Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

Improving Memory Management with Hardware-Generated Memory Access Profiles

Sergej Müller

Studienarbeit

Verantwortlicher Betreuer:	Prof. Dr. Frank Bellosa
Betreuender Mitarbeiter:	Dipl.-Inform. Raphael Neider

30.06.2009

Hiermit erkläre ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, den 30.06.2009

Sergej Müller

Abstract

With the rise of new memory technologies such as Flash we expect future systems to rely on a flatter memory hierarchy using multiple heterogeneous memory units in parallel. Operating systems face new challenges in managing these memories in a cost- and energy-efficient way while maximizing performance. We believe that the key to meet these challenges is to leverage the full potential of each memory technology by migrating pages between the different memories according to their current usage.

In this thesis we introduce a new memory profiling architecture that provides the operating system with hardware generated sets of promising candidate pages. The candidates are selected according to a customizable strategy (e.g., least frequently used) and can be employed by the operating system to make a migration decision.

Contents

Chapter 1

Introduction

From an application programmer's point of view the working memory should be indefinitely large and indefinitely fast. Unfortunately, there is no single memory technology available yet that can satisfy the programmer's desire. Hence, modern computing systems rely on a hierarchically organized memory architecture to leverage properties of multiple memory technologies. In fact the working memory is physically fragmented and scattered along the different levels of the memory hierarchy. To free the application programmer from handling different physical memories, virtual memory and caching are used to simulate a large, unified, and contiguous working memory. Most operating systems use paging to implement virtual memory along with some approximation of the *least recently used* (LRU) page replacement algorithm. Once a page is accessed, a *referenced bit* is set by the virtual memory system. By periodically checking and resetting this bit, the operating system is able to approximate the least recently used page.

With respect to improving overall cost and energy efficiency while maximizing performance we believe that more promising page replacing algorithms can be developed by providing the operating system a detailed insight into how the pages are actually used. Our beliefs are motivated by the following trends in technology.

1.1 Memory Hierarchy

Common memory technologies such as SRAM, SDRAM, and hard disk drives differ in properties such as latency, capacity, and price per bit. Faster memory is typically more expensive than slower memory and therefore can only be used to a moderate extent. To obtain a trade-off between these differences, a memory hierarchy with multiple levels is composed. Each level has higher bandwidth and lower latency but also a smaller size than lower levels. The idea is to provide the application programmer with as much cheap memory as possible at the bottom of the hierarchy while keeping access times low by using the fast memory at the top. The key to a hierarchical memory architecture is to recognize the two principles of memory locality [7] which are often referred to as *temporal* and *spatial locality.* Temporal locality addresses the observation that an access to a certain memory location is likely followed by another access to the same location shortly after again. On the other hand, spatial locality addresses the observation that accessing a word at a certain location is likely to be followed by an access to a close-by word in the near future. By transferring blocks of multiple words between the levels of the memory hierarchy, copies of the blocks in higher levels are created and can be leveraged with respect to the principles of locality during the next memory access. Thus, a trade-off between access times and price is attained.

The increasing number of processors in modern computing systems puts high demands regarding the performance of the memory subsystem. Multiple concurrent threads with interleaved memory access operations challenge the principles of memory locality. In addition to that, not only modern hardware, but also modern software technologies affect the principles of locality: The advent of object-oriented programming languages in combination with garbage collection, dynamic languages, and the use of data structures such as hash tables result in chaotic memory reference patterns and raise more questions about the presence of memory locality in modern systems.

LRU exploits temporal locality, but ignores benefits that are possible by also taking spatial locality into consideration [13]. By implementing a *least recently frequently used* (LRFU) page replacement algorithm [8], Lee et al. showed that taking also the access frequency of pages into account can result in up to 30% performance improvement over the basic LRU algorithm.

Therefore, we believe that making replacement decisions based on detailed knowledge about the actual usage of pages may foster further performance improvements.

1.2 Heterogeneous Memory Management

Multiple memory technologies in a single system are not only motivated by the advantages of the memory hierarchy. Memory technologies such as SRAM, DRAM, EEPROM and Flash differ not only in access latency and price, but also in bandwidth, writing properties, persistence, and energy consumption. Emerging non-volatile memory technologies [11] like MRAM, FeRAM or PCRAM may broaden the list of alternatives even more with their own technology dependent characteristics. System architects face a growing challenge in building modern computing systems while combining new memory technologies in an efficient way, especially when it comes to energy efficiency. Energy is becoming an important design consideration while the demand for higher performance still holds.

In order to tackle these demands, we expect the memory subsystem to flatten by supporting more than one memory technology at one hierarchy level. In this way each kind of memory can be employed to its full value and redundant copies of data in the hierarchy can be reduced, resulting in an overall more efficient system design. Small embedded systems tend to avoid memory hierarchies completely to aid real-time requirements and leave the decision where to place application data structures to the programmer. For instance, SRAM is fast in reading and writing properties and therefore perfectly suited for variables and data structures like stacks, that are constantly accessed and modified. On the other hand, SRAM is very cost intensive and therefore, depending on the usage of the data structure, DRAM may be a wiser choice. Another example is program data, where read-only code is frequently accessed in a random manner. Flash memory offers low latencies and good energy efficiency when it comes to read-only data, persistence and nowadays a moderate cost efficiency, and is therefore well suited for frequently used code regions.

In most current embedded systems, the above allocation problem is still solved by the programmer and results in unportable applications, since the memory configuration may differ from system to system. There has been only little work [3, 10] to release the programmer from this kind of burden.

We believe that dynamic approaches, where data (e.g., pages) is migrated to the memory that fits its current usage best, will play an essential role in future systems. However, detailed real-time information about the current usage of data structures is necessary to make effective migration decisions.

