Broadcast Filtering-Aware Task Assignment Techniques for Low-Power MPSoCs

Chun-Mok Chung
School of Computer Science and Engineering
Seoul National University
Seoul 151-742, Korea
chunmok@davinci.snu.ac.kr

Jihong Kim
School of Computer Science and Engineering
Seoul National University
Seoul 151-742, Korea
jihong@davinci.snu.ac.kr

ABSTRACT

Broadcast filtering technique is useful in reducing the snoop-energy consumption of shared bus-based MPSoCs by intelligently avoiding useless coherency-request broadcasts. Since the patterns of coherency-request broadcasts are highly dependent on how concurrent tasks are assigned to multiple processors, a broadcast filtering-aware task assignment is important in achieving a high-level of energy efficiency for MPSoCs with a broadcast filtering support. In this paper, we propose broadcast filtering-aware task assignment techniques for low-power MPSoCs, taking advantage of the patterns of coherency-request broadcasts of given tasks. We first propose a restricted *optimal* task assignment technique that can be useful only when the number of tasks is equal to the number of processors. Then, we describe a general task assignment heuristic that can be used for the arbitrary number of tasks.

Experimental results show that when the number of tasks is equal to the number of processors, the proposed optimal task assignment technique reduces the snoop energy consumption by 13% over naive task assignment cases. For general task sets, the proposed task assignment heuristic reduces the snoop energy consumption by 15% over naive task assignment cases.

Categories and Subject Descriptors

D.4.1 [**Software**]: Operating Systems—*Process Management*; B.3.2 [**Hardware**]: Memory Structure—*Design Styles*

General Terms

Algorithms, Design

Keywords

MPSoC, Energy reduction, Task assignment, Broadcast filtering, Snoop-based cache coherency

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MEDEA '07, September 16, 2007 Brasov, Romania Copyright 2007 ACM 978-1-9593-807-7/07/09... $5.00

1. INTRODUCTION

According to the development of architecture and VLSI technology, multiprocessor system-on-a-chips (MPSoCs) are now widely used in high-performance mobile embedded systems [1, 2]. They include multiple processor cores and on-chip memories in single chip die. As they can execute multiple context concurrently, they can meet the high performance demands for multimedia applications on mobile embedded systems. However, the higher performance means that they consume the more power. Since mobile embedded systems, such as cellular phones and personal game players, use batteries as their power sources, the importance of low power consumption increases in the design of MPSoCs.

As typical MPSoCs contain private local caches for each processor to enhance the performance, they have a cache coherency problem like general multiprocessor systems. In shared bus-based MPSoC environments, snoop-based schemes are widely used to solve the cache coherency problem. In snoop-based schemes, if a local cache requires or modifies data, it broadcasts a coherency request message and remote caches snoop on the broadcast and keep their data coherent. As an on-chip global wire occupies up to 25% of total chip power [3] and a tag lookup operation contributes up to 50% of cache energy [4], a cache coherency operation becomes one of major energy consumers in MPSoCs.

To reduce snoop-energy, a broadcast filtering technique [5] was proposed. It used a snooping cache and a split bus architecture so as to remove unnecessary coherency request broadcasts, and reduced 30% of snoop-energy in a shared bus-based four-way MPSoC. From the analysis of snoop-energy reduction by the broadcast filtering, we knew that the energy consumed by a cache coherency operation is variant according to the distance among a requestor cache, the snooping cache, and a data supplier cache. Because a coherency request is sent only to the remote caches which have the requested data and the requested data is copied from the closest supplier cache, the numbers of used bus segments, used splitters, and tag lookup operations is dependent on the distance among participated caches. Therefore, we propose a snoop-energy minimizing technique by making tasks related with a cache coherency operation to be executed on the processors located close.

Although, finding an optimal through the execution and comparison of all cases is the most confident method, it

takes too much time. We also propose an optimal detection technique which ensures the optimality without the execution of all cases. We analyze the execution characteristic of tasks and hardware characteristic of target MPSoC. With the analysis, our technique detects an optimal task assignment with only single execution and multiple calculations. We first propose an optimal task assignment technique, if the number of tasks is same to the number of processors. However, as the number of tasks is not always same to the number of processors in real computing environment, we extend our technique to be applicable even though the number of tasks is different from the number of processors, where a grouping technique is used to make the number of tasks be same to the number of processors. As the location of snooping cache can be changed in the early design step of target MPSoC, we detect optima for two cases, 1) the location of snooping cache is fixed as the middle of processors and 2) the location of snooping cache is not fixed.

Experimental results show that when the number of tasks is equal to the number of processors, the proposed optimal task assignment technique reduces the snoop energy consumption by 13% over naive task assignment case and the more energy is reduced with the more tasks. For general task sets, the proposed task assignment heuristic reduces the snoop energy consumption by 15% over naive task assignment case. If we compare the optimal task assignments before and after fixing up the location of snooping cache, most of cases, the same task assignment is selected for minimal snoop-energy. It says that snoop-energy is minimized when a snooping cache is located in the middle of processors. These results show that our approach is energy efficient and we expects that our software level snoop-energy reduction technique will help to design more energy efficient embedded systems.

