Current Trends and the Future of Software-Managed On-Chip Memories in Modern Processors

Shahid Alam  R.Nigel Horspool
Department of Computer Science
University of Victoria, Victoria, BC, Canada
E-mail: {salam, nigelh}@cs.uvic.ca

Abstract

Processors are unable to achieve significant gains in speed using the conventional methods. For example, increasing the clock rate increases the average access time to on-chip caches which in turn lowers the average number of instructions per cycle of the processor. On-chip memory system will be the major bottleneck in future processors. Software managed on-chip memories (SMCs) are on-chip caches where software can explicitly read and write some or all of the memory references within a block of caches. This paper analyzes the current trends for optimizing the use of these SMCs. We separate and compare these trends based on general classifications developed during our study. The paper not only serves as a collection of recent references, information and classifications for easy comparison and analysis but also as a motivation for improving the SMC management framework for embedded systems. It will also make a first step towards making them useful for general purpose multicore processors.

1 Introduction

A CPU (Central processing unit) in a modern processor (Including general purpose multicore processors (GPPs) and high performance embedded systems (ESs)) available today have several levels of caches [31] that are located either on-chip or off-chip. These on-chip and off-chip caches form a memory hierarchy. They are managed by hardware or software, or by a combination of the two. The purpose of using this cache hierarchy starting from the on-chip cache is to break the effect of the memory wall [40]. If the speed of an on-chip cache is almost equal to the speed of the CPU, as is the case in most modern processors, we can potentially break the effect of the memory wall if all the memory accesses pass through this memory without any delay. One option for accomplishing this is to let the compiler/software explicitly manage and somehow make the code and data available all the time in these high speed memories/caches. (1) But is it possible in practice? (2) What efforts have already been done in this area both in an ES and a GPP? (3) How successful are they? (4) And what major areas need more research to ease and optimize the use of on-chip caches specifically in GPPs? These are our motivations for the study carried out in this paper.

We define software-managed on-chip memories (SMCs) as on-chip caches where software can read and write all or some of the memory references within a block of caches. These can include locked caches, scratchpads and are high speed SRAMs.

Locked caches are caches which are locked by the hardware, or sometimes by the software [28], so the software can use either a portion of, or the whole cache as a scratchpad. Scratchpad memories (SPM) in one form or other have been used in ESs a long time. Recently [5] they have been recommended for ESs as an alternative to a cache. SPM is considered similar to L1-cache but it has explicit instructions to move data from and to the main memory, often using DMA (Direct memory access) based data transfer. A comparative study [5] shows that the use of scratchpad memory instead of a cache gives an improvement of 18% in performance for bubble sort, a 34% reduction in chip area, and uses less energy per access because of the absence of tag comparisons. From here onwards in this paper we use the abbreviation SMC to denote these memories. SMCs are currently only used in ESs including multicore processors [12, 13, 26, 34, 36]. There are also research efforts [20, 10, 9, 11] where SMCs have been developed and tested for use in a GPP. The main advantage as mentioned in [5] of using SMCs is the savings they provide in area and energy. They can also accelerate the speed of a program because of the close proximity to the CPU.

As SMCs are managed by software, operating systems and compilers (Especially dynamic/runtime compilers) will play a big role in their efficient use for op-
timizing the code. A multicore processor’s local data that does not need to be committed to the main memory or shared with other processors can efficiently utilize SMCs [27]. Threads in SMT (Simultaneous multithreading) [38] processors can share the SMC. In a multithreading application running on a multicore processor, threads that share data the most can be placed on a single SMT core to considerably decrease their communication time and memory bandwidth. As we increase the number of cores, a core needs to have its own private on-chip space to improve its performance characteristics. Recently IBM in its Cell processor [34] and Nvidia in its GPUs (Graphic processing units) [36] have been experimenting with SMCs. SMCs will play a big role in improving the performance of the next generation of microprocessors. Nvidia’s future GPU architecture, code name FERMI [14], will contain a parallel data cache hierarchy with configurable 64 KB private L1 caches for each streaming multiprocessor and a 768 KB shared L2 cache.