1.3 Approach

To improve memory management tasks such as page migration, we provide the operating system with hardware generated sets of promising candidate pages. These sets, which we refer to as *memory access profiles* or *profiles* for short, are generated by a dedicated device attached to the system's memory bus and reflect the current usage of the (physical) memory according to a *profiling policy.* Individual profiling policies can be implemented and may select, for instance, the least often accessed pages or the most often read pages as candidates. The profiles can be accessed by the operating system through memory mapped I/O and then be used to support a replacement decision.

This thesis is organized as follows. Chapter 2 discusses related work and the OpenProcessor platform on which our work is based. Chapter 3 presents our design of a memory profiling facility. Chapter 4 addresses implementation details, while Chapter 5 concludes and discusses future work.

Chapter 2

Background and Related Work

This chapter provides information on the OpenProcessor platform [12], which is being developed as a platform for research on hardware/software co-design with the main focus on interfaces to operating systems and positions our work among present research results.

2.1 OpenProcessor Platform

The OpenProcessor platform is a fully featured computation environment including all significant components of a common standalone computer system. It is entirely written in the hardware description language Verilog and implemented on a FPGA prototyping board. Besides other features the development board has the following highlights that are relevant to our work:

- Xilinx XC4VLX60-FPGA with 160 18 Kibit RAM blocks
- 360 KiB internal SRAM,
- 64 MiB DDR SDRAM, and
- 4 MiB Flash memory.

Figure 2.1 depicts an overview of the OpenProcessor platform. The *central processing unit* (CPU) is a reduced instruction set computer based processor with a custom load/store instruction set architecture. It features a four staged pipeline including load and result forwarding logic. The CPU features a software controlled *translation lookaside buffer* (TLB) to provide virtual memory support. Furthermore the CPU relies on separate instruction and data caches. The CPU is connected to a central bus that provides access to main memory and all of the OpenProcessor devices through memory mapped I/O. All three memory types (external DDR SDRAM, external

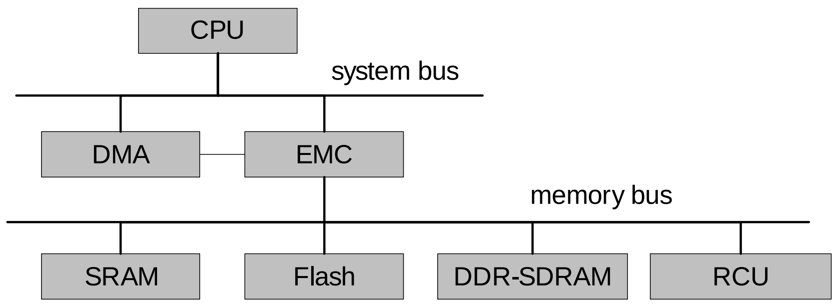

Figure 2.1: Overview of OpenProcessor Platform

Flash, and FPGA-internal SRAM) can be accessed through an *extended memory controller* (EMC) [1] in a uniform manner. All memory requests on the system bus are redirected to a dedicated memory bus by the EMC. Meta data such as the size and type of attached memory can be accessed through the EMC by the operating system using memory mapped I/O. Moreover, an additional *reference counter unit* (RCU) is attached to the memory bus. The RCU is capable of counting the number of memory accesses to a maximum of 512 regions. Size and location of each region are freely configurable by the operating system. A simple software interface is provided for reading the current counter values of individual regions.

Our approach relies on the above architecture, with the exception that we replace the RCU using a more advanced unit. Assume that we want to determine the least used memory region in a system's memory management task. Using the RCU would require the operating system to read all available counter values successively and evaluate them. Since reading one counter value requires one memory read operation, evaluating all counters would induce considerable overhead and affect the performance significantly. Our approach addresses this problem. The new unit is not only capable of counting memory references to configurable memory regions, but also evaluates these counters and generates memory access profiles such as the four least referenced memory regions (e.g., pages). This leads to a more efficient software interface which enables the operating system to determine the least referenced region by only using a small number of operations.

2.2 Related Work

For being able to improve memory management related tasks we need to understand how the memory is actually used. We believe that memory profiling is a key for approaching this problem. Current profiling techniques can be classified into two broad categories: software-based and hardware-based profiling. Common software-based profiling techniques induce additional code to profile desired code regions or instructions and hence not only introduce significant overhead but also change the application's execution behavior. In this section we concentrate on three hardware-based approaches that we believe are worth considering before building a new facility.

2.2.1 PMU Hardware

Many modern microprocessors provide a rich set of processor specific performance counters that may be used in order to aid application performance tuning. The performance counters are implemented inside a dedicated unit often refered to as the *performance monitoring unit* (PMU) and are capable of capturing internal processor events such as the number of elapsed cycles, cache misses, or instructions executed. Several PMU models can count events that indicate traffic between a CPU core and the memory subsystem. In [5], S. Eranian argues that performance counters are the crucial resource to understanding performance issues on today's hardware including x86, PowerPC, and Cell processors. S. Eranian also shows how performance counters can be used to collect interesting metrics related to the memory subsystem such as cache misses, bandwidth and access latencies. A generic kernel interface called perfmon2 [4] for accessing the performance counters of all major processors is also presented in this paper and offers kernel-level sampling buffers and event set multiplexing.

However, the PMU itself does not profile memory regions and is, for instance, not capable of detecting the most used memory region. The PMU can be configured to capture cache or TLB misses. Every captured miss increments a counter. Once a counter saturates, an interrupt is raised, the operating system takes control and evaluates the counter values. This procedure, often referred as to event-based sampling, is inaccurate because only few (commonly four) counters are available at the same time. Furthermore the actual physical address where the miss occurred is hard to determine. The introduced overhead and inaccuracy of event-based sampling makes it unfeasible to profile multiple memory regions efficiently.

