three lines set the number of processing elements (p=4), the number of memory blocks (n=128) and the type of the request selector (sch=1). The "socdmmu.vpp" file is then processed by the VPP to generate the custom file "socdmmu.v". □



```
`let n = 128
`let p = 4
`let sch = 1

module SoCDMMU ( . . . .);
.
.
.

`if (sch == 1)
FCFS scheduler( . . . .);
`else
PRIORITY scheduler(. . . .);
`endif

.
.
.
endmodule
```

```
Module SoCDMMU ( . . . .);
.
.
.
FCFS scheduler( . . . .);
.
.
.
endmodule
```

Figure 28: An example of custom Verilog generation using VPP.

5.2.3 Allocation Unit Optimization

The *Allocation Unit* is the most important part of the SoCDMMU as the critical path of the SoCDMMU goes through it. To generate a fast *Allocation Unit*, the *Allocation Unit* must be tweaked. The *Allocation Unit*, as described in Section 4.2.3, has two parameters: m and k which determine how the input data to the *Allocation Unit* is processed. The values of these two parameters need to be chosen for a given number of memory blocks (n) to generate a fast *Allocation Unit*.

A closer look at the *Allocation Unit* critical path (the dashed line in the Figure 29) shows that the *Allocation Unit* critical path goes through:

- One *0s counter* (which has almost constant delay time)
- k subtractors each of a constant delay D_s.

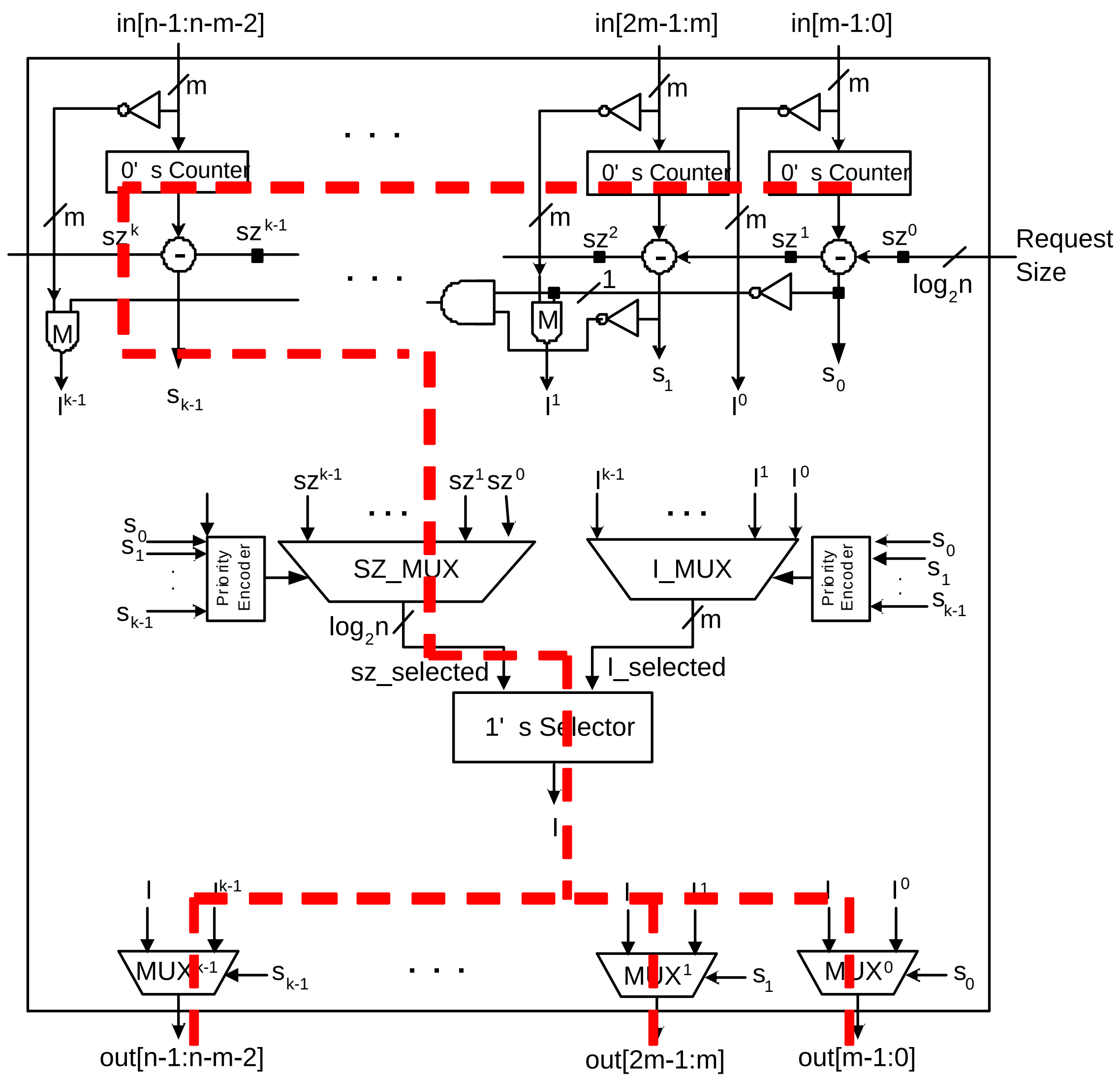


Figure 29: The *Allocation Unit* critical path.

- One multiplexer (SZ_MUX) which has almost constant delay time.

- One *1s selector* of delay = m x d_1, d_1 is the delay per stage

  The *1s selector* has an architecture similar to the unoptimized *Allocation Unit* described in Chapter 4. The *1s selector* delay depends on the number of stages, which depends on the number of input bits (m).

- One output multiplexer which has constant delay time.

The total delay time over the *Allocation Unit* critical path can be found be adding all the delay times of the components that on the critical path. The total delay over the critical path can be calculated using the following equation:

$D_{criticalpath} = C + k$ x $D_s + m$ x d_1

C is a constant that represents the sum of the delays of the constant delay elements (the multiplexers and the 0's counters).

Recalling that:

$n = m$ x k

Then:

$D_{criticalpath} = C + k$ x $D_s + (n/k)$ x d_1

Since $D_{criticalpath}$ is a convex function of k and it has a maximum value of infinity in the period $k > 0$, $D_{criticalpath}$ has a global minimum in the period $k > 0$. To find the value of k that minimizes $D_{criticalpath}$, we differentiate the above equation with respect to k and assigns the results to zero. This gives

$D_s - (n/k^2)$ x $d_1 = 0$

$k^2 = n$ x $d_1 \ / \ D_s$

$k = SQR(n$ x $d_1 \ / \ D_s) : k > 0$, k is power of 2

The CGI application uses the above equation to find the optimal value of k given the value of n. The constants C, d_1 and D_s are library dependent and can be found for every used library.