The contribution of this paper can be summarized into two aspects. First, we proposed a snoop-energy reduction technique by an optimal task assignment, which takes advantage of hardware and software characteristics. Second, we propose an energy efficient task assignment heuristic, without the restriction of the number of tasks and the number of processors.

The rest of paper is organized as follows. We describe related works and the broadcast filtering in Section 2 and Section 3, respectively. In Section 4, we explain how to detect a snoop-energy optimal task assignment. The experimental results are shown in Section 5. Finally, we draw conclusions of our study in Section 6.

2. RELATED WORK

To reduce cache energy in a cache coherency operation, cache lookup filtering and serial cache lookup have been proposed. Jetty [4] is a small structure attached to each cache. Before a cache lookup, Jetty was checked and it filtered out useless cache lookups. RegionScout [6] saved more cache energy than Jetty by using smaller sized filters. Jetty used one entry per cache block, whereas RegionScout used one entry per region which is a continuous memory area. Serial snooping [7] and flexible snooping [8] have been proposed as serial cache lookup techniques. Instead of broadcasting coherency requests to all remote caches in parallel, remote caches were checked serially, one by one.

To reduce bus energy, several researchers have proposed and used bus splitting techniques. Chen et al. [9] proposed the first bus splitting technique based on pass transistors and used a graph search technique to find an energy-efficient bus topology. Heish et al. [10] divided a bus into two segments and connected communication components based on a probabilistic model of communication to minimize bus energy. But they ignored the energy cost of the splitter. A multiple simultaneously accessible split-bus architecture has been proposed by Lu et al. [11]. Their goal was to enhance the bandwidth and latency of the shared bus. They did not consider energy. Guo et al. [12] proposed a design method for using a segmented bus. They minimized the number of bus segments by block ordering and the length of each bus segment by floor planning.

Optimal assignment and mapping problems happen in numerous design processes. Quadratic assignment problem [13] is widely used in the task of location electrical assemblies in given slots so as to minimize the total length of interconnecting wires [14] and in mapping strategy for parallel processing [15]. Recently, Guo et al. [16] proposed an energy efficient network topology using the solution of optimal communication spanning tree problem.

3. LOW-POWER MPSOCS WITH A BROADCAST FILTERING SUPPORT

As typical MPSoCs in an academic and commercial area contain multiple homogeneous processors and on-chip caches [1, 17], we specify the target MPSoC of this paper to be similar to them. Figure 1 represents the architecture of target MPSoC which is capable of the broadcast filtering. Shaded blocks are added logics to implement broadcast filtering. The MPSoC contains multiple homogeneous processors, and each processor has its own private L1 caches (I-cache and D-cache). The L1 D-cache has a duplicated tag to prevent processor delay during snooping. The MPSoC may contain a shared L2 cache to enhance performance. All processors share the memory (or L2 cache) and they communicate with each other through a shared bus. A snoop-based cache coherency protocol is used to keep data coherency between cache and memory.

It contains a directory, named as *snooping cache*, to determine if a requested data is contained in remote caches. The snooping cache has sharing information about data blocks in L1 caches. It is different from the conventional directory which keeps sharing information of data blocks in lower-level memory hierarchy. If the snooping cache receives a coherency request with a block address, it selects a set using an index part of address. After comparing a tag part of address with the tag array of the snooping cache, it outputs a corresponding flag vector and snoop-hit information.

To selectively broadcast coherency requests, the MPSoC adopts a split-bus architecture. A processor or snooping cache is connected to each bus segment. A splitter connects two adjacent bus segments and transfers signals between them only if it is activated. The split-bus works like a monolithic bus if all splitters are activated. An arbiter and the snooping cache control the splitters.

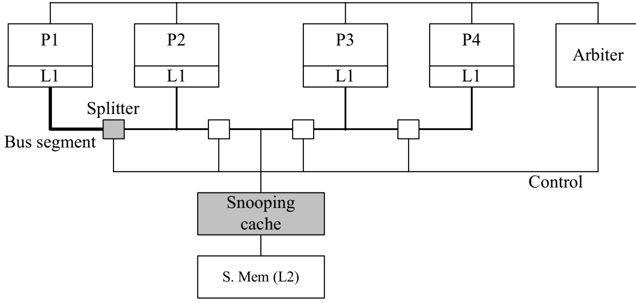

Figure 1: The architecture of target MPSoC with a broadcast filtering support.

