Journal of Systems Architecture 55 (2009) 196–208

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Broadcast filtering: Snoop energy reduction in shared bus-based low-power MPSoCs

Chun-Mok Chung, Jihong Kim *

School of Computer Science & Engineering, Seoul National University, Seoul 151-742, Republic of Korea

ARTICLE INFO

Article history:
Received 8 December 2007
Received in revised form 2 December 2008
Accepted 12 January 2009
Available online 20 January 2009

Keywords:
Multiprocessor system-on-a-chip (MPSoC)
Snoop energy reduction
Snooping cache
Split bus
Broadcast filtering

ABSTRACT

In multiprocessor system-on-a-chips (MPSoCs) that use snoop-based cache coherency protocols, a miss in the data cache triggers the broadcast of coherency request to all the remote caches, to keep all data coherent. However, the majority of these requests are unnecessary because remote caches do not have the matching blocks and so their tag lookups fail. Both the coherency requests and the tag lookups corresponding to a remote miss consume unnecessary energy.

We propose an architecture-level technique for snoop energy reduction, called *broadcast filtering*, which prevents unnecessary coherency requests from being broadcast to remote caches, and thus reduces snoop energy consumption by both the cache and bus. Broadcast filtering is implemented using a snooping cache and a split bus. The snooping cache checks if a block that cannot be obtained locally exists in remote caches before broadcasting a coherency request. If no remote cache has the matching block, there is no broadcast; and if broadcasting is necessary, the split bus allows coherency requests to be broadcast selectively to the remote caches which have matching blocks.

Experimental results show a reduction by 90% of cache lookups, by 60% of bus usage, and by 40% of snoop energy consumption, at a small cost in reduced performance. An analysis result based on the energy model shows the broadcast filtering technique can reduce by up to 55% of energy consumption per cache coherency operation.

1. Introduction

A multiprocessor system-on-a-chip (MPSoC) consists of several processor cores and memories on a single chip die. MPSoCs can execute multiple contexts in parallel, allowing them to meet the high computational demands of many multimedia applications, and are widely accepted as a next-generation architecture for high-performance mobile embedded systems [1,2] such as cellular phones and personal game players. Because these systems run on batteries, low power consumption is becoming increasingly important in designing MPSoCs.

Although there is as yet no clear consensus on the architecture and organization of MPSoCs, many early chips, such as ARM MPCore [1] and Stanford Hydra [3], employ a shared bus architecture. In shared bus MPSoCs, snooping is widely used to address the cache coherency problem. If a local cache requires or modifies data, it broadcasts a coherency request message and remote caches snoop on the broadcast to maintain data coherency. Since on-chip global wires are responsible for up to 25% of the total power consumption of a chip [4], and tag lookup operations contribute up to 50% of the cache energy consumption [5], cache coherency operations are a significant power consumer in MPSoCs, and therefore a prime target for energy-saving strategies.

Several researchers have already proposed cache energy reduction techniques for MPSoCs [5–7]. These techniques are all based on the observation that many coherency requests do not find matching blocks in remote caches, in which case both the coherency broadcasts and the subsequent remote cache lookups are useless. If *n* remote caches have the required block, we say that there are *n* remote hits. But, if none of the remote caches has the required block, it is a remote miss. Fig. 1 shows how many remote hits occurs during the execution of parallel applications on an MPSoC which has a similar configuration to the ARM MPCore (it consists of four processors and each processor has 32-Kbyte L1 data cache that is four-way set-associative with 32-byte blocks. MESI [19] is used as the snoop protocol). We can see that the remote miss ratio (the proportion of remote misses) is high, averaging 83% of all coherency requests.

However, most existing snoop energy reduction techniques are only partially effective, because they only focus on reducing the cache energy consumption. But cache coherency operations require bus transactions, and the energy consumed by the shared bus is a major contribution to the power requirement of the whole chip. Fig. 2 is a breakdown of the average snoop energy consumption for various remote miss ratios in a four-processor MPSoC, and shows how much of the snoop energy is used by the bus. As the

* Corresponding author.
E-mail addresses: chunmok@davinci.snu.ac.kr (C.-M. Chung), jihong@davinci.snu.ac.kr (J. Kim).

1383-7621/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2009.01.001

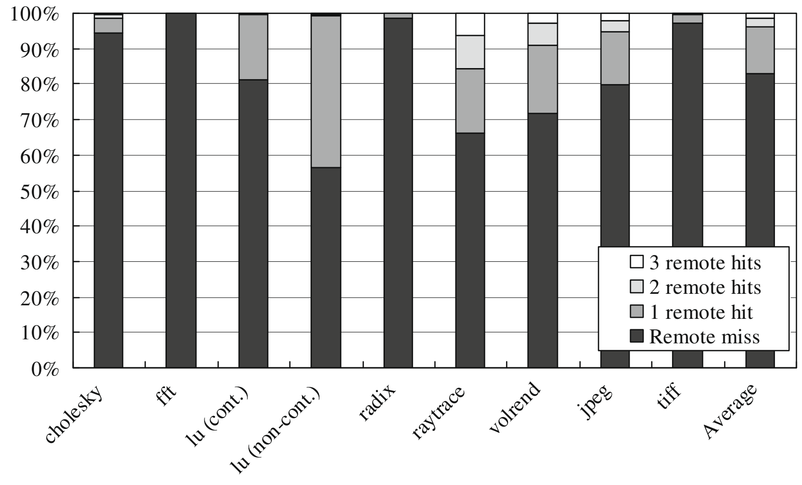


Fig. 1. Distribution of remote hits on a four-processor MPSoC.

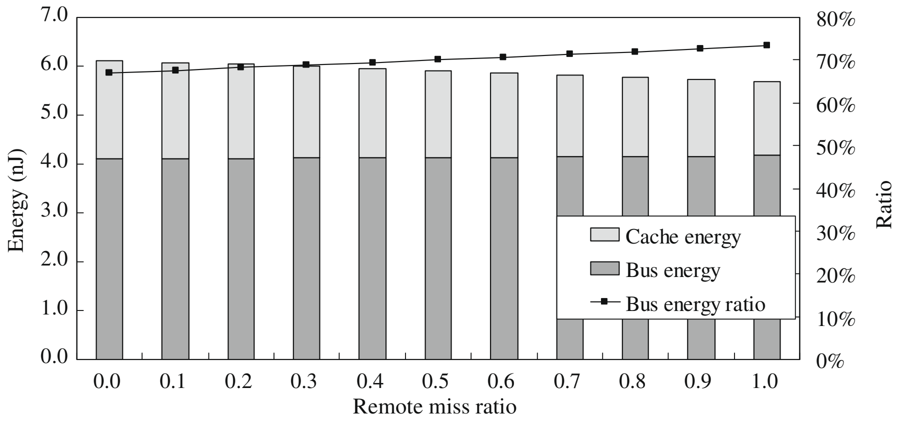


Fig. 2. Average snoop energy breakdown of a four-processor MPSoC.

remote miss ratio increases, the cache energy decreases, while the bus energy increases (because of static energy increase), so that the proportion of energy used by the bus reaches 73% of the average snoop energy at the highest remote miss ratio. This shows why it is inadequate to focus exclusively on cache energy reduction. A new approach is necessary to reduce the energy consumption of both the cache and the bus.

In this paper, we propose an architecture-level snoop energy reduction technique called *broadcast filtering*, which prevents the broadcast of unnecessary coherency requests to remote caches, and thus reduces snoop energy consumption in both the cache and the bus. Broadcast filtering is achieved by combining a snooping cache with a split bus. The snooping cache checks whether matching blocks exist in remote caches before broadcasting a coherency request. If no remote cache has the matching block, it cancels the broadcast. But if a broadcast is necessary, the split bus enables the coherency requests to be broadcast selectively to the remote caches that have matching blocks. Experiments for parallel applications show that our technique removes by about 90% of cache lookups and by about 60% of bus usage at the MPSoC containing 16 processors, resulting in a reduction by about 40% of energy consumption over the conventional architecture. In order to understand the efficiency of the proposed technique in a more general setting, we build an analytical energy model for systems with the broadcast filtering support. Based on the energy model, an analysis result shows that the broadcast filtering can reduce by 55% of energy consumption per cache coherency operation under various configurations of machine and program characteristics (such as the number of processors and remote miss ratio).

The rest of paper is organized as follows. In Section 2, we describe related work on reducing snoop energy consumption. The target MPSoC platform used in our work is described in Section 3. The details of broadcast filtering is described in Section 4. In Section 5, we show how snoop energy is reduced using simulations. In Section 6, we present an average snoop energy model and use it to show how the amount of snoop energy saved by the broadcast filtering technique is affected by the characteristics of the target hardware and software. We draw conclusions from this study in Section 7.

2. Related work

The main techniques that have been proposed to reduce the cache energy dissipated in cache coherency operations use cache lookup filtering [5,7] and serial cache lookup [9,10]. Jetty [5] is a small structure attached to each cache. It is based on the SMP (symmetric multiprocessing) system which has a large private L2 cache. Jetty is checked before cache lookup, and filters out unnecessary cache accesses. But Jetty does not save much energy in snooping operations in single-chip multiprocessor systems because of its won energy overhead [8]. RegionScout [7] saves more energy than Jetty because it uses a smaller filter. Whereas Jetty has one entry for each cache block, RegionScout has one entry per region, which is a continuous memory area. This reduces the space and energy costs of the filter. However, with these techniques, most coherency requests still have to be broadcast to all the remote caches, and all the filters are accessed because each filter is attached to an individual cache and only has information about the data blocks in that cache.