5.3 Advantages of using DX-Gt

DX-Gt allows ASIC designers to generate a custom optimized version of the SoC-DMMU without prior knowledge of the SoCDMMU architecture or the details of its hardware implementation. DX-Gt eliminate the learning period required by designers to understand the internals of the SoCDMMU; hence, it eliminates the need for an SoCDMMU expert designer in the design team. DX-Gt enables experienced and unexperienced ASIC designers to generate a custome optimized version of the SoCD-MMU in just few minutes using easy to use and understand interface. Also, DX-Gt makes it easy for the designer to use the SoCDMMU in different SoC designs with different configurations. The ability of DX-Gt to generate custom optimized version of the SoCDMMU shorten the design time and make it faster to bring the SoC design to the market. Moreover, using DX-Gt reduces the bugs and the errors that may result from configuring the SoCDMMU IP by hand.

5.4 The SoCDMMU Hardware Synthesis Results

This section presents the synthesis results of custom SoCDMMU generated by our tool (DX-Gt). We use the Synopsys Design Compiler [44] with a .25μm TSMC technology library from LEDA Systems [21].

Table 3: The SoCDMMU Synthesis Results (w/o the Memory Elements)

Lines (RTL Verilog)	**Area in TSMC 0.25u**
2100 lines	7550 Gates

In Table 3, we show the results for the synthesis of the SoCDMMU that works with an SoC with 256 memory *G_block*s and four PEs. The second column in Table 3 shows the number of NAND gate equivalents of the hardware required (not including the *Address Converter*s and the *Allocation Table*) when synthesized for a clock speed of 300MHz (the maximum achievable clock speed for this configuration; the critical path goes through the *Allocation Unit* as described in Section 5.2.3).

Table 4: The Address Converter and *Allocation Unit* Synthesis Results

Module	**Size of 6T SRAM with the same area**
Address Converter	1.22 KB
Allocation Table	0.66 KB

Table 4 shows the area of the *Address Converter* and the *Allocation Table*, which are mainly memory elements. The area is represented in equivalent 6T-Static Random Access Memory (SRAM) area. Please note that the Address Converter used in this system has 256 entries each of which is 16 bits wide.

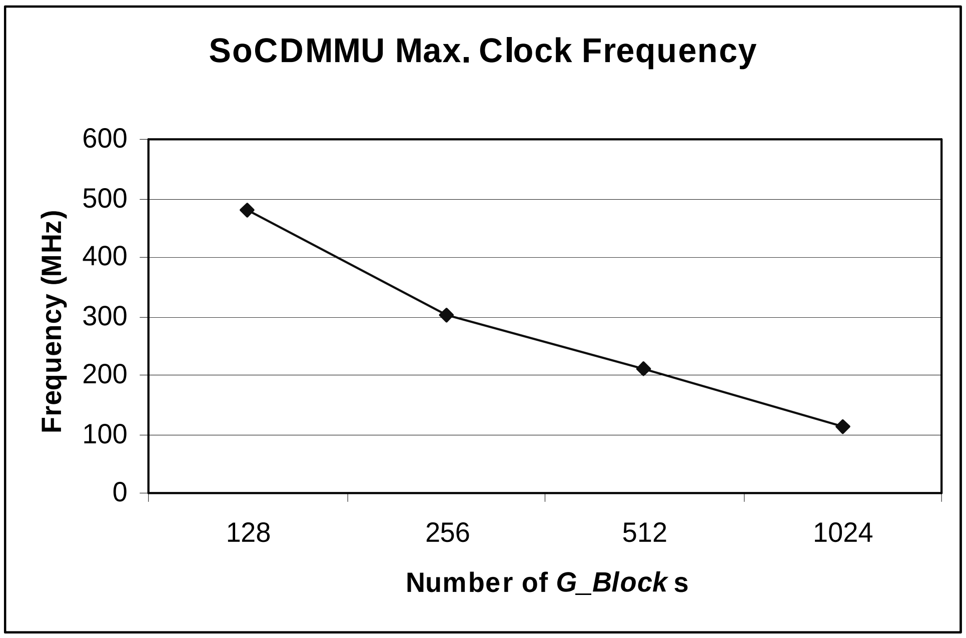


Figure 30: The maximum clock frequency of the SoCDMMU different numbers of *G_block*s.

Direct calculations of these numbers would make the reader conclude that the area Address Converter should be equivalent to the area of an SRAM of 512 bytes. However, Table 3 shows an Address Converter has the same area as 1.22KB SRAM. The extra area (the difference between 0.5KB and 1.22 KB) comes from the fact that the Address Converter uses associative memory which requires the use of comparators and other control logic in addition to the 512 SRAM bytes.

Figure 30 shows the maximum clock frequency (measured in MHz) of the SoCD-MMU for different numbers of *G_block*s (128, 256, 512 and 1024). As the number of *G_block*s increases the clock frequency decreases for TSMC 0.25u technology.

Figure 31 shows how the SoCDMMU area scales with the number of PEs (2, 4, 8 and 12) and *G_blocks* (128, 256, 512 and 1024). The area (represented by number of

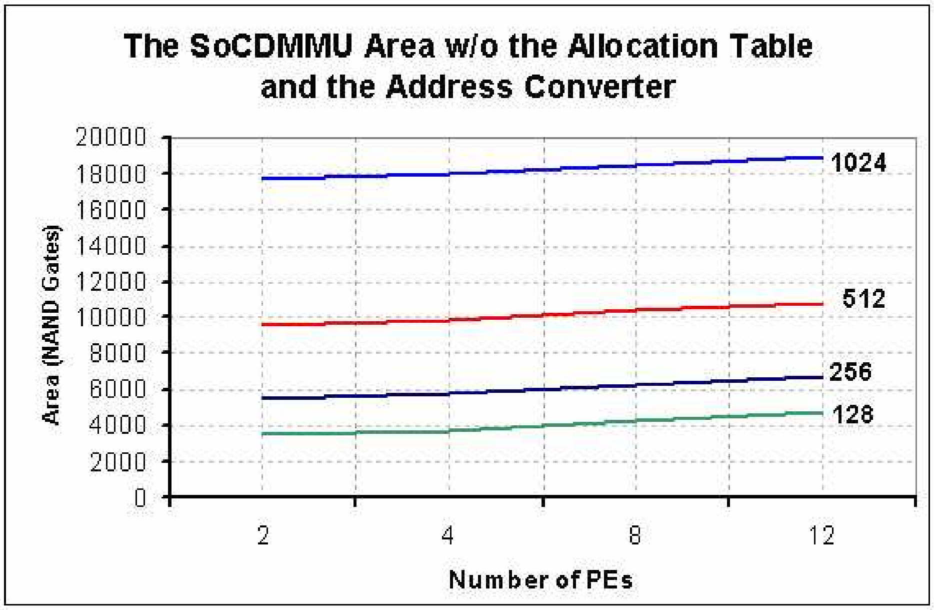


Figure 31: The Area of the SoCDMMU (w/o the *Allocation Table* and the Address Converter) for different numbers of PEs and *G_block*s.

equivalent NAND gates) scales linearly with the number of processors. Please note that the results were obtained using a clock frequency of 100MHz. We choose the 100MHz clock frequency to obtain the results as it is the maximum clock frequency we can achieve for an SoCDMMU that can handle 1024 *G_block*s.