The process of cache coherency operation is modified after applying the broadcast filtering. In case of BusRd, (1) the local cache requests bus use to the arbiter to perform a BusRd transaction. (2) the arbiter permits bus use. At the same time, it activates the splitters between the local cache and the snooping cache. (3) the local cache sends a BusRd request to the snooping cache. (4) the snooping cache checks if the requested data is shared by remote caches. If remote caches contain the requested block, it activates the splitters corresponding to remote caches. (5) if the requested data is shared, the snooping cache broadcasts the request to remote caches. Otherwise, it sends the request to the shared memory. (6) remote caches or the shared memory snoop the request and supplies the requested data to the local cache.

Because coherency requests are sent only to the remote caches that have the requested data and the requested data is copied from the closest supplier cache, the numbers of used bus segments, used splitters, and tag lookup operations is dependent on the distance between participating caches. Therefore, we can minimize snoop-energy by making the tasks that raise many cache coherency operations to be executed on the processors that is located close.

4. BROADCAST FILTERING-AWARE TASK ASSIGNMENT TECHNIQUES

We are given a task set $\mathbf{T}$ and a processor set $\mathbf{P}$ and they are defined as follows:

$$\begin{aligned}\mathbf{T} &= \{\tau_1, \tau_2, ..., \tau_k\} \\ \mathbf{P} &= \{p_1, p_2, ..., p_m\}.\end{aligned}$$

If $k = m = N$, the number of possible task assignments of $\mathbf{T}$ into $\mathbf{P}$ is as large as $N!$. Although, finding an optimal with execution and comparison of all cases is the most confident method, it takes too much time. So, we propose an optimal detection technique which ensures the optimality without execution of all cases. We analyze an execution characteristic of tasks and hardware characteristic of target MPSoC. With the analysis, our technique detects an optimal task assignment with only single execution and multiple calculations.

However, k is not always same to m in general computing environments. So, we first propose an optimal task assignment technique for the case of $k = m$ and extend our technique to cover the case of $k \neq m$.

4.1 Restricted optimal task assignment technique (RTA)

If we are given a cost matrix of n objects and a distance matrix of n locations, the problem of finding the assignment(a) over all permutations that minimizes the objective function(o) is called as a quadratic assignment problem (QAP) [13]:

$$o = \sum_{i,j} c_{ij} d_{a(i)a(j)}.$$

If $k = m = N$, RTA detects an optimal task assignment by formulating the problem with QAP. However, as the variables of two problems have difference, we will define a cost matrix, a distance matrix, and an objective function to detect an optimal with QAP solution.

4.1.1 *Definition of cost matrix*

As the number of cache accesses is proportional to the number of bus transactions, we use only a bus transaction frequency and infer cache energy from it using coefficients.

A cost matrix is defined as the number of bus transactions between tasks, as follows:

$$\mathbf{C} = \{(r_{ij}, b_{ij}) | 1 \leq i, j \leq N + 1\}.$$

A cost is represented as a tuple of (r, b), where r means the number of coherency requests and b means the number of block copies. We distinguished r and b, because the number of bus transactions per request and those per data block copy may be different according to a system configuration.

The cost matrix is built from a trace about cache coherency operations during execution of an application. Figure 2 shows the cost matrix generation algorithm. As a BusRd and a BusRdX send coherency requests to remote caches, r between the local cache and the snooping cache increases one. If a snoop-hit happens, as coherency requests are forwarded to remote caches, r's between the snooping cache and remote caches also increase one. We also add $1/h$ to b's between them. Although the local cache copies data from only one remote cache that is the closest to the local cache in the reality, as we cannot know the distance between the local cache and remote caches before task assignment, we assume the probability which remote cache is chosen as a data supplier is equivalent and add $1/h$ to b's between the local cache and all remote caches equivalently. In case of a BusUpgr, r between the local cache and the snooping cache is incremented by one. If a snoop-hit happens, r's between snooping cache and remote caches also increase one. As a BusUpgr does not need a data copy between caches, b's between them has no change. If a coherency type is a BusWB, r and b between the local cache and the snooping cache increase one. Because a dirty block is not shared by remote caches, no bus transaction happens between caches.

4.1.2 *Definition of distance matrix*

As a split-bus architecture is used the target MPSoC, bus energy is dependent on the number of bus segments used in a bus transaction. If all units (processors and snooping cache)

```
Input; cache coherency operation trace
Output; cost matrix C = {r_ij, b_ij}
Definition; l:local cache, e:snoop-hit remote caches,
            s:snooping cache, h:snoop-hit count
1:  for (all trace)
2:      Select one line of trace;
3:      if (coherency type is BusRd or BusRdX)
4:          r_ls += 1;
5:          if ( snoop-hit )
6:              r_le += 1;
7:              b_le += 1/h;
8:          end if
9:      else if (coherency type is BusUpgr)
10:         r_ls += 1;
11:         if ( snoop-hit )
12:             r_se += 1;
13:         end if
14:     else if (coherency type is BusWB)
15:         r_ls += 1;
16:         b_ls += 1;
17:     end if
18: end for
```