2 Current Trends in SMC Management and Optimization

Except for some pioneering work performed by Cheriton et al. in 1986 [10], this section reports on progress made in optimizing the use of SMCs from 2000 onwards. We label these works for comparison according to the type of work done and call this label as SMC Type. We only cover on-chip memories and exclude recent work done [32, 6, 22, 16] on software managed memory hierarchies that includes both on-chip and off-chip memories. Readers interested in a comparison of programming models for managing memory hierarchies (Both on-chip and off-chip) are referred to [33].

SMC-VMP: As mentioned above, the first work done on targeting SMCs is by Cheriton et al. [10]. They implemented software controlled caches in an experimental multiprocessor called the VMP [9]. Concepts learned in this experiment were later used in designing and developing the Paradigm architecture [11]. Paradigm consisted of a memory module and multiprocessor module groups. Each group consisted of: processors with on-chip caches (Private caches); an on-board cache (Shared cache); and an interbus cache module. It is unclear as to what extent the Paradigm system was completed. We can see that similar concepts are being used now in building commercial multicore processors [34, 36, 14].

The VMP processor was an experimental multiprocessor developed at Stanford University. It was a software/hardware architecture that combined the operating system, hardware and software as firmware-like cache management modules. The main motivation for building such a processor was to give more control to the software to manage cache access. The results presented were not very promising. The processor performance reduced by almost 50% with a cache miss rate of 1%. As mentioned by the authors [9], the real challenge of the VMP design was in the software and hence a lack of a good programming environment was one of the major reasons for these disappointing results.

SMC-IIC: The first effort to implement a runtime SMC is presented by Hallnor et al [20]. The SMC implemented is for L2 cache. There are two parts to this implementation: the hardware structure of the cache called IIC (Indirect index cache); and the cache line replacement algorithm which they called generational replacement. IIC uses a cache line table in hardware to make the cache replacement policy fully associative. Hash table entries are used to look up the tag for the block.

SMC-Optimal: Avisar et al [3] describes an optimal memory allocation scheme for SMC in ESs. The basic process includes collecting data, such as the size, frequency of access and total number of variables in the application, through profiling and passing this information to the compiler. The compiler also uses information on the size and latency of the memories. Based on this information, the compiler then formulates the problem of memory allocation as a linear optimization problem that is solved using Matlab, and allocates the memory accordingly. The objective is to minimize the total memory access time for the application.

Linear equations are used to represent the allocation of global and stack variables to the SMC. These linear equations combined with a set of constraints make it a linear optimization problem. The parameters used to form these linear equations do not include the times of access to the variables. They added this functionality to their second effort, described below as SMC-CT. In our opinion the information could be obtained at compile time, as is done in SMC-CT, but it may not be as accurate as when it is collected at runtime. Even so, by including this times in the equations, we may be able to further improve the solution.

SMC-ICache: Huneycutt et al [21] presents the first effort to implement SMC using dynamic binary rewriting for ESs. An instruction cache (I-Cache) is implemented in the software as a client-server model. A software cache controller at the client side handles hits and a hardware memory controller at the server side handles the misses. Instruction sequences are broken into chunks, which are basic blocks, at the hardware memory controller and sent to the software cache.
controller which places them in a cache on the client side called tcache. Instructions in the tcache can be relocated to anywhere. The binary rewriter dynamically modifies the code to include jumps to either off-chip or on-chip memory, depending on the location of the jump target. This way no matter whether the object is either on-chip or off-chip the code runs correctly.

The Software I-Cache is compared with a direct mapped hardware cache with 16 byte blocks. Results show a 19% slow down for the software cache as compared to a hardware cache. The reason is obvious but they proved that the software cache can be implemented without any help from the hardware on the client side. Implementing a I-Cache in software is good for ESs in a client server model but we should also take into account the communication cost between the client and the server, as the software cache management will create additional. The authors do not include or discuss this communication cost.

