Power-aware Bus Coscheduling for Periodic Realtime Applications Running on Multiprocessor SoC

Khaled Z. Ibrahim[1] and Smail Niar[2]
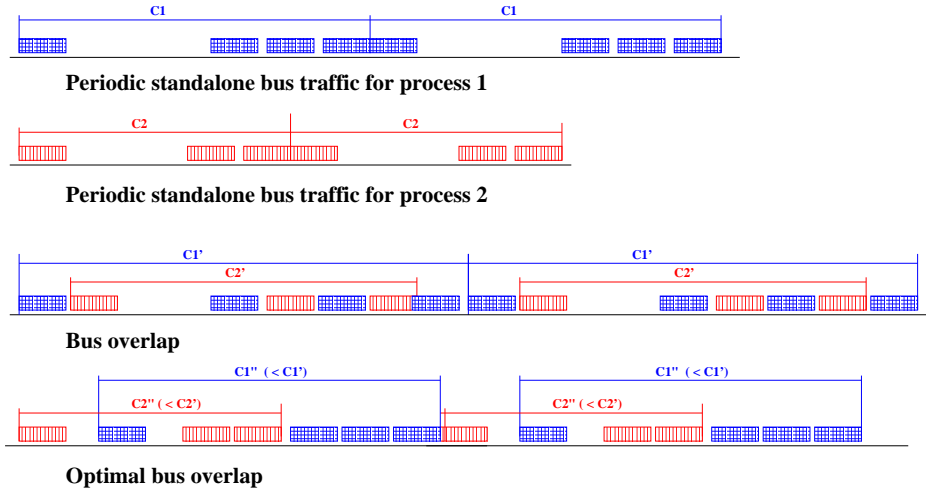
[1] Suez Canal University, 42563 Port Said, Egypt
[2] University of Valenciennes, 59313 Valenciennes Cedex 9, France

Abstract. Execution time for realtime processes running on multiprocessor system-on-chip platform varies due to the contention on the bus. Considering the worst case execution cycles necessitates over-clocking the system to meet the realtime deadlines, which has a negative impact on the system power requirements. For periodic applications coscheduled on multiprocessor with shared bus, the cycles needed by a memory transaction fluctuate based on the execution overlap between processes' activities on bus. In this work, we show the effect on execution cycles of different scheduling overlap of processes. Experiment results demonstrate that the execution cycles, and therefore the clock frequency, can be lowered by up to 24% on a 4 processor MPSoC. As the power consumption varies cubically with frequency, this reduction can lead to a significant power saving. Instead of exhaustively simulating all configurations to search for optimal scheduling overlap, we devise a scheme to predict the effect of scheduling. We propose the use of shift-variance of bus traffic profile of applications running individually on the system to predict the effect when scheduling these applications simultaneously. We show that the devised predictor of scheduling effect highly correlates to the behavior observed through simulations.

1 Introduction

In bus-based multiprocessor system, running multiple contending processes on the shared bus increases the completion time of bus transactions and consequently the number of cycles needed to finish these processes. In realtime system, *worst case execution time* (WCET) is usually considered while scheduling these processes. The system clock frequency is adjusted to meet the process realtime constraints.

Unfortunately, increasing the frequency (and possibly increasing the voltage, as well) to meet realtime deadlines negatively impacts the power consumption of the system. The dynamic power dissipation varies linearly with frequency and quadratically with supply voltage. With a linear relation between voltage and frequency, the increase in the number of cycles needed to execute a process can lead to a cubical increase in the dynamic power. The variation of application requirements and system state with time usually necessitates the dynamic adaptation of the system voltage and frequency. Dynamic

Periodic standalone bus traffic for process 1

Periodic standalone bus traffic for process 2

Bus overlap

Optimal bus overlap

Fig. 1. The effect of bus traffic overlap of two processes sharing a common bus.

voltage/frequency scaling (DVS) technique adapts the system frequency and voltage to the realtime constraints of the system [1,2], thus optimizing for the system power and energy.

Scheduling processes in multiprocessor system aims at coordinating the utilization of shared resources between competing processes [3,4,5]. The stress of each process on the memory system varies with time. The number of cycles to execute these processes can decrease if we could find an *optimal overlap* of the bus demands from processes running on the system. By "optimal overlap", we mean an overlap that minimize the average clock cycles that each process needs to complete a memory transaction.