PMUs were designed to help the developer to identify the location and cause for performance bottlenecks in an application and thus they are located where the application is executed, namely inside the CPU. They are not capable of capturing memory traffic that was not initiated by the CPU itself. Hence, traffic initiated by external entities such as a DMA controller cannot be captured. In our approach, we require non-intrusiveness and high accuracy for the memory profiling facility.

2.2.2 Frequent Loop Detection

Detecting the most frequently executed regions of code is an increasingly popular method for enabling dynamic software optimizations. In [6] Gordon-Ross et al. introduced a non-intrusive architecture for accurately detecting the most frequently executed loops of a program. The presented architecture is cache-based and resides between the CPU and the L1 memory. A frequent loop cache controller is connected to the address bus together with an additional signal that is asserted whenever a short backwards branch (sbb), a commonly used instruction in loops, is taken. The loop cache controller increments the iteration count of detected loops. The cache is indexed using the loop address. On a cache hit, the iteration count of the loop is read from the cache, incremented, and written back in the next cycle. On a miss, the instruction is added to the cache with a count of one. On a conflict miss, the new address replaces the old address in the cache. To prevent two frequent loops in the same address region from replacing each other, an associative cache is used. The saturation of a counter is handled by dividing all counters inside the cache by two using a simple right shift operation.

An adaptation of this architecture for our purposes sounds promising. We only need to replace the sbb operation trigger with a memory access operation like store or load. DMA operations can also be monitored by attaching the cache controller to the system memory bus. However, while this architecture will be capable of detecting the most frequently used memory regions, it will fail when it comes to detecting the least frequently used regions (a detailed explanation can be found in the design chapter). This is a highly desired feature when it comes to migrating unused system pages to a more economical memory technology. We address this essential feature with the architecture proposed in Chapter 3.

2.2.3 ProMem

Lysecky et al. introduced a new memory hardware-based architecture called ProMem [9] to enable profiling in prototype oriented real-time embedded environments. ProMem's architecture is based on a pipelined binary tree structure. Each tree level is implemented using a separate module and contains 2^{level} counting nodes with $level$ being the current tree level (the root has level $= 0$). ProMem is attached to an address bus, data bus, or a concatenation of both and is capable of handling incoming source patterns with a throughput of one pattern per clock cycle. All counters are configured through a software interface with target patterns. The captured source pattern descends from one tree level to the next using a greater than comparison with the current level's target pattern in order to decide whether to proceed with the left or right sibling. When the captured source pattern hits a tree node that matches the configured target pattern, the counter in this node is incremented. Reading is done by pushing the counter values in breadth-first order out of the tree.

We initially considered to adapt this architecture for our purposes, but encountered some troubles in incorporating the evaluation facility. Evaluating the counters with the intent to determine the least frequently matched patterns would require traversing all counters of all levels. A challenging task facing an architecture with multiple stages and different number of counter values per stage. Furthermore, evaluation must happen in parallel to the monitoring in order not to affect accuracy, but ProMem can only be used either in reading mode or in monitoring mode. Although ProMem can be extended to support ranges of target patterns, the binary tree architecture is based on the assumption that these patterns do not overlap. Overlapping pattern regions may come in handy when monitoring memory in a hierarchical way using different granularities. Our architecture is based on a bus structure that trades flexibility for costs in terms of area. A trade we can afford in our experimental environment.

Chapter 3

Design

As motivated in the introduction of this thesis a detailed understanding about how the memory is actually used is a key for improving memory management in systems with heterogeneous memory technologies. In order to employ usage aware page migration algorithms, a set of promising candidate pages for replacement must be selected.

In this chapter we present a design for a hardware based profiling facility that is capable of generating memory profiles using different profiling policies. In Section 3.1 we discuss the requirements for such a facility. Section 3.2 will analyze the best location for the facility. Section 3.3 will give an overview over the proposed architecture and its components, and the last three sections will discuss the individual components in detail.

3.1 Requirements

In the previous chapter we talked about existing profiling techniques and why they do not comply with our intent to improve memory management. In order to aid the operating system efficiently during memory management tasks such as paging, we believe that the following four requirements must be fulfiled:

1. **Non-intrusiveness**. Our major goal is to assist the operating system in making better decisions about placing pages among different memory technologies and thus improve the system's overall efficiency. In order to sustain the system's performance the profiling facility should not interrupt the system during normal program execution and thereby introduce changes in execution behavior that may result in additional run-time overhead. Thus, we demand that our aiding facility must operate without the need to interrupt the running system.

2. **Accuracy.** Preferably, the profiling facility should be completely accurate to enable potential page migration in real-time. Especially when

it comes to page migration with respect to energy efficiency, the quality of migrating algorithms depends on the accuracy of the generated profiles. We assume that memory references may occur every clock cycle and demand our facility not to miss any reference. In other words, our facility must feature a single cycle throughput of memory reference patterns in order to fulfil our accuracy requirements.

3. **Scalability and extensibility.** Profiling heterogeneous memory in a uniform manner is not a trivial task. Different memory architectures come along with diverse physical characteristics such as access latency and energy efficiency and thus must be addressed individually to leverage their full potential. In order to examine various memory technologies with customized profiles, it is desirable to be able to replace profiling metrics without the need for adapting the whole architecture of the profiling facility explicitly. Therefore, we request support for multiple profiling policies along with a feasible way to customize these strategies.

4. **Efficient software interface.** The system's overall performance does not only depend on its hardware abilities but also on the operating system's performance. Memory management tasks such as paging are critical to every operating system and must be handled efficiently. Since our goal is to leverage the generated profiles during these management tasks, we demand a transparent and fast way for accessing the profiles by the operating system.

In summary, our memory profiling facility must not interrupt the system during the profiling process, has to provide adequately accurate profiles at run-time, should be extensible, and has to expose an efficient software interface to the operating system. In the following sections we develop a profiling architecture that complies with these requirements.