Figure 32 shows the area of the *Address Converter* and the *Allocation Table*, which are mainly memory elements and are both part of the SoCDMMU, for different numbers of processors (2, 4, 8 and 12) and *G_blocks* (128, 256, 512 and 1024). The area is represented in equivalent 6T-Static Random Access Memory (SRAM) area.

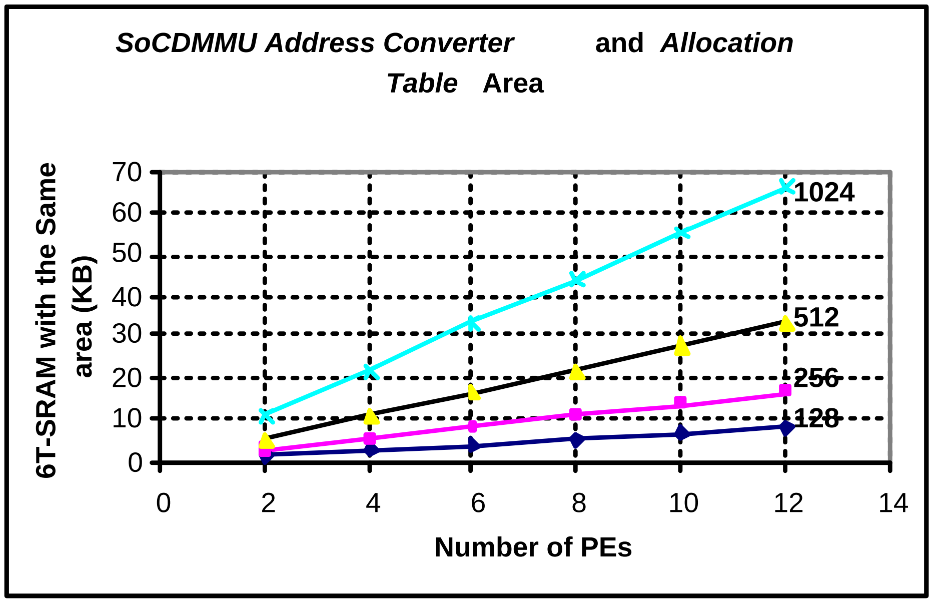


Figure 32: The Area of the *Address Converter* and the *Allocation Table* for different numbers of PEs and *G_block*s.

5.5 *Summary*

In this chapter, we presented a tool to automatically generate custom SoCDMMU hardware. The tool allows designers to customize and configure the SoCDMMU; hence, our tool reduces the time to market and helps to close the productivity gap.

In the next chapter, Chapter 6, we will show how to add the SoCDMMU to existing operating systems to enable designers to integrate the SoCDMMU with their system software.

CHAPTER VI

RTOS SUPPORT FOR THE SOCDMMU

In this chapter, we show how a real-time operating system (Atalanta) is modified to support the proposed SoCDMMU. First we will show how the memory allocation/deallocation is handled in the Atlanta RTOS; then we show how we modified the RTOS to support our hardware. Finally, we will outline set of guidelines for modifying any operating system to support the SoCDMMU.

6.1 Introduction

Dynamic memory management can consume a great amount of program's execution time. Moreover, memory management routines (e.g., *malloc()* and *free()*) do not have deterministic behavior. The most often used software allocator for dynamic memory allocation is sequential fit or segregated fit (refer to Chapter 2 for more information). A *malloc()* function that utilizes one of those allocators maintains the allocation status in a linked-list. Searching a linked-list sequentially to find free memory chunks takes time dependent on the length of the list; as the length of

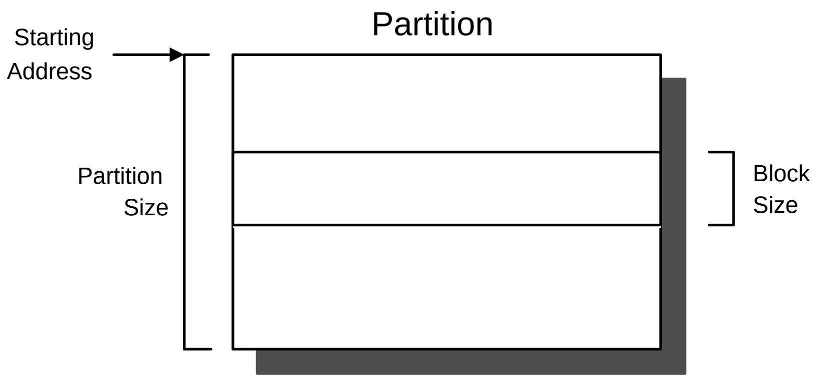


Figure 33: Memory partition in Atalanta.

the linked-list increases the time consumed to search the linked-list increases as well. Moreover, the searching time is not constant and changes dynamically. Small RTOSes typically do not support full dynamic memory allocation using *malloc()* and *free()*; instead, an RTOS usually divides the memory into fixed-sized allocation units from which any task can allocate only one unit at a time. Examples of such RTOSes are: eCos [30], pSOS [14], VRTXoc [26] and uCOS-II [20]. In this way the RTOS memory manager does not need to use complex data structures to store the status of allocated memory chunks. With this approach, an RTOS guarantees constant execution time for each memory management function.

6.2 Atalanta RTOS Memory Management

Atalanta is an open source RTOS developed at the Georgia Institute of Technology to be used for SoC [43]. 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 26 (please note that

Table 5: Atalanta Memory Management System Calls.

Function Declaration	**Description**
SYS_MEM asc_partition_gain(SYS_PARTITION,SYS_ERROR);*	Get free memory block from a partition (non-blocking)
SYS_MEM asc_partition_seek(SYS_PARTITION,SYS_TIME, SYS_ERROR);*	Get free memory block from a partition (blocking)
VOID asc_partition_free(SYS_PARTITION,SYS_MEM, SYS_ERROR);*	Free a memory block.
VOID asc_partition_reference(SYS_PARTITION, SYS_INFO, SYS_ERROR*);*	Get partition information.

a partition block is completely different than the *G_block* described in Chapter 3 – specifically, while a *G_block* size is usually quite large, e.g., 64KB in Example 4, partition block size is usually measured in bytes). The partitions are statically allocated (as static arrays) and cannot be created or deleted at run time. Multiple partitions of different block sizes may be created. A task can allocate or deallocate only one block at a time from a partition. Thus, external fragmentation (that sometimes results from dynamic memory allocation) does not occur. Consequently, memory compaction is not required. However, internal fragmentation may occur if the allocated memory block is not fully utilized because there is no partition with smaller block size. Atalanta provides only four Application Programming Interface (API) functions to manage the memory. These functions are summarized in Table 6.2. For further information please refer to [43].

