

Application-Aware Snoop Filtering for Low-Power Cache Coherence in Embedded Multiprocessors

XIANGRONG ZHOU, CHENJIE YU, ALOKIKA DASH, and PETER PETROV
University of Maryland, College Park

Maintaining local caches coherently in shared-memory multiprocessors results in significant power consumption. The customization methodology we propose exploits the fact that in embedded systems, important knowledge is available to the system designers regarding memory sharing between tasks. We demonstrate how the snoop-induced cache probings can be significantly reduced by identifying and exploiting in a deterministic way the shared memory regions between the processors. Snoop activity is enabled only for the accesses referring to known shared regions. The hardware support is not only cost efficient, but also software programmable, which allows for reprogrammability and customization across different tasks and applications.

Categories and Subject Descriptors: C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors); C.3 [Computer Systems Organization]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*

General Terms: Design, Experimentation

Additional Key Words and Phrases: Cache coherence, embedded multiprocessors, snoop filtering, low-power embedded systems

ACM Reference Format:

Zhou, X., Yu, C., Dash, A., and Petrov, P. 2008. Application-Aware snoop filtering for low-power cache coherence in embedded multiprocessors. *ACM Trans. Des. Autom. Electron. Syst.*, 13, 1, Article 16 (January 2008), 25 pages, DOI = 10.1145/1297666.1297682 <http://doi.acm.org/10.1145/1297666.1297682>

1. INTRODUCTION

Embedded applications, such as cell phones with data manipulation capabilities, personal organizers with various multimedia, network, and security functions, home entertainment, and various others, have become ubiquitous in our lives. Implementing these applications requires sophisticated computing

Authors' address: X. Zhou, C. Yu, A. Dash, P. Petrov (contact author), Department of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742; email: ppetrov@ece.umd.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org. © 2008 ACM 1084-4309/2008/01-ART16 \$5.00 DOI 10.1145/1297666.1297682 <http://doi.acm.org/10.1145/1297666.1297682>

ACM Transactions on Design Automation of Electronic Systems, Vol. 13, No. 1, Article 16, Pub. date: January 2008.

system platforms, frequently in the form of a complex multiprocessor systems-on-a-chip (MPSoC) [Barroso et al. 2000; Wolf 2004; Cumming 2003]. Such multiprocessor platforms are quite natural, as task parallelism and specialization are inherent for these embedded applications. However, most of these applications are extremely *energy constrained*, as they often need to operate on autonomous power sources. Even for many stationary devices with direct access to the power grid, such as set-top boxes and advanced network routers and gateways, energy efficiency is crucial due to the high cost of cooling and packaging.

Embedded systems have been traditionally designed using general-purpose hardware architectures and system software infrastructure in order to achieve flexible implementation, low design cost, and short time-to-market through the exploitation of off-the-shelf components and design reuse techniques [Wolf 2001; Sangiovanni-Vincentelli and Martin 2001]. However, such general-purpose computing architectures come with the price of excessive power consumption [Gonzalez 2000; Kathail et al. 2002], a characteristic of extreme importance for many wearable and battery-powered devices.

Shared-memory multiprocessor architectures are typically used for MP-SOC platforms, as they provide for low communication latency, well-understood programming models, and flexible platforms that can easily support heterogeneous processing units and hardware coprocessors. Since accesses to the shared memory can easily overwhelm the interconnect to memory, caches are used to bring data closer to the requesting processors, thus hiding the interconnect latency and reducing the required interconnect bandwidth. However, caches must be maintained coherently, since when a processor modifies cached data, other remote caches may be left with an older version of the same. To resolve this issue, cache-coherence protocols are commonly used. In this article we focus on one of the large classes of cache-coherence protocols, namely the snoop-coherence protocols. The snoop cache-coherence protocols are designed and very well suited for architectures in which a shared communication medium, most often in the form of a high-speed bus, is used to access the shared memory. The broadcast nature of the memory requests enables all the cache controllers to “snoop” the shared bus and through invalidation, updates, and write-backs to ensure that the local caches are maintained coherently respect to the shared memory. The provision for easily extendable multiprocessing platforms and their software-transparent implementation have made the snoop-based cache-coherence protocols easy to maintain, deploy, and reuse, while providing minimal impact on performance and memory access latency [Martin et al. 2002, 2003]. However, a major obstacle in utilizing these protocols in many modern embedded applications is their extreme energy inefficiency. The very general-purpose nature of this scheme, which has enabled their ease of integration, results in significant power consumption, thus preventing their utilization for energy-constrained embedded applications. It has been reported [Ekman et al. 2002; Loghi et al. 2005] that the power due to snoop-related cache lookup can amount to 40% of the total power consumed at the cache subsystem.

Snoop-based cache-coherence schemes are general purpose in nature, as no prior knowledge regarding the application structure, and sharing patterns, in particular, is available. It is assumed that all memory references can potentially

access shared data and need be treated as such. This is a very conservative approach, as each memory request triggers a cache lookup for all the remote processors in order to find out whether locally cached data needs to be invalidated, updated, or written back to the shared memory, thus leading to significant power consumption. Clearly, only the fraction of all memory accesses referring to shared memory potentially require remote cache invalidation or update. It has been shown [Nilsson et al. 2003] that only around 10% of the application memory footprint participates in cache-coherence activities. This power overhead can be drastically reduced if information regarding the application access patterns to shared memory are precisely captured and analyzed by compiler and system software, and subsequently dynamically exploited by the hardware.

Embedded processor and system customization approaches, where compiler, system software, and hardware are fine-tuned in their functionality and interaction with specific application requirements, have been shown powerful for achieving significant performance and power improvements [Gonzalez 2000; Lyonard et al. 2001; Kathail et al. 2002; Rowen 2004]. In this article, we introduce and evaluate a customizable snoop-cache controller architecture which filters out most of the memory requests placed on the shared bus and allows snoop-induced cache probing only for memory accesses relevant to the shared data accessed by each processor node. This is achieved by precisely identifying the shared memory regions for each task, and then providing this information to the operating system and cache snoop controller for runtime utilization. As the proposed approach identifies application information regarding shared memory regions, *close cooperation of the application code, thread package, operating system (OS) memory manager, and hardware architecture* is required. Consequently, we outline and discuss the required system software and hardware architecture augmentations needed by the proposed customization methodology; subsequently we discuss and evaluate their impact on the entire system.

Information regarding shared memory regions is captured and provided to the system software and hardware by the compiler or software developer. When a task is created, the system software is informed which set of addresses it is going to be using as shared memory regions to communicate with other tasks. The system software, in turn, captures this information and makes it available to the special hardware support, which we have introduced at the snoop-controller level. We explore several ways of capturing the information regarding the application shared regions and how to utilize this information at runtime, depending on whether a virtual memory system is being used and what major type of snoop cache-coherence protocol (update or invalidate) is being employed. The relevant application information regarding shared memory regions becomes part of the hardware context of each task and its value is stored and loaded each time the task is allocated for execution at that processor node. Since capturing the set of shared memory regions and utilizing them is implemented in a software-programmable way, a dynamic customization is achieved which enables the application of the proposed customization approach in an extremely flexible and platform/ISA-independent manner.

2. RELATED WORK

The impact and importance of cache-coherence protocols on the energy efficiency of chip multiprocessors have been recognized and addressed by the industry and research communities. A power-optimized snoop protocol for high-end general-purpose and server multiprocessors has been introduced in Moshovos et al. [2001]. The authors have introduced a cache-like hardware structure placed between the local L2 caches and the memory bus, whose purpose is to dynamically identify and subsequently filter all remote references, which are known to be not present in the local cache. As this is a hardware-only technique, the introduced snoop-filter cache is updated each time the state of the local cache is changed. An approach which dynamically tries to uncover the private memory regions for each processor node was introduced in Moshovos [2005]. This approach is based on the observation that many memory references miss in all remote processor nodes, as they are references to large private memory regions. With the help of small additional hardware, such private memory regions are identified at runtime and the corresponding snoop-induced cache lookups eliminated. In Ekman et al. [2002] a technique is proposed where virtually tagged caches, together with snoop reductions, are applied for chip multiprocessors. In order to determine whether a particular page is shared across the processors, the snoop broadcast includes a page sharing information placed on the introduced *shared vector bus*. By reading the page sharing information from the shared vector, the remote processors can eliminate snoop-induced cache lookups for pages which are privately accessed by a processor. A technique for runtime identification of data streams has been proposed in Wenisch et al. [2005]. A large number of coherence misses are eliminated by dynamically identifying sequences of memory accesses which correspond to a data stream. Coherence misses are eliminated by moving the data stream to the requesting processor in advance. In Cantin et al. [2005] a coarse-grained coherence tracking mechanism is proposed, where each processor keeps track of the coherence status of large memory regions. The snoop broadcasts are piggybacked with information regarding the coherence state of the requested address. In this way, large regions of memory are dynamically identified not to have been cached by remote processors, and as such, references to them are performed through a direct memory request. The effect of processor speculation on the power efficiency of snoop transactions is evaluated in Saldanha and Lipasti [2001]. A model for serial snoop transactions with reduced speculation is shown to reduce power, with no significant impact on performance.