3.2 Location

As seen in the related work section of the previous chapter the decision where to place our profiling facility is an important issue. For us, the following four locations seem to be worth considering: inside the CPU, more precisely, as a part of the *memory management unit* (MMU), inside each memory module, inside the memory controller, or directly attached to the memory bus.

Placing the profiling facility in the MMU results in an architecture similar to performance counters most modern microprocessors already provide. While this option offers a clean and fast interface for accessing the generated profiles by introducing a dedicated set of CPU registers, it makes satisfying our accuracy and non-intrusiveness requirements impossible. The strictly limited and expensive area available on the CPU die restricts the number of registers that can be provided to hold access related information of memory regions. This dramatically effects accuracy, since only monitoring few regions at the same time would be possible. Techniques like sampling could be used to allow multiplexing of the few registers in order to increase the number of monitored memory regions. However, this will not improve accuracy, since only samples of access related information at certain moments are taken. Furthermore, sampling requires periodic evaluation of the monitored regions and periodic reconfiguration of the profiling facility with a new set of regions that should be monitored next. Periodic evaluation and reconfiguration induces run-time overhead during normal operation and thus conflicts with our requirement for non-intrusiveness.

Another reason against placing the profiling facility in the MMU is that we will not be able to detect memory accesses issued by other CPUs or external devices such as DMA controllers. Hence, the resulting profiles would be based only on regions accessed by the corresponding CPU and not reflect the actual usage of the physical memory.

Memory specific profilers inside each memory module are indeed an interesting option to consider. Memory manufacturer would be able to incorporate internal memory characteristics to the full extent and provide completely accurate profiles. The individual profiles could be stored in the memory on the same module and accessed using common memory read instructions by the operating system. The only problem is that this option would distribute profiles among the different modules and make it more difficult for the operating system to evaluate the increased number of profiles.

The remaining two possible locations, inside the controller or as a separate device attached to the memory bus, are only of insignificant difference. In order to support multiple memory technologies, the memory controller can be extended as proposed by Ahues [1] in a previous work. Placing the profiling facility inside the extended memory controller or attaching it directly to the memory bus as an additional device does not have significant impact on our design decisions. For the ease of use we decided to go with the latter option and develop an additional profiling module that is attached to the memory bus. This solution is more flexible, since it depends only on an existing memory bus and may also be used in environments without the extended memory controller.

The next section focuses on satisfying the extensibility requirements while delving into the internal design of the facility.

3.3 Memory Profiling Unit

In the previous section we decided to attach our profiling facility directly to the memory bus as an additional device. Starting this section, we will refer to this profiling facility as the *memory profiling unit* (MPU). This section outlines the high level architecture of the MPU and addresses the requirements for an extensible architecture and a simple interface to the operating system.

In order to cope with the requirement for an efficient software interface, two major tasks need to be accomplished. First, we need to collect all necessary information about the memory we want to monitor and its actual usage by the system. More precisely, we want to monitor multiple regions of multiple memories independently of their type, size and location. The information we need to capture must reflect the usage of these regions. In most cases it is sufficient to focus on the number of read accesses and the number of write accesses. Other viable entities may include time related information such as the last timestamp a region was read from or written to.

Second, we need to process all collected information, generate condensed but substantial profiles and present them to the operating system. The operating system may access the profiles at any given time and will use them as reference for improving its memory management strategies.

Figure 3.1 shows the proposed high level architecture of the MPU. Since the MPU is directly attached to the memory bus it is able to analyze all incoming accesses to physical memory. This allows us to approach the first task by introducing monitoring units, which we refer to as *address monitors,* shown on the left hand side in Figure 3.1. They are responsible for monitoring the bus and storing the captured information about the usage of the memory.

We approach the second task by introducing dedicated profiling units, which we refer to as *profilers*, shown on the right hand side in Figure 3.1. Profilers analyse the access related information captured by address monitors and transforms it into compact and representative profiles that can be used by the operating system.

The *update logic* shown in the center of Figure 3.1 connects the address monitors with the profilers and is responsible for continuously presenting all access related information collected by the monitors to the profilers. Finally, the MPU acts like a facade and makes all profiles accessible to the operating system using conventional memory read instructions.

3.4 Address Monitors

Address Monitors are part of the MPU and responsible for gathering access related information on specified regions of the memory. As shown in Figure 3.1, an address monitor is directly attached to the memory bus and “listens” for incoming memory reference patterns. The system architect configures each address monitor to react to specified reference patterns. When an address monitor successfully detects a reference pattern, it is responsible for

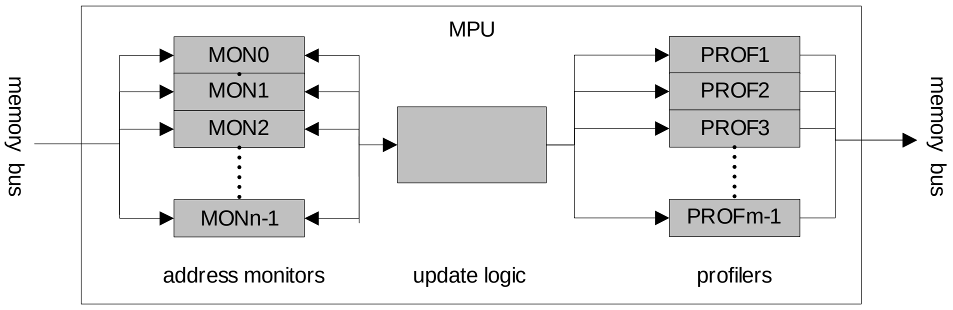

Figure 3.1: MPU High Level Architecture

storing and updating of the corresponding access related information. For instance, an address monitor may update an internal register with the current timestamp or increment a register that keeps track of the number of read accesses to a memory region represented by a target pattern. Multiple address monitors may be attached to the bus to enable monitoring of different regions. A single address monitor can also be configured to monitor multiple reference patterns. For this purpose, each address monitor supports retrieving of the stored information by passing an unique identifier of the corresponding target pattern.