6.3 Atalanta Support for SoCDMMU

We modified the Atalanta RTOS memory management to support the SoCDMMU. While modifying the Atalanta memory management system, we kept in mind the following issues. First, we add dynamic memory management for the global on-chip memory. Second, we do not alter the existing 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.

- 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 for a billion-transistor SoC 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 Chapter 3.

6.3.1 New API Functions

Since they did not already exist, 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 of sizes not modifiable at run-time).

Table 6: New API memory management functions introduced to the Atalanta RTOS

Function Name	**Description**
asc_partition_create(Size,Partition_blocks, Mode, Page_ID, Error)	Create a partition by requesting memory allocation from the SoCDMMU if necessary.
asc_partition_delete(SYS_PARTITION, Error)	Delete a partition and deallocate memory block if required.

Table 7: The *asc_memory_find* function

Function Name	**Description**
asc_memory_find(Size, Error)	Find a place in the PE address space to map the allocated memory.

Table 6 explains these two new functions. A task calls the *asc_partition_create* function to create a partition in the memory G_allocated to the PE (on which the task runs). If there is not enough memory or the PE's available memory has a different mode (e.g., read only or exclusive) than that of the requested partition, *asc_partition_create* requests a memory page (one or multiple *G_block*s) from the SoCDMMU.

The *asc_partition_create* has five arguments: the allocation size in *G_block*s, number of blocks per partition, the allocation mode (exclusive, read_only and read/write), the *Page ID* and pointer to the error structure. The *asc_partition_delete* function deletes a partition when it is not required anymore; *asc_partition_delete* will request memory deallocation from the SoCDMMU if the entire physical *G_block* that contained the partition is not in use anymore. The *asc_partition_delete* function has two arguments: the memory partition structure (SYS_PARTITION) and pointer to the error structure.

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 6 gives a description of this function.

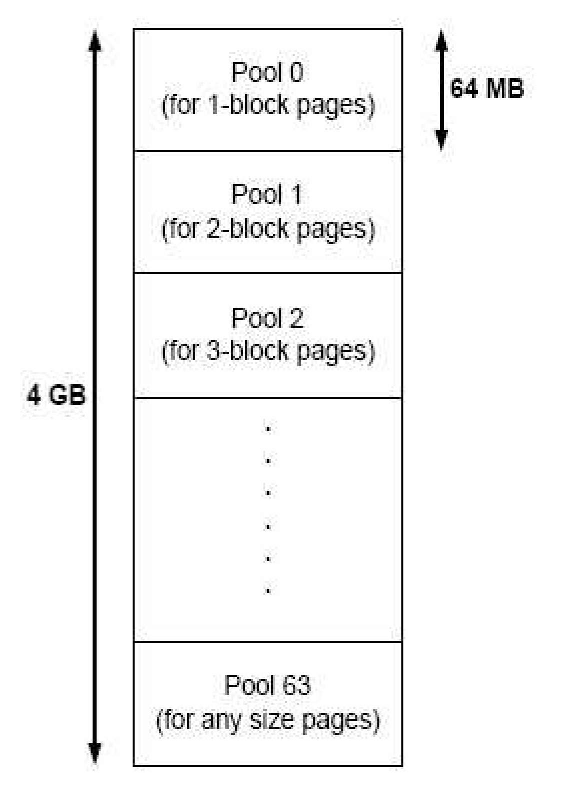


Figure 34: The PE's address space divided into pools.

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) as shown in Figure 34. 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 *G_block*, 2 *G_block*s, 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-*G_block* pages, the second pool for 2-*G_block* pages, etc., as illustrated in Figure 34.

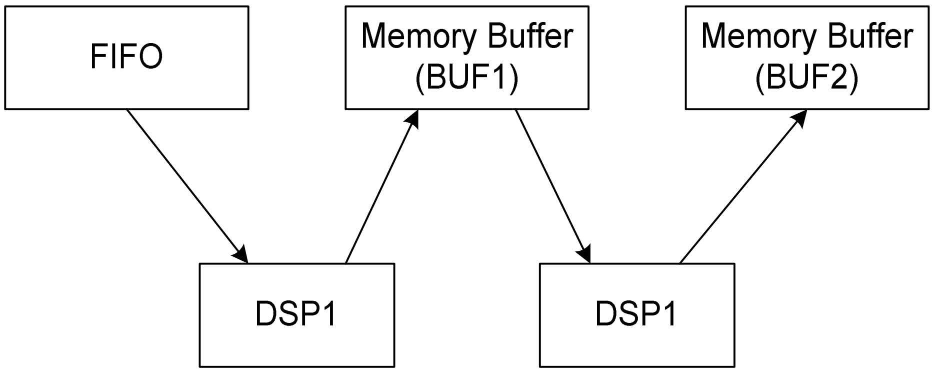


Figure 35: OFDM sub-system used in Example 13.

Example 13 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

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 36: Code snippets for the OFDM system in Example 13.

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. This sequence of operations is outlined in Figure 35. DSP1 allocates shared memory buffer (BUF1) as *read/write* while DSP2 allocates BUF1 as *read only*. DSP2 allocates a memory buffer (BUF2) for its exclusive use. Figure 36 shows for DSP1 and DSP2 the code snippets that perform the dynamic memory allocations. Please note that the values assigned to BUF1 and BUF2 are arbitrary values. Also, note that the value assigned to BUF1 (0x10) is the same for both DSP1 and DSP2 because it represents a shared memory page. For more information regarding the data types SYS_PARTITION, SYS_MEM and SYS_ERROR please refer to [43]. □

Table 8: Modifications done to Atalanta.

Item	**Orginal Atalanta Memory Mgmt.**	**New Atalanta Memory Mgmt.**
Number of Mem. Mgmt. Fn.	4	7
Number of C lines	330	450
SoCDMMU Device Driver	N	Y
Inlined Assembly Lines	0	10

Table 8 shows the modifications done to the Atalanta (in terms of the number of source lines and the number of functions) to add SoCDMMU support to Atalanta. The ten inlined assembly lines are used to access the SoCDMMU to send commands or read the status.

6.4 Operating Systems SoCDMMU Support Guidelines

In the previous sections, we discussed the modifications done to the Atalanta RTOS to support the SoCDMMU. In this section, we will give some guidelines to follow in order to add SoCDMMU support to any operating system.

6.4.1 Adding Low Level Functions to Access the SoCDMMU (Device Driver)

In order to be able to communicate with the SoCDMMU to send commands and receive resposes from the SoCDMMU, a set of new functions (usually not programmer accessible) need to be introduced. These new functions act as a device driver for the SoCDMMU. The new functions should allow the operating system to perform the following:

- Request global memory allocation from the SoCDMMU by sending *G_alloc* commands to the SoCDMMU.
- Deallocate global memory by sending *G_dealloc* command to the SoCDMMU.
- Re-map memory blocks in the virtual address space by sending *G_move* to the SoCDMMU.

