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

**Article** · April 2010



**1 author:**

Shahid Alam
Qatar Foundation
**22** PUBLICATIONS **38** CITATIONS

SEE PROFILE

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

Shahid Alam
Department of Computer Science
University of Victoria, BC V8P 5C2
E-mail: salam@cs.uvic.ca
March 15, 2010

**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

General purpose multicore processors (GPPs) and high performance embedded systems (ESs) available today use random access memories to store program's code and data. These memories can be static (SRAM) or dynamic (DRAM). SRAMs are costlier and speedier, almost equal to the speed of the processor, than DRAMs and are used as on-chip and off-chip caches. A cache stores copies of data or instructions, or a combination of two, from the main memory to reduce the average memory access time. A CPU (Central processing unit) in a GPP or a high performance ES has several levels of caches [40]. Caches closest to the ALU (Arithmetic logic unit) after the registers, i.e; on-chip are called L1-caches. Access time of a L1-cache in ES is usually 1 cycle and $1-3$ cycles in GPPs. L2-caches can be on-chip as found in multicore processors or off-chip. L3-caches if present are off-chip. Access time of L2-cache is more than the L1-cache and access time of L3-cache is more than the L2-cache. These on-chip and off-chip caches form a memory hierarchy and are either managed by the hardware or the software, or 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 [54]. 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 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 GPP? 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 [37], 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 ES a long time. Recently [8] they have been recommended for ES 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 [55, 8] 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 ES including multicore processors [16, 17, 35, 46, 49]. There are also research efforts [26, 14, 13, 15] where SMCs have been developed and tested for use in a GPP. The main advantage as mentioned in [55, 8] of using SMCs are 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.

The basic purpose of SMCs is to improve both performance and energy saving by optimizing the use of caches. Cache optimizations work on the principal of locality [19] which states that data recently used will be reused again in the near future. There are two kinds of localities. Spatial locality: Data located together will be referenced close together in time. Temporal locality: Data accessed recently will be accessed again in near future.

As SMCs are managed by software, operating systems (OSs) and compilers (Especially dynamic/runtime compilers) will play a big role in their efficient use by taking advantage of spatial and temporal locality of code and data. 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 [36]. Threads in SMT (Simultaneous multithreading) [52] 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 [46] and Nvidia in its GPUs (Graphic processing units) [49] 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 [18], 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.

This paper analyzes the current trends for optimizing the use of these SMCs. In Section 2 we present the current trends for managing and optimizing SMCs in software/hardware. In Section 3 we enumerate simple classifications developed in this paper that help us to provide an analysis and comparison of this study.

Section 4 separates, compares and analyzes these efforts based on these classifications. Section 5 concludes the paper.

## 2 Current Trends in SMC Management and Optimization

Except for some pioneering work performed by Cheriton et al. in 1986 [14], 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 [43, 9, 30, 21] 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 [44].

**SMC-VMP:** As mentioned before the first work done on targeting SMCs is by Cheriton et al [14]. They implemented SMCs in an experimental multiprocessor called VMP [13]. Concepts learned in this experiment were latter used in designing and developing the Paradigm architecture [15]. The 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 interbus cache module. It is unclear to what extent the Paradigm system was completed. We can see that similar concepts are being used now in building commercial multicore processors [46, 49, 18].

The VMP processor was an experimental multiprocessor developed at Stanford University. It was a software/hardware architecture that combined the OS, 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. Local memory, ie; on-chip cache, contained the software for cache management. A cache miss in the VMP is implemented as follows:

On a cache miss the cache controller issues an interrupt and generates a cache slot in the main memory to be brought in. The processor on interrupt saves its state on the (Supervisor processor) stack and jump to the cache miss handler routine stored in local memory. The cache miss handler routine maps the virtual address to the physical address of the cache page and tells the block copier to copy the main memory to the cache. If the data is not there a page fault occurs which is passed to the OS. The block copier works independent of the processor and the processor updates its data structures during the copy. When the copy completes, the processor resumes execution.

The VMP multiprocessor prototype was not ready at the time of experiments so they presented performance results based on trace-driven simulations. 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 [13], 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 scheme to implement a runtime SMC is presented by Hallnor et al [26]. The SMC implemented is for L2-cache. There are two parts to this implementation: hardware structure of the cache called IIC (Indirect index cache); and the replacement algorithm called generational replacement.

The IIC uses a cache line table in hardware to make the cache replacement policy fully associative. It

does not associate a tag entry to a special data block location and hence achieves fully associativeness. Hash table entries with a pointer to the data block are used to lookup the tag for the block. The IIC's replacement algorithm is as follows:

The use of data is divided into prioritized pools. The data is moved into pools based on the frequency of use. Instead of tracking the frequency of each data block they group them into smaller pools to make it easy to track the usage. The block to be replaced is chosen from the non-empty lowest priority pool.

Traces are generated on the Intel architecture running Windows NT 4.0 to run simulations. These traces contain instructions and data references to stress test the SMC. The generational replacement algorithm is compared with traditional cache design using different associativities, 4, 8 and 16. The average improvement on miss count is 45% on a block size of 512. It is not clear from the paper how the cache and the cache line table is simulated in the hardware.