In this work, we show the impact of coscheduling of processes on a shared bus for multiprocessor embedded system. We illustrate the variation in execution cycles based on the overlap of processes coscheduled on shared bus. Brute force search for optimal overlap of coscheduled processes requires simulations of the processes with all possible overlaps. This can be a prohibitively expensive process. Instead, we devise a scheme to predict optimal coschedule. This scheme narrows the search space for optimal overlap. We propose a process that comprises the following steps: identifying the initial phase of each process, finding periodicity in the process behavior, determining a common period between the coscheduled processes, finding profile of performance variation with all possible overlaps, and finally finding an optimal bus coschedule of the processes. We also introduce the use of scheduling barrier to maintain the optimal overlap of coscheduled processes.

The proposed scheme predicts the effect of coscheduling on the performance under all possible execution overlaps. This helps in identifying a schedule with minimum negative impact on performance (through minimizing bus contention)

and helps in reducing the number of cycles needed to execute each process. For realtime application reducing the number of cycles to execute a task reduces the power requirement because the system frequency and voltage can be reduced accordingly.

The proposed technique can be applied for coscheduling applications with periodic pattern of accessing the memory systems. For this class of applications, the same processing is usually applied on different frames of data and the processing is independent of the values of data processed. Even though the applicability of the proposed scheme is limited to this class of applications, specialized design process is common in embedded systems to achieve the best power consumption especially that these systems are usually dedicated to run a fixed set of applications.

The rest of this paper is organized as follows: Section 2 introduces the impact of contention on shared bus and its effect on the number of cycles a periodic task needs for execution. The simulation environment is described in Section 3. Our proposed technique to predict optimal bus scheduling is detailed in Section 4. We extend our formulation of the proposed scheme, in Section 5, to systems running arbitrary number of processes. Section 6 summarizes related work as well as future work. Section 7 concludes our work.

2   Impact of Bus Overlap on Performance

In this work, we constrain our discussion to multiprocessor system with applications known *a priori*. For clarity, we will consider system with two processes running concurrently. We will generalize our formulation in Section 5

In Figure 1, two processes with different traffic patterns are shown. The upper part of the figure shows the bus traffic for each application running individually on the system. Two different execution overlaps are shown in the lower part of the figure. The number of cycles (C_i) for each process depends on the overlap with the other processes running on the system. The clock frequency of the system f needed to meet deadline constraint is defined as C/T where C is number of cycles of a process with the realtime period T. Increasing the number of cycles C of a process necessitates increasing the frequency of the system f. Different proposals [2,6,7] describe how to adapt the frequency to the demand of a process with realtime constraints.

Increasing the frequency severely impacts the power requirements of the system. In CMOS based system, the commonly used technology in embedded systems, the power consumption is mainly due to dynamic power [8] that is given by $P = C_{ef} \cdot V_{dd}^2 \cdot f$ where C_{ef} is the effective switched capacitance, V_{dd} is the supply voltage. The frequency is almost linearly related to the supply
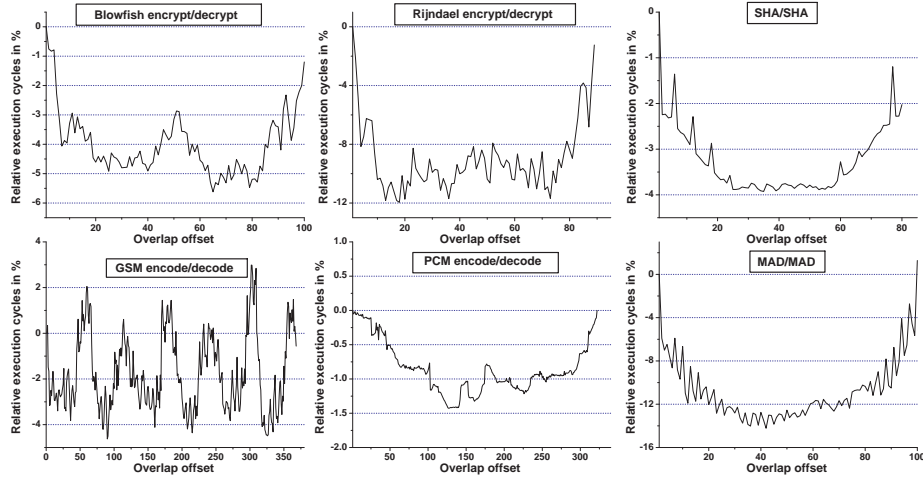
Fig. 2. Effect of all execution overlaps on the total execution cycles for six pairs of embedded applications. The effect is shown as a percentage of the execution cycles of an initial overlap.

voltage [8]. Consequently, the dynamic power is cubically related with the frequency ($P \propto f^3$). The energy (computed as $E = P \cdot T$) is quadratically related with the frequency, which is an important factor for battery powered devices.