SMC-CT: The work presented in [39] is an improvement on the previous work discussed in this survey as SMC-Optimal. They use compile time decisions to change static memory allocation to dynamic memory allocation (Section 3 for definition) that on average improves the performance by 40% and produce an energy saving by 31% as compared to SMC-Optimal.

Instead of forming linear equations, as is done in their first effort, they use heuristics to solve the problem. They first identify program points, which are points where it may be beneficial to insert code for copying a variable from DRAM to the SMC. They use variable size and timestamp to decide their eviction from the SMC.

SMC-As-FC: Baiocchi et al [4] presents a technique to manage a fragment cache (FC) in dynamic binary translators (DBT) using SMC with the help of flash and external memory in an ES. Their initial experiments without optimizations show that having FC in the external memory is better than FC in the SMC. After applying the optimizations they did achieve better results. But the improvement in speedup compared to using FC in external memory on average is just 2% for a SMC of size 32 KB. Other sizes of SMC show a reduction in the speedup compared to FC in the external memory. The only major improvement that was observed is that if SMC is used for FC than the amount of external memory required for a DBT is decreased.

One of the optimizations performed is victim compression. A victim (A block evicted from the cache upon replacement) is compressed and kept in the cache for easy retrieval. Compression and decompression is done in the external memory. In our opinion if the time needed is less than the time for accessing and retrieving the fragment from the external memory, then this scheme is profitable. Using this cost model before this optimization could give better results.

SMC-GPU: Silverstein et al [36] presents techniques to efficiently utilize SMC implemented in Nvidia’s GPU, which is based on a parallel computing architecture called CUDA [19], for memory bound algorithms. CUDA is a computing engine in Nvidia’s GPUs which is available to the programmers through the C language with Nvidia’s extensions. CUDA program is run by the hardware (Only Nvidia’s GPUs) on multiple threads. CUDA exposes a fast user manageable shared cache which can be used as a SMC among a subset of threads.

Here we just give an overview of the cache management strategy and the performance achieved. Preprocessing is done once by the CPU for deciding when and which data to be placed in the cache and then pass this information to the GPU in the form of metatables. The GPU uses metatables to manage the fetching and replacement of the data in the cache to be processed by the threads. With this user managed cache they, on average, achieved more than 150% performance improvement as compared to the use of texture cache [18].

SMC-Heap: There are two efforts which deal with heap data allocation in the SMC. The first scheme [15] does not allocate full heap data to SMC whereas the second [26] provides allocation for all heap data storage on the SMC. Therefore we just discuss the second effort that presents a SMC memory allocator (SMA) similar to the C language malloc() function. The SMA works as follows:

For large allocations it divides the SMC into a fixed number of blocks. The memory is allocated out of these blocks. For small allocations, the block is divided into small blocks of the size requested. SMA uses block sizes of 128 bytes and sub-block sizes of 8, 16, 32 or 64 bytes. In this way, SMC can be used as a memory pad where data is allocated by the software. It may not give good performance compared to hardware caches but it is space efficient.

The SMA is implemented for the Intel IXP network processor, which utilizes the Intel XScale microprocessor core. It is a heterogeneous multicore processor with two SMCs per core. One local and one shared. The results are compared with Doug Lea’s malloc() [23] implementation. Their SMA is on average 27% better in memory allocation time and 64% better in memory freeing time. It’s not clear how much this improvement is due to their allocation algorithm and how much to the fact that, compared to SMA, Doug Lea’s malloc() cannot use the on core SMC of the Intel IXP processor.
SMC-SMT: Metzlaff et al. [27] presents a design for a SMC that is managed dynamically in hardware to provide predictable timing behavior for a SMT processor. The SMC is called "function SMC" because it allocates a complete function inside the SMC.