Figure 37 shows a code snippet from the device driver showing how to send a command to the SoCDMMU that is mapped at the address 0x10000000 (code shown is in ARM assembly).

```
#define    SOCDMMU_PORT    #0x10000000    /* The SoCDMMU i/o Address */
.
.
 __asm {MOV r2,cmd1}
 __asm {MOV r1,SOCDMMU_PORT}
 __asm {STR r2,[r1,#0]}
 __asm {MOV r2,cmd2}
 __asm {STR r2,[r1,#0]}

.
.
```

Figure 37: Code snippet showing how to send a command to the SoCDMMU using a memory mapped i/o port.

6.4.2 Adding Heap Resizing Functions

The SoCDMMU only handles Level Two (as described in Chapter 3) memory allocation/deallocation. Level Two allocation/deallocation can be seen in most operating systems as heap resizing (size increasing or decreasing). Most operating systems that support Level One memory allocations in the heap have functions that can increase the size of the heap. Usually Level One memory allocators call these heap resizing functions if there is not enough space in the heap. For example, in a UNIX-like operating system, memory is managed using the *malloc()* and *free()* functions. If the heap is highly occupied, *malloc()* calls *sbrk()* function to increase the size of the heap [13].

To support the SoCDMMU, the operating system has to be modified so that the heap resizing function (e.g., *sbrk()* in UNIX-like operating systems) calls the SoCDMMU to allocate global memory blocks and remap them in the virtual address space to increase the size of the heap. In order to use the global memory in an efficient way, a new function should be added to decrease the size of the heap by calling the SoCDMMU to deallocate global memory blocks.

6.4.3 Adding Shared Memory Allocation/Deallocation Support

The SoCDMMU supports the allocation of different types of memory (exclusive, read only, read/write) as described in Chapter 3. Level One memory allocation functions (e.g., *malloc()*) need to be modified in a way to support the allocation mode. This can be done by either of the following:

- Add a new argument to Level One functions to indicate the allocation mode (exclusive, read only or read/write); or
- Have multiple versions of the same Level One functions each for a particular allocation mode.

6.5 Summary

In this chapter, we showed how to modify an RTOS to add support to the Two-Level memory management hierarchy and the SoCDMMU. Also, we gave general guidelines to help adding the SoCDMMU support to any operating system. In the next chapter, we will show some experimental results we conducted to show the benefits of using the SoCDMMU in multiprocessor SoCs.

CHAPTER VII

EXPERIMENTS AND RESULTS

In this chapter, we show the results obtained from a set of experiments. The first experiment is used to measure the execution time (in cycles) of different SoCDMMU commands using a co-simulation environment that is totally done in Verilog. In the second experiment, we compare the hardware SoCDMMU with the SoCDMMU implemented using a general purpose microcontroller. Finally, we show set of experiments we performed to compare a shared memory multiprocessor SoC that utilizes the SoCDMMU to the same multiprocessor SoC without the SoCDMMU.

7.1 SoCDMMU Command Execution Times

To test the effectiveness of our approach, we first simulated a model for an SoC that utilizes an SoCDMMU using Synopsys VCS^{TM} Verilog simulator. The simulated system looks like the system illustrated in Figure 38. The system has four PEs (two in-house developed MIPS like RISC processors and two CMU DSPs [9]), 16MB SRAM and the SoCDMMU. The memory is divided into 256 blocks; each block is

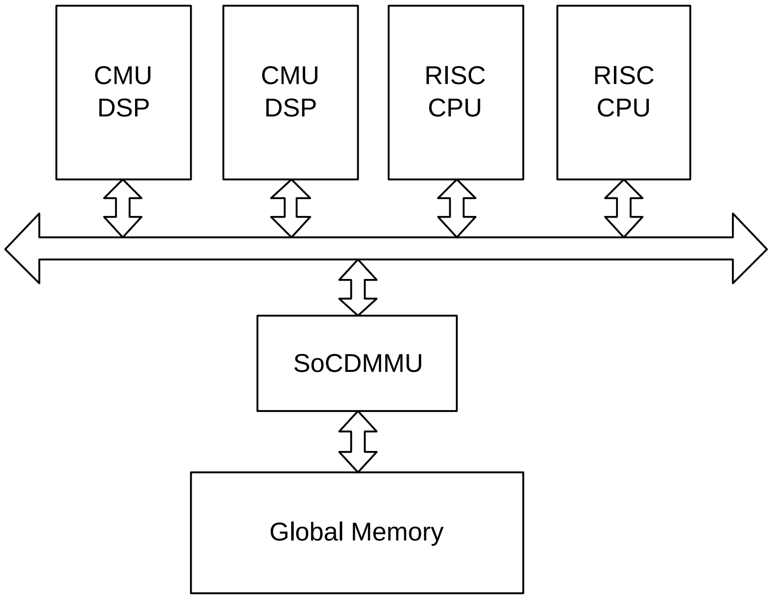


Figure 38: SoC with 2 RISC CPUs, 2 DSPs, SoCDMMU and memory.

64KB. The SoCDMMU utilizes a 256-bit *Allocation Vector* and an *Allocation Table* with 256 entries.

In this experiment we measured the worst-case execution time of the SoCDMMU commands. Table 9 summarizes the results. Please note that *G_alloc_ro* command executes in 3 cycles if the requested *page* was allocated as *read/write* by other PE and 4 cycles if not. We found that the worst-case execution time occurs when all PEs issue commands at the same time. The worst case execution time is 16 clock cycles.

Table 9: Execution Times in Cycles.

Command	**Number of Cycles**
G_alloc_ex	4
G_alloc_rw	4
G_alloc_ro	3 or 4
G_dealloc	4
Worst-Case Execution Time	16

7.2 Comparison with a Microcontroller Implementation

To demonstrate the importance of building the SoCDMMU in custom hardwired unit, we compared the hardwired SoCDMMU performance with the performance of SoCDMMU that uses software running on RISC microcontroller (Microchip PIC^{TM} microcontroller) to implement the functionalities of the BASIC SoCDMMU (as described in Chapter 4). Table 10 compares the worst-case execution time of the hardware SoCDMMU with the best-case execution time for the microcontroller implementation of the SoCDMMU in software. The comparison is shown in clock cycles. We assume that the hardware SoCDMMU and the microcontroller both have the same clock rate. As Table 10, the hardwired SoCDMMU is more than 10X faster than the SoCDMMU that uses software running on a general purpose microcontroller.

Table 10: A comparison between the SoCDMMU and Microcontroller E.T.