Figure 2: Cost matrix generation algorithm.

are labeled as $u_1 \sim u_{N+1}$, we define a distance matrix using the number of bus segments between two units as follows:

$$\mathbf{D} = \{d_{ij} | 1 \leq i,j \leq N+1\},$$
$$\text{where } \forall u_i, u_j : d_{ij} = \begin{cases} |i-j|+1, & \text{if } i \neq j \\ 0, & \text{if } i = j. \end{cases}$$

4.1.3 Definition of objective function

Total snoop-energy is the sum of consumed energy by L1 D-caches and the shared bus [5]. Cache energy is the sum of energy during tag lookups (E_{tag}), full cache accesses for data copy (E_{L1}), and snooping cache accesses (E_{sc}). Bus energy is the energy consumed by bus segments (E_{bs}) and splitters (E_{sp}) for bus transactions. So, the total snoop-energy can be expressed as follows:

$$\begin{aligned} E_{snoop} &= E_{cache} + E_{bus} \\ &= \sum E_{tag} + \sum E_{L1} + \sum E_{sc} \\ &\quad + \sum E_{bs} + \sum E_{sp}. \end{aligned}$$

However, full cache access energy and snooping cache access energy is have no relationship with a task assignment. So, the snoop-energy with an arbitrary task assignment (a) is proportional to the energy consumed in tag lookups, bus segments, and splitters.

$$E^a_{snoop} \propto \sum E_{tag} + \sum E_{bs} + \sum E_{sp}.$$

The goal of an optimal task assignment is to minimize snoop-energy. So, an objective function means the snoop-energy part dependent on task assignments and it is defined as follows using variables of cost matrix and distance matrix:

$$\begin{aligned} o &= E^a_{snoop} = \sum_{i,j} (r_{ij}, b_{ij}) d_{a(i)a(j)} \\ &= \sum_{i,j} \Big\{(1+\alpha) r_{ij} + \gamma b_{ij}\Big\} \hat{d}_{ij} + \beta(r_{ij} + \gamma b_{ij})(\hat{d}_{ij} - 1), \end{aligned}$$
$$\text{where } \hat{d}_{ij} = d_{a(i)a(j)}, \alpha = \frac{E_{tag}}{E_{bs}},$$
$$\beta = \frac{E_{sp}}{E_{bs}}, \text{ and } \gamma = \frac{\text{block size}}{\text{bus width}}.$$

α indicates the energy rate of cache and bus segment and β means that of splitter and bus segment. α and β are used to calculate the energy consumed by cache lookups and splitters during coherency operations. γ is the rate of cache block size and bus width. It means how many of bus transactions is necessary to copy a cache block. We use γ to calculate the number of bus transactions in cache coherency operations.

As we previously described in the cost matrix definition, we count only the number of bus transactions between tasks and estimate snoop-energy consumed by caches and splitters using the coefficients (α, β, γ). This is possible because the number of tag lookups is same to the number of requests and the number of used splitters is one less than the number of used bus segments. The coefficients can be determined from system configuration and energy consumption per component.

4.2 General task assignment heuristic (GTA)

The basic idea of GTA is to make the number of tasks to be same to the number of processors, and to detect an optimal task assignment using the RTA. We define a new task set $\mathbf{T}'$ which size is same to the number of processors and perform the RTA to assign $\mathbf{T}'$ to $\mathbf{P}$. From now, we will describe how to generate the new task set $\mathbf{T}'$ and a corresponding cost matrix $\mathbf{C}'$ according to the number of tasks.

4.2.1 A case that the number of tasks is bigger than the number of processors

If $|\mathbf{T}| = k > |\mathbf{P}| = m$, multiple tasks should be executed on the same processor and it needs definition of multitasking environment. As there are many kinds of programming models and task scheduling policies, we restrict the multitasking environment we consider in this paper.

Programming model: programmers can use multiple processors simultaneously using a multithreaded programming model [18]. This model is widely supported in embedded operating systems [19, 20].

Task scheduling policy: all processors use a shared task queue. However, each task is scheduled only to the specified processor. The processor ID of each task is determined on task creation and not changed until finish. This policy is widely used in embedded systems because the hardware and software of embedded systems are fixed in design step.

```
Input; original task set T, cost matrix C, time limit
Output; new task set T'
Definition; unified cost matrix C^U,
            et(T): execution time of T
1:  τ'_1 = τ'_2 = ... = τ'_m = {};
2:  for ∀c^u_ij ∈ C^U
3:      c^u_ij := r_ij + b_ij * γ;
4:  end for
5:  while (T ≠ {}) do
6:      Choose a maximum cost c^u_ij;
7:      T_M := {τ_i, τ_j};
8:      if (T_M ⊂ T)
9:          if ( Any task in T_s exists in τ'_i )
10:             T_C := T_M − τ'_i;
11:             if (et(τ'_i)+et(T_C) ≤ time limit)
12:                 Add T_C into τ'_i;
13:                 Remove T_C from T;
14:             end if
15:         else if (et(τ'_i)+et(T_M) ≤ time limit)
16:             Add T_M into τ'_i;
17:             Remove T_M from T;
18:         end if
19:     end if
20:     c^u_ij := 0;
21: end while
```

Figure 3: Task grouping algorithm.