**SMC-LT:** Kandemir et al [29] presents a SMC management framework focusing on optimizing the array based applications as found in image and video processing. The compiler divides the work into the following three phases:

- Data access: Loop transformations [2] are used to decrease the data transfer between SMC and off-chip memory and hence maximizing the use of the SMC. The portion of arrays required by the current computation is fetched and is called a tile. The selection criteria for these tiles are: they should have high reuse; and should fit in the SMC.
- Data partitioning: After loop transformations the compiler partitions the available space in the SMC among the arrays accessed. The partioning depends on how the loops are transformed in the first phase.
- Code modifications: Code is inserted into the program at compile time for the changes mentioned above.

The SMC management framework on average is 30% better than when the SMC is used as a hardware cache and is not able to improve upon the hand optimized version. The reason is the selection of tiles. In selecting the tiles the hand optimized version not only consider the loop nests [2] but also the tile reuse between multiple nests.

**SMC-No-Cache:** Banakar et al [8] recommends and establishes the use of a SMC instead of a cache in ES to save energy and area. This is the first time such recommendation has been made. A comparison is made between a 2-way set associative cache and the SMC. The results show that the area covered by the SMC is almost 34% less than the cache. The energy consumption on average is reduced by 40% using the SMC. An experimental compiler *encc* is used to generate code, which identifies the frequently used code and data and maps them to the SMC using the knapsack algorithm.

**SMC-Optimal:** Avissar et al [6] presents an optimal memory allocation scheme for SMC in ES. The optimality depends on the data collected by the profiler at compile time. The paper assumes that the target ES has atleast two writable memories and no cache. Focus of this paper is on global and stack variables. The basic process includes collecting data like size, frequency of access and total number of variables in the application by profiling. This information is passed to the compiler. Compiler also gets the size and the latency of the memories. Based on this information compiler formulates the problem of memory allocation into linear optimization problem that is solved using Matlab.

The scheme presented assumes the heap data to be allocated to the external DRAM. Heaps are allocated dynamically, i.e; at runtime, and there is no way to know the size and allocation frequency of heap data at compile time. Linear equations are formed for allocating global and stack variables to the SMC. With these linear equations following constraints are defined to turn memory allocation problem into a linear optimization problem: a variable can only be allocated to one memory unit; and sum of all the sizes of variables allocated cannot exceed the size of the memory unit. For stack variables they propose the following two options for allocation:

- Multiple stacks are allocated in SMC and DRAM. Because of more overheads this is feasible for large number of variables.
- One stack is allocated to either SMC or DRAM. Because of less overheads this is feasible for small number of variables.

The basis of the optimality is the formulation of the data collected by the profiler into linear optimization problem. The parameters used to form the linear equations does not include the time of access to the variables. In our opinion this 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 these times in the equations, we may be able to further improve the solution. Results show that on average the SMC allocation achieves over 50% speedup than the all DRAM allocation. A comparison with a hardware cache could have produced more real results.

**SMC-ICache-1:** Huneycutt et al [27] presents the first effort to implement SMC using dynamic binary rewriting for ES. 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. This way the workload is divided between a client which does not need to be powerful, hence saving energy in an ES, and a server which can be far more powerful. Instruction sequences are broken down into chunks, which are basic blocks, at the hardware memory controller and send 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, i.e; tcache is fully associative. Instructions accessed recently are placed in the tcache.

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. By rewriting the instructions (Branch instructions) there is no need to check for cache tags. Not all the tags can be avoided and replaced in this way. Only tags for the branch instructions whose destinations are known at the time of rewriting are replaced and hence the technique only deals with the common case of the branch instructions. The design for a data cache is also proposed but not implemented in the paper.

The software I-Cache is compared with a direct mapped hardware cache with a 16 bytes block. Results show 19% slow down of software cache than hardware cache. But they are successfull in proving that the software cache can be implemented without any help from the hardware and its performance is close to the hardware cache. Implementing I-Cache in software is good for ESs in a client-server model but we should also take into account the communication between the client and the server. In these environemnts a client

needs to communicate with the server for other purposes, like command and control, but the software cache management will add more to this communication. The authors do not include or discuss this communication cost.

**SMC-ICache-2:** The second effort of designing a software instruction cache is by Miller et al [37]. This software I-Cache has been implemented on the MIT RAW prototype microprocessor [51]. There are two parts to this design: a runtime; and a preprocessor.

Preprocessor: The preprocessor consists of a binary rewriter for code modifications, to add instruction caching to the code, and is located in the main memory. Preprocessing is carried out before linking of the object file. The preprocessor divides the cache into blocks. These blocks refer to the program basic blocks in the CFG (Control flow graph) [2]. Basic blocks in a CFG have different sizes so to keep their sizes same NOP (No operation) instructions are added. It is not clear from the paper what maximum size is kept for the basic block. We assume its the size of the SMC. But, what if the size of a basic block is greater than size of the SMC? The binary rewriter creates a destination table to store physical addresses along with the virtual addresses of the control instructions which are at the end of a basic block in the CFG. This table is stored in the main memory and consulted by the runtime to fetch the appropriate data for each control instruction. In our opinion this way the runtime incurs a call to the main memory each time it jumps to the next block.

Runtime: The runtime is located in the cache. When the runtime receives control from one of the blocks it looks up the physical address, in a block data table as described above that contains information about the current basic block, based on the virtual address passed. If the block is present it jumps to the new block otherwise it asks the main memory to send the block. When it receives a response it copies the block to a specific memory location in the cache and jumps to the new block.

For cache replacement FIFO or FLUSH is used. FIFO evicts the oldest cache, and FLUSH flushes the entire cache and starts fresh. A pin system is implemented for the software cache which allows a programmer to specify what functions to pin/lock for time predictability in real-time systems. The pinned/locked code in the cache cannot be evicted and therefore has predictable and consistent time when it executes.