Most of these studies focus on general-purpose processor architectures and in most of the cases the introduced approach relies on dynamic, runtime identification and utilization of program patterns. All of these approaches rely on the utilization of small tables or cache-like structures to dynamically identify and utilize patterns in shared memory accesses. While applicable to general-purpose server and scientific multiprocessor systems featuring very large L2 caches targeted by the snoop controllers, such approaches would prove difficult to utilize in resource-constrained embedded systems, where the area and power overhead of the introduced hardware would be prohibitively large.

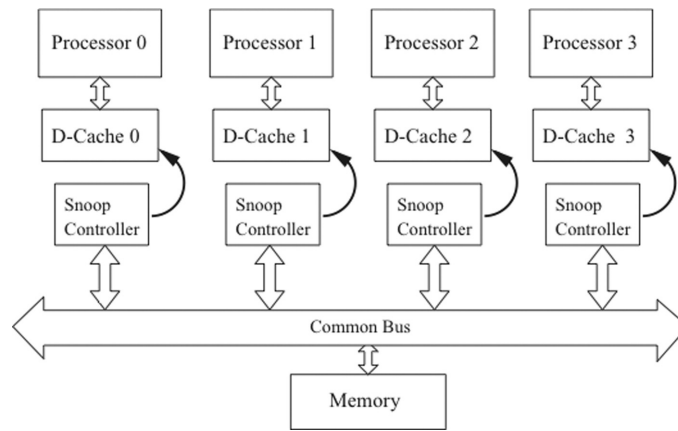


Fig. 1. Snoop cache-coherence architecture for bus-based shared memory multiprocessors.

The methodology proposed in this article targets embedded applications where application knowledge is available in advance and can be captured, analyzed, and exploited in a deterministic manner. This, in turn, results in significantly smaller and much more efficient hardware support.

3. APPLICATION-AWARE SNOOP FILTERING

The traditional general-purpose snoop cache-coherence protocols enforce cache probing at all remote processors for all memory requests placed on the shared memory bus. The general architecture of this class of protocols is shown in Figure 1. When a node places a memory request due to a read miss at the local cache, all the remote processors observe it and check whether they need to write back the requested data, in case it is present and modified in their local cache. Similarly, when the memory request on the bus is due to a write miss at a local cache, remote processors would provide the data on the bus (write-back) if it is modified in a remote cache and, subsequently, invalidate the cache line of the requested data. In this way, the requesting processor would have an exclusive copy of the written data in its local cache. This functionality is followed by the *write-invalidate* cache-coherence protocols, while in the case of a *write-update* snoop protocol, the remotely cached data is updated each time the data is written to by some processor. Write-update snoop protocols are implemented in combination with write-through caches in order to ensure that all writes to a cached data are always propagated to main memory and, hence, observed by the remote processors. A combination of write-back caches and write-invalidate snoop protocols is the variant followed by the majority of multiprocessor systems, as it results in a smaller number of coherence-related cache lookups and much less bus traffic compared to the write-update coherence scheme. In practice, write-invalidate protocols combined with write-back caches have been overwhelmingly preferred due to the extremely high cost of write-update protocols in both bus traffic and power consumption. In this article we evaluate the proposed snoop filtering scheme in the context of both these major cache-coherence approaches.

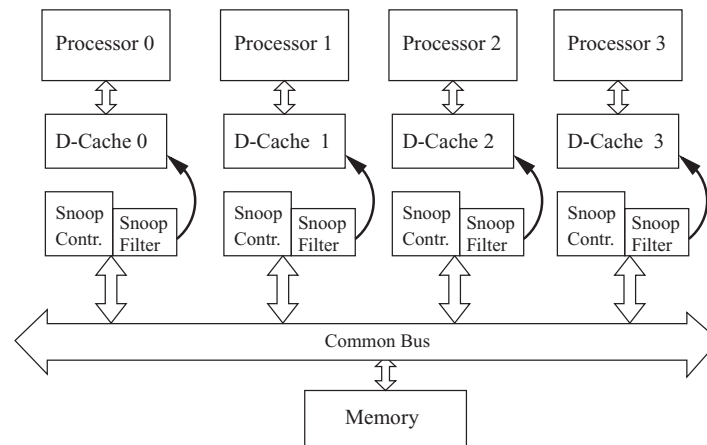


Fig. 2. Snoop-filtering architecture.

The snoop-induced cache probeds at all remote processor nodes performed at each memory request placed on the bus constitute the major reason for the excessive power consumption of snoop-coherence protocols. Clearly, in general-purpose systems where no application knowledge regarding sharing memory regions is available at hardware and system software levels, such a conservative approach is needed. It is evident, however, that snoop-induced cache lookup would hit and change the cache line status only when this line contains shared data for the task executing on the local processor. In the extreme case where all the processors execute independent tasks with no sharing of data between them, then no snoop activity would be needed at all to maintain caches coherently; only cache lines containing shared data can become incoherent, when several processors are manipulating this data at different locations.

3.1 Architecture for Selective Snooping

In the presence of application knowledge regarding shared memory regions, the snoop controllers can selectively ignore all memory requests which are irrelevant to their local processor's dataset. For example, all accesses to private data of remote processors could have no impact on the cache-coherence at a particular processor node. Only accesses to memory regions which the local processor node shares with other processors should be acted upon by the snoop controller. The fundamental approach of the technique proposed in this article is to expose to the snoop controllers application information regarding shared memory regions and program the snoop controller to react only to those memory requests referring to the shared memory regions of their processors.

The general architecture of the proposed application-aware snoop-filtering scheme is depicted in Figure 2. The *snoop filter (SF)* block is introduced with the purpose of capturing and utilizing at runtime the shared memory information from the application. The SF then allows cache probing only for memory requests to shared data which can potentially be cached at the local node. There are two important issues regarding the SF module that need to be considered

and resolved. First, how do we obtain the application information regarding shared memory regions of each processor node in the system? The second major issue is how is this information represented and provided to the SF so that it is exploited dynamically at runtime?

3.2 Shared Memory Identification and Exploitation

Information regarding shared data is readily present to the embedded software designer and, to some degree, is also available to the compiler. Typically, the embedded application which is to be mapped on a shared-memory multiprocessor is designed as a set of parallel software threads. Each thread works on various pieces of the data set and possibly communicates with other threads. Multiple software threads run within the same address space, and as such, access the same data memory. The actual communication between the threads, however, is achieved through common access to a small set of data items, typically a few data arrays.

The problem of statically identifying the shared data objects between software procedures (and software threads, in particular) has been one of the major problems in the compiler design research community for the last decade. This has been known as the *pointer analysis problem* [Hind 2001]. In general, pointer analysis attempts to determine all possible runtime values of a pointer in order to obtain information pertaining to the memory objects which can be potentially accessed by that pointer. In this way, pointer analysis, together with escape analysis (determining when a pointer leaves a scope of a procedure and hence can be used to access the data object to which it was initialized within the procedure), are used to identify the shared data objects between threads [Rugina and Rinard 1999]. Even though it has been shown that static pointer analysis is an undecidable problem [Landi 1992; Ramalingam 1994], various approximation algorithms have been offered which produce solutions with different degrees of precision. An imprecise solution may be quite conservative and include data objects which are not accessed by the pointer. Recent solutions have achieved efficient runtime complexity with high accuracy of the analysis results [Das 2000; Berndt et al. 2003]. A recent work offers an efficient solution for pointer and escape analysis for multithreaded applications [Salcianu and Rinard 2001]. The algorithm analyzes the interaction between threads and identifies the shared data objects accessed across them; the approach has been shown to be suitable for unstructured forms of multithreading, such as the one offered by the traditional POSIX threads library.