To make the number of tasks to be same to the number of processors, we define new the task set $\mathbf{T}'$ as follows:

$$\begin{aligned}\mathbf{T}' = \{\tau'_1, \tau'_2, ..., \tau'_m\}&,\\ \text{where } \forall \tau'_i : \tau'_i \subset \mathbf{T}&,\\ \forall \tau'_i, \tau'_j : \tau'_i \cap \tau'_j = \{\},&\ \text{and}\\ \tau'_1 \cup ... \cup \tau'_m = \mathbf{T}&.\end{aligned}$$

As the address space of all tasks are shared in our multitasking environment, no cache flush is necessary for context switching. So, as the cache is shared by the tasks assigned into the same processor, there is no cache coherency problem. Therefore, we cluster the tasks with many cache coherency operations into the same task group. However, if all tasks are clustered into the same task group, the execution time will be extended. So, we added a time constraint to finish all tasks. The Figure 3 shows our task grouping algorithm without violation of time constraint.

With the original task set, the original cost matrix, and time constraint, we generate the new task set. Initially, we make all elements of the new task set empty and generate a unified cost set, which consists of the number of bus transactions without distinguishing the number of requests and the number of block copies. To classify frequently communicating tasks into the same task group, we select two tasks with the maximum cost. If one of them is already classified into a task group and if the other can be assigned to the same task group without violating the time constraint, we assign them into the same task group and remove the tasks from the original task set. Otherwise, we assign the selected tasks into other task group which does not violate the time constraint. Until all tasks in the original task set are grouped, we repeat the grouping process.

The grouping heuristic may not generate optimal task groups, because it uses cache coherency frequencies between tasks (when a processor executes a task) as the metric of grouping. If multiple tasks are assigned into the same processor, as the cache is shared by the assigned tasks, cache usage pattern may be changed because of conflict misses between the tasks. But it is difficult to predict cache usage pattern of task group before grouping, so we assume the cache coherency frequencies of tasks after grouping will follow those of individual executions.

After the new task set is generated, we generate a new cost matrix for it. As each task in the new task set indicates a task group, the costs are changed from the costs between tasks to the costs between task groups. As an original cost consists of the number of requests and the number of block copies, if a tuple (r'_{ij}, b'_{ij}) means a new cost between two task groups τ'_i and τ'_j in the new task set, the new cost matrix is defined as follows:

$$\begin{aligned}\mathbf{C}' = \{(r'_{ij}, b'_{ij}) | 1 \le i, j \le m+1\}&,\\ \text{where } r'_{ij} = \sum_{n,l} r_{nl},\ b'_{ij} = \sum_{n,l} b_{nl}&,\\ \tau_n \in \tau'_i,\ \tau_l \in \tau'_j,\ \text{and}&\\ \tau'_{m+1} = \text{snooping cache}&\ .\end{aligned}$$

4.2.2 A case that the number of tasks is smaller than the number of processors

If $|\mathbf{T}| = k < |\mathbf{P}| = m$, we add empty tasks to make the number of tasks and the number of processors same. So, the new task set $\mathbf{T}'$ is defined as follows:

$$\begin{aligned}\mathbf{T}' = \{\tau'_1, \tau'_2, ..., \tau'_m\}&,\\ \text{where } \tau'_i = \begin{cases} \tau_i, & \text{if } 1 \le i \le k \\ \emptyset, & \text{if } k+1 \le i \le m. \end{cases}\end{aligned}$$

The new cost matrix $\mathbf{C}'$ is defined as a superset of $\mathbf{C}$. As the size of cost matrix increases from $(k+1) \times (k+1)$ to $(m+1) \times (m+1)$, we set a tuple (0,0) as the cost related to the added empty tasks.

4.3 Range of optimal assignment

According to the precedence of task assignment operation in an embedded system design process, an optimization range can be changed. If we assign tasks before determining the location of snooping cache, the location of snooping cache can be optimized, too. So, we detect optimal assignments for following two cases.

OptTask: the location of snooping cache is fixed as the middle of processors and we find an optimal assignments of tasks. We consider only the task assignment functions which assign the snooping cache into $u_{N/2}$. It means the case that a hardware floor plan is determined before a task assignment operation.

OptAll: the snooping cache can be placed in any location. It can be the side of bus or the middle of any two processors. So, we find an optimal assignment not only tasks but also the location of snooping cache. It means the case that a

Table 1: Simulation parameters for target MPSoCs.