3.5 Profilers

The purpose of the profilers is to transform the huge amount of the access related information gathered by the address monitors into a condensed but representative form. While in most cases the implementation of an address monitor is straightforward by counting how often the a memory region was accessed, profilers must be able to implement customized profiling policies based on different metrics to generate the desired profile. Therefore the design has to be flexible and must not enforce many restrictions for the implementor.

We demand from all profilers to accept two signals. The first signal denotes a key that can be used to globally identify a data register inside an address monitor. The second signal denotes the actual data that corresponds to the key. How the data is actually interpreted and processed is up to the implementation of the individual profiler. Each profiler will eventually see all keys/value pairs from the address monitors. It is left for the profiler to decide what key/value pairs are of interest for generating the profile. We also make no restrictions about the contents of the generated profiles. A profiler is free to implement its own metric along with the corresponding results.

3.6 Update Logic

Data gathered by the monitors must be presented to the profilers to allow them to update their profiles. The following three update policies are worth considering: on demand, on memory access, or periodically.

Updating profiles on demand, i.e., whenever a profile is read by the operating system, requires the update logic to present the access related information of those memory regions that are subject to the profiler's configuration. In a common configuration where thousands of memory regions are profiled by a single profiler, this would require thousands of cycles until the profile is up-to-date and may be used by the operating system. Thus, updating the profiles on demand results in a slow software interface and induces significant performance overhead during memory management tasks.

Another approach to keep the profiles up-to-date is to propagate the access related information of memory regions whenever a memory access takes place. However, this approach restricts the number of possible profiling policies. Consider a scenario where the monitors are used to count the number of read accesses to a set of regions of memory. Furthermore assume two kinds of profilers. One profiler yields the most often used memory region, and the other one yields the least often used memory region. Propagating only the updates to the first profiler, which is based on a "greater than" comparison of the counter values would not cause any problems and result in the most used region as requested. However, if we feed the second profiler, which uses a "less than" comparison on the counter values, with the same sequence of memory accesses the result will not be necessarily accurate. The "less than" profiler will only get the chance to see updates on memory regions that were actually accessed and thus never take notice of untouched memory regions. This behaviour is futile, especially when we want to find a region that was actually never accessed.

In order to cope with the issue of the above "less than" profiler it is necessary to traverse the access related data of each memory regions. To avoid performance overhead as seen in the on demand approach, an update logic using periodic updates is reasonable. In this case the update logic periodically presents the captured access related data of all monitored regions, one by one, to the profilers. The profilers update their generated profiles constantly and thus allow a fast interface for reading the profiles by the operating system.

In order to enable support for multiple address monitors and profilers embedded into one MPU we interconnect both, address monitors and profilers, using a bus-like interface. It keeps the profilers up-to-date by cycling through all possible keys used by the address monitors to globally identify the access related information that belongs to a memory region and presenting the corresponding key/value pairs to all profiles.

Chapter 4

Implementation

To validate the design from Chapter 3, we implemented a memory profiling unit (MPU) on the OpenProcessor platform presented in Chapter 2. Our implementation shall determine the most often used regions and the least often used regions of memory, as they are the most promising candidates to be moved between the different memory technologies. For this purpose, we implemented address monitors that count the number of read/write accesses and profilers to determine the most and the least often accessed regions.

The following sections present selected implementation issues of both the address monitor and the profiler modules.

4.1 Address Monitors

We restrict our implementation to a fundamental address monitor that is capable of counting read and write references to a fixed number of contiguous and equally sized memory regions. In the first subsection we present a facility for holding the required counters and then utilize this facility to implement an address monitor in the second subsection.

4.1.1 Reference Counter Block

Reference counter blocks are responsible for storing a large number of counters efficiently. For instance, if we intend to monitor a memory range of 64 MiB with a granularity of 8 KiB, we require a total of 2^{13} counters. Being interested in monitoring both the number of reads and the number of writes separately, we require twice as many counters. Needless to say, implementing all counters using registers is unfeasible, especially on our prototyping platform that is entirely implemented on FPGA. Fortunately, modern FPGAs feature a large number of dual-port block RAM modules. Our platform offers about 200 block RAM modules, each with a capacity of 18 Kibit. In other words, a block RAM module is capable of storing 512 36 bit wide counters.

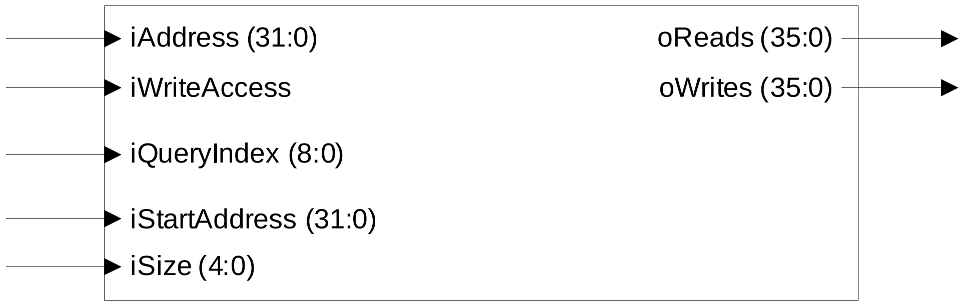

Figure 4.1: Address Monitor Module Schematic

Since we intent to count how often a memory region was accessed, we require logic that is capable of incrementing the value of any counter stored on the block RAM within a single clock cycle. This problem can be solved by leveraging the dual-port feature provided by block RAMs as suggested by Ahues [1]. However, our design additionally demands a facility for retrieving the value of the counters periodically in order to feed the profilers. To accomplish both, incrementing and retrieving counters independently requires a memory with at least two read ports and one write port. Unfortunately there was no native support for such memory on our prototyping platform. We solved this problem by emulating quad-port memory using normal dual-port memory clocked twice the system clock as described in [14]. This approach provides us with two internal memory clock cycles each external system cycle. We leverage the first internal cycle for a read-increment-write operation and utilize the second cycle for reading the counter value of an externally selected counter.