Chaining is used to modify the code inside the cache the first time when a block is loaded by the runtime. This changes the destination of the jump which requested the block. In this way, second time, the new block is automatically executed without going through the runtime, which saves some clock cycles. According to the authors it saves 40 clock cycles. Chaining is good for FLUSH because unchaining is not needed when the block needs to be evicted. For indirect jumps, which are jumps that might have different target addresses, each time all the target addresses are chained. This chaining is only done for function jumps, which according to the authors have small number of different targets, and for FLUSH.

The experimental results presented in the paper are not very encouraging but they also prove, as is proved in SMC-ICache-1, that an I-Cache can be implemented in software where hardware cache is not present and improves convenience of programming. The I-Cache implemented neither improves performance nor energy. Its major difference than the previous such effort, SMC-ICache-1, is that its implementation is not based on a client-server model. Because of this it improves performance and energy saving compared to SMC-ICache-1 as shown in Table 1.

**SMC-CT:** The technique presented in [53] is an improvement on the previous work discussed in this survey

as SMC-Optimal. Compile time decisions are used to change static memory allocation to dynamic memory allocation (Explanation of these terms is given in Section 3) that on average improves the performance by 40% and energy saving by 31% compared to SMC-Optimal. When compared with all hardware direct mapped cache implementation the improvement in overall performance is neglegible and is 1.7%. Out of 9 benchmarks used only 3 of them show improvements in performance. Two of these show minor improvements but the third benchmark G.721 shows a 100% improvement in performance, which considerably improves the overall results. G.721 is one of the data compression techniques (Speech codecs) used in audio signal processing. We are not sure why this discrepency is there as the memory use of G.721 is almost the same as some of the other benchmarks as shown in Table I in [53].

The basic process/heuristic used consists of first identifying program points, which are points where its beneficial to insert code for copying a variable from the DRAM to the SMC. A point is beneficial if: gain in speed by having the variable in the SMC is greater than the cost of moving the variable to the SMC. Profiling is used to find out this cost and benefit model. The compiler evicts some of the existing variables from SMC to make space for incoming variables that makes the allocation dynamic. Variables with minimum size are removed first to make the eviction simple and to keep the runtime lower. In a case of a tie the compiler chooses the variable with higher timestamp.

The timestamps are a dynamic execution order of the running program and are generated by using a data program relationship graph (DPRG). The DPRG is created by time stamping the call graph [2] of the program in a depth first traversal. Each node in the DPRG is a program point as described above. The DPRG is a directed acyclic graph as it does not handle recursive calls. Recursive cycles in the DPRG are collapsed to a single node and are allocated to the DRAM. A sample program and its DPRG is shown in Figure 1.

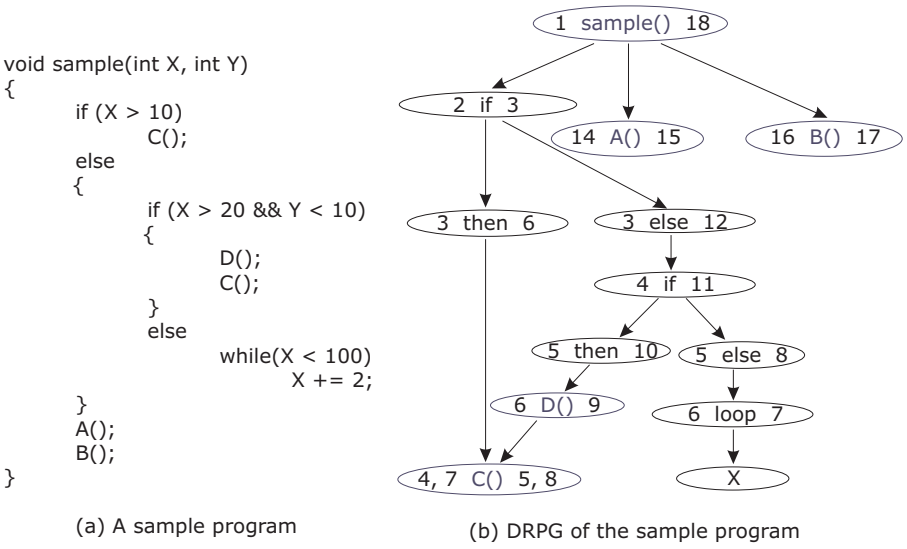



Figure 1: **A Sample Program and its Data Program Relationship Graph (DPRG)**

For allocating global and stack variables to the SMC the algorithm first traverses each program point in the DPRG in the partial order of their timestamps. In the first traversal it transfers variables to the SMC in decreasing order of their frequency of access. This frequency is computed at compile time by profiling the application. The second time before transferring a variable to the SMC the algortihm checks the cost and

benefit model, as described above, and transfers and evicts only if its feasible. An extension is presented to include program code for allocation to the SMC. It is not clear from the paper [53] if the authors have incorporated this extension in the implementation before evaluating it.

**SMC-As-FC:** Baiocchi et al [7] 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. A FC is used to keep dynamically translated instructions called fragments which are the application's translated code working set to keep the DBT from retranslating the previously translated code. Their initial experiments without optimizations show that having FC in the external memory is better than FC in the SMC. Based on these experiments and results following three optimizations are applied to improve the use of FC in the DBT using SMC. These optimizations are implemented using Strata [45], a cross platform infrastructure for building a DBT:

- Footprint Reduction: The DBT uses a trampoline (A short snippets of code) for translating the target at the end of a basic block. In the case of a branch taken it adds a branch instruction to the new target and in the case of a branch not taken it returns control to the DBT. Depending on the number of basic blocks these trampolines can expand the instruction count in the program. To reduce this instruction count only one trampoline function is used that can be shared by all the branches. For speed this function resides inside the SMC.