Parameter	Value
Processor	ARM core
# of processors	2, 4, 8, 16
Private L1 I-, D-cache	32KB, 32-byte blocks, 4-way
Cache coherency protocol	MESI [24]
Interconnect	32-bit shared bus
E_{tag}	0.0882nJ
E_{bs}	0.0074nJ/mm
E_{sp}	0.0163nJ

hardware floor plan is not fixed and can be changed after a task assignment process.

5. EXPERIMENTS

5.1 Environment

We evaluated the snoop-energy reduction effect by the proposed task assignment techniques using simulation based experiments. We used the CATS simulator [21], which is an extended version of the SimpleScalar tool [22] for an MPSoC simulation. It supports multiple processors, private cache, shared bus, shared memory, and snoop-based cache coherency protocol. We configured CATS similar to MPCore specification [23], as it is one of representative MPSoCs and widely used in an industry field. Table 1 shows the detailed simulation parameters for baseline MPSoCs.

We executed programs in the SPLASH-2 suite [25] with various number of tasks (2, 4, 8, and 16) and generated traces about cache coherency operations such as coherency type, address, and cache updates. Although SPLASH-2 is not developed for embedded system evaluation, as it is widely used in multiprocessor platforms and there is not any widely used parallel applications for embedded system, we selected it and used a small size input data.

With the traces from the CATS, we generated the cost matrix. The distance matrix was generated using the number of bus segments between processors from the configuration of simulator. With the matrixes, we detected snoop-energy optimal task assignments using the RTA and the GTA. In case of 16 processors, as there are too many possible assignments (=17!), we used the FANT [26], one of heuristic searching algorithms for QAP, in the RTA. With the optimal assignment and coherency traces, we estimated snoop-energy using energy parameters in Table 1, which are gained from the CACTI [27] and Heish et al's work [10].

5.2 Objective function verification

Before inspecting the snoop-energy reduction effect by the proposed techniques, we verified if our objective function correctly represents snoop-energy. For this, we compared the relationship between snoop-energy and objective function value of all possible task assignments. However, as the objective function includes only the task assignment variant snoop-energy, direct comparison is not elibigle. Instead, we examined if the following proposition was kept true.

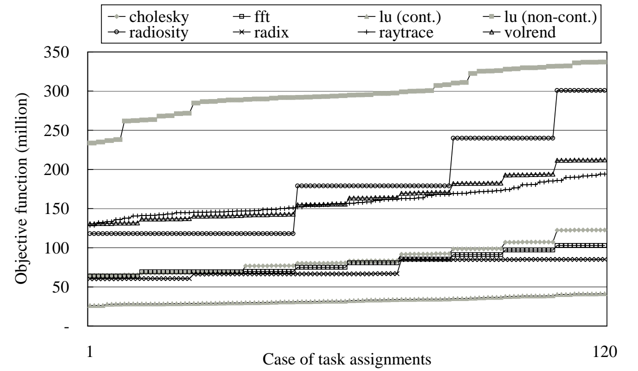

Figure 4: Increase of objective function according to increase of snoop-energy.

$\forall\ a_i,\ a_j \in$ {possible assignments}:
if snoop-energy with a_i > snoop-energy with $a_j \rightarrow$
objective function with a_i > objective function with a_j.

Figure 4 shows the objective function of task assignments. In an X-axis, task assignments are sorted in increasing order of snoop-energy. In all applications, as the snoop-energy increases, objective function also increases. So, we can know that our objective function always makes true the proposition and represents snoop-energy enough correctly. In some applications, there are ranges where the objective function does not increases. However, it is not the fault of objective function, but there are different task assignments with the same snoop-energy.

5.3 Snoop-energy reduction

To evaluate the snoop-energy reduction by the RTA, we executed two, four, 8, and 16 tasks on the MPSoCs with the same number of processors and estimated snoop-energy. Figure 5 shows the snoop-energy of four-way MPSoC with different task assignments. In the figure, *Baseline* means the snoop-energy when tasks are assigned sequentially. *RTA*s indicate the snoop-energies when only tasks(*OptTask*) or tasks and the snooping cache(*OptAll*) are optimally assigned by the RTA. All bars are normalized to the *Baseline*.

The RTA reduced snoop-energy by minimizing snoop-energy of frequently performed cache coherency operations. On average, it reduced snoop-energy by 13% over baseline model. In case of *radix*, there is little energy reduction with the RTA. It is because the frequencies of cache coherency operation between arbitrary two tasks are almost the same, so snoop-energy has no relationship with task assignment. In other side, in case of *radiosity*, the RTA reduced much of snoop-energy. As specific tasks perform much more cache coherency operations, by assigning those closely, 26% of snoop-energy is saved. So, the more snoop-energy is reduced, if cache coherency operations are distributed more partially. *OptTask* and *OptAll* reduce almost the same amount of snoop-energy at most of all applications, except *raytrace* and *volrend*. It means that the optimal location of snooping cache is the middle of processors. It is because of frequent bus transactions between caches and the snooping cache.