Since the proposed snoop-filtering methodology exploits knowledge regarding the shared data objects between software threads, any of the aforementioned alias analysis techniques can be applied to extract the required information. An overly conservative solution would simply translate into more snoop-induced cache lookups as compared to exactly identifying the shared memory regions, but will have no correctness or performance impact on the application program. In many cases, nonetheless, the information regarding shared memory objects between software threads can be easily provided by the software developer to the compiler/linker infrastructure. When designing and developing

the multitasking embedded application, the embedded designer partitions the functionality into multiple threads with very well-defined communication and synchronization behavior. Consequently, the actually shared data buffers can be easily tagged by the programmer by using a compiler directive. In our experimental study, we have found that for many multithreaded applications, such as those from the Splash-2 benchmark suite [Singh et al. 1992] or the MPEG video encoder from ALPBench [Li et al. 2005], the shared data arrays are explicitly defined and provided to the worker threads, thus making them relatively easy to manually identify them. Consequently, in the presentation of the proposed snoop-filtering techniques, we have assumed that the software developer would provide the information regarding the interthread shared data objects to the compiler and operating system, and have shown what would be the best mechanisms for achieving this. This is the approach we have also followed for our experimental study presented in Section 6. The shared memory arrays and variables are manually identified and their address ranges provided to the multiprocessor simulator, while the remaining memory objects accessed by each thread are considered private data and marked as such.

It is noteworthy that the proposed snoop-filtering customization approach deterministically captures and exploits the application knowledge regarding shared data items. At all times during program execution it is assumed that the shared dataset is known and fixed and that all references to addresses within this shared dataset will result in snoop-induced cache lookups. In this respect, the proposed approach needs to identify all data items that can potentially be shared during program execution and to account for them as shared. Once this is done and the information about the shared data is sent to the snoop-filtering hardware, any dynamic changes in access patterns to the shared data will not result in any changes in the snoop-filtering mechanism. In this respect, the proposed technique is conservative and does not at runtime track changes in the shared dataset, which in our estimation may require a significant hardware overhead (both area and, more importantly, power) to pay off for the eventual slight improvement of the snoop-filtering mechanism. However, coarse-grained changes in the shared dataset can be tracked by the compiler and operating system, since the proposed hardware architectures for snoop-filtering are software controlled. For such coarse-grained compiler-based tracking, special care need be taken when the shared dataset is reduced, since cache lines containing data from the larger set can still be present in the cache and lead to consistency problems. In this article we focus on and evaluate the basic approach, where the dataset is assumed permanent and all the data items that can be eventually shared are assumed a part of the shared dataset.

When the compiler is made aware that certain data arrays are used for data communication between threads, this application information needs to be provided to the snoop-filtering hardware. Depending on the memory management system offered by the multiprocessing platform and system software, this can be effectively achieved through two different methodologies which we outline and evaluate in this work. The first methodology addresses the case when no virtual memory support, in the form of MMU with address translation functionality, is offered by the platform. For these platforms we outline the *shared*

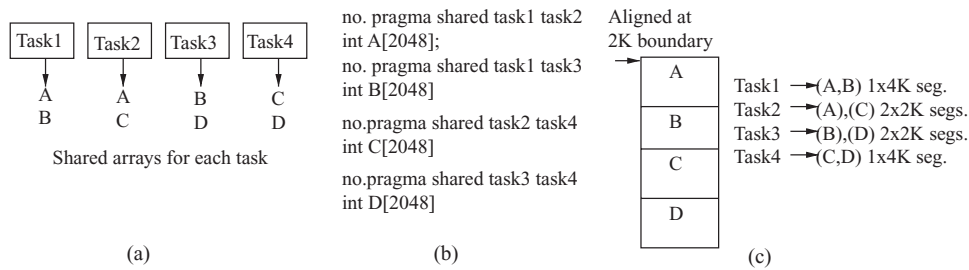


Fig. 3. SAS compiler support.

address segments (SAS) mechanism for capturing and identifying shared memory at each processor node. Here, the compiler/linker has complete knowledge and control of the task of allocating data items into system memory. The second methodology for identifying and capturing shared memory regions is the *shared page sets (SPS)* mechanism. The SPS approach is suited for platforms supporting paged virtual memory. For such platforms, the compiler/linker has control and knowledge only regarding the data memory map in the virtual address space. The actual physical memory addresses which are used to access the system memory and observed by the snoop controllers are determined by the operating system memory manager.

4. SHARED ADDRESS SEGMENTS (SAS) ARCHITECTURE

The proposed SAS methodology is applied on platforms with no virtual memory. In these platforms it is the responsibility of the compiler to layout the data items in memory, as there is no support from the hardware and operating system for physical memory allocation. The processor nodes generate addresses which are directly used to index caches and access main memory. For this reason, the compiler/linker has complete control and knowledge regarding the memory map of the program data items. The SAS architecture requires that the software designer provides information regarding shared data arrays for each thread to the compiler through means of a compiler directive (i.e., “pragma” in gcc parlance). The compiler needs this information, as it is responsible to propagate this shared data information to the snoop-filtering hardware at each processor node.

4.1 Compiler Support for SAS

As parallel tasks communicate through shared data arrays, the compiler needs to be informed which arrays are shared by which tasks. Based on this information the compiler then generates special setup code at the entrance for each task function, the purpose of which is to set up the SF hardware of the processor node with the shared data information.

Figure 3 shows an example of an application containing four tasks mapped to four processors. The tasks communicate through a total of four data arrays, where each pair of tasks actually shares data with two other tasks. For instance, *task1* shares array A with *task2* and array B with *task3*; all of the arrays in

this example are assumed to be of 2KB size. Figure 3(a) shows the shared arrays for each task. This information is transferred to the compiler by using a special directive, as illustrated in Figure 3(b). The compiler will use this information in order to analyze and transfer it to the snoop-filtering hardware support.

In systems with no virtual memory support, data arrays are always allocated by the compiler and their size is known at compile time. If array A is allocated by the compiler at starting address multiple of 2K ($= 2^{11}$ bytes), then the entire address range of this array can be uniquely identified by the 21 ($= 32 - 11$) most significant bits from the address (we have assumed a 32-bit address space). Similarly, all of the four arrays in this example, if aligned properly by the compiler, can be easily identified at runtime through the 21 most significant address bits, which we refer to as a *segment identifier (SegId)*. In this way the proposed SAS methodology identifies each shared data segment with a subset of the most significant bits (MSB) of the addresses, where the number of MSBs is determined from the segment size. In order to have a single such SegId for each shared array, we require the compiler to allocate the array at an address which is aligned to the array size. The gcc compiler offers a standard “pragma” directive for enforcing memory alignment at user-specified address boundaries. Compiler-aligned shared data is usually required for other purposes as well, for example, to avoid false sharing at cache-line level, as well as at page level for distributed memory multiprocessors. Requiring such an alignment does not constraint the compiler in any significant way since typically it will start with the largest data item and subsequently allocate the smaller items with fewer or no alignment constraints. As explained next, the existence of such an alignment is not a strong requirement, as it can be shown that the proposed approach will still work correctly without this, even though resulting in more snoop-related cache lookups.

Figure 3(c) shows a memory map of the four arrays from the example; array A is aligned at 2K address boundaries and the rest of the arrays are allocated after it. Since a processor node can have several shared memory segments, the snoop-filter hardware support must be able to capture and dynamically utilize several segment identifiers. However, it is often possible to combine shared segments into a single shared segment, since from a cache-coherence perspective it is of no importance which particular shared data is accessed at any given moment. As long as it is known that the memory access refers to any of the shared data items for that processor node, a snoop-induced cache lookup is performed. As can be seen from the example in Figure 3(c), the shared data for *task1* can be combined into a single shared segment of size 4K, for which the SegId consists of 20 ($= 32 - 12$) most significant address bits.

Exact identification of a shared data item through its segment identifier requires that the data item is of a power-of-2 size and that it is aligned in the address space. In all of the benchmarks that we have utilized for our experimental results, the shared data has been of power-of-2 size. In situations where this is not the case, the data item would still be allocated and aligned at address boundaries corresponding to the smallest power-of-2 larger than the data item size. Clearly, the segment identifier of this (larger than the data item) segment would also cover addresses which are outside the shared data item.

The proposed methodology, however, would still work correctly, as the snoop-filtering mechanisms would be slightly more conservative and would enforce cache lookups even for addresses which are outside the shared data for that processor node. Even though these few extra lookups will not result in any coherence actions on the cache lines and, as such, would be unnecessary, the correctness of the coherence protocol will be preserved. For the rare cases where such alignment is not feasible, the shared data will be assigned a segment identifier which corresponds to the next power-of-2 size that covers the unaligned shared data item. As discussed in the next subsection, such an approach that combines several segments into a single larger one will still be operational in snoop-filtering, even though more conservative in allowing snoop-related cache probing for addresses which are outside the shared data items yet inside the larger segment that covers the shared regions.

After the compiler has allocated and possibly merged some of the shared segments for each task, it will insert a special setup code at the beginning of each task, the purpose of which is to store the segment identifiers for the tasks into special registers introduced in the snoop-filtering hardware support. In this way the hardware will be provided with application information regarding shared data regions for each task and as such will be able to exploit them at runtime to allow snoop-related cache activity only for memory requests within the shared data regions for each processor node.