SoCDMMU Worst-Case Execution Time	16 Cycles
Microcontroller Best-Case Execution Time	221 Cycles

7.3 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 to (ii) a fully shared-memory multiprocessor system. For the comparison with the SoCDMMU, we used two memory managers: the ARM Software Development Toolkit (SDT) [4] embedded memory heap manager [5] and the uClibc embedded C library memory manager [46].

7.3.1 Simulation Setup

The simulation is carried out using the Mentor Graphics Seamless CVE^{TM} Co-simulation environment [24]. For hardware simulation, we used Synopsys VCS^{TM} Verilog simulator. For software emulation, we used the XRAY debugger [25]. We used ARM SDT v2.5 [4] and the GNU C Compiler (*GCC*) [12] for software development.

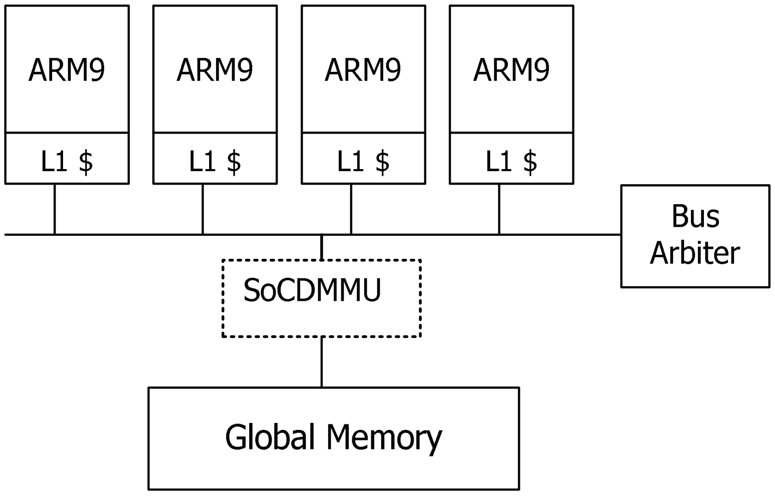


Figure 39: Four-PE SoC with an SoCDMMU.

The simulated system shown in Figure 39 consists of four ARM9TDMI cores each of which has Level one (L1) data and instruction caches each of 64KBytes. All four PEs share a global bus. A shared memory of 16 MBytes of SRAM is connected to the same bus. We assume it takes five cycles to get the first word from the global memory in Figure 39. The bus arbiter shown in Figure 39 controls the access of the cache controllers to the memory. The system (including the SoCDMMU) is clocked at 150MHz.

7.3.2 Speedup of a single *malloc()* or *free()*

Table 11: E.T. of *malloc()* and the SoCDMMU Allocation (Atalanta API)

	Execution Time (Average Case)	**Execution Time (Worst Case)**
SDT2.5 embedded malloc()	106 cycles	559 cycles
uClib *malloc()*	222 cycles	1646 cycles
SoCDMMU allocation	28 cycles	199 cycles
Speed up over SDT *malloc()*	3.78X	2.8X
Speed up over uClib *malloc()*	7.92X	8.21X

Table 11 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 (note that, if desired, the Atalanta RTOS could be rewritten to have *malloc()* and *free()* functions instead of *asc_partition_create()*, *asc_partition_gain()* and *asc_partition_free()* functions). As Table 11 shows, memory allocation done by the SoCDMMU is faster than memory allocation using the *malloc()* function.

Table 12: E.T. of *free()* and the SoCDMMU Deallocation (Atalanta API)

	Execution Time (Average Case)	**Execution Time (Worst Case)**
SDT2.5 embedded free()	83 cycles	186 cycles
uClib *free()*	208 cycles	796 cycles
SoCDMMU deallocation	14 cycles	28 cycles
Speed up over SDT *free()*	5.9X	6.64X
Speed up over uClib *free()*	14.8X	28.42X

Table 12 compares the execution time of the *free()* function in cycles to that of the *asc_partition_free()* that utilizes the SoCDMMU and shows that memory deallocation

Table 13: 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

using the SoCDMMU is up to 28X faster than using the standard *free()* function.

7.3.3 Speedup during transition from the MPEG-2 player to the OFDM receiver

In this experiment we assume that the SoC in Figure 39 is used for a handheld device like the that used in the fictitious story described in [27]. That device was able to download a medical record, make a phone call and play a video clip. The handheld device can be used for OFDM communication as well as other personal applications (e.g., a video player). Table 13 shows the memory requirements for the MPEG-2 video player [39] and the OFDM receiver [32]. We assume that other applications take up 13.9 MBytes leaving 2.1 MBytes available for the OFDM receiver or the MPEG-2 player (depending on which is running). During the transition from the MPEG-2 player to the OFDM receiver, six memory deallocations and seven memory allocations are executed.

From the results, shown in Table 14, we can see that using the SoCDMMU yields a 4.4X improvement over the SDT2.5 embedded memory manager and, as shown in

Table 14: Memory Management E.T. Comparison (SoCDMMU vs ARM SDT2.5)

	Using the SOCDMMU	**Using uClib *malloc()*and *free()***	**Speedup**
Average Case	280 cycles	1240 cycles	4.4X
Worst Case	1244 cycles	4851 cycles	3.9X

Table 15, 9.26X over uClibc memory management functions in average case execution time. In worst-case execution time, on the other hand, Table 14 shows that using the SoCDMMU yields a 3.9X improvement over the SDT2.5 embedded memory manager while Table 15 shows a 12.46X improvement over the uClibc memory management functions.

Table 15: Memory Management E.T. Comparison (SoCDMMU vs uClibc)

	Using the SOCDMMU	**Using uClib *malloc()*and *free()***	**Speedup**
Average Case	280 cycles	2593 cycles	9.26X
Worst Case	1244 cycles	15502 cycles	12.46X

For worst-case execution calculation times we assumed 100% misses in L1 caches. The Atalanta RTOS memory manager uses unoptimized data structures which are accessed excessively during the execution of *asc_partition_create()* and *asc_partition_free()* functions; the ARM SDT2.5 memory manager, on the other hand, uses a more efficient data structure which is not accessed as often. Since Atalanta RTOS memory management functions perform more memory accesses than the equivalent functions of the ARM SDT2.5 memory manager, the speedup improvement in the worst-case is less than that of the average-case (when comparing Atalanta memory management functions that use the SoCDMMU to equivalent functions of the ARM SDT2.5).

7.3.4 Speedup during application run-time

In this experiment, we used the same system described in Section 7.3.1 except we used *GCC* [12] and *glibc* [13] instead of ARM SDT v2.5 and *uclibc*. We simulated several benchmarks taken from the SPLASH-2 [42] application suite: Blocked LU Decomposition (LU), Complex 1D FFT (FFT) and Integer Radix Sort (RADIX) [50]. We measured the execution time of each benchmark. The selected benchmarks initially used static memory allocation which is unsuitable for mobile applications with limited memory resources and tens of applications that reside in main memory. The selected benchmarks source files were modified to replace all the static memory arrays by arrays that are dynamically allocated at run time and deallocated upon completion. In this way, the benchmarks could be dynamically downloaded and run on a handheld device, which is the kind of ability we want this research to enable and make more practical.