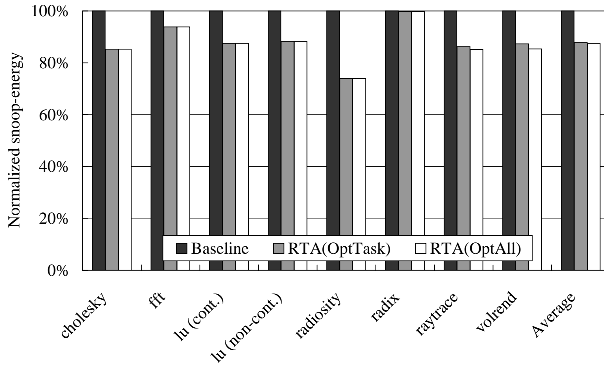

Figure 5: Snoop-energy when four tasks are executed on a four-way MPSoC.

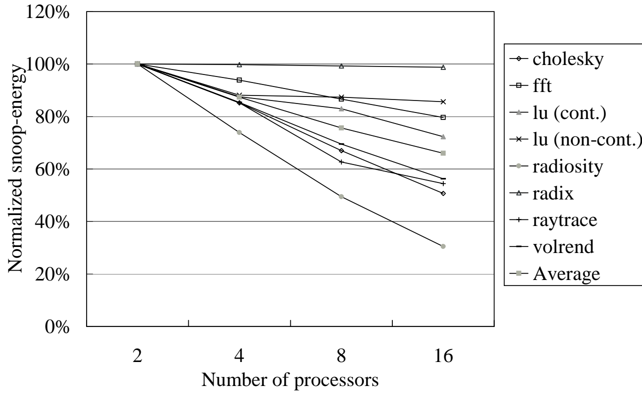

Figure 6: Snoop-energy with the RTA on various number of processors.

To perform cache coherency operation, all caches send coherency requests to the snooping cache. Also, if a snoop-miss is detected, the local cache copies data blocks not from a remote cache but from the shared memory through the snooping cache.

Figure 6 shows the snoop-energy reduction by the RTA in MPSoCs with two, four, 8, and 16 processors. All values in graph are normalized to the baseline model with the same number of processors. In all programs, the more snoop-energy is saved as the number of processors (tasks) increases. On the average, snoop-energy is reduced to 87% of baseline in four-way MPSoC. It decreases to 66% of baseline, if an MPSoC contains 16 processors.

To evaluate the snoop-energy reduction effect by the GTA, we executed 16 tasks on a four-way MPSoC and estimated snoop-energy. Figure 7 shows the result. *Baseline* indicates the snoop-energy when tasks are assigned to processors according to a round-robin scheme. As the GTA is performed in two steps (the grouping and the RTA), we distinguished the snoop-energy after each step. *Grouping* indicates the snoop-energy after only the grouping is performed and task groups are assigned to processors sequentially. *GTA* indicates the snoop-energy when tasks are assigned to processors through all steps of GTA. All bars are normalized to the *Baseline*.

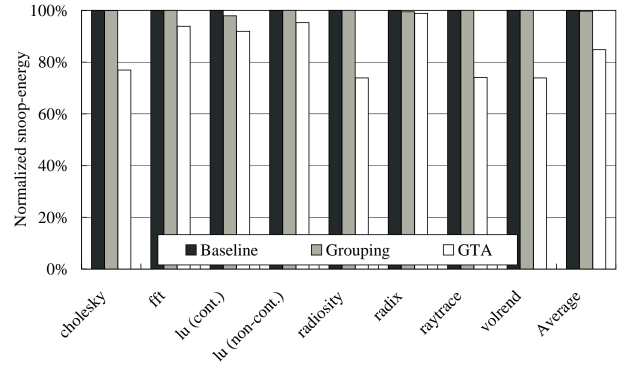

Figure 7: Snoop-energy when 16 tasks are executed on a four-way MPSoC.

If we compare *Baseline* and *GTA*, snoop-energy decreases maximally to 74% of baseline model with the GTA, and on average, the GTA reduces 15% of snoop-energy over the baseline model. As the coherency operation distribution between tasks and task groups after grouping is different, the snoop-energy reduction effect by the grouping step and the RTA step is different according to the distribution change. In cases of *cholesky*, *radiosity*, *raytrace*, and *volrend*, as the partiality of distribution is large, the RTA step reduces considerable snoop-energy. In other side, the grouping step reduces little snoop-energy. It reduce 1% $\sim$ 2% of snoop-energy, only in cases of *lu(cont.)* and *radix*. It is because the number of cache coherency operations is not reduced by the grouping. Although, the grouping removes the cache coherency operations between the tasks in the same task group, it results in conflict misses in caches and cache coherency operations between tasks allocated to different task groups.