4.2 Hardware Support for SAS

The outlined shared address segment architecture for snoop-filtering requires a special hardware support in the form of the snoop-filtering (SF) modules. The purpose of this hardware is to capture the segment identifiers for the processor's shared segments and at runtime to filter out all memory requests placed on the bus that refer to addresses outside the set of shared segments. As each processor node may exhibit several shared-data items, the snoop-filtering hardware needs to support several shared SegIds. It is noteworthy that support for even a single segment identifier may provide for significant snoop-filtering. In this case all the shared segments need to be merged into one large segment that covers all of the initial segments. Of course, it is very likely that this single large segment will also cover a large amount of addresses which reside outside the shared data items of the processor node, thus resulting in very conservative snoop-filtering. In our scheme we experiment with support for four (4) segment identifiers. Our experimental results indicate that four segment identifiers are always enough to cover the shared data items with practically full precision.

The hardware architecture for the SAS methodology is shown in Figure 4: A snoop-filtering architecture supporting two shared-data segments is shown. Each shared data segment is uniquely associated with its segment identifier corresponding to a portion of the most significant address bits. As an extreme case, a segment identifier may consist of the entire 32-bit address, and as such would correspond to a single data word in memory. Clearly, such precision is rarely needed; the shared data arrays are typically of size in the order of hundreds of bytes. The hardware architecture outlined here supports segment identifiers

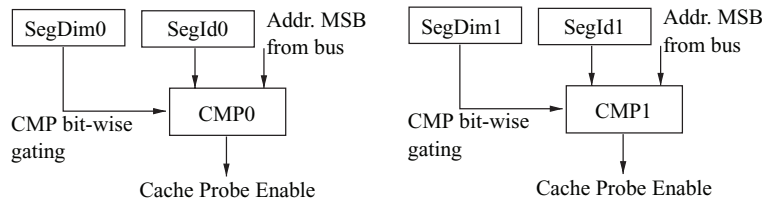


Fig. 4. SAS hardware architecture for snoop filtering.

from 32 bits to a single bit. The segment identifier is loaded by means of software to the SegId register in the beginning of the task execution. Together with the SegId register, the compiler also makes sure to inform the hardware as to the width or dimension of the segment, which it stores into the *Segment Dimension (SegDim)* register. This is a 32-bit register containing a bit-mask defining the number of address MSBs to be used as a segment identifier. For instance, a 4KB segment would require 12 address MSBs and as such, its SegDim register will need to be loaded with a bit-mask value consisting of twelve 1's followed by zeroes. The SegDim register is used as a control value to the comparators used to compare the shared data segment identifiers with the actual address bits placed on the common bus. The SegDim bit-mask is used to gate (disable) the least significant bits of the comparators and to leave active only the most significant bit positions corresponding to the segment identifier.

Overhead analysis. The *area overhead* of the introduced hardware corresponds to several registers and comparators. In the case of support for four distinct segment identifiers, a total of eight 32-bit registers and four 32-bit comparators are needed. The comparators are simple value comparators (not range comparators) and as such consist of a single XOR gate per bit position and a single wire-or transistor to aggregate the XOR outputs. The introduced silicon area is only a very small fraction of the entire cache silicon area (much smaller than 1% of a typical 32K cache). As cache-coherence maintenance is on the backend of the processor architecture and does not reside on the critical cache access path, the introduced hardware results in *no performance degradation*. In any case, the comparator delay is extremely small compared to other delays in the system, such as the bus propagation and cache probing delays. The *power overhead* of the proposed hardware is minimal as well, compared to the power needed to probe the cache. It is well worth the power consumption of a single (or few) comparators in order to decide whether to proceed or completely discard a coherence-related cache lookup. As the comparators are heavily gated, only the needed number of comparators will be active for a particular task. For example, if a task exhibits only a single shared segment, then only one of the comparators and its corresponding SegId and SegDim registers will be active. Furthermore, not all of the 32-bit comparator cells (XOR gates) will be active. Only the most significant XOR gates corresponding to the dimension of the segment identifier will be active and consume power. In our experimental results we have taken into account the very small energy expenditure consumed by the proposed snoop-filtering hardware.

5. SHARED PAGE SET (SPS) ARCHITECTURE

Increasingly, many high-end embedded microprocessor cores, such as the Intel XScale [Intel 2007] and ARM720 [Furber 2000], have started offering hardware support for virtual memory in the form of memory management units (MMUs). Employing virtual memory has a number of significant advantages, such as transparent and efficient memory allocation as well as isolation and protection for the running software tasks. In the presence of virtual memory, the processors generate virtual addresses which are then translated through the MMU and in some cases, with some OS intervention, to the actual physical addresses used to access the main memory. In order to control the complexity of the translation process, the virtual address space is separated into pages of fixed size which are uniquely identified by their *virtual page number (VPN)*: a set of the most significant addresses which are a constant for that page. The VPN is then translated into a *physical page number (PPN)*, which is a page of the same size residing in the physical memory address space.

Address translation from VPN to PPN typically happens when the address leaves the processor and before it is placed on the memory bus. In this way, the processor nodes share the physical address space, but each node can be running a task which has its own virtual address space. Address translation is also needed for the processor to access its own local cache, as typically data caches are accessed with a virtual index but a physical tag. Such a cache indexing scheme is needed in order to avoid problems such as aliasing and cache synonyms [Cekleov and Dubois 1997]. Consequently, physical addresses are placed on the common memory bus and the snoop controllers need to work with such addresses. However, physical pages are mapped to virtual ones by the operating system memory manager and this is done in a way completely transparent to application program. Therefore, the compiler and application developer have no prior knowledge regarding the actual physical addresses to which the task's shared data items will be mapped. In the previous section, we described the SAS methodology which captures and transfers application information regarding shared data address segments to the snoop-filtering hardware. In multiprocessor platforms with virtual memory support, such a scheme would not be feasible as the actual physical addresses observed by the snoop controllers are not known during compile/link and application development time, and moreover, can even change dynamically while the application tasks are executing. In such platforms the operating system needs to assume an active role in providing this application information to the hardware support.

5.1 Compiler and OS Support for SPS

The proposed shared page set (SPS) technique captures and identifies the shared data items by using the help of the operating system memory manager. The software developer provides information regarding the shared memory regions to the OS (or to the thread library and then the library code would contact the OS). Because the application program has access to the virtual addresses of its data items, it cannot directly provide the shared memory information to the snoop-filtering hardware, as in the SAS architecture. The memory

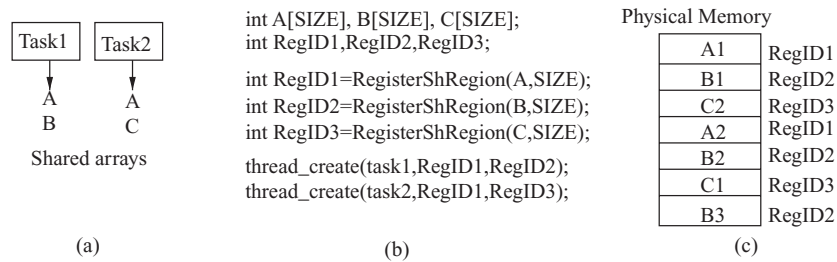


Fig. 5. SPS compiler and OS support.

manager of the OS is the only entity that can capture, identify, and provide this information.

It is noteworthy that the SAS architecture, with slight modifications, can be applied for platforms with virtual memories as well if the shared data spans only a limited number of pages. The SAS segment identifiers would correspond to the physical page numbers (PPN) and the segment dimension is always fixed to the page size. Additionally, it will be the responsibility of the operating system memory manager to write and update the values of the SegId registers, since PPNs are determined at runtime and in some situations can change if memory remapping is needed. For instance, PPNs can change when secondary storage in the form of FLASH memory is used to temporarily store some data and code pages. However, when operating with larger shared data arrays where each of the arrays can be mapped to several noncontiguous physical pages, the SAS architecture would need a large number of SegId registers in order to capture the shared data. In the case when physical pages are dispersed in the address space, combining them into a single larger segment may be unpractical, as the new segment would cover a large number of address ranges besides the shared one, thus the snoop-filtering will be very conservative.