- Victim Compression: The FC is divided into two regions: a compressed fragment region (CFR); and an uncompressed executable fragment region (EFR). The CFR is used to save the evicted fragment (A victim - a block evicted from the cache upon replacement) from the FC. The basic idea is to store the victim in the CFR after compressing it for easy retrieval. Compression and decompression is done in the external memory. In our opinion if the time for compressing and decompressing the fragment when 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. We are not clear if the scheme presented follow this model. The FC is partitioned dynamically between the CFR and the EFR. More priority is given to the EFR. When the FC is filled completely with the EFR then the EFR is compressed and becomes the new CFR.

- Fragment Pinning: A fragment in FC can be pinned (Locked) so that it persists across different flushes to avoid unnecessary overhead of compressing and decompressing such a fragment. A pinned fragment region (PFR) is used for this purpose and is inter mixed with the EFR for best utilization. Victims from the previous FC which are part of the working set of the DBT are one of the targets for pinning. Pins are released when the size of the PFR reaches a certain threshold value, which is computed experimentally. There is no specific policy (For example in what order) given in the paper for releasing the pins.

After applying these optimizations the results improved. 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 speedup compared to FC in the external memory. The only major improvement that was obsereved is that if SMC is used for FC than the amount of external memory required for a DBT is decreased.

In our opinion if size of the SMC and the FC allows, it is beneficial to keep more than one CFRs (Old copies of EFR). This may produce better results if the data presents such a temporal locality. But will increase the complexity of the SMC management for the DBT.

**SMC-GPU:** Silberstein et al [49] presents techniques to efficiently utilize SMC implemented in Nvidia's GPU, which is based on a parallel computing architecture called CUDA [25], 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 and the OpenCL [23] framework. CUDA SDK (Software development kit) is available for Windows and Linux. 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 this information is passed 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. The preprocessing also includes the determination of the replacement policy for each function in the program. If a function exceeds the size of the cache available that function is accessed directly from the main memory bypassing the cache. Spatial locality is improved by restructuring the data layout. With this user managed cache on average they achieve more than 150% performance compared to the use of texture cache [24]. Textures are read only data and present spatial optimization oppportunities. Textures are used to map images onto the surfaces of three dimensional objects. For example mapping a grassy image to an uneven surface of a mountain. A texture cache in a GPU provides faster access to these textures.

**SMC-Heap:** There are two efforts which deal with heap data allocation to SMC. The first [20] does not allocate full heap data to SMC whereas the second [35] provides allocation of full storage of heap data to 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 fixed number of blocks. The memory is allocated out of these blocks. For small allocations a block is divided into sub-blocks of the size requested, which should be equal to a valid size, if not then it is rounded to a valid size. Valid size for the SMA is a power of two. The 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 provides simple and semi-automatic management of SMC. It may not give good performance compared to hardware caches but it is space efficient.

The experiments and results are shown for Intel IXP network processor, which utilizes Intel XScale [28] microprocessor core. The IXP is a heterogeneous multicore processor with two SMCs per core. One local and one shared. The results are compared with Doug Lea's malloc [31] implementation, which is the standard implementation used in Linux allocator in the GNU C library. According to the paper this is considered as one of the fastest and space efficient allocators available. The SMA on average is 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 the SMA, the Doug Lea's malloc cannot use the on core SMC of the Intel IXP processor.

**SMC-SMT:** Metzlaff et al [36] presents a design for a SMC that is managed dynamically in hardware to

provide predictable timing behavior for a SMT processor. The SMC designed gets help from the software in the form of a flag i.e; why we call it SMC in this paper. 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. This flag indicates the end of the function in the linked code.

The selected benchmarks for experimenting list the largest function size. The comparison is done with a system without on-chip cache. Experiments are carried out with different SMC sizes. SMC minimum size is selected according to the largest function's size listed. The scheme shows improved instructions per count compared to the system without on-chip cache. On average improvement is over 100%. A comparison with an on-chip locked cache could have produced more real results.

**SMC-GC:** Li et al [32] 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.

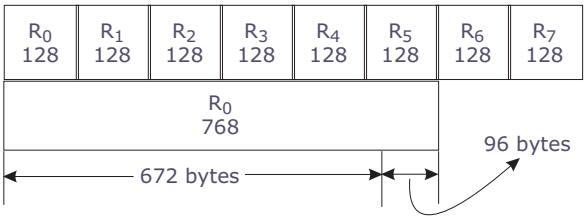
The promising idea presented is the partitioning of the SMC into a register file. That is how they map the SMC allocation problem to register allocation and hence to graph coloring problem. The complete algorithm for the SMC partioning is given in [33]. Here we illustrate it in Figure 2 and show that 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. Figure 2(a) shows the alignment of arrays 'A', 'B' and 'C' at 8 bytes using the size of the smallest array 'A'. The SMC shown in Figure 2(b) of size 1024 bytes is divided into 8 registers each of size 128 bytes, because of the size of the smallest array 'A'. Array 'C' whose original size is 668 bytes fits into 6 registers with the last register having $(96 + 4)$ bytes of unused space.