Each processor, implemented using SystemC processor simulator, has a local SMC with a controller (SPC) which is responsible for all reads and writes from and to the SMC. The execute stage of the pipeline passes the function call and return information to the SPC which then loads the current function and any function that is nested in the current function. The SPC also maps a function to the SMC. If the function size is greater than the SMC, SPC wraps around and copies the left over instructions from the start overwriting some of the instructions of the current function. This can create some complications. For example the size of the largest function in the application must not exceed the size of the SMC. This is a constraint of this paper which in our opinion may limit the use of this scheme to relatively few applications. SPC does not have any information at runtime about the size of the function to be copied. This information is passed via the compiler through a flag, i.e; why we have included this work as SMC in this paper. This flag indicates the end of the function in the linked code.

SMC-GC: Li et al. [24] presents the first effort which maps the SMC management problem to the graph coloring (GC) problem. GC is the way to color the vertices of a graph such that no adjacent vertices shares the same color.

The promising idea presented is the partitioning of the SMC into a register file. That is how they map the SMC allocation problem to register allocation and hence to the graph coloring problem. We illustrate the SMC partitioning in Figure 1 and show that for some of the array sizes the algorithm may not be able to utilize SMC space efficiently by showing some unused space in the SMC with a simple example.

After performing liveness analysis for arrays, a live range of an array is splitted into subranges. Given these arrays and the register file, an existing graph coloring algorithm [30] is used to determine where these arrays are going to reside in the SMC.

SMC-USize: Nguyen et al. [29] presents the first effort which deals with an unknown size (USize) SMC at compile time. The basis of their technique is a binary rewriter (BW). The BW computes the size of the SMC and then accordingly modifies the code to fit the SMC size. The profiler collects information such as frequency of access, allocation and memory layout at compile time for every possible SMC size and the results are stored in a compact form. This saves a lot of computation at install time. For allocating code, the program code is divided into regions at compile time based on the frequency of access. At install time these regions are placed in the SMC. To preserve the control flow, branches are inserted. Results show a reduction when compared with one of their previous work discussed as SMC-Optimal in this paper. We believe this is because of the overheads in computing the SMC size at install time.

SMC-DLDP: This [13] is the first effort which presents a technique to specifically deals with the data layout decision problem (DLDP) in the SMC for the regular and irregular data access patterns usually found in multimedia applications. DLDP is defined as a problem of finding a layout for data to fit in the memory, in this case, the SMC, to maximize energy savings. There are two parts to the technique they develop to solve this problem: selection of data to be moved to the SMC based on the data access patterns; and placement of this data in the SMC to reduce memory fragmentation after solving DLDP.

Data elements are selected based on their frequencies of access and their lifetimes, and are formed into clusters. A heuristic is used to solve the problem to find the locations. The heuristic is based on the divide and conquer principle. Clusters are loaded in the SMC at these locations using DMA. For dynamic address translation of data references, they use an address translation buffer to optimize address generation code. This address translation buffer is implemented by a set of registers in the hardware and is updated by the operating system when the application is loaded.

SMC-MC: The SMC implemented in this [34] work is a 4-way set associative cache in the IBM Cell processor that has 8 general purpose cores and one special core. Each of the 8 cores has its own local SMC which uses DMA to access main memory. The 4-way set associative cache implemented in software uses fully associative replacement policy and hence gives a low cache miss overhead. The authors use a cache line table to
map a Tag to the cache line.

The authors present an adaptive algorithm to choose the cache line size and the replacement policy. The algorithm learns and adapts to the characteristics of the specific loop. There are five cache line sizes to select from. These are selected dynamically by running the loops and comparing the TPIs (Execution times per iteration). The size with the lowest TPI is selected. This way an optimal size is selected for the running loop. The replacement policy is selected out of: clock algorithm, LRU and FIFO in the similar way. On average, the results show an improvement of 20% over SMC-IIC. We believe the main reason is the tag comparison done in SMC-IIC.