In order to cover high-end embedded platforms which execute complex applications with large shared datasets, we propose and evaluate the shared page set (SPS) architecture, where the shared data regions are identified at page level. The OS is ultimately responsible to uniquely tag all the physical pages (possibly noncontiguous) allocated to a particular shared data item, and to subsequently pass this information to the snoop-filtering hardware. The OS memory manager needs to be informed by the application regarding the shared memory items accessed by each parallel task. For this, a slight modification to the memory manager is required in order to provide for interface functions as a part of the existing application program interface (API) to the application. Figure 5 illustrates this process through a simple example. Two tasks, each accessing two shared arrays are shown in Figure 5(a). The embedded software developer provides information about the status of arrays A, B, and C to the operating system by using the API. As shown in Figure 5(b), each array is registered with the OS as a shared region, and in turn the OS assigns to it a unique *region identifier (RegID)*. The bit-width of the RegID is determined by the number of distinct shared regions that we can support within the multiprocessor platform. As our experimental results indicate, supporting up to 8 shared regions

is enough for a four-processor platform, thus requiring 3-bit region identifiers. For platforms with more processors, the number of supported shared regions can be easily extended to 16 or even 32, with the small increase of 1 or 2 bits to the region identifier. A RegID of 0 is applied to all pages which do not belong to any shared data items declared by the application software. Note that the application passes to the OS the beginning address of the array, which is a virtual address, and its size. This is enough for the OS to determine the set of VPNs into which the array is allocated and their corresponding physical pages. A unique identifier is assigned to this set of pages and captured as part of the VPN-to-PPN translation information in the *page table* for that process. As there is a page table entry for each VPN, the unique RegID mapped to the shared regions is assigned to all the pages comprising that region. The unique RegID assigned to each shared memory region is used by the snoop-filtering hardware to determine whether a memory request observed on the common bus refers to a shared region.

After each shared array is assigned a unique identifier and registered as a shared region with the OS, the parallel threads can be created. As part of the creation process, the OS and thread library need to be informed as to which shared regions the thread is associated with. This is needed because when the thread is allocated a processor node, the snoop-filtering hardware of that node needs to be provided with that information in order to trigger coherence-cache lookups for references to these shared regions. The thread creation API can be slightly extended to include a list of region identifiers associated with this thread. An example of such a thread creation function is shown in Figure 5(b). In the example, the two parallel tasks are created and the OS is informed that *task1* refers to shared regions A and B, while *task2* refers to B and C. This information, as shown in Figure 5(c), is used by the OS memory manager to assign each page of these arrays with their unique RegID. Even though the physical pages comprising the arrays may be allocated in arbitrary order, each page is associated with its RegID as part of the VPN-to-PPN translation information maintained in the page table. During program execution, this information is captured in the *translation lookaside buffer (TLB)* as well. The TLB is simply a cache for page table entries, and as such is slightly extended to capture the region identifier for each physical page. As the TLB is looked up each time a memory request is placed on the bus, and in fact for all cache accesses in order to obtain the physical tag, the RegID for each requested memory address can be provided on the bus as a part of the memory request bus transaction. In this way, the region identifier is made available to the snoop-filtering hardware.

5.2 Hardware Support for SPS

The purpose of the SPS snoop-filtering hardware support is to capture the set of shared regions of each task and to use it at runtime to allow coherence-induced cache lookups only for accesses to the node's shared regions. This information becomes a part of the state of the task and is loaded by the operating system or thread library when the parallel task is mapped to a processor node for execution. Since the only information that the snoop-filtering hardware requires is

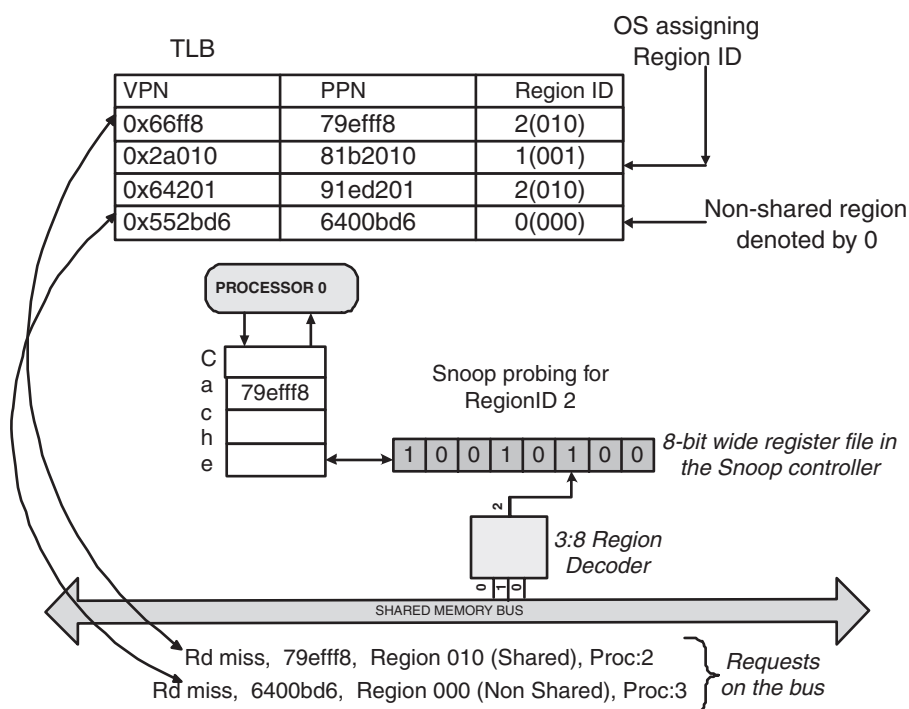


Fig. 6. SPS hardware architecture.

the set of shared regions accessed by its local processor node, this can be easily implemented through a bit-mask register. The width of this register is equal to the number of distinct shared regions supported by the platform. Consequently, each distinct region identifier can be used as an index to the bit position in the mask register, which indicates whether this shared region is accessed by the processor. For example, if the bit at position 3 is set to 1, then shared region with RegID = 3 is being accessed by the processor node.

Figure 6 depicts the proposed hardware architecture and also explains the global workings of the proposed SPS snoop-filtering architecture. As mentioned in the previous subsection, the address translation entry for each VPN is extended with the 3-bit register identifier, signifying that this VPN belongs to a particular shared memory region. This RegID is carried with the page table translation entry to the TLB as well, as shown in Figure 6. In presence of virtual memory, caches are usually physically tagged in order to prevent problems such as address aliasing and cache synonyms. Therefore, the TLB is looked up at each cache accesses (in parallel while indexing the cache) in order to obtain the physical tag of the address being accessed. In case of a cache miss, the complete physical address is placed on the memory bus to fetch data from memory. At this point in the proposed SPS architecture, the memory controller places on the common memory bus the region identifier for that address as provided by the TLB. No new bus lines are needed, as the memory request transaction caused by read or write misses to the bus carries an address only. Therefore, the

data bus lines are available and the SPS architecture places the RegID value on the least significant data bus lines. After the RegID is made available to all the remote processor's snoop controllers as part of the bus data, the hardware support needed at each SF node for using this information is rather simple. As explained earlier, this hardware consists of a single bit-mask register which is indexed by the RegID found on the least significant data bus lines of the memory access transaction.

Clearly, the SPS architecture as described here can only be used with write-invalidate snoop protocols, since these protocols react on the bus traffic caused by cache misses only. Write-update protocols, on the other hand, react to memory writes as well. However, a write bus transaction uses both the address and the data bus lines, and, as such, without adding extra bus lines or serializing the bus transaction, cannot carry the region identifier needed by the SPS architecture. Given the fact that write-invalidate coherence protocols are overwhelmingly preferable, especially in systems where power consumption is of significant concern, we restrict the SPS architecture to write-invalidate snoop protocols only.

Overhead analysis. The *hardware area overhead* of the proposed SPS snoop-filtering methodology is confined to the introduced bit-mask registers and the slightly extended TLB entries. The silicon area used by the region bit-mask register is negligible and extremely small compared to the area occupied by the data cache (well below 0.1%). The TLB extension also has no impact on silicon size in practice, as on-chip TLBs usually consist of a few entries, typically in the range of 32 to 64. Since the page table, maintained by the OS kernel in its own memory space, captures a shared region identifier for each translation entry, slightly more memory may be required by the kernel to append this new information to the page table entries. The page table entries are usually allocated in large data buffers which can capture additional information regarding access rights, besides the physical page number. In many cases, the small RegID can easily fit within the entry and thus result in no memory overhead. The worst case is when the region identifier cannot fit and a new table is allocated in addition the page table, which contains a 3-bit entry per VPN (logically extending the page table entries with 3 bits). The memory requirements for this new structure are minimal as embedded applications typically have memory footprints on the order of tens of memory pages (e.g., a very large footprint of 200 pages requires 600 bits = 75 bytes of additional memory).