Our approach is different from these methods because we use a central directory (called a snooping cache) which is shared by all caches and it is only accessed once for each coherency request. It tells us which remote caches need to receive a coherency request, and the coherency request is only broadcast to those caches. This is possible because the snooping cache contains data block information for all the caches.

Serial snooping [9] and flexible snooping [10] are serial cache lookup techniques. Serial snooping targets a hierarchical bus whereas flexible snooping is designed for a ring-based bus. Instead of broadcasting a coherency request to all the processors in parallel, remote caches are checked sequentially. The idea is that, when a miss occurs in a local cache, it should often be possible to find the required block in a nearby remote cache without checking all the remote caches. A coherency request is sent to the nearest remote cache and the cache looked up its data (snoop). If the required block was found there, the snoop transaction is complete. Otherwise, a snoop transaction is issued to the next cache (forward). This is called as a "snoop then forward" scheme. Flexible snooping extends this process by means of "forward then snoop" and "forward" schemes, which enhance snoop performance if a cache far from the requester has the required block.

Several researchers have used bus splitting techniques [11,13,14] to reduce bus energy consumption. Chen et al. [11] proposed the first bus splitting technique, based on pass transistors, and used graph search to find an energy-efficient bus topology. They segmented the bus at the logic level and reduced the bus energy by exploiting information about the communication ratio between modules. Hsieh and Pedram [13] divided the bus into two segments and connected communication components based on a probabilistic model of communication. But they only considered splitting the bus into two segments and they ignored the energy cost of the splitters. Guo et al. [14] showed how to design a segmented bus in which the number of bus segments is reduced by block ordering, and the length of each segment is then minimized by floor planning. Their goal was the automated design of a segmented bus architecture. Although we also use a split bus architecture, our goal is to save snoop energy by filtering unnecessary coherency requests, an approach which has not been considered in previous research.

Recently, application-driven snoop-based cache coherence customization techniques are proposed. Yu and Petrov [21] proposed a producer–consumer model-aware cache coherence customization. They eliminated unnecessary snoop-induced cache tag lookups. For this, they used a hint from application regarding data sharing between producer and consumer. They filtered cache tag lookups for data out of the sharing data. Moreover, they proposed a write-update-based coherence customization with a special "write-and-flush" instruction to reduce cache misses and bus transactions [22]. For this, they analyzed the last write per cache block in compile-time and generated the special instruction to perform one write-update transaction per cache block. However, those techniques are eligible only for producer-consumer model where a write-after-read sharing is moderate and contiguous writes to shared data is generated. In a write-after-write sharing with distributed data, as frequent write-update overwhelms bus bandwidth, their techniques are not efficient in energy and performance aspect.

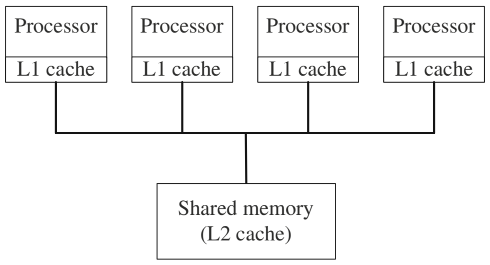


Fig. 3. The baseline MPSoC model with four processors.

3. Target MPSoC platform

Typical MPSoCs, both academic and commercial, contain multiple homogeneous processor cores and on-chip caches [1,3], and we use a similar baseline architecture for our target MPSoC platform. Fig. 3 shows this baseline MPSoC, which has four processors, although the technique that we are proposing is not restricted to a particular number of processors. Each processor has its own private L1 caches, comprising a separate I-cache and D-cache. Each D-cache has a duplicated tag to prevent cache coherency operations delaying the processor. The MPSoC may also contain a shared L2 cache to enhance performance. All the processors share the memory (or the L2 cache) and access it through a shared bus. A snoop-based cache coherency protocol is used to maintain data coherence between cache and memory.

In a snoop-based protocol, if a cache miss occurs at a processor, or a data block in the cache is modified, then the cache (called the local cache) uses a bus transaction to keep its data coherent with the other processors' caches (called the remote caches). In the case of the MESI protocol [19], the occurrence of a read miss in a local cache causes a read transaction (BusRd). The local cache broadcasts a block address and a BusRd signal to remote caches and memory. Remote caches snoop on the bus transaction and a remote cache containing the requested data supplies the data to the local cache. If no remote cache contains the requested data, the shared memory supplies the data. A write hit is processed as an upgrade transaction (BusUpgr). The local cache notifies remote caches that the data has been modified. The remote caches snoop on the BusUpgr signal, and those that contain the same data block invalidate their copies. If a write miss occurs, the local cache executes a read exclusive transaction, by sending a BusRdX signal to all remote caches to notify them that it needs to become the exclusive holder of the data block. A remote cache containing the required block transfers it to the local cache and all remote caches invalidate their copies of that block. If the cache block to be evicted is dirty, the local cache writes the dirty block back into the shared memory using a write-back transaction (BusWB). Because the same block cannot exist in remote caches, those caches do not perform any operation after snooping on a BusWB signal.

4. Reducing snoop energy consumption by broadcast filtering

Fig. 1 shows that many coherency requests do not find matching blocks in any remote cache. The broadcasts and subsequent remote cache lookups associated with these remote misses consume energy without any improvement in performance. We therefore propose a broadcast filtering technique which reduces snoop energy consumption by filtering unnecessary coherency broadcasts. We will describe how to detect and remove these redundant broadcasts when no remote cache has the required data block. We will also explain how snoop energy consumption can be reduced even if some of the remote caches have the requested data blocks, so that some coherency request broadcasts are necessary.

4.1. Detecting a remote miss and filtering unnecessary broadcasts

If we know that the required data does not exist in any remote cache, there is no point in broadcasting a coherency request. We use a directory, called as *snooping cache*, to determine if the required data is available in remote caches. A coherency request is processed in a two-hop bus transaction: one hop from the local cache to the snooping cache and another from the snooping cache to the remote caches. Fig. 4 shows a two-hop request: (i) a local cache sends a coherency request to the snooping cache instead of broadcasting it to the remote caches and (ii) if a remote hit is detected in the snooping cache, it broadcasts a coherency request to the remote caches, but if a remote miss is detected in the snooping cache, it requests the block direct to the shared memory.

The information in snooping cache relates to the sharing of data blocks among L1 caches. This is different from a conventional directory which records the ownership and sharing information about data blocks in shared memory. The snooping cache is organized as a set-associative cache, consisting of a tag array and a flag vector (FV) array. It does not have a data array. If an MPSoC contains N processors and each processor has a W-way set-associative cache with S sets, the snooping cache will have $N \times W$ ways and S sets. Because the purpose of the snooping cache is to determine whether a block is available in the other caches, a set of tags consists of all the tags with the same index in all the L1 caches. We need to choose the number of ways conservatively because the same indexed sets may have different tags in all L1 caches (if no data is shared by the L1 caches). The length of a tag is the same as that in the L1 caches, and a flag vector consists of N flag bits. Each flag bit corresponds to an L1 cache and indicates if the corresponding L1 cache contains the same tag. Fig. 5 shows the snooping cache organization when an MPSoC consists of four processors ($N = 4$) and each L1 cache is direct-mapped ($W = 1$).

Like a traditional set-associative cache, the snooping cache receives a coherency request with a block address, and selects a set using the index part of that address. After comparing the tag part of the address with a tag array, it outputs a corresponding flag vector and remote hit information. To keep the tag array and the flag vector array up to date, the snooping cache is updated whenever any L1 cache is updated. When a new tag is added to any L1 cache, it is also added to the tag array and the corresponding flag bit is set. If the tag is already present in the tag array, only the corresponding flag bit is set. When a tag is invalidated in any L1 cache, the corresponding flag bit in the flag vector is cleared. If all bits in the flag vector are cleared, the corresponding tag in the tag array is also invalidated.

4.2. Broadcasting to remote caches selectively

Suppose that processor P2 requests a block, which is only in the cache of P3 in Fig. 4. Because a remote hit is detected in the snooping cache, a coherency request is broadcast to all the remote caches. Although P1 and P4 do not have the data, they snoop on the coherency request and then perform unnecessary cache lookups. But if the coherency request is selectively broadcast to P3 alone, the energy consumed in snooping will be reduced.

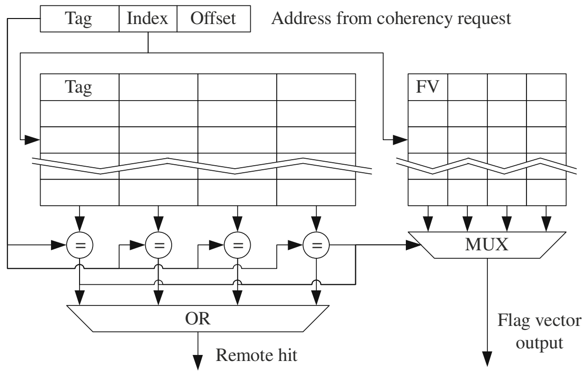


Fig. 5. The organization of a snooping cache in a four-processor MPSoC.

We will now explain how to select the remote caches to receive a selective broadcast and how that broadcast is performed. Because the flag vectors in the snooping cache have information about data sharing, we know which remote caches need to receive the coherency request. To broadcast the coherency request selectively, we use a split bus architecture. This is an extension of the bus segmentation technique due to Hsieh and Pedram [13], who divided a bus line into two segments and connected frequently communicating components to the same bus segment so as to reduce bus energy consumption. We extend this model by introducing multiple segments; their number is determined by the number of processors.