4.1.2 Read/Write Monitors

Our implementation of the address monitors uses two of these reference counter blocks (described in the previous subsection). One block is used for counting write accesses and the other block is used for counting read accesses to a memory region. Figure 4.1 shows the schematic of an address monitor. Each address monitor has two input signals: `iAddress` and `iWriteAccess` which are connected to the internal memory bus inside the MPU. The former signal represents the actual address bus which is 32 bit wide on the OpenProcessor platform. The latter signal denotes whether a write or a read access is currently being done.

The address monitors are optimized for counting references of contiguous memory regions. Due to the limited capacity of a block RAM used in our reference counter block modules, we are bound to at most 512 possible monitored regions per address monitor. The start address of the first region and the actual size of the entire monitored memory range is configured using

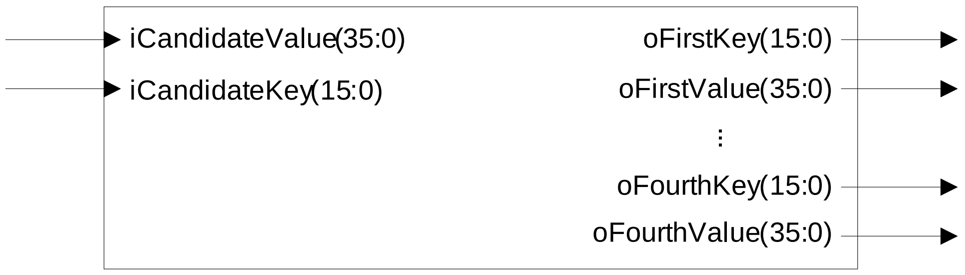

Figure 4.2: TOP4 Module Schematic

the two inputs signals `iStartAddress` and `iSize`. While the start address may be any valid memory address, the size is determined by the following equation:

$$\mathit{scope\ in\ bytes} = 2^{(iSize+9)}$$

The monitor observes the incoming signal `iAddress` and tests if the address is within the monitored memory range. If so, the monitor maps the incoming address to a counter index via

$$\mathit{counterIndex} := (\mathit{iAddress} - \mathit{iStartAddress})\ \texttt{SHR}\ \mathit{iSize} \quad .$$

Depending on the `iWriteAccess` signal, the counter at index `counterIndex` is either incremented in the reference counter block for read accesses or in the reference counter block for write accesses.

The input signal `iQueryIndex` selects a counter pair, whose value is then presented at the output signals `oWrites` and `oReads` representing the corresponding number of writes respectively the number of reads to the associated region.

4.2 Profiler

We also implemented a profiler that is capable of generating profiles which reflect the four most and the four least used memory regions. First, we develop a helper module that serves for maintaining a sorted list of key/value pairs. Then we implement the profiler based on this sorting facility.

4.2.1 TOP4/FLOP4

The TOP4 module implements a sorted list of arbitrary key/value pairs in hardware and works as follows. The module monitors a bus for incoming key/value pairs. For each cycle it compares the captured pair to a list of top pairs rated by value. If the captured key already exists in the list, the value of the corresponding key is updated and the list is re-sorted. If the list does not contain the captured key, the value of the captured pair is used in order to determine whether the captured pair will make it into the top. If so, the pair with the lowest value in the list is replaced by the captured pair and the list is re-sorted. Figure 4.2 shows the schematic of the TOP4 module.

At first glance, the procedure resembles a full associative cache, with an additional sorting feature. In order to fulfil our requirement for single cycle throughput we decided to settle for a small list with 4 elements. Reducing the number of elements to be sorted allows us to sort on insertion respectively on updates. Thus, we implemented a finite state machine that swaps the elements depending on the position of the matched key in the list and the position the new value would fit in. Table 4.1 shows the state transitions for the implemented state machine. At every clock cycle we check whether the captured key is already in the list. If so, the current position is determined. The key position column in 4.1 denotes the detected position. The case where the key was not in the list is denoted with a hyphen. The value position column in 4.1 denotes the position where the captured key/value pair should be placed according to its ranking. The case where the new pair fails to make it into the four top pairs is also denoted with a hyphen. Based on these two input variables the state machine updates the items of the list. For instance, if the key of a captured pair is not inside the list, but the value is ranked to position 3, the list would be updated as follows according to table 4.1: The 1st and 2nd place requires no modifications. The new key/value pair is inserted at the 3rd place. Finally the 4th place is updated with the key/value pair of the previous 3rd place. The previous holder of the 4th place is removed from the list.

So far, we discussed the implementation of a module that holds the top four key/value pairs ranked by value with highest value as number one. But we also require a module that holds the flop four key/value pairs ranked by value with smallest value as number one. This feature can be easily integrated into a general TOP4 module by adapting the search algorithm for the desired position inside the list to support TOP and FLOP search. This is possible without modifying the state machine because it only reacts to the current key and the desired value position and does not determine these two parameters itself.