Table 16: E.T. of some SPLASH-2 Benchmarks using *glibc malloc()* and *free()*

Benchmark	**E.T. (Cycles)**	**Memory Management E. T. (Cycles)**	**% of E. T. used for Memory Management**
LU	318307	31512	9.90%
FFT	375988	101998	27.13%
RADIX	694333	141491	20.38%

Table 16 shows the execution time of the benchmarks in cycles and the total number of cycles consumed in memory management when the benchmarks use a

Table 17: E.T. of some SPLASH-2 Benchmarks using the SoCDMMU

Bench-mark	**E.T. (Cycles)**	**Memory Mgmt. E. T. (Cycles)**	**% of E. T. used for Memory Mgmt.**	**% Reduction in time used to Manage Memory**	**% Reduction in Benchmark E. T.**
LU	288271	1476	0.51%	95.31%	9.44%
FFT	276941	2951	1.07%	97.10%	26.34%
RADIX	558347	5505	0.99%	96.10%	19.59%

conventional memory allocation/deallocation techniques (*glibc malloc()* and *free()*).

Table 17 shows the same information introduced in Table 16 but with the benchmarks using the SoCDMMU for memory allocation/deallocation. Also, Table 17 shows the reduction in the memory management execution time because of using the SoCDMMU instead of using *glibc malloc()* and *free()* functions. This reduction in the memory management execution time yields speed ups in the benchmarks execution times. As we can see in Table 17, using the SoCDMMU tends to speed up the application execution time and this speed up is almost equal to the percentage of time consumed by conventional software memory management techniques.

7.4 Area Estimation of the SoC

In this section, we try to estimate the total size of the SoC we simulated for our experimental results of Section 7.3. Our goal is to find an estimate of the extra area required to incorporate the SoCDMMU into our candidate SoC. The SoC consists of four ARM9TDMI cores. Each ARM9TDMI has Level 1 (L1) data and instruction caches, each of which is 64KBytes. The SoC has a global on-chip memory of

16MBytes. The global on-chip memory is divided into 256 *G_block*s each of 64KBytes.

Table 18: SoC Area

Element	**Number of Transistors**	%
4 ARM9TDMI Cores	4 x 112K = 448K Transistors	
4 L1 Caches (64KB+64KB)	4 x 6.5M = 26M Transistors	
Global On-Chip Memory (16MB)	134.217M Transistors	
SoCDMMU (w/o memory elements)	30K Transistors	
SoCDMMU *Allocation Table*	30K Transistors	
SoCDMMU *Address Converters* (4)	4 x 60K = 240K Transistors	
SoCDMMU (total)	**300K Transistors**	
SoC (total)	**160.965M Transistors**	
SoCDMMU to 4 ARM9 & 4 L1 $		1.134%
SoCDMMU w/o mem. elements to SoC		0.0186%
SoCDMMU to SoC		**0.186%**

Table 18 shows the sizes of the different SoC components expressed in number of transistors. The four ARM9TDMI cores consume 448K transistors [33]. Assuming 6-Transistor (6T) SRAM cells [16], the eight Level 1 caches consume 26 Million transistors. The global on-chip 16MB DRAM consumes approximately 134 Million transistors assuming 1-transistor DRAM cells [16]. From Table 3, the SoCDMMU non-memory elements consume approximately 7500 2-input NAND gates, each of which consists of four transistors; this totals 30K transistors. From Table 4, the SoCDMMU memory elements (one *Allocation Table* and four *Address Converter*s) area is approximately equivalent to the area of 5.5KB 6T SRAM, which consumes approximately 270K (5.5K x 8 x 6) transistors. The SoCDMMU, in total, consumes 300K transistors while the SoC consumes almost 161 million transistors, which means that the SoCDMMU consumes only 0.186% of the total number of transistors of the

SoC. Thus, for a tiny increase in area, the SoCDMMU improves memory management performance by 4-12X, a significant improvement for general purpose applications and a potentially critical improvement for real-time applications as well.

Please note that we did not do a full chip layout for this chip area estimate. For example, we do not have the layout of the ARM9TDMI available, nor do we have a memory compiler available to generate more exact memory layouts.

7.5 Summary

In this chapter we showed experimental results from a multiprocessor SoC that utilizes the SoCDMMU. We showed an example where our approach gives up to 9.2X overall speedup in the average case execution time of memory management during application transition time when compared to a fully shared memory system with the same memory organization and number of processors. Also, we showed speedups of 10%-26% in application execution time. Thus, for a tiny increase in area (0.186% of 161M transistor SoC), the SoCDMMU improves memory management performance.

CHAPTER VIII

CONCLUSION

In this thesis, we presented an approach to handle on-chip memory allocation between PEs in a heterogeneous multiprocessor SoC. Our approach is based on an SoCDMMU that provides a dynamic, fast way to allocate/deallocate global on-chip memory (L2). Moreover, when implemented in hardware, the SoCDMMU allocation/deallocation of the memory blocks is completely deterministic, which makes it suitable for real-time SoC applications. The introduction of the SoCDMMU introduces a new memory management hierarchy we call Two-Level Memory Management. Level Two, in Two Level Memory Management, is the management of the global on-chip memory among the on-chip processing elements. Level One, on the other hand, is the management of memory allocated to a particular on-chip Processing Element, e.g., an operating system's management of memory allocated to a particular processor. We gave the details of a possible hardware implementation of the SoCDMMU which implements management of Level Two memory. We showed how an RTOS might be adapted to support the SoCDMMU. Also, we introduced a tool to automatically customize the hardware SoCDMMU according to the user's input. We showed experimental results where our approach gives overall speedup of 4.4X to 9.2X in the average case execution time of memory management during application transition when compared to a fully shared memory system with the same memory organization and number of processors. Also, our approach tends to speed up the application execution time and this speed up almost equals to the percentage of time consumed by conventional software memory management techniques. For three benchmarks we showed improvements from 10% to 26%.

REFERENCES

[1] "Special Issue on Distributed Shared Memory Systems," *Proceedings of IEEE*, Vol. 87, No. 3, pp. 397–532, 1999.

[2] Ackland, B., "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, 2000.

[3] ARC International. http://www.arc.com.

[4] ARM Limited, *ARM Software Development.* http://www.arm.com/devtools/.

[5] ARM Limited, *Embedded Heap Management, ARM Application note 58.* http://www.arm.com/armwww.nsf/html/Application_Notes?OpenDocument.

[6] Artisan Components, Inc., *Artisan memory generators.* http://www.artisan.com/products/memory.html.

[7] Bruyninckx, H., "Real-Time and Embedded Guide," http://people.mech.kuleuven.ac.be/~bruyninc/rthowto/.