Since, nowadays, repeaters (using buffers or inverters) are inserted in bus lines for power and performance enhancement [12], the split bus can be implemented by replacing some repeaters by splitters (using tri-state buffer). Therefore, the length of split bus is same to that of monolithic bus'. Also, there is no difference in bus wire width and spacing between monolithic bus and split bus. Fig. 6 shows our split bus architecture applied to the target MPSoC. The processors and the snooping cache are each connected to a bus segment. A splitter connects two adjacent bus segments and transfers signals between them only if it is enabled. If all the splitters are enabled, the split bus works like a monolithic bus. A bus arbiter and the flag vector in the snooping cache control the enabling of the splitters. If a remote hit is detected in the snooping cache, the bits in the flag vector are used to ensure that the coherency request is only broadcast to the remote caches that contain the required data. The details of this procedure will be described in the next subsection.

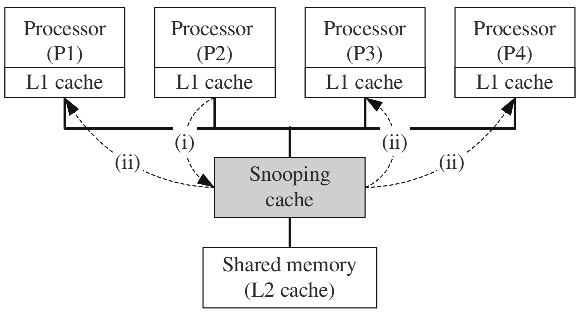


Fig. 4. Sending a two-hop coherency request through a snooping cache.

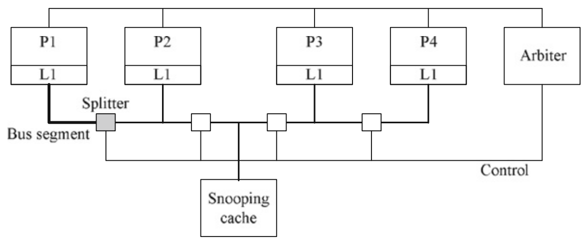


Fig. 6. A four-processor MPSoC with a split bus.

4.3. Cache coherency operations with broadcast filtering

Sequences of bus transactions are modified by broadcast filtering. We will describe the sequence of bus transactions that takes place during a cache coherency operation to show that broadcast filtering is applicable to all kinds of cache coherency operations.

The read transaction (BusRd): Fig. 7 shows the operation sequence when processor P2 requests a data block which is in the cache at P3: (i) If the cache miss occurs in P2's cache, it sends a bus request signal to the arbiter to perform a BusRd transaction. (ii) The arbiter replies with a bus acknowledgement signal and permits bus use. At the same time, it enables splitter S2 between the local cache and the snooping cache. (iii) After receiving the bus acknowledgement signal, the local cache at P2 sends a coherency request (BusRd) to the snooping cache. (iv) The snooping cache checks the tag array and the flag vector array. The bits that are set in the flag vector enable all the splitters needed (only S3 in this example) to connect with the appropriate remote cache. Then a flag bit corresponding to the local cache is set. (v) If a remote hit is detected, the snooping cache broadcasts a BusRd request to the remote caches. But if a remote miss is detected, it omits this broadcast and sends the request to the shared memory. (vi) Remote caches (only the cache at P3 in this case) snoop on the request and supply the required data to the local cache at P2.

The read-exclusive transaction (BusRdX): This operation sequence is similar to that of BusRd. However, the BusRdX request causes the snooping cache to clear the flag bits corresponding to the remote caches in step (iv), because the remote caches will invalidate their cache tags for this data block.

The upgrade transaction (BusUpgr): This is similar to BusRdX, except that remote caches do not supply the required data to the local cache in step (vi), because it is already there.

The write-back transaction (BusWB): This performs steps (i) to (iii) in the same way as BusRd. Then (iv) the snooping cache clears the flag bit corresponding to the local cache, as the local cache will invalidate the corresponding tag. (v) The snooping cache sends a BusWB request and then writes the data to the shared memory without broadcasting, as the block to be written back is dirty and therefore cannot be shared by the remote caches.

As regards performance, broadcast filtering increases the time taken by a cache coherency operation, at most by the latency for one bus transaction and one snooping cache access. This is because a coherency request is processed as a two-hop bus transaction and the snooping cache has to be checked to discover whether any of the remote caches contain the required data. However, if a remote miss is detected in the snooping cache, it does not broadcast a coherency request to the remote caches, so that only a one-hop bus transaction is required, and no tag lookups occur in the remote caches. Using a snoop-based protocol, the shared memory must wait until it is certain that no cache will supply the requested data before driving the bus [19]. So the time taken during a snooping cache lookup can be offset by the elimination of remote cache lookup time, assuming that the snooping cache access and an L1 tag lookup both take the same number of clock cycles. Therefore, the performance overhead of broadcast filtering is proportional to the remote hit ratio. We will evaluate the performance overhead experimentally.

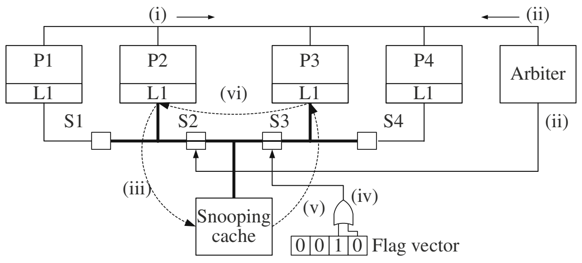


Fig. 7. The operation procedure of BusRd after applying broadcast filtering.

5. Experiments

We evaluated the effect of broadcast filtering by simulation. The results illustrate its performance in terms of component utilization, energy consumption, and latency. We also compared the component utilization and energy consumption with the Jetty and RegionScout techniques. As RegionScout has been shown to outperform Jetty, we focused on comparisons with RegionScout.

5.1. Experimental setup

We implemented an MPSoC simulation within CATS framework [15], an extended version of the SimpleScalar tool [16], which supports multiple processors, private caches, a shared memory, and a snoop-based cache coherency protocol. As far as possible, we configured CATS to follow the specification of MPCore [20], which is a representative MPSoC that is widely used in industry. Table 1 shows the detailed simulation parameters of our target MPSoC. Our implementation of RegionScout consists of 16KB-regions, 16-entry direct-mapped NSRTs, and 256-entry CRHs, the configuration that was originally proposed [7] for a low-power single-chip multiprocessor.

We used benchmark programs from SPLASH-2 [17] and MiBench [24]. Table 2 describes the benchmark programs and the input data. Although SPLASH-2 was not developed for embedded system evaluation, as it has been widely used as a benchmark for multicore and multiprocessor platforms, and there are no other widely used parallel applications for embedded systems. Small data sets were chosen to make the SPLASH-2 applications as appropriate as possible for MPSoCs. As the programs in MiBench were designed for single core processor, we parallelized them using posix thread-like CATS multitasking model.

We executed the programs on CATS and generated traces about cache coherency operations. The trace consists of cache coherency request type, data address, remote hit/miss results according to remote cache tag lookups, and cache block state changes of all L1 data caches according to the cache coherency protocol. To estimate the energy consumed by cache coherency operations, we designed an energy estimator. It inputs the trace from CATS and accumulates the number of snooping cache accesses and remote cache lookups during each cache coherency operation. It also counts the number of used bus segments and splitters during each bus transaction. Then, it estimates the dynamic energy consumed by the snooping cache, L1 data caches, bus segments, and splitters by multiplying the number of accesses by the dynamic energy parameter of each

Table 1
Simulation parameters of the target MPSoC.

Parameter	Value
Processor	ARM core
Processor frequency	200 MHz
Numbers of processors	2, 4, 8, 16
Private L1 I-cache and D-cache	32 kB, 32-byte blocks, four-way set-associative
Cache hit latency	One clock cycle
Cache miss latency	80 clock cycles
Cache coherency protocol	MESI
Interconnect	Shared bus
Bus latency	One clock cycle
Bus segment length	6 mm
Snooping cache latency	One clock cycle

Table 2
Benchmark programs and their input data.

Program	Description of program	Input data
cholesky	Blocked Cholesky factorization on a sparse matrix	tk14.o
fft	Complex 1D version of the six-step FFT	16K points
lu (cont.)	LU factorization of contiguously allocated blocks	256 × 256 matrices, 16 × 16 blocks
lu (non-cont.)	LU factorization of non-contiguously allocated blocks	256 × 256 matrices, 16 × 16 blocks
radix	Integer radix sort	256K integers, radix 1024
raytrace	Rendering a 3D scene using ray-tracing	teapot.env
volrend	Rendering a 3D volume using ray-casting	head-scaleddown2
jpeg	Compressing an image to a JPEG format	small.bmp
tiff	Converting a color TIFF image to greyscale	small.tif

component. It also estimates the static energy consumption of each component by multiplying the execution time information from CATS by the static power parameter of each component.

Table 3
Dynamic energy parameters (nJ).

Parameter	Number of processors			
	2	4	8	16
Snooping cache	0.1017	0.1589	0.2423	0.3558
L1 cache	0.6626			
L1 cache tag	0.0882			
Bus line	0.0005/mm			
Splitter	0.0005			

Table 4
Static power parameters (μW).