In terms of *performance overhead* we need to consider the impact on the system software and hardware architecture. The only difference in the kernel operation is that when the page table is traversed, additional information will be extracted from the page table entry. This clearly has no performance implications. In terms of hardware performance overhead, the delay of indexing a simple 8-bit register is rather small compared to the cache lookup delays and even the bus transmission delay. The power overhead of the SPS architecture consists of that needed to obtain the RegID from the TLB of the requesting processor, that of transmitting the identifier on the memory bus, and that of indexing the bit-mask registers at all remote processor nodes. It is noteworthy

that the RegID is obtained from the TLB only in the case of a cache miss, thus its energy overhead applies only to memory accesses which leave the processor. Since the index to the TLB entry has already been computed as part of the cache access, the RegID bits are obtained through a simple indexing into the very small TLB data array. Transmitting the few RegID bits on the bus similarly constitutes a very small fraction of the energy needed to look up a cache structure, as the common bus is typically very short and the processors are placed in close proximity to each other in order to minimize the bus communication latency and for easy and efficient chip routing. Memory accesses outside the shared regions are tagged with a 0 region identifier, and since a larger number of memory references are to such private data, the constant 0 value on the few data bus lines would require a negligible amount of energy. In our experimental studies we have taken into account these energy overheads and we present more details regarding this in Section 6.

6. EXPERIMENTAL RESULTS

To quantitatively evaluate and analyze the proposed snoop-filtering techniques, we have performed detailed multiprocessing simulations on a number of application benchmarks, including audio/video processing and parallel numerical and signal processing programs. For our simulations we have used the M5 multiprocessing simulator [Binkert et al. 2006]. We have utilized the user-level simulator extended with a collection of thread synchronization primitives. For our experiments, a four-processor platform has been instantiated and simulated upon. Because of limited interconnect bandwidth shared memory multiprocessors, especially for resource-constrained embedded applications, we are usually limited to several processors placed on a shared memory bus. If more processors are needed, a more complex and oftentimes hierarchical interconnect is employed. Snoop cache-coherence protocols will be applied only locally in such architectures, while cross-layer coherence would be maintained through point-to-point communication, usually using coherence protocols based on directories [Lenoski et al. 1990]. For our study we have chosen a four-processor platform as a representative for embedded shared-memory multiprocessors. For applications where all threads share the same data, the achieved reductions would be largely independent from the number of processors, but would only depend on the ratio of shared and private data as well as the threads' miss rates. For applications where the data is shared only between two or a small subset of processors, the reductions achieved by the proposed approach would increase linearly with the number of processors, as accesses to this "locally" shared data will be filtered at all remote processors.

The simulated processors exhibit the Alpha ISA, a classical RISC ISA, the principles of which have been used in many embedded processors, such as the ARM, MIPS, and Tensilica's Extensa processor cores. A gcc cross-compiler has been used to compile the multithreaded benchmarks. We have evaluated both the write-update and write-invalidate snoop cache-coherence protocols. The write-invalidate protocol is traditionally coupled with write-back caches, while the write-update protocol is combined with write-through caches (note that the

Table I. Tag Per-Access Energy Expenditure (in pJ) for the Cache Architectures

	32K DM	32K 4-SA	16K DM	16K 4-SA
Tag Energy	35.97	62.56	26.35	54.89

write-update coherence policy can only work with write-through caches). The energy expenditure for both the baseline architectures and proposed methodologies has been estimated by using the Cacti v4.2 tool [Tarjan et al. 2006]. Cacti is a well-known and used tool for estimating area, delay, and energy expenditure of standard cache architectures. We have evaluated the proposed methodologies on four different cache organizations: 32K and 16K direct-mapped and 4-way set associative architectures. Table I shows the tag energy consumption for the four cache organizations, with a process technology node of 0.18 μm .

We have used several types of multiprocessing benchmarks to evaluate the proposed methodology. All of the benchmarks have been ported for the synchronization library as offered by the extended M5 simulator; the synchronization library includes support for basic *locks* and *barriers*. The first part of our benchmarks consists of the three numerical kernel benchmarks *fft*, *lu*, and *radix*, from the Splash-2 benchmark suite [Singh et al. 1992]. These kernels are typically encountered in various signal processing applications. The remaining benchmarks from the Splash-2 suite are scientific applications and their relevance to the embedded domain is limited. To provide for a broader set of benchmarks, we have developed two additional parallel application benchmarks, each consisting of four well-known embedded algorithms and kernels. The first, which we refer to as *S1*, is comprised of the *advanced encryption standard (rijndael)*, the *Lempel-Ziv-Oberhumer (LZO)* data compression algorithm, the *matrix multiplication (mmul)* algorithm, and the *lu* numerical kernel. The four programs are organized to run on a stream of data passed through them in a pipelined manner. They have been load-balanced and synchronized through barriers. In these benchmarks, each of the four tasks communicates with two of the other tasks that are mapped on its neighboring processor nodes. The second benchmark of this type, which we refer to as *S2*, consists of the *adpcm* speech coder, the *fft* signal processing kernel, the *lu* numerical kernel, and the *cyclic redundancy check (crc)* computation. The organization and synchronization of *S2* is similar to *S1*. Finally, we have evaluated the parallelized MPEG video encoding application from the ALPBench [Li et al. 2005] set.

Table II reports snoop-related cache activities for the baseline architectures with 32K data caches, while Table III shows the same type of data but for 16K data caches. All the numbers in these two tables, as well as in the subsequent tables are in thousands. Each entry contains a pair of numbers: the first corresponds to a direct-mapped cache organization, while the second is for 4-way set associative organization. The first three rows of these tables are compound where each consists of four subrows reporting the data for the four processors in the system. The first row shows the cache read misses at each of the four processors, the second row shows the write misses, while the third row reports on the memory writes generated by each processor. The number of writes is

Table II. Baseline Snoop Activity ($\times 1000$) for 32K (DM/4-SA) Caches

	S1	S2	fft	lu	Radix	MPEG
Read	1499/2380	1508/2384	2/3	1948/851	23/20	36/28
Misses	554/118	3050/458	2/2	1874/746	24/21	38/28
	151/42	48/41	2/2	1896/746	22/19	35/27
	304/191	0.09/0.08	2/2	1965/781	25/20	71/22
Write	104/95	137/128	4/4	1146/482	25/20	31/25
Misses	264/33	1840/554	2/2	1046/361	28/23	30/24
	9/8	25/21	2/2	1056/362	25/21	27/23
	155/104	0.04/0.07	2/2	1098/379	26/21	31/16
Writes	11221/11230	11453/11462	58/58	22544/23208	636/640	4864/4869
	8391/8622	8369/9655	17/17	10280/10964	633/637	5339/5345
	5403/5404	3974/3978	17/17	10266/10960	631/636	5376/5379
	2083/2133	132/132	17/17	10775/11493	635/639	5014/5029
Snoops (WI)	9132/8921	19871/10769	74/74	36127/14135	605/504	7584/12713
Snoops (WU)	90415/91080	91637/86440	387/389	197675/183998	8194/8151	62674/62446

Table III. Baseline Snoop Activity ($\times 1000$) for 16K (DM/4-SA) Caches

	S1	S2	fft	lu	Radix	MPEG
Read	4119/4521	4146/4523	5/4	2613/876	27/20	83/34
Misses	1636/1114	3069/475	3/3	2104/769	31/23	67/34
	169/43	50/41	3/3	2024/770	25/20	68/33
	338/241	0.09/0.08	3/3	2208/804	3322	68/27
Write	139/110	172/143	5/5	1188/488	30/24	45/28
Misses	288/33	1843/571	2/2	1140/364	33/28	32/28
	11/8	26/21	2/2	1101/364	31/25	32/27
	168/122	0.08/0.07	2/2	1196/381	32/26	44/20
Writes	11185/11215	11418/11447	58/58	22502/23202	631/637	4849/4866
	8367/8622	8366/9638	17/17	10186/10962	628/633	5337/5341
	5401/5404	3973/3978	17/17	10221/10958	626/632	5371/5376
	2069/2115	132/132	17/17	10676/11491	628/635	5001/5025
Snoops (WI)	20620/18584	27945/17337	89/82	57544/14455	738/572	15469/13390
Snoops (WU)	101677/100646	99583/92908	399/395	201475/184282	8259/8169	63199/62517

relevant for write-through caches with a write-update snoop-coherence protocol. In our experimental studies we have evaluated the two major cache-coherence protocols. The fourth row in Tables II and III, labeled by *Snoops (WI)*, shows the total number of snoop-induced cache lookups for the write-invalidate protocol. The last row in both tables shows the total number of snoop-related cache lookups for the write-update case. It can be seen that with write-update, the number of snoop events is significantly increased (around $5\times$), which can be easily explained by the fact that memory writes result in additional snoop events as compared to write-invalidate protocols.

