Software Managed Manycores

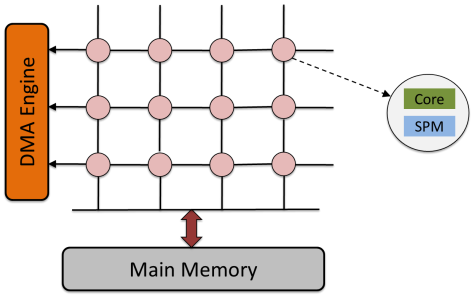

Fig. 1: *A Software programmable Memory Manycore (SMM) architecture has multiple cores, each core with a Software Programmable Memory (SPM). The data transfers between the SPMs and the main memory of the system takes place through Direct Memory Access (DMA) instructions that must be explicitly specified in the software.*

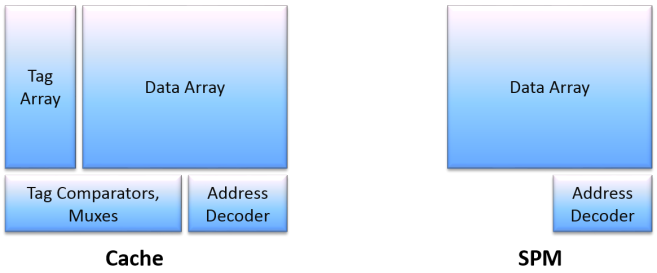

Fig. 2: *Difference between Cache and SPM—the hardware view: SPM is just a raw memory without the hardware mechanism to manage it (as is present in caches).*

Caching as a concept has been highly successful in various aspects of computer system design. Examples of cache memory abound in contemporary designs: a disk buffer cache is a small amount of buffer memory present on a hard drive to speed up the access to the disk; a web cache provides a mechanism for temporary storage of web documents to improve user experience; a DNS cache stores queried results for a period of time in the domain name system to make DNS lookup faster; P2P caching is a technique to reduce bandwidth costs for content on peer-to-peer networks; and database caching is a mechanism used to cache database content in multi-tier applications.

Perhaps the earliest use of cache memory was in processor design, where caching in various forms (e.g., data caching, instruction caching, page table caching) was exploited for improving performance by reducing accesses to slower, off-chip memories. Caches store frequently accessed data in a memory close to the processor to make the memory accesses faster and consume lower power. Traditionally caches in the processor have been implemented in hardware. Here the movement of the data between the caches and main memory (i.e., caching) is performed automatically by hardware—with the software oblivious to the caching mechanisms. Caching in computer architecture is typically implemented in hardware to reduce latency because each cache access is typically in the timing critical path of instruction execution. Another advantage of hardware caching is that programmers can develop software without worrying about caching since it is handled automatically in hardware. Furthermore, in order to bridge the increasing latency of memory accesses, multiple levels of caching (organized as a cache hierarchy) became popular in processor architectures.

However, as we scale to manycore systems, it becomes increasingly challenging to scale the corresponding cache-based memory hierarchies. One important reason is because the overhead of coherence logic increases rapidly with the number of cores. Some processors have already tried to alleviate this problem by removing hardware cache coherence from processors either partially or completely, e.g. Intel SCC, Kalray MPPA-256. In these architectures, the coherence—whenever needed by the application/system—must be implemented in software. However in these systems, caching—without coherence—is still implemented in hardware. The fact that hardware caching in manycore architectures becomes power-hungry due to the complexity of caching logic is another challenge hardware implemented caches are facing in scaling to manycore architectures.

An alternative mechanism is to deploy software caching mechanisms for smart data management, using the raw memories in the processor. Here the data movement between the close-to-processor memory and the main memory has to be done explicitly in software, typically done through the use of Direct Memory Access (DMA) instructions. We refer to such architectures as Software programmable Memory Manycore architectures (SMM), and the raw memories in such processors as Software Programmable Memories (SPM). Figure 1 shows an example of a typical SMM architecture.

A Software Programmable Memory (SPM) refers to the internal data and instruction memory array incorporated into a processor or System on Chip (SoC) architecture and controlled by software (the application itself, compiler, operating system, or a combination of them), e.g. scratchpad memory in the IBM Cell processor, TCM in ARM processors, or configurable memories in TI DSP processors. SPM is attached to the processor in much the same way as the L1 cache. However,

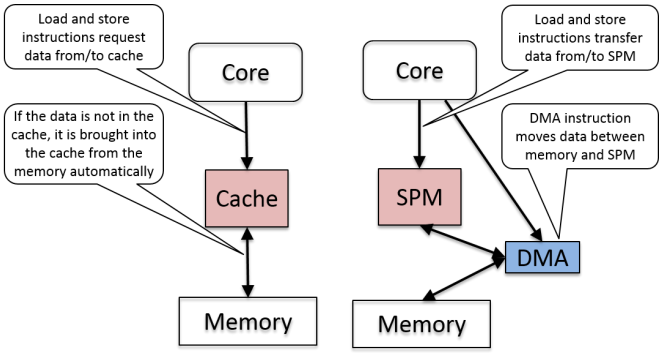

Fig. 3: *Difference between Cache and SPM—the software view: the data movement to and from the cache is performed automatically in hardware, in SPM-based systems, it must be present in the software in the form of data movement instructions.*