Parameter	Number of processors			
	2	4	8	16
Snooping cache	1.4480	2.8746	5.7279	11.4266
L1 cache	3.4032			
Bus line	1.7302/mm			
Splitter	0.6302			

Tables 3 and 4 show the dynamic energy and the static power parameters of components used in our energy estimation. The dynamic energy parameters of caches were taken from CACTI [18] with 130 nm CMOS technology. We also referred to the capacitance parameter by Hsieh and Pedram [13] for the dynamic energy parameter of splitter. We obtained the static powers of snooping cache and L1 cache using HotLeakage model [23]. For the dynamic energy and static power of bus lines, equations derived by Banerjee and Mehrotra [12] were used. These equations took into account an optimal repeater size/spacing. For 130 nm technology nodes and beyond, the coupling capacitance can be as high as the sum of the area and fringing capacitance of wire. The role of coupling capacitance will be even more dominant in the future as feature size shrink [25]. The bus energy model took into account not only an area and fringing capacitance (wire-to-silicon substrate capacitance) but also a coupling capacitance (wire-to-wire capacitance) based on International Technology Roadmap for Semiconductors (ITRS). The model used not only a three-dimensional (3D) capacitance model for bus wire capacitances (i.e. coupling capacitance, area capacitance, and fringing capacitance) [27], but also an input/output parasitic capacitance of repeater. According to the bus wire capacitance model [26], the coupling capacitance is proportional to a wire thickness and a dielectric thickness and inversely

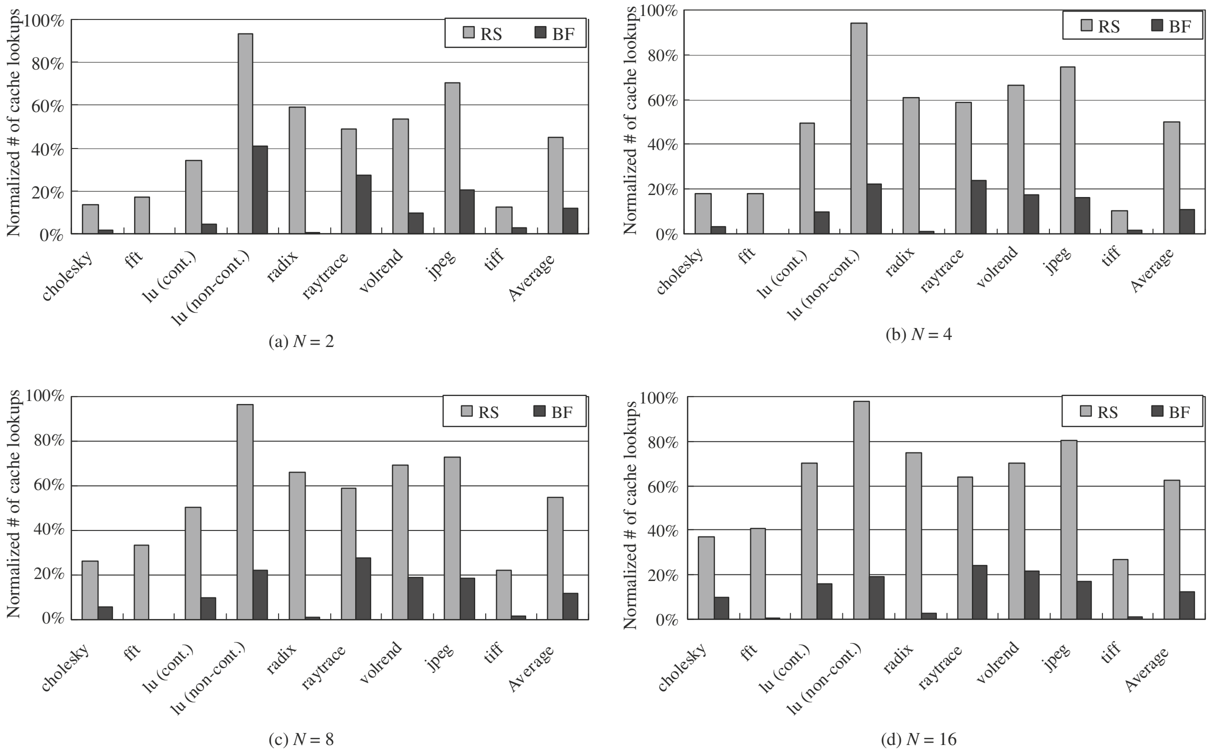


Fig. 8. The number of cache lookups in snooping operations by RegionScout (RS) and broadcast filtering (BF), for *N* processors, normalized to the baseline MPSoC.

proportional to an inter-wire spacing. The area and fringing capacitance is proportional to a wire with and inversely proportional to a dielectric thickness. Therefore, since our split-bus implementation did not change the wire thickness and the inter-wire spacing, we assumed the capacitance of split-bus was the same to that of monolithic bus'. The static power of splitter was calculated as 22-times of the static power of bus (an inverter), because the splitter requires two tri-state buffers and 20 gates to implement enable logics of the tri-state buffers [13].

5.2. The utilization of cache and bus

To find out how many of the unnecessary remote cache lookups are filtered by our technique, we counted the number of remote cache lookups and the bus segments used during cache coherency operations. Fig. 8 shows the number of remote cache lookups that occurs for different numbers of processors. The results are all normalized to the baseline MPSoC.

Both the broadcast filtering and RegionScout reduced the number of remote cache lookups for all applications and all numbers of processors. For both techniques, the reduction tracked the proportion of remote misses in each application because remote cache lookups which would have led to remote misses were filtered out. However, the broadcast filtering reduced the numbers of remote cache lookups much more than RegionScout. On average, the broadcast filtering eliminated about 90% of remote cache lookups, whereas the best performance of RegionScout, with two processors, only reduced by 55% of remote cache lookups. The gap in performance between the broadcast filtering and RegionScout increased with the number of processors, from 32% with two processors to 50% with 16 processors. We can explain these results in terms of the granularity of the sharing information; whereas the broadcast filtering uses cache blocks, RegionScout uses regions. A region is much larger than a cache block, so that RegionScout makes a conservative estimate of cache block sharing, allowing for local caches to send many redundant broadcasts (because of false sharing) and for remote caches to look up their tags unnecessarily.

For the broadcast filtering, the number of remote cache lookups increases a little with the number of processors, except for LU factorization of non-contiguously allocated blocks. This is because some residual unnecessary lookups are not filtered. For example, in Fig. 7, suppose that the data which P2 requests is only in the cache at P4. Nevertheless, the snooping cache forwards the requests to both P3 and P4, and they look up their caches. If we could map the tasks which will generate cache coherency operations to processors which were close to each others on the bus, there would be scope to save more snoop energy. But we will not consider task mapping in this paper.

As well as reducing cache lookups, the broadcast filtering also affects bus segment usage, so we counted the numbers of bus segments used during snoop operations and the totals are shown in Fig. 9. The baseline model and RegionScout both use a monolithic bus, which is replaced with $N+1$ bus segments in a split bus, where N is the number of processors. We therefore took $N+1$ bus segments as our baseline model.

The use of bus segments by broadcast filtering decreases as the number of processors increases. With two processors, the broadcast filtering reduces bus usage to 68% of the baseline. With 16 processors, the bus usage is reduced to 38%. In the figure, lu (non-cont.) reduced the most bus segment usage in the almost all numbers of processors. The increase of reduction amount was also the highest according to the increase of the number of processors. Although fft has the highest remote miss ratio, it reduced less bus segments usage than lu (non-cont.) which has the lowest remote miss ratio. This is because the bus segments are not only used to broadcast coherency requests, but also to copy a required data blocks. Moreover, as the cache block size (32 bytes) is larger than the data bus width (4 bytes), multiple bus transactions are required to copy a data block between caches. So the reduction of bus segment usage achieved by eliminating data copy operation is larger than that needed to broadcast a request which only uses one bus transaction. The reduction of bus segments usage is therefore dependent on both the remote miss ratio and the distance between the requester and supplier of data.

5.3. Energy and performance

Fig. 10 shows the energy breakdown of cache coherency operations in baseline model (Base), with RegionScout (RS), and with the broadcast filtering (BF). The figure shows not only total energy but also energy consumed by each component. Each energy result is sum of dynamic energy and static energy consumed by each component. Since snoop energy reduction techniques require additional logics (NSRT and CRH in the case of RegionScout, and the snooping cache and splitters for broadcast filtering), the energy consumed by these components is also shown.

Because dynamic energy consumption is proportional to the numbers of cache lookups and bus usage, we could expect it to be reduced by both techniques, but in fact RegionScout reduced

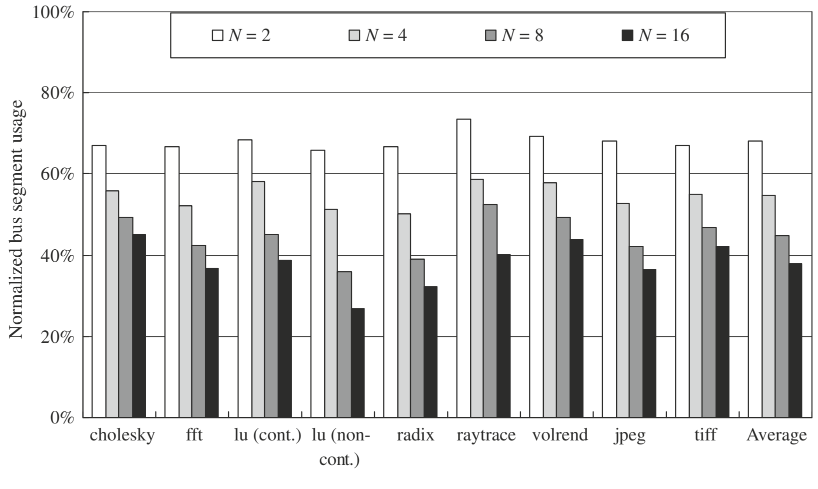


Fig. 9. The bus segment usage of broadcast filtering, for different numbers of processors (N), normalized to the monolithic bus ($N+1$ segments) of baseline MPSoC.