Tables IV and V present the data for the proposed snoop-filtering methodology: The first table reports for 32K, while the second is for a 16K data cache. Each entry contains a pair of numbers: the first corresponding to a direct-mapped cache organization, while the second one refers to a 4-way set associative organization. The first three rows report the data for write-invalidate snoop protocols; the last three rows refer to a write-update protocol. For all the benchmarks it has been possible to capture precisely the shared data items and to provide their address segments (for the SAS approach), or the set of shared

Table IV. Snoop Activities ($\times 1000$) for the Proposed Snoop-Filtering Methodology; 32K (DM/4-SA) Caches

	S1	S2	fft	lu	Radix	MPEG
Snoops (WI)	253/68	131/33	11/12	8840/3373	69/62	85/43
	1555/2437	1572/2458	3/3	547/291	69/61	92/50
	645/134	4752/974	4/4	479/262	69/62	93/50
	16/16	34/33	.51/.52	497/262	67/61	72/49
Total	2469/2656	6490/3497	19/20	10362/4188	274/247	342/192
Red.(%)	73.0/70.2	67.3/67.5	74.4/73.2	71.3/70.4	54.7/51.1	95.5/98.5
Snoops (WU)	253/68	131/33	58/59	39718/36339	117/115	601/563
	1591/2474	1707/2593	12/12	838/583	117/115	611/589
	4649/4361	4752/2257	16/16	770/554	114/112	615/576
	139/139	148/147	0.66/0.68	788/554	109/109	592/576
Total	6632/7043	6739/5030	87/88	42113/38030	457/452	2419/2289
Red.(%)	92.7/92.3	92.7/94.2	77.5/77.5	78.7/79.3	94.4/94.5	96.1/96.3

Table V. Snoop Activities ($\times 1000$) for the Proposed Snoop-Filtering Methodology; 16K (DM/4-SA) Caches

	S1	S2	fft	lu	Radix	MPEG
Snoops (WI)	1291/1065	131/33	13/13	9577/3444	87/73	197/71
	4173/4584	4189/4602	4/4	564/307	83/72	205/76
	690/161	4753/1005	5/5	556/277	87/73	203/76
	17/16	35/33	.68/.59	513/277	79/71	164/76
Total	6170/5827	9108/5673	22/22	11210/4305	336/290	769/299
Red.(%)	70.8/68.7	67.4/67.3	74.9/73.3	80.5/70.2	54.5/49.3	95.0/97.8
Snoops (WU)	1921/1065	131/33	60/59	40227/36405	126/116	706/580
	4194/4606	4308/4722	12/12	854/595	123/116	717/589
	4678/4388	4753/2273	17/17	846/565	123/113	718/592
	140/139	148/147	.72/.74	803/565	115/110	678/592
Total	10302/10198	9341/7175	90/89	42728/38131	486/456	2818/2354
Red.(%)	89.8/89.9	90.6/92.3	77.5/77.4	78.8/79.3	94.1/94.4	95.5/96.2

pages (for the SPS approach) to the simulation environment. The first compound row presents the number of snoop-induced cache lookups for each of the four processors in the system. The total number of snoop-induced cache lookups is reported in the next row, labeled *Total*. The third row presents the achieved reduction of snoop-related cache lookups as compared to the baseline architecture; the reductions are reported as percentages. It can be observed that the reductions range from 50% up to 98%. On the low end of this range is the *radix* benchmark, the reason for this being its relatively large shared arrays as well as the very small amount of private data for each task. The largest reductions have been achieved for the MPEG application. The MPEG benchmark is the most complex one, with a large amount of private data items being accessed within the parallel threads. Consequently, the ability to distinguish the shared data for each node and to filter all other memory references, results in significant snoop reductions in the range of 95% to 98%. The last three rows present the same data but for the write-update coherence protocol. Here the reductions are larger than the reduction for write-invalidate, which is due to the fact that many redundant snoop events caused by consecutive writes within the shared data are completely eliminated. The reductions for write-update protocols are in the

Table VI. Energy (in μJ) for Baseline and Proposed Snoop-Filtering Methodology; 32K (DM/4-SA) Caches

	S1	S2	fft	lu	Radix	MPEG
Base(WI)	329/558	715/674	2.7/4.6	1300/884	21.8/31.6	273/795
Base(WU)	3252/5698	3296/5407	14.0/24.3	7111/11510	295/510	2255/3906
SAS(WI)	99.9/177	258/232	0.77/1.3	417/279	11.7/16.1	21.5/27.5
Red.(%)	69.6/68.3	64.0/65.6	71.0/71.2	67.9/68.4	51.3/49.1	92.1/96.6
SAS(WU)	349/551	354/420	3.6/6.0	1755/2603	26.4/38.2	163/219
Red.(%)	89.3/90.3	89.3/92.2	74.2/75.5	75.3/77.4	91.0/92.5	92.8/94.4
SPS(WI)	114/191	288/248	0.88/1.4	472/301	11.5/16.8	33.2/46.9
Red.(%)	65.3/65.8	59.7/63.1	66.8/71.2	63.7/66.0	47.1/46.7	87.9/94.1

Table VII. Energy (in μJ) for Baseline and Proposed Snoop-Filtering Methodology; 16K (DM/4-SA) Caches

	S1	S2	fft	lu	Radix	MPEG
Base(WI)	543/1020	736/952	2.4/4.5	1516/793	19.5/31.4	407/735
Base(WU)	2679/5524	2624/5099	10.5/21.7	5309/10115	218/448	1665/3431
SAS(WI)	188/342	274/332	0.7/1.3	365/254	9.7/16.6	39.1/32.7
Red.(%)	65.5/66.4	62.8/65.1	70.3/71.1	76.0/68	49.9/47.1	90.4/95.6
SAS(WU)	395/682	367/507	2.8/5.4	1371/2317	22.9/35	151/205
Red.(%)	85.3/87.7	86.0/90.1	72.9/75.2	74.2/77.1	89.5/92.2	90.9/94.0
SPS(WI)	219/371	317/359	0.8/1.4	454/276	10.9/17.5	62.8/53.2
Red.(%)	59.6/63.6	57.0/62.3	64.5/68.3	70.1/65.2	44.09/44.3	84.6/92.8

range from 70% up to 96%. Interestingly, only for MPEG are the write-update reductions slightly smaller than the write-invalidate reductions. This can be explained by the much larger number of reads to shared data items and the relatively few writes to them occurring only at the end of the thread processing.

The reductions presented in Tables IV and V correspond to the ideal case, where no overheads exist in achieving application-specific snoop-filtering. In such an ideal yet unachievable situation, the energy reduction would exactly match the reduction in snoop-related cache loops. To demonstrate the utility of our methodologies, the next two tables show the actual amount of energy expenditure for the baseline and proposed SAS and SPS snoop-filtering approaches. The energy reported for the SAS and SPS architectures takes into account all the power overhead introduced by the additional snoop-filtering hardware.

Tables VI and VII show the actual energy numbers for the baseline and proposed SAS and SPS application-specific snoop-filtering architectures. The first two rows report the energy expenditure for write-invalidate and write-update snoop protocols, respectively. It is evident that the actual energy needed by write-update protocols is significantly larger than that for write-invalidate. The next three pairs of rows report the energy consumed by the proposed SF architectures. The SAS architecture is evaluated for both write-invalidate and write-update protocols and reported in the SAS(WI) and the SAS(WU) rows, respectively. The actual energy and actual reductions as compared to the baseline energy is reported in the next rows. For the SAS architecture we have accounted for the energy taken by the address segment comparators. The energy for these comparators has been obtained by the Cacti tool and conservatively have been set to identify segments of size 8K, even though for most of the benchmarks the

shared data is of larger size, which leads to even smaller active comparators. It can be seen that the difference between the SAS(WI) reductions and the ideal reductions is in the range of 2% to 4%. A similar difference range is observed for the SAS(WU) case, as reported in the subsequent two rows.