[8] Cam, H., Abd-El-Barr, M., and Sait, S., "A High-Performance Hardware-Efficient Memory Allocation Technique and Design," *Proceedings of International Conference on Computer Design*, pp. 274–276, October 1999.

[9] Carnegie Mellon University, *CMU Low Power Group Benchmarks.* http://www.ece.cmu.edu/lowpower/benchmarks.html.

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

[11] Chang, J., Srisa-an, W., Lo, C., and Gehringer, E., "DMMX: Dynamic Memory Management Extensions," *Journal of Systems and Software*, Vol. 63, No. 3, pp. 187–199, 2002.

[12] The Free Software Foundation, The GNU Project, *The GCC Compiler.* http://gcc.gnu.org/.

[13] The Free Software Foundation, The GNU Project, *The GNU C Library.* http://www.gnu.org/manual/glibc-2.2.5/html_node/.

[14] Integrated Systems Inc., *pSOS System Concept [Users Manual]*, 1996.

[15] Isoda, S., Goto, E., and Kimura, I., "An Efficient Bit Table Technique for Dynamic Storage Allocation of 2^n-word Blocks," *ACM Communications*, Vol. 7, pp. 580–592, September 1971.

[16] Itoh, K., *VLSI Memory Chip Design.* New York, NY: Springer-Verlag Telos, 2001.

[17] Knowlton, K., "A Fast Storage Allocator," *ACM Communications*, Vol. 8, No. 10, pp. 623–625, 1965.

[18] Knuth, D., *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms.* Redwood City, CA: Addison Wesley Longman Publishing Co., Inc., 1997.

[19] Kozyrakis, C., "Scalable processors in the billion-transistor era: IRAM," *IEEE Computer*, Vol. 30, No. 9, pp. 75–78, 1997.

[20] Labrosse, J., *MicroC/OS-II, The Real-Time Kernel.* Manhasset, New York: CMP Books, 2002.

[21] LEDA Systems. http://www.qualcorelogic.com/.

[22] Levy, M., "Chip Combines Four 1GHz Cores," *Microprocessor Report*, pp. 12–14, October 2002.

[23] Li, Y. and Wolf, W., "Hardware/Software Co-Synthesis with Memory Hierarchies," *IEEE Transaction Computer-Aided Design of Integrated Circuits and Systems*, Vol. 18, No. 10, pp. 1405–1417, 1999.

[24] Mentor Graphics Corporation, *Seamless Hardwarre/Software Co-Verification.* http://www.mentor.com/seamless/.

[25] Mentor Graphics Corporation, *XRAY Debugger.* http://www.mentor.com/xray/.

[26] Mentor Graphics Corporation, *VRTXoc User's Guide and Reference*, 1999.

[27] Morgan, S., "Jini to the rescue," Vol. 37, No. 4, pp. 44–49, 2000.

[28] Panda, P. R., Dutt, N. D., and Nicolau, A., "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, 1997.

[29] Puttkamer, E., "A Simple Hardware Buddy System Memory Allocator," *IEEE Transaction on Computers*, Vol. 24, No. 10, pp. 953–957, 1975.

[30] Redhat, *eCos Reference Manual*, 2003. http://sources.redhat.com/ecos/docs-latest.

[31] Rompaey, K., Verkest, D., Bolsens, I., and Man, H., "Coware - a design environment for heterogeneous hardware/software systems," *Proceedings of the EURO-DAC'96 European Design Automation Conference with EURO-VHDL*, pp. 357–386, October 1996.

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

[33] Segars, S., "The ARM9 Family - High performance Microprocessors for Embedded Applications," *Proceedings of the International Conference on Computer Design*, pp. 230–235, October 1998.

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

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

[36] Shalan, M., Shin, E., and Mooney, V., "DX-Gt: Memory Management and Crossbar Switch Generator for Multiprocessor System-on-a-Chip," *Proceedings of the 11th Workshop on Synthesis And System Integration of Mixed Information technologies*, pp. 357–364, April 2003.

[37] Shalan, M. A. and Mooney, V. J., "Hardware Support for Real-Time Embedded Multiprocessor System-on-a-Chip Memory Management," Tech. Rep. GIT-CC-03-02, Institute of Technology, Atlanta, Georgia, 2003.

[38] Shin, E. S., *Automated Generation of Round-robin Arbitration and Crossbar Switch Logic.* PhD thesis, School of Electrical and Computer Engineering, Georgia Institute of Technology, USA, Novembre 2003.

[39] Soderquist, P. and Leeser, M., "Memory Traffic and Data Cache Behavior of an MPEG-2 Software Decoder," *Proceedings of the International Conference on Computer Design*, pp. 417–422, October 1997.

[40] Srisa-an, W., Lo, C., and Chang, J., "A Hardware Implementation of Realloc Function," *Proceedings of the IEEE Annual Workshop on VLSI*, pp. 106–111, April 1999.

[41] Srisa-an, W., Lo, C., and Chang, J., "A Performance Analysis of the Active Memory Module (AMM)," *Proceedings of IEEE International Conference on Computer Design*, pp. 493–496, September 2001.

[42] Stanford University, *Stanford Parallel Applications for Shared Memory (SPLASH).* http://www-flash.stanford.edu/apps/SPLASH/.

[43] Sun, D., Blough, D. M., and Mooney, V. J., "Atalanta: A new multiprocessor rtos kernel for system-on-a-chip applications," Tech. Rep. GIT-CC-02-19, Institute of Technology, Atlanta, Georgia, 2002.

[44] Synopsys, Inc, *Synopsys Logic Synthesis.* http://www.synopsys.com/products/logic/logic.html.

[45] Tensilica, Inc. http://www.tensilica.com.

[46] uClibc, *A C library for embedded systems.* http://www.uclibc.org/.

[47] Verkest, D., "Matisse: a system-on-chip design methodology emphasizing dynamic memory management," *Journal of VLSI Signal Processing*, Vol. 21, No. 3, pp. 219–232, 1999.

[48] VPP, *Verilog PreProcessor.* http://www.surefirev.com/vpp/.

[49] Wilson, P., Johnstone, M., Neely, M., and Boles, D., "Dynamic Storage Allocation: A Survey and Critical Review," *Proceedings of the International Workshop on Memory Management*, pp. 1–78, Sept. 1995.

[50] Woo, S., Ohara, M., Torrie, E., Singh, J., and Gupta, A., "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 24–36, June 1995.

[51] Wuytack, S., Silva, J., Cathoor, F., Jong, G., and Ykman, C., "Memory Management for Embedded Network Applications," *IEEE Design and Test of Computers*, Vol. 18, No. 5, pp. 533–544, 1999.

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

[53] Zorn, B., "The Measured Cost of Conservative Garbage Collection," *Software-Practice and Experience*, Vol. 32, No. 7, pp. 733–756, 1993.