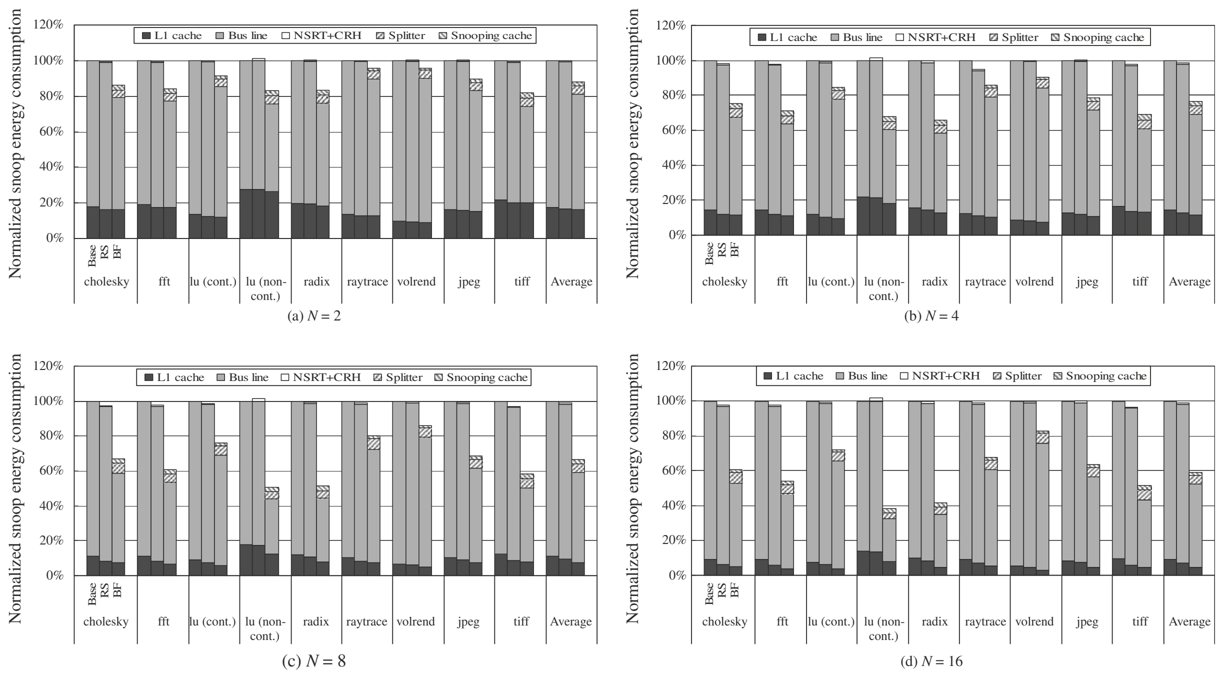


Fig. 10. The snoop energy consumed by baseline model (Base), RegionScout (RS), and broadcast filtering (BF), for N processors, normalized to the baseline MPSoC.

little snoop energy consumption. It is because NSRT and CRH consumed additional energy. The broadcast filtering is much more effective, because it reduced the energy requirement of the bus as well as that of the cache's. The more energy was saved as the number of processors increased, because the more number of unnecessary broadcasts were eliminated. Due to the dynamic and static energy overhead by the snooping cache and splitters, the energy saving was smaller than the reduction of cache lookups and bus usage. With 16 processors, snoop energy decreased to 59% on average, to 38% by maximum, of the baseline model. The energy consumed by splitters and snooping cache represents about 7% of the total snoop energy.

As we have already described, the performance overhead incurred by the broadcast filtering when a remote hit was detected (because of the two-hop broadcasting and the snooping cache lookup). To evaluate this performance overhead, we calculated the average latency of cache coherency operations, which is the time from a local cache's sending a coherency request to the return of the required data, for the bus latencies of one cycle (L_{B1}) and 10 cycles (L_{B10}). The latencies of the L1 cache and snooping caches were one cycles in both cases (we verified these latencies using CACTI [18]). If $\overline{L}_{\text{snoop}}$ and $\overline{L}^{BF}_{\text{snoop}}$ are the average snooping latencies before and after applying broadcast filtering, respectively, the performance overhead can be calculated as follows:

$$T_{\text{overhead}} = \frac{(\overline{L}^{BF}_{\text{snoop}} - \overline{L}_{\text{snoop}}) \times \text{remote hit ratio}}{\overline{L}_{\text{snoop}}}.$$

Table 5 shows the performance overhead incurred by the broadcast filtering in an MPSoC with four processors. The performance overhead is small in all cases, and this can be explained in terms of two characteristics: First, because the snooping cache only broadcasts coherency requests to remote caches if a remote hit is detected, the overhead is proportional to the remote hit ratio and this is small for many programs. Second, the cache coherency operation latency is dominated by the time taken to the required data, and the cost of the two-hop request is relatively insignificant, so that $\overline{L}_{\text{snoop}}$ and $\overline{L}^{BF}_{\text{snoop}}$ make little difference.

6. Energy efficiency analysis

As the experimental results show, the amount of snoop energy reduction by the broadcast filtering varies significantly depending

Table 5
The performance overhead of broadcast filtering in a four-processor MPSoC.

Application	Remote hit ratio (%)	T_{overhead} (%)	
		L_{B1}	L_{B10}
cholesky	5.4	1.1	0.7
fft	0.1	0.0	0.0
lu (cont.)	18.8	3.8	2.3
lu (non-cont.)	43.5	8.7	5.3
radix	1.4	0.3	0.2
raytrace	33.7	6.7	4.1
volrend	28.1	5.6	3.4
jpeg	20.1	4.0	2.4
tiff	2.7	0.5	0.3
Average		3.4	2.1

Table 6
Definition of parameters in the snoop energy model.

Parameter	Definition
N	The number of processors (L1 caches)
$E_{\text{tag,dynamic}}$	Dynamic energy of L1 cache tag lookup
$E_{\text{L1,dynamic}}$	Dynamic energy of L1 cache access
$E_{sc,\text{dynamic}}$	Dynamic energy of snooping cache access
$E_{bs,\text{dynamic}}$	Dynamic energy of bus segment
$E_{sp,\text{dynamic}}$	Dynamic energy of splitter
$E_{\text{bus,dynamic}}$	Dynamic energy of monolithic bus ($= (N+1) \times E_{bs,dynamic}$)
A	The number of bus transactions per block ($\frac{\text{block size}}{\text{bus width}}$)
p_{hit}	Remote hit ratio
p_{rd}	Probability of a BusRd operation
p_{rdx}	Probability of a BusRdX operation
p_{upgr}	Probability of a BusUpgr operation
p_{wb}	Probability of a BusWB operation
$P_{\text{cache,static}}$	Static power of L1 cache
$P_{\text{bus,static}}$	Static power of monolithic bus
$P_{sc,\text{static}}$	Static power of snooping cache
$P_{\text{splitter,static}}$	Static power of splitter

on the execution characteristics of a parallel application as well as the number of active processors in the MPSoC. In order to better understand the efficiency of broadcast filtering under different machine configuration and program execution characteristics, we define an analytical snoop energy model. Our snoop energy model includes not only dynamic energy consumption but also static energy consumption. So, it will be useful in evaluating the energy efficiency of broadcast filtering more accurate in general environment.

It is important to note that the goal of our energy model is not to design a more accurate energy model of each component but to design an overall energy model of cache coherency operation. However, for the accuracy of analysis, our model refers to the energy parameters from the widely accepted detail (dynamic and static) power/energy models for cache [18,23], bus [12], and splitter [13]. Using the dynamic and static energy parameters, we design an energy consumption model of cache coherency operation and evaluate energy efficiency of our technique.

6.1. Baseline average snoop energy model

An energy consumption consist of dynamic energy and static energy. Dynamic energy is consumed when a logic performs some operations. Static energy is always consumed even the logic performs no operation. Since the dynamic energy depends on the number of cache coherency operations and static energy depends on execution time of applications, it will be more useful to define an average snoop energy model which is independent of the characteristic of individual applications. Different cache coherency operations consume different amounts of energy. For example, while BusRd and BusRdX perform cache-to-cache data copy as well as remote cache lookup, BusUpgr only performs remote cache lookup. We therefore construct our snoop energy model to reflect the average distribution of coherency types.

