EVALUATION SEARCH OPERATION FOR TRADITIONAL PROCESSOR AND A DIFFERENT MEMORY PROCESSOR FOR GARBAGE COLLECTION

Viral V Kapadia\textsuperscript{1}, Dr. V K Thakar\textsuperscript{2}

\textsuperscript{1}Computer Science & Engg Dept, The M S University of Baroda
\textsuperscript{2}Head & Professor, A D Patel Institute of Technology

ABSTRACT

The major issue in today’s world of computing is improvement in performance of memory. The major issue at processor level is to identify the data block or data that is required for processing either address or data, which is stored in memory. The data needs to be fetched from memory (RAM) to register and vice versa. Hence the performance of the processor depends on the searching time taken to allocate or deallocation of data in memory for processing. Here we are trying to measure the searching time when we use co-processor for garbage collection than the traditional processor that performs memory operation.

Keywords: VHDL, Verilog, Garbage Collection, Processor, Clock Cycle, Delay, Xilinx.

I. INTRODUCTION

Memory management is one of the most vital researched areas in computer engineering since their goals are to understand and improve the usage of the two most crucial resources in virtually every computer system memory and processor. In both research areas, work has been done in many directions including theory, hardware, algorithms, and implementation and so on. A typical methodology is to investigate issues in only one area, assuming the system’s behaviour in the other area has no influence at all, or at most, very little influence on the issues under investigation. Although such a methodology is very successful in terms of reducing research scope and simplifying the targeted issues, it is not sufficient to reflect the system level nature of processor and memory usage, which is that they interact and even conflict with each other. Therefore, the two dimensions of time and space must be considered together in many cases so that the result will be a well designed system that has acceptable performance in both dimensions, rather than being extremely good in one dimension but unacceptable in the other one. This is the famous time-space tradeoff [1].
Memory management on which researchers are mainly focusing are the different techniques related to memory management i.e. garbage collection techniques, scheduling, real time system, user oriented design and many more. The major problem found out in memory management is processor and operating system delays. We don’t have good performance in Real time system since we are using the software approach which ultimately depends on operating system. So virtually we try to say we are having real-time memory management but physically not performing the intended task more convincingly. Here we are trying to design a hardware design that can perform all memory management tasks with additional power of VLIW instruction word and parallel processing.

Real-time requirements are far and wide in modern applications, from multimedia players to communication controls, aviation systems, safety critical control systems, etc. Garbage collection is widely acknowledged to speed up software development while increasing security and reliability. However, garbage collectors that support real-time applications are notoriously hard to build. Traditional garbage collectors stop the application to perform the entire collection or even just to initiate or finish up the collection. Stopping all threads creates a computation pause which is unacceptable for real-time systems [1].

Such atomic operations create ordering constraints that do not leave much room for reordering of memory accesses. Thus, a more sophisticated compaction mechanism must be used. As part of our real-time system, we want to provide a novel concurrent collector, which provides a solution to the above challenge.

Real-time requirements are far and wide in modern applications, from multimedia players to communication controls, aviation systems, safety critical control systems, etc. Garbage collection is widely acknowledged to speed up software development while increasing security and reliability. However, garbage collectors that support real-time applications are notoriously hard to build. Traditional garbage collectors stop the application to perform the entire collection or even just to initiate or finish up the collection. Stopping all threads creates a computation pause which is unacceptable for real-time systems [1].

Classical real-time systems typically operate with a certain amount of slack. If there is no slack, then the system can’t guarantee to meet its bounds. But interactive systems may become sluggish during bursts of other activity on the machine, and in some cases this is the desired behaviour. Queued systems may absorb temporary load spikes and tolerate a certain fraction of outliers depending on a service-level agreement [1].

Such, atomic operations create ordering constraints that do not leave much room for reordering of memory accesses. Thus, a more sophisticated compaction mechanism must be used. As part of our real-time system, we want to provide a novel concurrent collector, which provides a solution to the above challenge.

To measure responsiveness of our collector, we propose a mutator responsiveness measure, attempting to check how responsive the system is for events that get generated at a high frequency. Currently there is no Distributed Object Store that Implement an Algorithm of Distributed Garbage Collectors that can do processor management and provide adaptive garbage collection. We have to use different garbage collection techniques for different environment. There is partly memory allocation and deallocation Implementation of the System function which doing memory allocation and deallocation that can be optimize in our implementation of algorithm.