Static power usually has negligible contribution to total power for embedded systems with low frequency. Static power is proportional to supply voltage V_{dd} [9,10], and thus can benefit linearly from decreasing frequency.

Figure 2 shows the effect of all execution overlaps for six pairs of embedded applications. Execution overlaps are generated by sliding the execution of one application with respect to the other application in the pair. These pairs of applications are taken from three categories of MiBench suite [11], namely the security, the telecommunication and the consumer categories. Each group is run on a dual processor system with shared bus. Each application exhibits a periodic behavior in accessing the bus. Figure 2 shows the percentage of change in execution cycles when we have different overlaps for each pair of applications. Depending on overlap (or shift), the execution cycles change by up to 5.5% for Blowfish, 12% for Rijndael, 4% for SHA, 6.5% for GSM, 1.5% for PCM, and 12.5% for MAD. These changes consider the difference between the maximum and the minimum execution cycles. The changes in the execution time are due to the memory latency changes that arise because of the different contention scenarios that these applications face on the bus. The details of the simulation environment are given in Section 3.

During the run of these applications, the execution cycles alternate between the values shown in Figure 2. To guarantee meeting deadline for real-time application, system designer usually considers worst-case execution cycles thus

necessitating overclocking the system. Based on the early discussion, if we can enforce a coscheduling that provide the minimum execution cycles then we can obtain a saving in the dynamic power consumption, relative to the power associated with worst-case execution cycles, from 4% for PCM up to 33% for MAD assuming a dual-core MPSoC. Even if the system bus and memory are not affected by the frequency scaling and only the processor core and caches are affected, the gain in reducing the processor power is very large. This is attributable to the large ratio of power consumed by the processor core and caches compared with the power consumed by the bus and memory subsystems. For PCM, this ratio is 50:1 and the ratio for MAD is 4:1. The saving due to frequency reduction is especially important for battery powered systems.

The cubic scaling of power with frequency is one of the main motives for building MPSoC for realtime systems running concurrent jobs, in constrained power environment. This alternative is based on using multiple processor cores with a lower frequency. The other, less efficient, alternative is to run the concurrent jobs on one processor core with higher frequency. Replicating the number of cores almost linearly increases the power demands while increasing frequency increase power demands cubically.

The above discussion shows the importance of coscheduling processes and its severe impact on the system power requirements. The main problem is to search for optimal configuration. This involves simulation of all possible overlaps which can be excessively expensive. We simulated 99 configurations for Blowfish, 90 for Rijndael, 79 for SHA, 368 for GSM, 321 for PCM, and 98 for MAD. The number of simulated configurations for each application is chosen to avoid repeated simulations and is determined by the common periodicity detected for each pair of applications as will be detailed in Section 4.3. Searching for a local minimum execution cycles using a technique such as steepest descent can face difficulty because of the existence of multiple minima and the existence of fast-varying change (ripples) as an additional component to the more important slowly-varying change. It will also require many simulation runs.

The determination of the optimal scheduling is an additional dimension to the design space for MPSoC. Multiple hardware configurations are usually explored in designing such systems. It is also common to have systems that run different combinations of applications simultaneously. For every set of applications, we need to find an optimal coschedule that saves execution cycles as well as energy. In this work, our objective is to predict the effect of coscheduling of multiple processes in a simple and accurate way that enables fast and precise design process. The details of our proposed technique are presented in Section 4.

Table 1. Embedded Benchmarks used in this study.

| benchmark	Description
Blowfish	Symmetric block cipher with multiple lengths key.
Rijndael	A Symmetric block cipher encryption/decryption algorithm that is chosen by the National Institute of Standards as Advanced Encryption Standard (AES).
SHA	Secure Hash Algorithm (SHA) that produces a 160-bit message digest.
GSM	Global Standard for Mobile communication.
ADPCM	Adaptive Differential Pulse Code Modulation (ADPCM).
MAD	MPEG Audio Decoder. It supports MPEG-1, MPEG-2, and MPEG-2.5.

3 Simulation Environment

The simulation environment is based on MPARM [12] environment. MPARM models a Multiprocessor System-on-Chip (MPSoC). The processor models ARM 7 processors based on a software implementation called SWARM [13]. The system is interconnected using an AMBA [14] communication standard (architecture for high performance embedded systems). Multiple bus arbitration schemes are implemented by the MPARM. We choose a simple round-robin scheme. The simulated system has data cache (4KB size, 4-way set-associative) and instruction cache (2KB size, 2-way set-associative). Both caches are with writeback policy. The clock frequencies of all CPUs are the same, *i.e.*, homogeneous processor cores. Uncontended memory latency is 40 cycles.