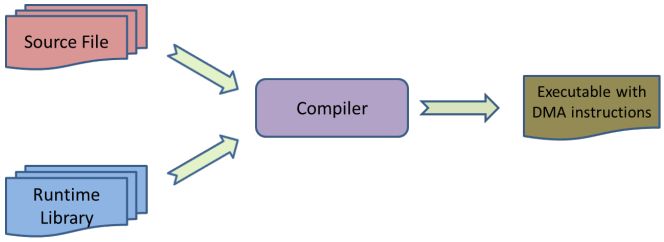

Fig. 4: *General flow of compilation of a SPM-aware compiler.*

SPM is *raw memory*, in the sense that it only contains decoding and column access logic, without the complex circuitry required to achieve hardware control of replacement policies, and managing coherence (tag directory, tag look-up circuitry, etc.). As Figure 2 shows, while a cache stores both the data and its address, an SPM only stores data, avoiding the extra lookup circuitry. As a result SPMs are about 30% smaller, slightly faster, yet consume about 30% less power than caches (for the same data capacity).

Functionally SPMs are similar to caches, in that they allow for fast access to frequently used data, but with lower power and latency. However, replacing caches with SPMs comes with its own set of challenges, as in Figure 3. Using caches is automatic; if desired data is not present in the cache, hardware mechanisms are built to bring the requested data into the cache, potentially preventing the necessity of a repeated operation if the data is reused. However, SPM contains no such hardware mechanism to automatically bring the data that is requested to the SPM. It must be brought in explicitly through memory transfer instructions that trigger Direct Memory Access (DMA) transfers. Furthermore, once data brought in, it must be accessed using its new address in the SPM, and not the original address in the main memory.

While there are some challenges in using SPMs instead of caches, the promise is that execution on SPM-based systems can be more efficient. Caches are a one-size-fits-all approach. They have one way of managing data, regardless of how the data is actually accessed. Whether some data is accessed randomly, or is accessed in a first-in-first-out manner, on a cache-based system, it will always be accessed in the manner implemented in hardware. On the other hand, SPM-based systems allow more efficient management of data by exploiting application semantics and knowledge of data access patterns, thereby enabling customization of data movement across the memory hierarchy. For example, SPMs can manage stack at stack-frame level, which is much more natural than managing them by cache blocks. By employing a coarser granularity of data management, we may be able to reduce the number of times thus the overhead for checking if the requested data is in the fast memory. More importantly, by analyzing application data access patterns, we can achieve further efficiency. If we know two stack frames will always be in the SPM at the same time by analyzing the call graph, we can bring these two stack frames from the main memory into the SPM at once, instead of fetching each of them separately. By doing so, we can reduce number of memory transfers and alleviate the overall start-up cost, and further eliminate status checking of stack frames whenever either one calls the other.

SPMs offer many advantages over caches. When application designers have deep understanding of the data requirements of their applications—especially in embedded systems—the use of SPMs allows developers to exploit application semantics effectively to achieve efficient execution. SPMs offer many other advantages over caches. The first is power efficiency by eliminating the hardware overhead of traditional caching. The second is predictability, a critical factor for real-time systems. It is hard to estimate the worst-case execution time (WCET) of software executing in cached architectures, since cache replacement policies executing in hardware result in unpredictable execution times for cache hits and misses. On the other hand, SPMs allow predictable estimation of WCET since all memory accesses are explicitly controlled in software. Third, there is potential for performance improvement by orchestrating the management of data transfers explicitly in software. Other potential benefits include the ability to explicitly manage data accesses for thermal and wearout constraints, particularly for emerging memory technologies (e.g., non-volatile memories). While there is a large body of work on using SPMs for guaranteeing WCET (e.g., for real-time applications), this article focuses on the use of SPMs for efficiency, such as in improving average-case performance, reducing power consumption, managing thermal constraints, mitigating the effects of aging, etc.

Early efforts in programming SPM-based architectures required application developers to insert data management (DMA) instructions by hand. However, with the increasing complexity of embedded software, as well as the diversity of the underlying architectures, automated techniques are required to understand the application and insert data management instructions automatically. While automatic insertion of data management instructions may be achieved in different ways,—statically by programmers or the compiler, or

dynamically through runtime systems that execute additional instructions to achieve the desired effect—the project aims to provide a generic (yet efficient) compiler-based solution for SPM management that should require no more extra hardware other than a DMA engine. The advantages of such a compiler-based approach include i) improved programmability: software develops can write their code as if hardware caching is provided, so that they can focus on their core which eventually speeds up the development cycle. ii) enhanced portability: the same application code can be reused on different versions or even different SPM-based architectures, with slight attunement of the compiler. iii) simplified hardware design: processor designers can design simple yet power-efficient manycore processors with proper compiler support. Other than the readily perceivable merits, there is another subtle yet decisive reason for choosing compiler-based approaches: iv) delivery of comparable or even better application performance than hardware caching. With deliberately designed compiler analyses, we can greatly reduce the overhead incurred by SPM management (transferal of values between the SPM and main memory) in applications. On the other hand, as we have argued, hardware caching provides a one-size-fits-all solution and cannot be benefited much with compiler analyses.

Despite of its charms, a satisfactory compiler-based SPM management is not that easy to design, since finding the optimal solution to minimize the memory transfers between the SPM and main memory is an intractable problem. Instead, we develop heuristics that will deliver high-quality (sub-optimal) results in a reasonable time period.

The general flow of a compiler-based approach is shown in Figure 4. Our compiler takes as input a source file written for the cache-based architecture and our management libraries, performs necessary compiler analyses which inserts memory transfer requests (typically DMA instructions), and generates an executable that can be run on an SMM architecture.