II. GARBAGE COLLECTION

One of the potential disadvantages of garbage collection compared to the explicit freeing of objects is that garbage collection gives programmers less control over the scheduling of CPU time devoted to reclaiming memory. It is in general impossible to predict exactly when (or even if) a garbage collector will be invoked and how long it will take to run. Because garbage collectors
usually stop the entire program while seeking and collecting garbage objects, they can cause arbitrarily long pauses at arbitrary times during the execution of the program. Such garbage collection pauses can sometimes be long enough to be noticed by users. Garbage collection pauses can also prevent programs from responding to events quickly enough to satisfy the requirements of real-time systems. If a garbage collection algorithm is capable of generating pauses lengthy enough to be either noticeable to the user or make the program unsuitable for real-time environments, the algorithm is said to be disruptive. To minimize the potential disadvantages of garbage collection compared to the explicit freeing of objects, a common design goal for garbage collection algorithms is to minimize or, if possible, eliminate their disruptive nature [1].

The process of distributed garbage collection can be categorized by two approaches:

A. Software approach

The software approach for garbage collection is very famous and it is implemented in many of the application software such as JAVA, .Net etc. the main disadvantage of this system is works under the control of operating system and for each and every instruction operating system calls the main processor through any interrupts. As whenever there is a requirement of memory in application, it runs the garbage collection process and in that process also the main processor is interrupted. The main processor is wasting its time in garbage collection where as other process are waiting for processor. The software approach is easy to implement compare to hardware approach. At the same time is also slower compare to hardware approach. We will discuss the software approach in detail.

A widely accepted method to control software complexity and to increase software quality is automatic dynamic memory management, also referred to as garbage collection. Unfortunately, it is difficult to implement garbage collection for real-time systems. Because of high overhead and unpredictable pauses, most embedded system designers still consider garbage collection an unaffordable luxury. Consequently, they have to resort to manual memory management with the well-known problems: Freeing memory too early causes “dangling references”, while freeing memory too late or not at all gives rise to memory leaks. But worst of all, manual memory reclamation requires the programmer’s global view of a software system and contradicts modularization [5].

A widely accepted method to control software complexity and to increase software quality is automatic dynamic memory management, also referred to as garbage collection. Unfortunately, it is difficult to implement garbage collection for real-time systems. Because of high overhead and unpredictable pauses, most embedded system designers still consider garbage collection an unaffordable luxury. Consequently, they have to resort to manual memory management with the well-known problems: Freeing memory too early causes “dangling references”, while freeing memory too late or not at all gives rise to memory leaks. But worst of all, manual memory reclamation requires the programmer’s global view of a software system and contradicts modularization [5].

Real-time applications require a small upper bound on any pause the garbage collector might cause. Garbage collection in software, however, faces an inevitable trade-off between the granularity of garbage collection (i.e. the maximum pause length) and the code-size and runtime overhead caused by synchronization. To prevent this overhead from becoming unjustifiably high, software collectors have to rely on indivisible operations such as processing an entire object or the complete root set [4]. Since the duration of these operations is potentially unlimited, these software collectors are not suited for hard real-time environments [5].
B. Hardware approach

Following Programming Tools are available for hardware designing.

1) **VHDL**: VHDL (VHSIC hardware description language; VHSIC: very-high-speed integrated circuit) is a hardware description language used in electronic design automation to describe digital and mixed-signal systems such as field-programmable gate arrays and integrated circuits. VHDL is a fairly general-purpose language, and it doesn't require a simulator on which to run the code. There are many VHDL compilers, which build executable binaries. It can read and write files on the host computer, so a VHDL program can be written that generates another VHDL program to be incorporated in the design being developed. Because of this general-purpose nature, it is possible to use VHDL to write a test bench that verifies the functionality of the design using files on the host computer to define stimuli, interacts with the user, and compares results with those expected.

   It is relatively easy for an inexperienced developer to produce code that simulates successfully but that cannot be synthesized into a real device, or is too large to be practical. One particular pitfall is the accidental production of transparent latches rather than D-type flip-flops as storage elements.

   The key advantage of VHDL when used for systems design is that it allows the behaviour of the required system to be described (modeled) and verified (simulated) before synthesis tools translate the design into real hardware (gates and wires).

   Another benefit is that VHDL allows the description of a concurrent system (many parts, each with its own sub-behaviour, working together at the same time). VHDL is a Dataflow language, unlike procedural computing languages such as BASIC, C, and assembly code, which all run sequentially, one instruction at a time.