4.2.2 TOP/FLOP Profiler

Using TOP4/FLOP4 modules we implemented a simple but effective profiler for our MPU. The profiler is capable of generating profiles that represent the four most referenced memory regions and the four least referenced memory regions, each with respect to the number of write and the number of read accesses. Figure 4.3 shows the schematic of our profiler. As demanded in the design chapter, the profiler accepts a key via the `iCounterKey` signal, along with the corresponding data via the `iCounterReads` and `iCounterWrites`

key pos.	**value pos.**	**1st**$_{t+1}$	**2nd**$_{t+1}$	**3rd**$_{t+1}$	**4th**$_{t+1}$
-	**-**	-	-	-	-
-	**1**	new	1st$_t$	2nd$_t$	3rd$_t$
-	**2**	-	new	2nd$_t$	3rd$_t$
-	**3**	-	-	new	3rd$_t$
-	**4**	-	-	-	new
1	**-**	2nd$_t$	3rd$_t$	4th$_t$	update(1st$_t$)
1	**1**	update(1st$_t$)	-	-	-
1	**2**	2nd$_t$	update(1st$_t$)	-	-
1	**3**	2nd$_t$	3rd$_t$	update(1st$_t$)	-
1	**4**	2nd$_t$	3rd$_t$	4th$_t$	update(1st$_t$)
2	**-**	-	3rd$_t$	4th$_t$	update(2nd$_t$)
2	**1**	update(2nd$_t$)	1st$_t$	-	-
2	**2**	-	update(2nd$_t$)	-	-
2	**3**	-	3rd$_t$	update(2nd$_t$)	-
2	**4**	-	3rd$_t$	4th$_t$	update(2nd$_t$)
3	**-**	-	-	4th$_t$	update(3rd$_t$)
3	**1**	update(3rd$_t$)	1st$_t$	2nd$_t$	-
3	**2**	-	update(3rd$_t$)	2nd$_t$	-
3	**3**	-	-	update(3rd$_t$)	-
3	**4**	-	-	4th$_t$	update(3rd$_t$)
4	**-**	-	-	-	update(4th$_t$)
4	**1**	update(4th$_t$)	1st$_t$	2nd$_t$	3rd$_t$
4	**2**	-	update(4th$_t$)	2nd$_t$	3rd$_t$
4	**3**	-	-	update(4th$_t$)	3rd$_t$
4	**4**	-	-	-	update(4th$_t$)

Table 4.1: State transition table for the TOP4/FLOP4 module

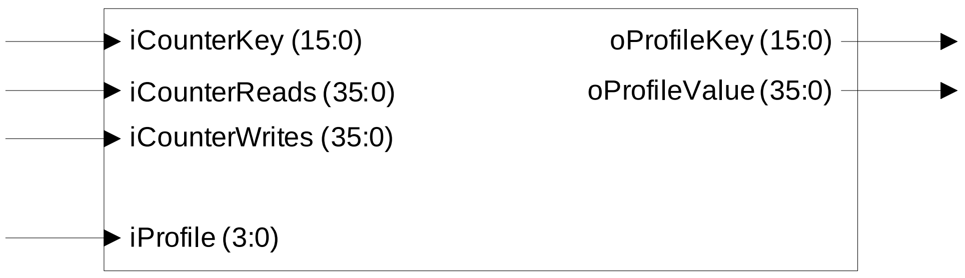

Figure 4.3: Top/Flop Profiler Module Schematic

signals. Each profiler is configured by the system architect with a start and an end key that dictate what memory regions the profiler is going to profile. All three input signals are monitored synchronously with the system clock. If the monitored key is within the configured range the profile is updated using the current number of reads and the current number of writes along with the key to identify corresponding memory region.

Each profiler instantiates four TOP4 modules to keep track of the current top key/value pairs. Two modules are initialized as TOP4 lists. The other two modules are initialized as FLOP4 lists. All four profiles are updated, whenever the the incoming key fulfils the above criteria. The profile entries can be accessed from outside the module by applying the `iProfile` input signal that selects the profile register to be read. Each profile entry is associated with an unique index of four bits. The most significant two bits select the profile according the following scheme:

0 top most by read access

1 top least by read access

2 top most by write access

3 top least by write access.

The least significant two bits select the position inside the a profile, where 0 denotes the top most, respectively the least most element in the list. The results are available at the signals `oProfileValue` and `oProfileKey` by the next positive clock edge.

4.3 Memory Profiling Unit and Update Logic

The implementation of our first MPU consists of 32 address monitors, two profilers, and an update logic as shown in Figure 4.4. The memory address bus is directly wired to all address monitors. The monitors are grouped into two banks: BANK0 and BANK1. Monitors of the first bank are configured to cover a memory range of 0..64 MiB. Covering 64 MiB with 16 monitors and 512 counters per monitor results in a coverage of 4 MiB per monitor or 8 KiB per counter. The second bank is configured to cover a range of 0..32 KiB with a coverage of 2 KiB per monitor and 4 byte per counter. Since all monitors are independently of each other there is no restriction on how the ranges are configured. Regions can overlap or reside completely inside each other. While a top level region with coarse granularity may offer a global view, nested regions with fine granularity can concentrate on crucial regions in detail.

The update logic is implemented by using a 14 bit counter to scan all counters inside the monitors. The least significant 9 bits are used to address