An interprocedural control flow analysis [2, 3] is performed to build an interprocedural CFG (ICFG). The ICFG consists of CFGs of all the functions in the program and all possible interprocedural flow edges across the CFGs. Liveness analysis is performed for arrays. An array is live at a program point if some of its elements may be used (Read) before they are defined (Killed) in an ICFG. They split a live range of an array into subranges, which can be allocated to different registers in the SMC. Only arrays in hot loops are splitted and allocated. Profiling is used at compile time to find these hot loops.

The SMC partitioning and the live range splitting create arrays to be allocated to the SMC. Given these arrays and the register file an existing graph coloring algorithm [39] is used to determine where these arrays are going to reside in the SMC. The results are compared with [53] discussed as SMC-CT in this study. The SMC-GC on average shows an improvement of almost 3% in speedup.

| Arrays | Original Sizes in bytes | Sizes in bytes after aligned @ 8 bytes |
|---|---|---|
| A | 124 | 128 |
| B | 128 | 128 |
| C | 668 | 672 |

(a) Arrays 'A', 'B' and 'C' with there original and aligned sizes.




(b) Partioning of SMC of size 1024 bytes into a Register File. Array 'C' fits into 6 registers with the last register $R_5$ having (96 + 4) bytes of unused space. 4 bytes added for alignment to array 'C'.

Figure 2: **An Example of SMC Partitioning into a Register File**

**SMC-USize:** Nguyen et al [38] 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. Here we are going to look into three things: how and where this BW gets installed; how the data and instructions are allocated to the SMC; and how the executable is modified to make these changes.

The BW inserts code into the application executable for a customized installer. The installer is called just before the main() routine in the application and it runs just after code is loaded into the memory. The SMC size is calculated by making an OS call or by probing addresses in the memory using binary search.

The install time allocator does two jobs: profiling and allocation. Profiling is done at compile time which computes the frequency of data access. Variables with greater frequency of access are allocated first to the SMC. Other information that is required at install time like allocation and memory layout are also collected at compile time for every possible SMC size. This information is stored in a compact form. This way lot of computation and space is saved at install time. To further save space all the accesses of variables are stored in a linked list.

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 at two places, which is called code patching: start of region i.e, from the original location to the SMC; and end of region i.e, from the SMC to the original location.

Lot of information required as described above is collected at compile time. The code needs to be compiled to collect this information. Therefore only statically linked libraries with source code should be used for better results. Such libraries are recompiled to include their variables in SMC allocation. Libraries without source code are not optimized.

Results are compared with one of the author's previous work [6] on SMC discussed as SMC-Optimal in

this study, which requires the size of the SMC at compile time. On average results show a decline of $-4\%$ in performance and a reduction of 5% in energy saving. We beleive the overheads are in computing the SMC size at install time. Results are also compared with hardware cache and are not very promising. On average results show a reduction of 3% in performance and an improvement of 8% in energy saving.

**SMC-DLDP:** This [16, 17] is the first effort which presents a dynamic technique to specifically deals with data layout decision problem (DLDP) in the SMC for 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 SMC, to maximize energy saving. There are two parts to the technique to solve this problem: selection of data to be moved to the SMC based on the data access patterns; placement of this data in the SMC to reduce memory fragmentation after solving DLDP.

Data selection (At compile time) algorithm depends on data reusability factor (DRF) and the lifetime (LT) of a data element. Profiling is used at compile time to find the frequency of data access to compute the DRF of a data element. DRF is a ratio of frequency of access of an element to its estimated size in words. Data elements with DRF of more than 1 are selected. Usually these elements are large in numbers so a cluster is formed, to move them to the SMC using DMA. The lifetime is computed in two steps: First LT of an element is computed, which is the difference between it's final and initial accesses. Then LT-D is computed, which is the difference between LTs of two elements in an array. Now the data cluster is formed which is a union of data elements that have the most beneficial LT-D. In this way two kinds, first using DRF and the other using LT-D, of data clusers are formed.

The DLDP solver (At compile time) finds an order/layout for these clusters selected to fit them in the SMC. The DLDP is formulated into a two dimensional (Time and space) knapsack problem. A heuristic is given to solve this problem to find the locations, which is based on divide and conquer principle, and then clusters are loaded to the SMC at these locations using DMA. For dynamic address translation of data references, which are created by the DLDP solver, a address translation buffer in hardware is used to optimize address generation code. This address translation buffer is implemented by a set of registers and is updated by the operating system when the application is loaded. Replacement policy is decided at runtime but nothing is mentioned about how and when the data is replaced in the SMC.

The scheme presented in [17] is an improvement over their previous scheme [16]. These improvements are mentioned below:

- Tracking of data access patterns and data layout is changed from static to dynamic. To accomplish this a data access record table (DART) is implemented in the hardware. The DART records the runtime data access history, as memory addresses and frequency counters, to support the decision of data placement at runtime by the operating system. Only highly accessed memory addresses (Called working memory locations - WMLs) are kept in the DART, which are computed by profiling at compile time. The operating system updates the memory addresses inside the DART.

- Introduction of new operating system components to automatically manage the contents of the SMC. At runtime the operating system SMC manager performs two tasks: data transfer; and data access trace comparison for selecting a data layout scenario. These scenarios are computed during compile time by

the profiler and passed to the operating system before runtime.