The last two rows of Tables VI and VII report the actual energy numbers for the SPS architecture. As explained before, the SPS architecture is applicable to write-invalidate protocols only, which cover the large majority of actual snoop-coherence implementations due to the very great energy impact and extremely large bus traffic caused by the write-update protocols. The SPS overhead includes three components. The first overhead component consists of obtaining the region identifier from the TLB before placing it on the bus, together with the entire bus transaction. As explained in the previous section, this operation is performed only when a cache miss occurs and placing a memory request on the common bus is inevitable. Since the TLB index has already been computed for a cache access, what is left to be done on a cache miss is to use this index to obtain the 3-bit region identifier from the data arrays. For this, by using Cacti we have modeled a 64-entry SRAM array (assuming 64-entry TLB, a typical number for high-end embedded processors) with 3 bits per entry. The next overhead component is placing the region identifier on the data bus lines. To estimate the energy overhead of this component, we have used the on-chip bus parameters reported in Bashirullah et al. [2003] for a 0.35 μm technology process. This implementation study reports that each bus line exhibits a total of 2.56pF of capacitance for a length of 1 cm. This is a rather conservative situation, as memory buses are typically much shorter due to the compact nature of shared-memory multiprocessors and the limited number of processors connected to that shared bus. We have scaled the data to 0.18 μm process technology with $V_{dd} = 1.66\text{V}$ as reported by Cacti, and have subsequently computed the energy corresponding to 3 bus lines, assuming 50% transition activity. Finally, the snoop-filtering hardware at each node requires a bit-mask register capturing the processor's shared regions. It is indexed by the region identifier provided on the bus. We have modeled this 8-bit register as a collection of eight latches, where each latch consists of four logic gates. It can be observed now that, as expected, the SPS architecture results in a slightly larger gap between the ideal, no-overhead case and what is achieved in reality due to the hardware overhead. The reduction difference as compared to the no-overhead baseline is from 4% up to 10%. The actual gap differs across the various cache organizations. It can be seen that for set associative cache organizations, which are more energy consuming due to the parallel tag lookup and compare operations, the overhead gap for both the SPS and SAS architectures is smaller and close to around 4% for SPS and 2% for SAS. This can be explained by noticing that the small and constant overhead is being used to eliminate lookups which are more energy expensive than in the case of direct-mapped caches.

7. CONCLUSIONS

In this article, we have presented a low-power methodology for cache coherence for embedded multiprocessor systems. The proposed approach exploits

application information regarding shared memory regions of parallel tasks in order to eliminate a large number of power-consuming snoop-induced cache probings. When created, each parallel task provides the system software and snoop-filtering hardware with information regarding the shared arrays and variables used to communicate with other tasks mapped on remote processor nodes. We have proposed two different architectures for achieving such an application-specific snoop-filtering. The shared address segment (SAS) architecture is well suited for systems with no virtual memory support, or systems with relatively small and not significantly dispersed shared data items. The shared page set (SPS) architecture is suited for high-end embedded multiprocessors featuring virtual memory as well as large and scattered shared data items. The proposed methodology is very cost efficient, as the required additions to the system software and hardware architectures are minimal and impose no performance or area overheads. Such an approach would be of great utility to a large number of modern embedded applications for which both high performance and low power are of significant importance.

REFERENCES

- BARROSO, L., GHARACHORLOO, K., MCNAMARA, R., NOWATZYK, A., QADEER, S., SANO, B., SMITH, S., STETS, R., AND VERGHESE, B. 2000. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. ACM Press, New York, 282–293.
- BASHIRULLAH, R., LIU, W., AND CAVIN, R. K. 2003. Low-Power design methodology for an on-chip bus with adaptive bandwidth capability. In *Proceedings of the Design Automation Conference (DAC)*. ACM Press, New York, 628–633.
- BERNDL, M., LHOTAK, O., QIAN, F., HENDREN, L., AND UMANEE, N. 2003. Points-To analysis using BDDS. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 103–114.
- BINKERT, N., DRESLINSKI, R., HSU, L., LIM, K., SAIDI, A., AND REINHARDT, S. 2006. The m5 simulator: Modeling networked systems. *IEEE Micro*. 26, 4, 52–60.
- CANTIN, J. F., LIPASTI, M. H., AND SMITH, J. E. 2005. Improving multiprocessor performance with coarse-grain coherence tracking. *SIGARCH Comput. Archit. News* 33, 2, 246–257.
- CEKLEOV, M. AND DUBOIS, M. 1997. Virtual-address caches. Part 1: Problems and solutions in uniprocessors. *IEEE Micro*. 17, 5 (Sept.), 64–71.
- CUMMING, P. 2003. The TI OMAP platform approach to SoC. In *Winning the SOC Revolution*. Kluwer Academic.
- DAS, M. 2000. Unification-Based pointer analysis with directional assignments. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 35–46.
- EKMAN, M., DAHLGREN, F., AND STENSTROM, P. 2002. TLB and snoop energy-reduction using virtual caches in low-power chip-microprocessors. In *Proceedings of the International Symposium on Low-Power Electronics and Design (ISLPED)*, 243–246.
- FURBER, S. B. 2000. *ARM System-on-Chip Architecture*. Addison-Wesley, Boston, MA.
- GONZALEZ, R. E. 2000. Xtensa: A configurable and extensible processor. *IEEE Micro*. 20, 2, 60–70.
- HIND, M. 2001. Pointer analysis: Haven't we solved this problem yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*.
- INTEL CORPORATION. 2007. Intel XScale Microarchitecture. <http://www.intel.com/design/intelxscale/316283.htm>.
- KATHAIL, V., ADITYA, S., SCHREIBER, R., RAU, B. R., CRONQUIST, D. C., AND SIVARAMAN, M. 2002. Pico: Automatically designing custom computers. *IEEE Comput.* 35, 9, 39–47.
- LANDI, W. 1992. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.* 1, 4 (Dec.), 323–337.

- LENOSKI, D., LAUDON, J., GHARACHORLOO, K., GUPTA, A., AND HENNESSY, J. 1990. The directory-based cache-coherence protocol for the dash multiprocessor. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. ACM Press, New York, 148–159.
- LI, M.-L., SASANKA, R., ADVE, S., CHEN, Y.-K., AND DEBES, E. 2005. The ALPbench benchmark suite for complex multimedia applications. In *Proceedings of the International Symposium on Workload Characterization*, 34–45.
- LOGHI, M., LETIS, M., BENINI, L., AND PONCINO, M. 2005. Exploring the energy efficiency of cache-coherence protocols in single-chip multi-processors. In *Proceedings of the 15th Great Lakes Symposium on VLSI (GLSVLSI)*, 276–281.
- LYONNARD, D., YOO, S., BAGHDADI, A., AND JERRAYA, A. 2001. Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip. In *Proceedings of the Design Automation Conference (DAC)*. ACM Press, New York, 518–523.
- MARTIN, M. K., HILL, M. D., AND WOOD, D. A. 2003. Token coherence: Decoupling performance and correctness. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. ACM Press, New York, 182–193.
- MARTIN, M. M. K., SORIN, D. J., HILL, M. D., AND WOOD, D. A. 2002. Bandwidth adaptive snooping. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 251–262.
- MOSHOVOS, A. 2005. Region scout: Exploiting coarse grain sharing in snoop-based coherence. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Washington, DC, 234–245.
- MOSHOVOS, A., MEMIK, G., CHOUDHARY, A., AND FALSAFI, B. 2001. Jetty: Filtering snoops for reduced energy consumption in SMP servers. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Washington, DC, 85–96.
- NILSSON, J., LANDIN, A., AND STENSTROM, P. 2003. The coherence predictor cache: A resource-efficient and accurate coherence prediction infrastructure. In *Proceedings of the International Symposium on Parallel and Distributed Processing*. IEEE Computer Society, Washington, DC, 10–17.
- RAMALINGAM, G. 1994. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.* 16, 5, 1467–1471.
- ROWEN, C. 2004. *Engineering the Complex SOC. Fast, Flexible Design with Configurable Processors*. Prentice Hall, NJ.
- RUGINA, R. AND RINARD, M. 1999. Pointer analysis for multithreaded programs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI) 34*, 5, 77–90.
- SALCIANU, A. AND RINARD, M. 2001. Pointer and escape analysis for multithreaded programs. In *Proceedings of the Symposium on Principles and Practices of Parallel Programming (PPoPP)*, 12–23.
- SALDANHA, C. AND LIPASTI, M. 2001. Power efficient cache-coherence. In *Workshop on Memory Performance Issues*.
- SANGIOVANNI-VINCENTELLI, A. AND MARTIN, G. 2001. Platform-Based design and software design methodology for embedded systems. *IEEE Des. Test Comput.* 18, 23–33.
- SINGH, J. P., WEBER, W.-D., AND GUPTA, A. 1992. Splash: Stanford parallel applications for shared-memory. *SIGARCH Comput. Archit. News* 20, 1, 5–44.
- TARJAN, D., THOZIYOOR, S., AND JOUPPI, N. 2006. Cacti 4.0: An integrated cache timing, power and area model. Tech. Rep., HP Laboratories, Palo Alto, CA. June.
- WENISCH, T. F., SOMOGYI, S., HARDAVELLAS, N., KIM, J., AILAMAKI, A., AND FALSAFI, B. 2005. Temporal streaming of shared memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Washington, DC, 222–233.
- WOLF, W. 2001. *Computers as Components: Principles of Embedded Computing Systems Design*. Morgan Kaufmann, San Francisco, CA.
- WOLF, W. 2004. The future of multiprocessor systems-on-chips. In *Proceedings of the Design Automation Conference (DAC)*, 681–685.

Received May 2006; revised May 2007; accepted July 2007