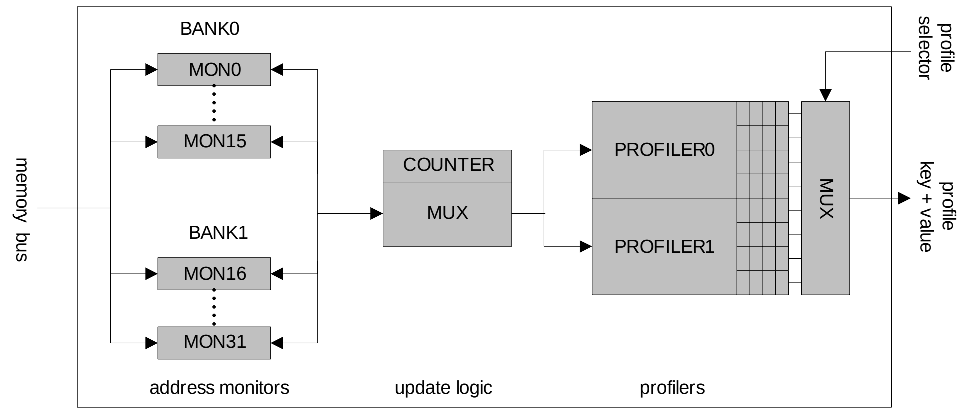

Figure 4.4: MPU Internal Design

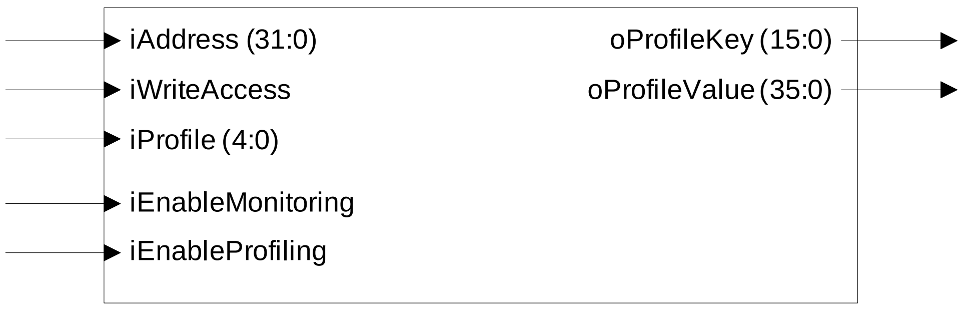

Figure 4.5: MPU Module Schematic

the counter inside one monitor. The 5 most significant bits are used to select the current monitor. The counter is implemented as a roll-over counter and keeps scanning all monitors continuously. The result of the currently selected monitor is multiplexed to a profiling bus where both profilers are attached. The first profiler is configured to profile the monitors of the first bank. The second profiler profiles the monitors of the second bank. Figure 4.5 shows the schematic of the MPU module.

Chapter 5

Conclusions

The goal of this thesis was to provide the operating system with profiles that reflect how memory pages are actually used by the system. The profiles should represent candidates that are subject to memory management strategies such as page replacement algorithms and enable dynamic migration of pages among different memory technologies according their usage. The profiles should be accurate and generated at run-time without affecting the system's performance. No previous approach to memory profiling was able to cope with these requirements.

We introduced a new hardware based memory profiling architecture that is capable of capturing every memory access issued by the system and of generating accurate run-time memory access profiles of multiple memories. Both the captured data and the profiling policies can be customized and allow individual profiling policies.

Finally, we validated our design by implementing a profiling facility that is able to generate profiles to determine the four most often read/written regions and the four least often read/written regions of memory. All four profiles can be accessed through memory mapped I/O and can efficiently be used by the operating system.

An important question not yet answered is how to handle saturation of counters. Since we are generally interested in qualitative profiles, a viable solution would be to divide all counters by two once a counter saturates. Since the division preserves the ratios of the counter values, the generated profiles should not be adversely affected.

Future work can start by evaluating page replacement algorithms that make use of the profiles generated in our implementation. Other profiling policies could also be studied in future work. Especially energy aware policies appear interesting.

Our profiling facility does not migrate pages itself, it supports the operating system by selecting promising candidates. Future work develop a facility that can migrate pages between different memories transparently and autonomously according their respective usage.

Bibliography

[1] Ahues, B. Entwurf und Implementierung einer erweiterten Speicherkontrolleinheit. Study thesis, System Architecture Group, University of Karlsruhe, Germany, 2008.

[2] Bach, S. Design and Implementation of a Debugging Unit for the OpenProcessor Platform. Study thesis, System Architecture Group, University of Karlsruhe, Germany, 2008.

[3] Ece, O. A., Barua, R., and Stewart, D. Heterogeneous Memory Management for Embedded Systems. In *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'01)*, ACM, pp. 34–43.

[4] Eranian, S. The perfmon2 project. `http://perfmon2.sourceforge.net`.

[5] Eranian, S. What can performance counters do for memory subsystem analysis? In *Proceedings of the 2008 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC'08)*, ACM, pp. 26–30.

[6] Gordon-Ross, A., and Vahid, F. Frequent Loop Detection Using Efficient Nonintrusive On-Chip Hardware. *IEEE Trans. Comput. 54*, 10 (2005), 1203–1215.

[7] Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Approach*, fourth ed. Morgan Kaufmann, September 2006.

[8] Lee, D., Choi, J., Choe, H., Noh, S., Min, S., and Cho, Y. Implementation and Performance Evaluation of the LRFU Replacement Policy. In *in Proceeding of the 23th Euromicro Conference* (1997), IEEE Computer Society, pp. 106–111.

[9] Lysecky, R., Cotterell, S., and Vahid, F. A Fast On-Chip Profiler Memory Using a Pipelined Binary Tree. *IEEE Trans. Very Large Scale Integr. Syst. 12*, 1 (2004), 120–122.

[10] MARQUET, K., AND GRIMAUD, G. An Object Memory Management Solution for Small Devices with Heterogeneous Memories. In *Proceedings of the 5th Workshop on Intelligent Solutions in Embedded Systems (WISES'07)* (2007), IEEE, pp. 227–237.

[11] MÜLLER, G. Emerging Non-Volatile Memory Technologies, 2003.

[12] NEIDER, R. OpenProcessor v1.

[13] PENDSE, R., AND BHAGAVATHULA, R. Performance of LRU Block Replacement Algorithm with Pre-Fetching. In *MWSCAS '98: Proceedings of the 1998 Midwest Symposium on Systems and Circuits*, IEEE Computer Society, pp. 86–89.

[14] XILINX. Quad-Port Memories in Virtex Devices (XAPP228).