SimpleScalar [5] is used for simulation and CACTI [48] for energy estimation. Comparisons, with different hardware cache configurations using LRU replacement policy: 1, 2, 4, 8 way set associative and different SMC sizes: 2, 4, 8 KB, are made. The results presented in [16]: improves 30% energy consumption compared to caches, similar results are shown by [8] discussed as SMC-No-Cache in this study; on average improves runtime by 18%, but 8-way set associative hardware cache gives better runtime on average 5% better than using the SMC. The improvements carried out in [17] improves the overall results by 6% compared to [16].

**SMC-MC:** The SMC implemented in this [46] work is a 4-way set associative cache in the IBM Cell processor [41] that has 8 general purpose and one special cores. 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 use fully associative replacement policy and hence gives a low cache miss overhead. A cache line table is used to map the tag to the cache line.

The replacement algorithm used is a modification of the reuse replacement algorithm [42]. The original reuse replacement algorithm keeps a reuse counter for each cache line starting with 0 and increments upto 3. Looking for a victim cache it searches and evicts the first cache line with 0 reuse counter. While searching it also decrements each of the non-zero reuse counters. The authors claim that this algorithm may introduce more misses by selecting the zero counter. The replacement algorithm modify this and initializes the counter to less than or equal to 3.

To avoid thrashing (Generation of cache misses when the working set of a parallel loop is greater than the cache size) loop distribution/fission [2] is applied, which splits the loop into multiple loops to decrease the working set. 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.

Eight OpenMP [11] applications are ported to the runtime developed for evaluation. The results are compared against indirect indexed cache [26] discussed as SMC-IIC in this study. 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 sixteen 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 works are done for desktops with one of them, SMC-VMP, designed for a multiprocessor. SMC-VMP showed poor results and SMC-IIC did not prove to be successful, results shown in column PI (Performance Improvement) of Table 1. As mentioned in SMC-VMP the reason for poor performance is the lack of a good software system or a programming environment for managing SMCs.

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 texture cache. So out of the sixteen works surveyed we consider SMC-GPU to give the best results. We list SMC-GPU as an ES in Table 1 because it is designed for GPUs, special purpose graphic processors, that are embedded inside either a GPP or a high performance ES. These GPUs are from Nvidia Corporation, which provides one of the best graphic programming environments including software and hardware called CUDA [25] as discussed before. One of the disadvantages of CUDA is that it is highly customized and can only be used for Nvidia's GPUs. Other significant programming models are: Brook [12] used by AMD and RapidMind [34] used by the new language called Ct [22], currently under development at Intel, specifically designed for GPP. Ct is an extension of C/C++ language and has a compiler and a runtime to automatically parallelize and optimize the program, that is written for a single core CPU, for a multi core CPU. It is not clear if these models provide any help for managing the SMCs.

There are also some, SMC-GPU, SMC-MC and SMC-DLDP, successful efforts in multicore processors but are all developed for ES. If SMCs can be successful in ES they can also be successful in GPPs. Unlike ES, because of the nature of applications, for any system software to be successful in GPPs it has to provide an easy to understand and programmable framework and a transparent software/hardware interface to the application programmer.

Less than half (Six) of the work discussed use profiling and are all type static, Table 1. The reason for this small number is that most of the SMCs are used in ES as shown in Table 1. ES are designed to run specific applications. Its easier to optimize the program for a specific application than for a general purpose application without profiling information.

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 factor 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 [12] used by AMD and RapidMind [34] used by the new language called Ct [22], 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 ES discussed as SMC-GPU, SMC-MC and SMC-DLDP in this paper. Application programmers for GPP 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.

| | Allocation | | | | Results | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| SMC Type | Kind | Type | Method | [1]Prof | Compared With | [2]PI | [3]E | [4]H/S | ES | GPP |
| SMC-VMP | Dynamic | ✗ | Dynamic | ✗ | Traced simulations | D | ✗ | ✓ | ✗ | ✓ |
| SMC-IIC | Dynamic | ✗ | Dynamic | ✗ | [5]HC | C | ✗ | ✓ | ✗ | ✓ |
| SMC-LT | Dynamic | [6]Var | Static | ✗ | [7]HO SMC/SMC HC | D/C | ✗ | ✗ | ✓ | ✗ |
| SMC-No-Cache | Static | Code,Data | Static | Static | HC | ✗ | C | ✗ | ✓ | ✗ |
| SMC-Optimal | Static | Var,Stack | Static | Static | Main memory | B | ✗ | ✗ | ✓ | ✗ |
| SMC-ICache-1 | Dynamic | Code | Dynamic | ✗ | HC | D | ✗ | ✓ | ✓ | ✗ |
| SMC-ICache-2 | Dynamic | Code | Dynamic | ✗ | HC | D | ✗ | ✓ | ✓ | ✗ |
| SMC-CT | Dynamic | Code,Var,Stack | Static | Static | SMC-Optimal/HC | C/C | ✗ | ✗ | ✓ | ✗ |
| SMC-As-FC | Dynamic | Code | Dynamic | ✗ | FC in Main Memory | C | ✗ | ✗ | ✓ | ✗ |
| SMC-GPU | Dynamic | Var | Dynamic | ✗ | Texture cache | A | ✗ | ✓ | ✓ | ✗ |
| SMC-Heap | Dynamic | Heap | Dynamic | ✗ | [8]DLMalloc | C | ✗ | ✗ | ✓ | ✗ |
| SMC-SMT | Dynamic | Code | Dynamic | ✗ | No cache | A | ✗ | ✓ | ✓ | ✗ |
| SMC-GC | Dynamic | Var | Static | Static | SMC-CT | C | ✗ | ✗ | ✓ | ✗ |
| SMC-USize | Static | Code,Var,Stack | Static | Static | HC/No cache | D/C | C/C | ✗ | ✓ | ✗ |
| SMC-DLDP | Static | Var | Static | Static | HC | C | C | ✓ | ✓ | ✗ |
| SMC-MC | Dynamic | Code | Dynamic | ✗ | SMC-IIC | C | ✗ | ✓ | ✓ | ✗ |

1 Profiling
2 Performance improvement
3 Energy saving
4 Implemented using both hardware and software
5 Hardware cache
6 Variables
7 Hand optimized SMC/SMC as hardware cache
8 Doug Lea's malloc() [31]

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

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. [47] presents a dynamic application profiler for space conservation and [10] 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 [50, 4]. 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 GPP as described above.