3 Classifications Developed

We develop general classifications also called parameters to distinguish, compare and analyze the thirteen works discussed above. Table 1 lists these works based on these classifications. Section 4 provides analysis and gives some of the comparison examples using this table. As mentioned initially in the paper, the most important aspect of managing a SMC is to allocate as much program code and data to the SMC as possible. Our classifications are mostly based on memory allocations and are defined below:

1. Allocation Kind Static: Memory allocation can not change at runtime, i.e; the cache blocks cannot be replaced. It’s easier to manage but is not very flexible.
2. Allocation Kind Dynamic: Memory allocation can change at runtime, i.e; the cache blocks can be replaced. It’s difficult to manage but is more flexible.
3. Allocation Type Code: If program instructions are allocated to the cache.
4. Allocation Type Data: If program data is allocated to the cache. We further subdivide data allocation into three categories:
   (a) Variables: These can be scalars or arrays and local or global, and are allocated at compile time or runtime.
   (b) Stack: Data using the stack and is allocated at compile time or runtime.
   (c) Heap: Memory area allocated during runtime and used as dynamic memory.
5. Allocation Method Static: Techniques used for allocation are carried out at compile time.
6. Allocation Method Dynamic: Techniques used for allocation are carried out at runtime.
7. Profiling Static: Compile time profiling. The Program is executed with generated sets of input data to collect profiling information.
8. Profiling Dynamic: Runtime Profiling. Profiling information is collected as the program executes with actual (Real) input data.
9. System Compared: The System that is compared with the system developed/presented.
10. Results: We divide the results compared to the system above into two categories:
   (a) Performance: An improvement or a reduction in the execution time.
   (b) Energy Saving: An improvement or a reduction in the energy saved.
   (c) We use the following grades to compare the above two: A: (100% and up) B: (50% to 99%) C: (0% to 49%) D: (−1% to −49%) F: (−50% and less)

4 Synthesis

In this Section we use the classifications defined above to distinguish, compare and analyze the approaches used for SMCs as described in Section 2. In this synthesis we determine and reason some of the basic characteristics of a framework for optimizing the management of SMCs, and list them at the end of this Section.

All the work discussed in this paper uses software to manage SMCs and over half (Seven) of them use both software and hardware as shown in Table 1. One of them SMC-SMT is implemented in hardware (Simulated) but needs a flag from the compiler to be passed to indicate the size of a function. Less than half (Five) of the schemes use profiling which is of type static as shown in Table 1.

Only two, SMC-VMP and SMC-IIC, of these schemes are done for GPPs. SMC-VMP showed poor results and SMC-IIC did not prove to be successful as shown in column PI (Performance improvement) of Table 1. We strongly believe that one of the major reasons for this poor performance is the lack of a good software system or a programming environment for managing the SMCs.
<table>
<thead>
<tr>
<th>SMC Type</th>
<th>Allocation</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Method</td>
<td>1Prof</td>
</tr>
<tr>
<td>SMC-VMP</td>
<td>Dynamic</td>
<td>x</td>
</tr>
<tr>
<td>SMC-IIC</td>
<td>Dynamic</td>
<td></td>
</tr>
<tr>
<td>SMC-Optimal</td>
<td>Static</td>
<td>6Var,Stack</td>
</tr>
<tr>
<td>SMC-ICache</td>
<td>Dynamic</td>
<td>Code</td>
</tr>
<tr>
<td>SMC-CT</td>
<td>Dynamic</td>
<td>Code,Var,Stack</td>
</tr>
<tr>
<td>SMC-As-FC</td>
<td>Dynamic</td>
<td>Code</td>
</tr>
<tr>
<td>SMC-GPU</td>
<td>Dynamic</td>
<td>Var</td>
</tr>
<tr>
<td>SMC-Heap</td>
<td>Dynamic</td>
<td>Heap</td>
</tr>
<tr>
<td>SMC-SMT</td>
<td>Dynamic</td>
<td>Code</td>
</tr>
<tr>
<td>SMC-GC</td>
<td>Dynamic</td>
<td>Var</td>
</tr>
<tr>
<td>SMC-USize</td>
<td>Static</td>
<td>Code,Var,Stack</td>
</tr>
<tr>
<td>SMC-DLDP</td>
<td>Static</td>
<td>Var</td>
</tr>
<tr>
<td>SMC-MC</td>
<td>Dynamic</td>
<td>Code</td>
</tr>
</tbody>
</table>

1 Profiling 4 Implemented using both hardware and software 5 Hardware cache 6 Variables

Table 1: Allocations, results and platforms supported by the SMCs based on the classifications developed in Section 3

There are two schemes which based on our study get a grade of A in the results as shown in column PI of Table 1. One is SMC-SMT which is compared with a system using no cache and the other is SMC-GPU which is compared with a system using a texture cache. So out of the thirteen works covered we consider SMC-GPU to give the best results.