Benchmarks used in this study are taken from security, telecommunication, and consumer categories of MiBench suite of embedded benchmarks [11]. Except for SHA and MAD, all benchmarks have a decode functionality in addition to the encode functionality. In this study, we run these two functionalities in pairs. For MAD and SHA, we run similar copies of the application. These pairs of applications can normally run concurrently in a multiprocessor embedded system.

Rijndael and MAD represent memory intensive applications with large percentage of cache misses, while PCM is less memory intensive application with small percentage of cache misses. The average memory access time is affected by the miss penalty that increases with the contention on shared bus. These applications are sequential applications that are run in parallel with non-shared memory spaces.

4 Finding Optimal Bus Coschedule

This section presents our technique to search for an optimal coschedule for two processes running on system with a shared bus. The approach is generalized
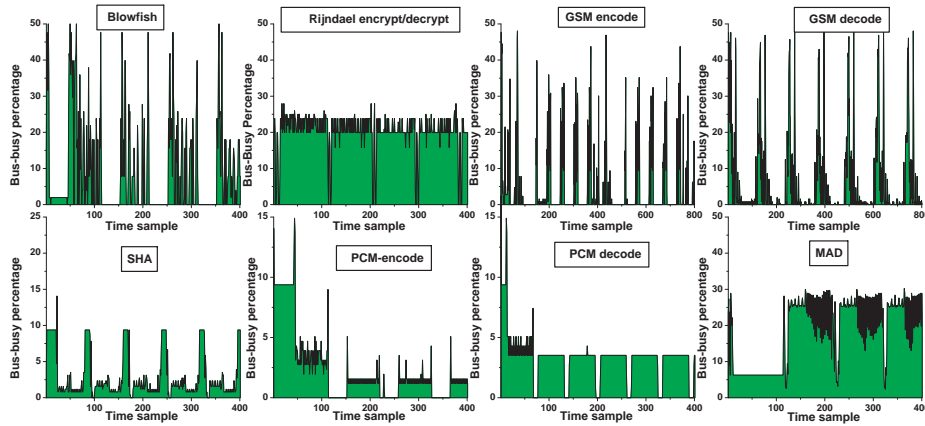
Fig. 3. Percentage of bus-busy for applications running standalone on the system bus.

to more than two processes in Section 5. We target finding an optimal static coschedule between two running processes that exhibit periodicity. Many embedded applications exhibit periodicity in their execution and consequently periodicity in the traffic sent to the memory system. This periodic behavior usually appears after an initialization period and may be trailed by a termination period. We are more interested in the periodic part of the application because it dominates the execution time of the application.

We start the search for optimal coschedule by running each process individually on the system. We record the bus-busy percentages over fixed interval of cycles. These recordings constitute a time-series of measurements b_k, $k = 1, \ldots, n$, where b_k is the bus-busy percentage at time interval k.

Figure 3 shows the percentages of the bus-busy cycles for our benchmarks. The time samples are 0.5K cycles for Blowfish and Rijndael; 2K cycles for GSM and PCM; and 20K cycles for SHA and MAD. These choices for sampling sizes are empirically chosen to compromise between having enough bus traffic details and limiting the number of scheduling decisions (bus traffic periodicity) to facilitate verifications. Decreasing the sampling interval increases the number of samples per application periodicity. While the proposed scheme has no difficulty in predicting performance with any number of samples per periodicity; it will be very difficult to simulate all configurations to verify the correlation between the performance predicted by our model and the outcomes of the simulations.

As shown in Figure 3, Blowfish and GSM bus traffics have burstiness in accessing the memory. Execution is divided into phases of large bus traffic followed by almost idle phases. Applications such as Rijndael, on the other hand, have slowly varying bus traffic. The encode and the decode functionalities produce different traffic profile for GSM and PCM. For Rijndael and Blowfish, the
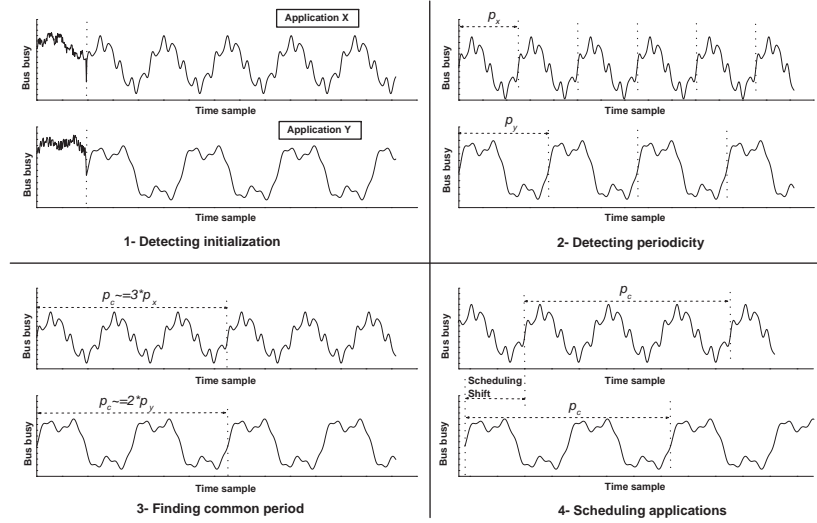
Fig. 4. Four steps in scheduling processes in MPSoC.