4. **Flexible**: With different sizes of SMCs and the different data patterns presented by applications running on ES and GPP, 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 ES and 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 GPP.

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 ES and introducing and making it successful for GPP.

# References


[1] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock Rate Versus IPC: the End of the Road for Conventional Microarchitectures. *SIGARCH Comput. Archit. News*, 28(2):248–259, 2000.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc, Boston, MA, USA, 2007.

[3] Randy. Allen and Ken. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, San Francisco, CA, USA, 2002.

[4] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, 1997.

[5] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, 2002.

[6] Oren Avissar, Rajeev Barua, and Dave Stewart. An Optimal Memory Allocation Scheme for Scratch-Pad-Based Embedded Systems. *ACM Trans. Embed. Comput. Syst.*, 1(1):6–26, 2002.

[7] Jose Baiocchi, Bruce R. Childers, Jack W. Davidson, Jason D. Hiser, and Jonathan Misurda. Fragment Cache Management for Dynamic Binary Translators in Embedded Systems with Scratchpad. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 75–84, New York, NY, USA, 2007. ACM.

[8] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad Memory: Design Alternative for Cache On-Chip Memory in Embedded Systems. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78, New York, NY, USA, 2002. ACM.

[9] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 1–10, New York, NY, USA, 2008. ACM.

[10] Erik Berg and Erik Hagersten. Fast data-locality profiling of native execution. In *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 169–180, New York, NY, USA, 2005. ACM.

[11] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 3.0*. Available Online: April 18, 2010 @ http://www.openmp.org/mp-documents/spec30.pdf, 2008.

[12] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.

[13] D. R. Cheriton, A. Gupta, P. D. Boyle, and H. A. Goosen. The VMP Multiprocessor: Initial Experience, Refinements, and Performance Evaluation. In *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 410–421, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[14] D. R. Cheriton, G. A. Slavenburg, and P. D. Boyle. Software-Controlled Caches in the VMP Multiprocessor. In *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*, pages 366–374, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[15] David R. Cheriton, Hendrik A. Goosen, and Patrick D. Boyle. Paradigm: A Highly Scalable Shared-Memory Multicomputer Architecture. *Computer*, 24(2):33–46, 1991.

[16] Doosan Cho, Sudeep Pasricha, Ilya Issenin, Nikil Dutt, Yunheung Paek, and SunJun Ko. Compiler Driven Data Layout Optimization for Regular/Irregular Array Access Patterns. In *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 41–50, New York, NY, USA, 2008. ACM.

[17] Doosan Cho, Sudeep Pasricha, Ilya Issenin, Nikil D. Dutt, Minwook Ahn, and Yunheung Paek. Adaptive Scratch Pad Memory Management for Dynamic Behavior of Multimedia Applications. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(4):554–567, 2009.

[18] NVIDIA Corporation. Nvidia's Next Generation CUDA Compute Architecture, Fermi. *Whitepaper NVIDIA Corporation*, © 2009.

[19] Peter J. Denning. The Locality Principle. *Commun. ACM*, 48(7):19–24, 2005.

[20] Angel Dominguez, Sumesh Udayakumaran, and Rajeev Barua. Heap data allocation to scratch-pad memory in embedded systems. *J. Embedded Comput.*, 1(4):521–540, 2005.

[21] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: programming the memory hierarchy. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83, New York, NY, USA, 2006. ACM.

[22] A. Ghuloum, T. Smith, G. Wu, X. Zhou, J. Fang, P. Guo, B. So, M. Rajagopalan, Y. Chen, and Chen B. Future-proof data parallel algorithms and software on intel multi-core architecture. *Intel Technology Journal, 2007*, 04, November 2007.

[23] Khronos OpenCL Working Group. *The OpenCL Specification Version: 1.0 Document Revision: 48*. Available Online: April 18, 2010 @ http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf, 2009.

[24] Ziyad S. Hakura and Anoop Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. *SIGARCH Comput. Archit. News*, 25(2):108–120, 1997.

[25] Tom R. Halfhill. Parallel Programming with CUDA Nvidias High-Performance Computing Platform uses Massive Multithreading. *The Insider Guide to Microprocessor Hardware*, 2008.

[26] Erik G. Hallnor and Steven K. Reinhardt. A Fully Associative Software-Managed Cache Design. *SIGARCH Comput. Archit. News*, 28(2):107–116, 2000.

[27] Chad M. Huneycutt, Joshua B. Fryman, and Kenneth M. Mackenzie. Software Caching Using Dynamic Binary Rewriting for Embedded Devices. In *ICPP '02: Proceedings of the 2002 International Conference on Parallel Processing*, page 621, Washington, DC, USA, 2002. IEEE Computer Society.