   A final point is that when a VHDL model is translated into the "gates and wires" that are mapped onto a programmable logic device such as a CPLD or FPGA, then it is the actual hardware being configured, rather than the VHDL code being "executed" as if on some form of a processor chip.

![](Figure 1: Coprocessor Architecture)
III. IMPLEMENTATION

In digital systems encompasses a range of systems from low-level components to complete system-on-a-chip and board-level designs. If we are to encompass this range of views of digital systems, we must recognize the complexity with which we are dealing. The most important way of meeting this challenge is to adopt a systematic methodology of design. If we start with a requirements document for the system, we can design an abstract structure that meets the requirements. We can then decompose this structure into a collection of components that interact to perform the same function.

Each of these components can in turn be decomposed until we get to a level where we have some ready-made, primitive components that perform a required function. The advantage of this methodology is that each subsystem can be designed independently of others. We use the term model to mean our understanding of a system. The model represents that information which is relevant and abstracts away from irrelevant detail. The implication of this is that there may be several models of the same system, since different information is relevant in different contexts. In our design we have define the CMS.vhd for advance code morphing software and ALU.vhd that performs the task of arithmetic and logical unit and various required data manipulation functionalities.

![Figure 2: DSM using shared variables](image)

Distributed shared memory management modelling scenario presents distributed shared memory management techniques and their performance impact for multicore environments. By removing imbalance between processor and memory performance it becomes more important to optimize the shared memory management features to obtain the maximum possible performance out of the memory subsystem. It has consideration that no of processors that communicating through shared variables. Here, figure 2 below shows the view of shared variable used concurrently by three processors P_1, P_2 and P_3 to manipulate the data in the system. The system can have more no of processor and large shared memory size depends on the requirements.

System Architecture

In distributed shared memory systems processes share data transparently across node boundaries; data faulting, location, and movement is handled by the underlying system. While the performance implications cannot be ignored, the advantages of the shared memory programming model are well known:

Programming with shared memory is a well-understood problem.
Shared memory programs are usually shorter and easier to understand than equivalent message passing programs.

Consumes less design time compared to others
Large or complex data structures may easily be communicated.
Shared memory gives transparent process-to-process communication.
Cheaper to implement system model using shared memory

IV. RESULTS

Figure 3: Simulation window consisting of Input and Output

The objective over here is to prove the difference of efficiency of memory management at the processor level and traditional approach. Here we have taken an 8 bit processor to prove the same. Figure 3 is the snapshot of processor that we have simulated in Xilinx ISE 9.2. It won’t make a difference for the higher level of processors also as we are providing the comparison between an 8 bit traditional processor and 8 bit memory processor. The same is to implied for the higher level of processors also. The first row in figure 3 indicate the CLK (clock) for the processor. Processing is done after the delay of 3micro seconds to let the signal stabilize.

D0 to D4 stands for the data / address lines. S0, S1 and S2 are control signals. Hence we have 8 instructions set.

I0, I1, I2, I3, I4 and Y are output signals used for checking various operations performed by the processor as shown in the figure 3.

<table>
<thead>
<tr>
<th>Experiment No</th>
<th>Memory status (Occupied)</th>
<th>Memory Block to be allocated</th>
<th>Search in Traditional Manner</th>
<th>Search with Coprocessor</th>
<th>% Time cycles Improved</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25%</td>
<td>1</td>
<td>16</td>
<td>1</td>
<td>16</td>
<td>Linear Memory Allocation</td>
</tr>
<tr>
<td>2</td>
<td>50%</td>
<td>1</td>
<td>32</td>
<td>1</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>75%</td>
<td>1</td>
<td>48</td>
<td>1</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100%</td>
<td>1</td>
<td>64</td>
<td>1</td>
<td>64</td>
<td></td>
</tr>
</tbody>
</table>
V. CONCLUSION

The Best Way to provide trade of between Garbage collection and processor utilization is to have another software coded processor dedicated for Garbage collection which works independently of main processor. Since technological and economical reasons have put a stop to the increase of clock frequency, performance improvements now come in the form of additional parallelism.
In the simulator the check the exact behaviour of the processor the input needs to be in the form of wave form and hence we get a wave form as an output we can map the in the input signal at a particular clock cycle and at the same time we can also observe the output wave form on the same clock cycle that provides us the information of processing of the processor. Output wave form is missing in figure 6 as it would be generated after the processing.

REFERENCES