Now we list and discuss, based on our classifications and the analysis above, what we consider to be some of the basic characteristics of a framework for optimizing the management of SMCs:

1. **Transparent Software/Hardware Interface**
   We believe this area is one of the most important factors for improving the use of SMCs especially in a GPP. The best example of a transparent software/hardware system for managing SMCs discussed in this paper is SMC-GPU. The CUDA framework used in SMC-GPU is highly optimized for and only runs on Nvidia’s GPUs. Other significant programming models not discussed in this paper are: Brook [8] used by AMD and RapidMind [25] used by the new language called Ct [17], currently under development at Intel, specifically designed for multi core CPUs. They are still under development and we are not sure how much support they provide for SMCs. Most of the successful work done in multicore processors is in ESs discussed as SMC-GPU, SMC-MC and SMC-DLDP in this paper. Application programmers for GPPs need a general easy to understand and programmable interface. So making it general and transparent is one of the major hurdles for adapting SMCs to a GPP.

2. **Dynamic Profiling**
   Profiling is a very important part of any software optimizing system. Dynamic profiling provides more exact information than static profiling. The challenge of dynamic profiling is that it takes time and space and hence increases the execution time and area of the running program. [35] presents a dynamic application profiler for space conservation and [7] is a recent effort that presents a dynamic fast profiler for data locality. Almost all modern processors have hardware performance monitors/counters that can be used for profiling the running program [37, 2]. But to our knowledge there is no such effort where they have been used for profiling to optimize the use of SMCs. We did not find any work that uses dynamic profiling for SMC management. We believe this is one of the
major areas where more research is needed.

3. **Dynamic Memory Allocation**: The ideal situation would be to allocate all the code and data of the current working set of the running program to the SMC without any delay. Much work has been done on allocation of code and data including stack and global variables to the SMC. There is a need to do more work on SMC management for heap data. The only work we know of on allocating the heap to the SMC is SMC-Heap. The other areas are the kind and method of allocation. Based on the results presented in Table 1 we believe that both the method and the kind of allocation should be dynamic. Dynamic allocation takes time and can increase the execution time of the running program. To reduce time, we recommend obtaining help from the hardware as is done in some of the schemes listed in Table 1 but should be transparent to the application programmer especially for the GPPs as described above.

4. **Flexible**: With different sizes of SMCs and the different data patterns presented by applications running on ESs and GPPs, there is a need for the SMC management framework to be flexible. This will enable it to learn, change and adapt to these changing environments. This is done in SMC-MC, which adapts and selects different cache line sizes and replacement policies based on the loop characteristics, and the technique presented in SMC-USize works with an unknown SMC size.

5 **Conclusion**

We have analyzed the current trends and reasoned about some of the basic characteristics of a framework for managing and optimizing SMCs in an ES and a GPP. A general classification has been developed to compare, analyze and distinguish these trends. Table 1 lists the division based on these classifications for easy analysis and comparison.

With aggressive clock rates, the average access time to a L1-cache will typically be 3 - 7 cycles and 30 - 50 for L2-caches, which will adversely affect the average number of instructions per cycle [1]. Conventional processors at best will be able to achieve an annual gain of 12% rather than 55% in speed [1] if Moore’s Law continues to apply to chip density. This is the main reason multicore processors have already taken over from single core processors. The on-chip memory system will be the major bottleneck in future processors and there is a need to do more research and work on managing these memories especially for GPPs.

We hope this paper will not only serve as a collection of recent references, a source of information and classifications for easy comparison and analysis but also a motivation for improving SMC management framework for ESs and introducing and making it successful for GPPs.

**References**