A cache coherency operation consists of the cache accesses for tag lookup and cache-to-cache data coping, and the bus transactions needed to transfer a coherency request and the required data block between caches, so the average snoop energy can be partitioned into dynamic energies of cache and bus and static energy of them, as follows:

$$\overline{E}_{\text{snoop}} = \overline{E}_{\text{cache,dynamic}} + \overline{E}_{\text{bus,dynamic}} + \overline{E}_{\text{static}}. \tag{1}$$

As dynamic energy is dependent on cache and bus operations and a major energy consumer, if we does not describe static energy, it means dynamic energy in our energy model. The cache energy is the sum of the energy consumed during a remote cache access for tag lookup and a full cache access for copying data between caches or between the shared memory and the local cache. So the average cache energy can be expressed as follows:

$$\overline{E}_{\text{cache,dynamic}} = \overline{E}_{\text{tag,dynamic}} + \overline{E}_{\text{L1,dynamic}}. \tag{2}$$

where, $\overline{E}_{\text{tag}}$ is the average tag lookup energy and $\overline{E}_{\text{L1}}$ is the average amount of energy required for a full L1 access. As tag lookup is performed at remote caches, its energy requirement is the product of the number of remote caches and the energy consumed in each lookup. However, as no remote cache checks its tag using BusWB, the probability of a BusWB operation should be excluded in determining the average tag lookup energy, which can then be formulated as follows:

$$\overline{E}_{\text{tag,dynamic}} = (N-1) \times (p_{rd} + p_{rdx} + p_{\text{upgr}}) \times E_{\text{tag,dynamic}}. \tag{3}$$

where the variables are defined in Table 6. If a remote hit occurs during a data copy operation, the cache-to-cache data copy is performed by BusRd or BusRdX, and a data block is read from a remote cache and written to a local cache. But if a remote miss occurs during BusRd or BusRdX operations, or BusWB is invoked, a memory-to-cache or cache-to-memory data copy is performed. Therefore the cache access energy expended during a data copy can be expressed as follows:

$$\overline{E}_{\text{L1,dynamic}} = \{p_{hit} \times 2(p_{rd} + p_{rdx}) + (1 - p_{hit}) \times (p_{rd} + p_{rdx}) + p_{\text{wb}}\} \times E_{\text{L1,dynamic}}. \tag{4}$$

The bus energy is the product of the number of bus accesses and the energy consumed in one access. The bus is always used once to send a coherency request, and many times in a data copy operation, but its usage differs with the type of coherency. The BusRd, BusRdX, and BusWB operations always use the bus to copy data between the local cache, the remote cache, and the shared memory. But, BusUpgr does not use the bus because no data is copied between caches. So the average bus energy can be expressed in terms of the probabilities of coherency types, as follows:

$$\overline{E}_{\text{bus,dynamic}} = \{1 + (p_{rd} + p_{rdx} + p_{\text{wb}}) \times A\} \times E_{\text{bus,dynamic}}. \tag{5}$$

The static energy consumed by cache and bus during a cache coherence operation can be estimated by multiplying total static power of all caches and bus by the average latency of snoop operation, as follows:

$$\overline{E}_{\text{static}} = (N \times P_{\text{cache,static}} + P_{\text{bus,static}}) \times \overline{L}_{\text{snoop}}. \tag{6}$$

6.2. Average snoop energy model for broadcast filtering

When broadcast filtering is applied to the baseline model, the dynamic and static energy consumed by the snooping cache and splitters must be considered, and we will now reconstruct the average snoop energy model to take this into account:

$$\overline{E}^{BF}_{\text{snoop}} = \overline{E}^{BF}_{\text{cache,dynamic}} + \overline{E}^{BF}_{\text{bus,dynamic}} + E_{\text{sc,dynamic}} + \overline{E}^{BF}_{\text{static}}.$$

We do not need to determine an average energy requirement for the snooping cache, because it is always accessed in all kinds of cache coherency operations.

The average cache energy is the sum of the average tag lookup energy and the average full cache access energy, as follows:

$$\overline{E}^{BF}_{\text{cache,dynamic}} = \overline{E}^{BF}_{\text{tag,dynamic}} + \overline{E}_{\text{L1,dynamic}}.$$

The average amount of energy used by the full cache during a cache-to-cache data copy remains unchanged from the baseline model. But the average tag lookup energy is affected by broadcast filtering because only the remote caches which contain the required data look up their tags. So the number of tag lookups is now dependent on the numbers of remote hits that occur as a result of each coherency request. If we define p_n to be the probability that there are n remote hits, the expression for the average tag energy is now:

$$\overline{E}^{BF}_{\text{tag,dynamic}} = \sum_{i=1}^{N-1} (i \times p_i) \times (p_{rd} + p_{rdx} + p_{\text{upgr}}) \times E_{\text{tag,dynamic}}.$$

Broadcast filtering uses a split bus consisting of bus segments and splitters, which all contribute to energy consumption. The energy consumed by the bus segments is simply the product of the number of segments used and the energy consumed by a single segment. The same is true of the splitters. The average number of bus segments used is the sum of the average number used in a snooping cache access ($\bar{n}_{bs,sc}$), in broadcasting a coherency request ($\bar{n}_{bs,bc}$), and in a cache-to-cache or cache-to-memory data copy operation ($\bar{n}_{bs,cp}$). The average number of splitters used can be calculated in the same way. The average bus energy can then be expressed as follows:

$$\overline{E}^{BF}_{\text{bus,dynamic}} = (\bar{n}_{bs,sc} + \bar{n}_{bs,bc} + \bar{n}_{bs,cp}) \times E_{bs,\text{dynamic}} + (\bar{n}_{sp,sc} + \bar{n}_{sp,bc} + \bar{n}_{sp,cp}) \times E_{sp,\text{dynamic}}. \tag{7}$$

During a snooping cache access, between 2 and $\frac{N}{2}+1$ bus segments are used, depending on the distance between the local cache and the snooping cache. If we define the probability that n_{bs} bus segments are used in snooping cache access to be $p_{n_{bs}}$, then the average number of bus segments used in a snooping cache access ($\bar{n}_{bs,sc}$) is the sum of the products of these probabilities and the number of bus segments used in each case:

$$\bar{n}_{bs,sc} = \sum_{n_{bs}=2}^{\frac{N}{2}+1} (p_{n_{bs}} \times n_{bs}).$$

If one or more remote hits are detected and the coherency type is BusRd, BusRdX or BusUpgr, the snooping cache broadcasts the coherency request. Depending on which remote cache contains the data, between 2 and $N+1$ bus segments are used to broadcast the coherency request. The average number is the sum of the products of the probabilities above and the number of bus segments used in each case. So the average number of bus segments used in broadcasting a coherency request ($\bar{n}_{bs,bc}$) can now be expressed as follows:

$$\bar{n}_{bs,bc} = \sum_{n_{bs}=2}^{N+1} (p_{n_{bs}} \times n_{bs}) \times (p_{rd} + p_{rdx} + p_{\mathrm{upgr}}) \times p_{hit}.$$

The average number of bus segments used in a data copy operation depends on the coherency type. We first calculate the average numbers of segments for each coherency type and then combine them into an average that covers all the coherency types. In the case of BusRd and BusRdX, if a remote hit occurs, between 2 and $N+1$ bus segments are used to copy data from a remote cache to the local cache. Otherwise, if there is a remote miss, between 2 and $\frac{N}{2}+1$ bus segments are used to copy data from the shared memory to the local cache. BusWB also uses between 2 and $\frac{N}{2}+1$ bus segments to write data back to the shared memory. Since BusUpgr does not copy data, its bus usage is zero. The average number of bus segments used in a data copy operation for all kinds of coherency can now be expressed as the sum of the products of the probability of each type of request and the average numbers of bus segments used in each case, as follows:

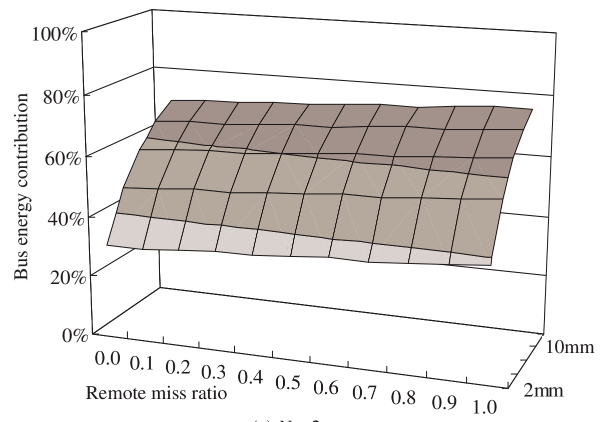
$$\bar{n}_{bs,cp} = \sum_{n_{bs}=2}^{N+1} (p_{n_{bs}} \times n_{bs}) \times (p_{rd} + p_{rdx}) \times p_{hit} \times A + \sum_{n_{bs}=2}^{\frac{N}{2}+1} (p_{n_{bs}} \times n_{bs}) \times \{p_{wb} + (p_{rd} + p_{rdx}) \times (1 - p_{hit})\} \times A.$$

The average usage of splitters three operations $\bar{n}_{sp,sc}$, $\bar{n}_{sp,bc}$ and $\bar{n}_{sp,cp}$ can be calculated in a similar way, allowing for the fact that the number of splitters is always one less than the number of bus segments in use. The three averages can therefore be expressed as follows:

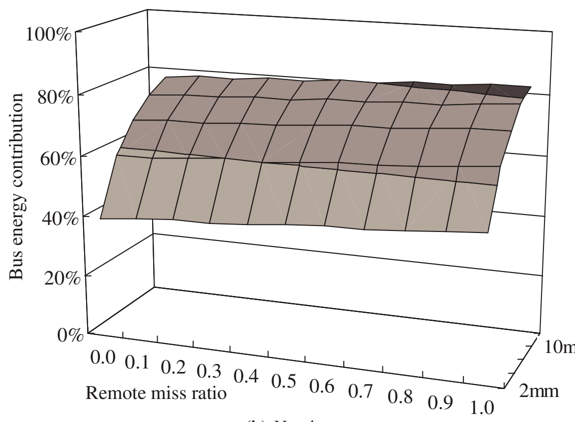
$$\bar{n}_{sp,sc} = \sum_{n_{sp}=1}^{\frac{N}{2}} (p_{n_{sp}} \times n_{sp}).$$

$$\bar{n}_{sp,bc} = \sum_{n_{sp}=1}^{N} (p_{n_{sp}} \times n_{sp}) \times (p_{rd} + p_{rdx} + p_{upgr}) \times p_{hit}.$$