6. CONCLUSIONS

We proposed an optimal task assignment techniques to minimize snoop-energy by considering the broadcast filtering and cache coherency operation among tasks. We explained how to detect an optimal task assignment technique, if the number of tasks is same to the number of processors and extended our technique to be applicable even though the number of tasks is different from the number of processors.

Experimental results show that when the number of tasks is equal to the number of processors, the proposed optimal task assignment technique reduces the snoop energy consumption by 13% over naive task assignment cases. For general task sets, the proposed task assignment heuristic reduces the snoop energy consumption by 15% over naive task assignment cases.

As the design of embedded systems require both of hardware and software level optimizations, we expect that our software level snoop-energy reduction technique will help to design more energy efficient embedded systems.

7. ACKNOWLEDGEMENTS

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

8. REFERENCES

[1] J. Goodacre, A. N. Sloss, "Parallelism and the ARM instruction set architecture," *IEEE Computer*, 38(7), Jul. 2005.

[2] D. Courtright, "MIPS32 M4K core for multi-CPU applications," *Embedded processors forum*, Apr. 2002.

[3] N. Magen, A. Kolodny, U. Weiser, and N. Shamir, "Interconnect-power dissipation in a microprocessor," *System level interconnect prediction*, Feb. 2004.

[4] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary, "Jetty: filtering snoops for reduced energy consumption in SMP servers," *HPCA*, Jan. 2001.

[5] C. M. Chung, J. Kim, and D. Kim, "Reducing snoop-energy in shared bus-based MPSoCs by filtering useless broadcasts," *GLSVLSI*, Mar. 2007.

[6] A. Moshovos, "RegionScout: exploiting coarse grain sharing in snoop-based coherence," *ISCA*, June 2005.

[7] C. Saldanha and M. Lipasti, "Power efficient cache coherence," *Workshop on memory performance issues*, June 2001.

[8] K. Strauss, X. Shen, and J. Torrellas, "Flexible snooping: adaptive forwarding and filtering of snoops in embedded-ring multiprocessors," *ISCA*, June 2006.

[9] J. Y. Chen, W. B. Jone, S. Wang, H. I. Lu, and T. F. Chen, "Segmented bus design for low-power systems," *IEEE Trans. on VLSI sys.,* 7(1), Mar. 1999.

[10] C. T. Heish and M. Pedram, "Architectural energy optimization by bus splitting," *IEEE Trans. on CAD of int. cir. & sys.*, 21(4), Apr. 2002.

[11] R. Lu and C. K. Koh, "A high performance bus communication architecture through bus splitting," *ASP-DAC,* Jan. 2004.

[12] J. Guo, A. Papanikolaou, P. Marchal, and F. Catthoor, "Physical design implementation of segmented buses to reduce communication energy," *ASP-DAC,* Jan. 2006.

[13] M. Hanan and J. M. Kurtzberg, "A review of the placement and quadratic assignment problems," *SIAM Review*, 14(2), Apr. 1972.

[14] S. H. Bokhari, "On the mapping problem,", *IEEE Trans. on comp.*, C-30(3), Mar. 1981.

[15] S. Y. Lee and J. K. Aggarwal, "A mapping strategy for parallel processing,", *IEEE Trans. on comp.*, C-36(4), Apr. 1987.

[16] J. Guo, A. Papanikolaou, and F. Catthoor, "Topology exploration for energy efficient intra-tile communication," *ASP-DAC,* Jan. 2007.

[17] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun, "The Stanford Hydra CMP," *IEEE Micro,* 20(2), Mar.-Apr. 2000.

[18] B. Nicols, D. Buttlar, and J. P. Farrell, *Pthreads Programming*, O'Reilly & Associates, Inc., 1996.

[19] Red Hat, Inc., "eCos," *http://ecos.sourceware.org/*

[20] J. J. Labrosse, *MicroC/OS-II*, CMP Books, 1999.

[21] D. Kim, S. Ha, and R. Gupta, "CATS: Cycle accurate transaction-driven simulation with multiple processor simulators," *DATE*, Apr. 2007.

[22] D. Burget and T. Austin, "The SimpleScalar tool set version 4.0," *http://www.simplescalar.com.*

[23] ARM, "MPCore multiprocessor technical reference manual," *ARM DDI 0360A,* Feb. 2005.

[24] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel computer architecture: a hardware/software approach*, Morgan Kaufmann publishers, 1999.

[25] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," *ISCA*, June 1995.

[26] E. D. Tailiard and L. M. Gambardella, "Adaptive memories for the quadratic assignment problem," *Technical report IDSIA-87-97*, Oct. 1997.

[27] P. Shivakumar and N. P. Jouppi, "CACTI 3.0: an integrated cache timing, power, and area model," *WRL research report 2001/2*, Aug. 2001.