[28] Intel Corporation Inc. *3rd Generation Intel XScale(R) Microarchitecture Developer's Manual*. Available Online: April 18, 2010 @ http://www.intel.com/design/intelxscale/316283.htm, 2007.

[29] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic Management of Scratch-Pad Memory Space. In *DAC '01: Proceedings of the 38th annual Design Automation Conference*, pages 690–695, New York, NY, USA, 2001. ACM.

[30] Timothy J. Knight, Ji Young Park, Manman Ren, Mike Houston, Mattan Erez, Kayvon Fatahalian, Alex Aiken, William J. Dally, and Pat Hanrahan. Compilation for explicitly managed memory hierarchies. In

*PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 226–236, New York, NY, USA, 2007. ACM.

[31] Doug Lea. *A Memory Allocator Called Doug Lea's Malloc or dlmalloc for Short.* Available Online: April 18, 2010 @ http://gee.cs.oswego.edu/dl/html/malloc.html, 1996.

[32] Lian Li, Hui Feng, and Jingling Xue. Compiler-Directed Scratchpad Memory Management via Graph Coloring. *ACM Trans. Archit. Code Optim.*, 6(3):1–17, 2009.

[33] Lian Li, Lin Gao, and Jingling Xue. Memory Coloring: A Compiler Approach for Scratchpad Memory Management. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 329–338, Washington, DC, USA, 2005. IEEE Computer Society.

[34] Michael D. McCool. Data-parallel programming on the cell be and the gpu using the rapidmind development platform. In *GSPx Multicore Applications Conference, 2006GSPx Multicore Applications Conference, 2006*, Santa Clara, CA, USA, October 2006.

[35] Ross McIlroy, Peter Dickman, and Joe Sventek. Efficient Dynamic Heap Allocation of Scratch-Pad Memory. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 31–40, New York, NY, USA, 2008. ACM.

[36] Stefan Metzlaff, Sascha Uhrig, Jörg Mische, and Theo Ungerer. Predictable Dynamic Instruction Scratchpad for Simultaneous Multithreaded Processors. In *MEDEA '08: Proceedings of the 9th workshop on Memory performance*, pages 38–45, New York, NY, USA, 2008. ACM.

[37] Jason E. Miller and Anant Agarwal. Software-Based Instruction Caching for Embedded Processors. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 293–302, New York, NY, USA, 2006. ACM.

[38] Nghi Nguyen, Angel Dominguez, and Rajeev Barua. Memory Allocation for Embedded Systems with a Compile-Time-Unknown Scratch-Pad Size. *ACM Trans. Embed. Comput. Syst.*, 8(3):1–32, 2009.

[39] Jinpyo Park and Soo-Mook Moon. Optimistic Register Coalescing. *ACM Trans. Program. Lang. Syst.*, 26(4):735–765, 2004.

[40] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[41] D.C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P.M. Harvey, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D.L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. Overview of the Architecture, Circuit Design, and Physical Implementation of a First-Generation Cell Processor. *Solid-State Circuits, IEEE Journal of*, 41:179–196, Jan 2006.

[42] Moinuddin K. Qureshi, David Thompson, and Yale N. Patt. The V-Way Cache: Demand Based Associativity via Global Replacement. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 544–555, Washington, DC, USA, 2005. IEEE Computer Society.

[43] Manman Ren, Ji Young Park, Mike Houston, Alex Aiken, and William J. Dally. A tuning framework for software-managed memory hierarchies. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 280–291, New York, NY, USA, 2008. ACM.

[44] Scott Schneider, Jae-Seung Yeom, Benjamin Rose, John C. Linford, Adrian Sandu, and Dimitrios S. Nikolopoulos. A comparison of programming models for multiprocessors with explicitly managed memory hierarchies. *SIGPLAN Not.*, 44(4):131–140, 2009.

[45] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and Reconfigurable Software Dynamic Translation. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 36–47, Washington, DC, USA, 2003. IEEE Computer Society.

[46] Sangmin Seo, Jaejin Lee, and Zehra Sura. Design and Implementation of Software-Managed Caches for Multicores with Local Memory. In *HPCA*, pages 55–66. IEEE Computer Society, 2009.

[47] Karthik Shankar and Roman Lysecky. Non-intrusive dynamic application profiling for multitasked applications. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 130–135, New York, NY, USA, 2009. ACM.

[48] Premkishore Shivakumar and Norman P. Jouppi. Cacti 3.0: An Integrated Cache Timing, Power, and Area Model. *Compaq Western Research Laboratory Report*, 2001.

[49] Mark Silberstein, Assaf Schuster, Dan Geiger, Anjul Patney, and John D. Owens. Efficient Computation of Sum-Products on GPUs Through Software-Managed Cache. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 309–318, New York, NY, USA, 2008. ACM.

[50] Peter F. Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David Grove, and Michael Hind. Using hardware performance monitors to understand the behavior of java applications. In *VM'04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.

[51] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, 22(2):25–35, 2002.

[52] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 533–544, New York, NY, USA, 1998. ACM.

[53] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. Dynamic allocation for Scratch-Pad Memory Using Compile-Time Decisions. *ACM Trans. Embed. Comput. Syst.*, 5(2):472–511, 2006.

[54] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

[55] Lehrstuhl Informatik Xii, Rajeshwari Banakar, Stefan Steinke, Bo sik Lee, M. Balakrishnan, and Peter Marwedel. Comparison of Cache and Scratch-Pad Based Memory Systems with Respect to Performance, Area and Energy Consumption. In *Technical Report 762, University of Dortmund*, 2001.