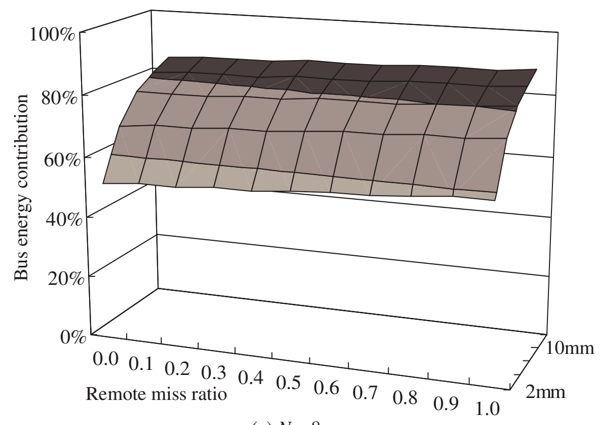
$$\bar{n}_{sp,cp} = \sum_{n_{sp}=1}^{N} (p_{n_{sp}} \times n_{sp}) \times (p_{rd} + p_{rdx}) \times p_{hit} \times A + \sum_{n_{sp}=1}^{\frac{N}{2}} (p_{n_{sp}} \times n_{sp}) \times \{p_{wb} + (p_{rd} + p_{rdx}) \times (1 - p_{hit})\} \times A.$$



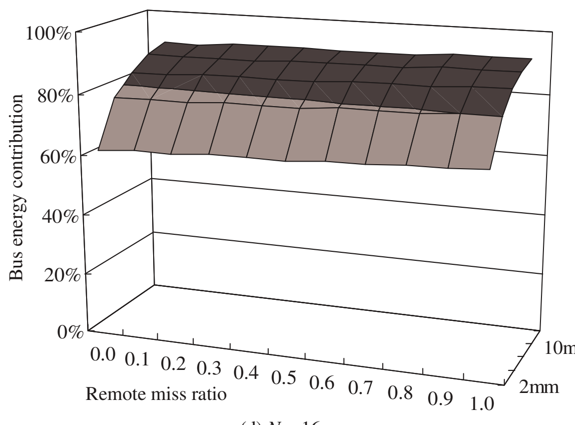
(a) $N = 2$



(b) $N = 4$



(c) $N = 8$



(d) $N = 16$

Fig. 11. The contribution of the bus energy to the total snoop energy in the baseline model for different remote miss ratios and numbers of processors (N).

The static energy can be estimated by multiplying total static power of caches, bus, snooping cache, and splitters by the average latency of snoop operation. If we assume that the sum of bus segment length of split-bus is same to the monolithic bus in the baseline model, the static energy of broadcast filtering can be expressed as follows:

$$\overline{E}^{BF}_{\text{static}} = (N \times P_{\text{cache,static}} + P_{\text{bus,static}} + P_{sc,\text{static}} + N \times P_{\text{splitter,static}}) \times \overline{L}^{BF}_{\text{snoop}}.$$

Although additional control lines are required to allow the arbiter and snooping cache to control the splitters, they are much fewer than the number of bus lines for addresses and data (by a factor of 32 or 64), and we can neglect their energy requirement.

6.3. The contribution of the bus line to snoop energy

We analyzed the average snoop energy model of a baseline MPSoC based on Eqs. (1)–(6) and determined the percentage of energy used by the bus, which is $\frac{\overline{E}_{\text{bus,dynamic}}+P_{\text{bus,static}}\times\overline{L}_{\text{snoop}}}{\overline{E}_{\text{snoop}}}$. The relative probabilities of the BusRd, BusRdX, BusUpgr and BusWB transactions were set to the average values found from SPLASH-2 applications. The average snoop energy was calculated using the parameters in Table 3. As the total length of the bus line depends on the design and implementation of the MPSoC, we analyzed the five cases, in which there are 2, 4, 6, 8, and 10 mm of bus segments in each case.

Fig. 11 shows how the percentage contribution of bus energy to the average snoop energy varies with bus length and remote miss ratio, for MPSoCs with 2, 4, 8, and 16 processors. Irrespective of the number of processors and the bus length, the contribution of the bus to the total energy consumption increases with the remote miss ratio. In the case of four processors, the energy contribution of the bus is between 40% and 77% when the remote miss ratio is 0.0, but it increases to between 48% and 82% when the remote miss ratio is 1.0. As the remote miss ratio rises, the number of cache-to-cache data copy operations drops and the contribution of cache energy decreases. This tells us that we must concentrate on the bus if we want to reduce the snoop energy consumption for applications with high remote miss ratios.

Although the contribution of the bus to the total energy consumption increases with bus length, as we would expect, the increase is not proportional, because the total snoop energy also increases. For four processors and a remote miss ratio of 0.5, the contribution of buses of length 4, 6, 8, and 10 mm bus segments were, respectively, 1.32×, 1.59×, 1.73×, and 1.82× that of a 2 mm bus segment. As the number of processors increases, bus energy becomes a higher proportion of snoop energy. For a remote miss ratio of 0.8, which is approximately average for the SPLASH-2 applications and a 6 mm bus segment, the bus consumed 63%, 72%, 80%, and 85% of the snoop energy for 2, 4, 8, and 16 processors, respectively, showing how the importance of bus energy saving increases with the number of processors. The energy contribution of the bus increases smoothly with the remote miss ratio but the rate of increase declines with the number of processors. This is because more processors means more tag lookups and the relative effect of the cache-to-cache data copy operations decreases, reducing the extent to which the energy contribution of the cache depends on the remote miss ratio. With two processors, bus energy is a small proportion of the snoop energy when

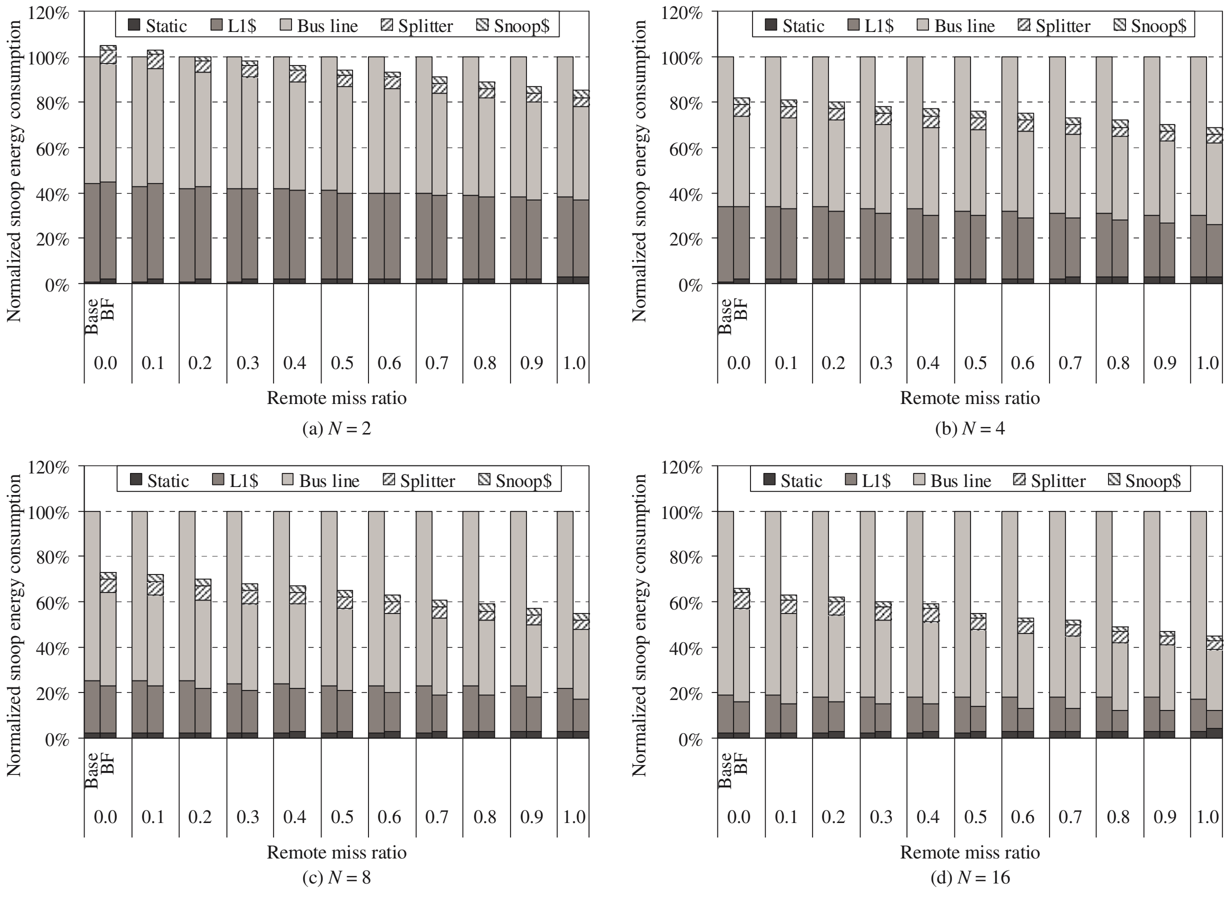


Fig. 12. A average snoop energy against remote miss ratio for different numbers of processors (*N*).

the remote miss ratio is low, but its contribution increases with the remote miss ratio; but when there are 16 processors the contribution of the bus energy increases a little (by maximum of 2%) as the remote miss ratio goes from 0.0 to 1.0.

6.4. Average snoop energy reduction with broadcast filtering

To predict the amount of energy that we might expect to serve with our technique, we used the energy model with appropriate component parameters. The baseline model, is an ARM MPCore-like cache configuration [20] with a 6 mm 32-bit bus segment, and uses the energy parameters in Tables 3 and 4. Using broadcast filtering Eq. (7) shows that the energy used by the bus depends on the number of bus segments and splitters between the requesting cache and the cache that supplies the data. Average values of these parameters were obtained from traces of SPLASH-2 applications. Fig. 12 shows how the average snoop energy depends on the remote miss ratio for 2, 4, 8, and 16 processors. It shows dynamic energy consumption of each component and accumulated static energy consumption of all components.

Irrespective of the number of processors, the amount of energy saved increased with the remote miss ratio. In the case of four processors and a remote miss ratio of 0.0, broadcast filtering saved 18% of the snoop energy. For a remote miss ratio of 1.0, this increases to 32%. The pattern is similar for 8 and 16 processors. When there are more than two processors, broadcast filtering reduces the average snoop energy, and the extent of this reduction increases with the number of processors. For the average remote miss ratio of SPLASH-2 applications, which is approximately 0.8, our technique reduced by 12%, 29%, 41%, and 51% of snoop energy, compared with the baseline, for 2, 4, 8, and 16 processors, respectively. This in largely because the contribution of bus energy increases with the number of processors (as Fig. 11 shows) and broadcast filtering saves bus energy.

The broadcast filtering also reduces the energy requirement of the cache to some extent, by the prevention of unnecessary tag lookups. Even though the snooping cache requires additional energy, the overall cache energy (i.e. L1 cache energy + snooping cache energy) is reduced. More overall cache energy is saved as the number of processors and the remote miss ratio increases. In four processors, the overall cache energy is 95% of the baseline requirement; but it drops to 70% in 16 processors with a remote miss ratio of 1.0.

Unfortunately broadcast filtering actually consumes a little more energy when there are two processors and the remote miss ratio is less than 0.3. This energy is consumed dynamically and statically by additional components, i.e. the snooping cache and splitters. In the aspect of static energy, because broadcast filtering needs additional logics such as a snooping cache and splitters, and it increases the latency of cache coherency operation, static energy consumption increases a little. As the amount of increase is less than 2%, static energy increase by broadcast filtering can be assumed to be negligible.

7. Conclusions and future work

The broadcast filtering reduces the snoop energy consumed by both the cache and the bus in an MPSoC. It most cases, it can detect when coherency requests will result in remote misses, and then it prevents unnecessary broadcasts being sent to remote caches which do not have the required data. The broadcast filtering is implemented using a snooping cache and a split bus. The snooping cache checks whether matching blocks exist in remote caches before broadcasting coherency requests. If a remote miss is detected, the snooping cache filters out the broadcast. If one or more remote hits are detected, only a part of the split bus may be used, so that the request is selectively broadcast to the remote caches which have the matching data.

By experiment we showed that our technique can eliminate by about 40% of snoop energy consumption. This saving is larger than that achieved by other techniques, because the broadcast filtering reduces not only cache energy but also bus energy. We designed an average snoop energy model, which enabled us to predict the saving in snoop energy consumption for applications with different the remote miss ratios, and MPSoCs with different numbers of processors. An energy model analysis showed that the broadcast filtering could reduce by 55% of the energy consumption per cache coherency operation. This combination of experimental and analytical results demonstrates the effectiveness of our approach and we expect the broadcast filtering to be used as an architecture-level energy cache coherency scheme in low-power MPSoCs.

In future work, we intend to enhance the effect of broadcast filtering by exploiting a knowledge of the data sharing characteristics of the tasks in parallel applications. For example, the broadcast filtering selectively broadcasts coherency requests to remote caches to obtain shared data, and the energy consumption of each snoop operation is affected by the distance between the caches which are sharing the data. If the caches are nearer to each others, less energy is consumed during a snoop operation, so we could reduce the energy cost of snoop operations by assigning tasks which share data to closer processors.

Acknowledgement

This work was supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MEST) (No. R0A-2007-000-20116-0), and the Brain Korea 21 Project in 2008. The ICT at Seoul National University provided research facilities for this study.

References

[1] J. Goodacre, A.N. Sloss, Parallelism and the ARM instruction set architecture, IEEE Computer 38 (7) (2005).
[2] D. Courtright, MIPS32 M4K core for multi-CPU applications, Embedded Processors Forum, April 2002.
[3] L. Hammond, B.A. Hubbert, M. Siu, M.K. Prabhu, M. Chen, K. Olukotun, The stanford hydra CMP, IEEE Micro 20 (2) (2000).
[4] N. Magen, A. Kolodny, U. Weiser, N. Shamir, Interconnect-power dissipation in a microprocessor, System Level Interconnect Prediction, February 2004.
[5] A. Moshovos, G. Memik, B. Falsafi, A. Choudhary, Jetty: filtering snoops for reduced energy consumption in SMP servers, in: International Symposium on High-Performance Computer Architecture, January 2001.
[6] M. Ekman, F. Dahlgren, P. Stenström, TLB and snoop energy reduction using virtual caches in low-power chip-multiprocessors, in: International Symposium on Low Power Electronics and Design, August 2002.
[7] A. Moshovos, RegionScout: exploiting coarse grain sharing in snoop-based coherence, in: International Symposium on Computer Architecture, June 2005.
[8] M. Ekman, F. Dahlgren, P. Stenström, Evaluation of snoop energy reduction techniques for chip-multiprocessors, in: Workshop on Duplicating, Deconstructing, and Debunking, May 2002.
[9] C. Saldanha, M. Lipasti, Power efficient cache coherence, in: Workshop on Memory Performance Issues (in conjunction with ISCA), June 2001.
[10] K. Strauss, X. Shen, J. Torrellas, Flexible snooping: adaptive forwarding and filtering of snoops in embedded-ring multiprocessors, in: International Symposium on Computer Architecture, June 2006.
[11] J.Y. Chen, W.B. Jone, S. Wang, H.I. Lu, T.F. Chen, Segmented bus design for low-power systems, IEEE Transactions on Very Large Scale Integration Systems 7 (1) (1999).
[12] K. Banerjee, A. Mehrotra, A power-optimal repeater insertion methodology for global interconnects in nanometer designs, IEEE Transactions on Electron Devices 49 (11) (2002).
[13] C.T. Hsieh, M. Pedram, Architectural energy optimization by bus splitting, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 21 (4) (2002).
[14] J. Guo, A. Papanikolaou, P. Marchal, F. Catthoor, Physical design implementation of segmented buses to reduce communication energy, in: Asia and South Pacific Design Automation Conference, 2006.

[15] D. Kim, S. Ha, R. Gupta, CATS: cycle accurate transaction-driven simulation with multiple processor simulators, in: Design Automation and Test in Europe, April 2007.
[16] D. Burget, T. Austin, The SimpleScalar Tool Set Version 4.0. <http://www.simplescalar.com/v4test.html>.
[17] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta, The SPLASH-2 programs: characterization and methodological considerations, in: International Symposium on Computer Architecture, June 1995.
[18] P. Shivakumar, N.P. Jouppi, CACTI 3.0: an integrated cache timing, power, and area model, WRL Research Report 2001/2, 2001.
[19] D.E. Culler, J.P. Singh, A. Gupta, Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufman, 1999.
[20] ARM, MPCore Multiprocessor Technical Reference Manual (ARM DDI 0360A), ARM, 2005.
[21] C. Yu, P. Petrov, Aggressive snoop reduction for synchronized producer–consumer communication in energy-efficient embedded multi-processors, in: International Conference on Hardware–Software Codesign and System Synthesis, 2007.
[22] C. Yu, P. Petrov, Latency and bandwidth efficient communication through system customization for embedded multiprocessors, in: Design Automation Conference, 2008.
[23] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, M. Stan, HotLeakage: a temperature-aware model of subthreshold and gate leakage for architects, University of Virginia Department of Computer Science Technical Report CS-2003-05, 2003.
[24] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, R.B. Brown, MiBench: a free, commercially representative embedded benchmark suite, in: Workshop on Workload Characterization, December 2001.
[25] M. Kuhlmann, S.S. Sapatnekar, Exact and efficient crosstalk estimation, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 20 (7) (2001).
[26] S.C. Wong, G.Y. Lee, D.J. Ma, Modeling of interconnect capacitance, delay, and crosstalk in VLSI, IEEE Transactions on Semiconductor Manufacturing 13 (1) (2000).
[27] K. Nabors, J. White, FastCap: a multipole-accelerated 3-D capacitance extraction program, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 10 (11) (1991).

Chun-Mok Chung received BS in computer engineering from Kwangwoon University, Seoul, Korea in 1997. He received MS in computer science from Yonsei University, Seoul, Korea in 1999. He is a PhD Student in School of Computer Science & Engineering, Seoul National University, Seoul, Korea. His research interests include computer architecture, low-power design, and embedded system. He is a student member of ACM.

Jihong Kim received his BS in computer science and statistics from Seoul National University, Seoul, Korea in 1986, and MS and PhD degrees in computer science and engineering from the University of Washington in 1988 and 1995, respectively. He is a Professor in the School of Computer Science & Engineering, Seoul National University. Before joining Seoul National University in 1997, he was a Member of Technical Staff in the DSPS R&D Center of Texas Instruments in Dallas, Texas. His research interests include embedded software and systems, low-power systems, computer architecture, image/multimedia systems and real-time systems. He is a member of IEEE Computer Society and ACM.