bus traffic for encrypt and decrypt follows the same profile. All these applications show periodic behavior in dealing with the system bus. To find an optimal bus coschedule, we propose the following steps:

1. Isolation of the initialization part of the application from the periodic part for each application, Section 4.1.
2. Identification of the periodicity of the bus traffic for each application individually, Section 4.2.
3. Creation of common coscheduling period for all applications designated for coexistence on the system, Section 4.3.
4. Analysis of the effect of different coscheduling overlaps/shifts, for the common coscheduling period, on the execution cycles, Section 4.4.

Figure 4 summarizes the four steps proposed to find optimal scheduling decision for a pair of processes. With the outlined technique, using simulation to exhaustively search for optimal coschedule is not needed. Enforcement of coscheduling decision to guarantee repetitiveness using *scheduling barriers* is introduced in Section 4.5. We asses the goodness of our technique in the prediction of optimal bus coschedule in Section 4.6.

4.1 Identifying Initialization Period

To identify the bus-traffic initialization phase of each process, we start by forming an *initialization approximate vector* (IAV). The IAV vector is formed by taking an initial subset of the vector b_k. We arbitrary choose a divisor d for the number of samples n. The IAV is chosen as $b_k, k = 1, \ldots, n/d$. Then, we compute the
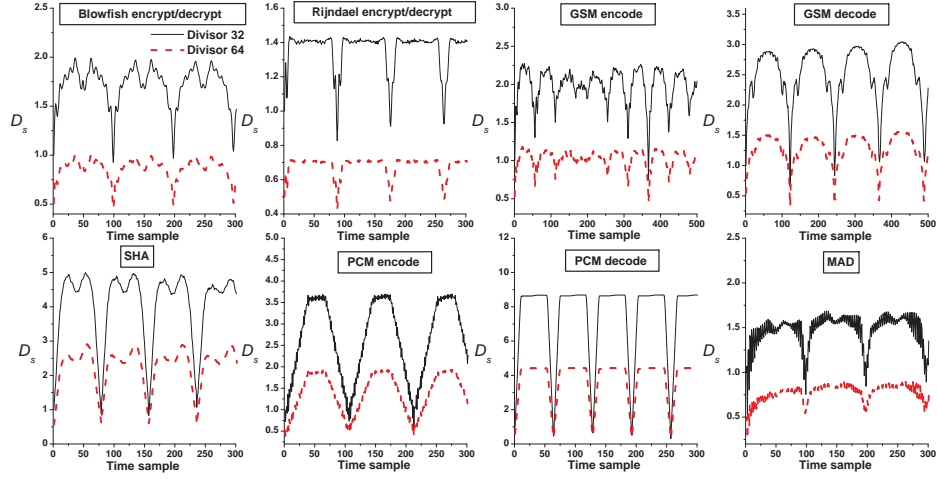
Fig. 5. Initialization phase detection based on difference graph of bus traffic.

difference between the IAV and shifted versions of the original time series, as $\mathfrak{D}_s = \sum_{j=1}^{n/d} |b_{j+s} - b_j|, s = 1, \ldots, 2n/d$. We compute an approximation of the second order difference as $\Delta\mathfrak{D}_s^2 = (\mathfrak{D}_{s+1} - 2\mathfrak{D}_s + \mathfrak{D}_{s-1}), s = 2, \ldots, 2n/d - 1$.

The peaks of the second degree difference $(\Delta\mathfrak{D}_s^2)$ occur at the possible end of initialization points. It is notable that multiple local maxima may arise due do the inclusion of part of the periodic behavior within the IAV. We used multiple divisors d to get confidence of the result. The outcome of initialization identification is independent of d as long as the initialization is a subset of IAV.

This approach is analogous to that used in finding application initialization based on basic block difference graph [15]. After identifying the m initial intervals, we form a new time series g_k with the initialization phase stripped; such that $g_k = b_{k+m}, k = 1, \ldots, r$ where $r = n - m$.

Figure 5 shows $\mathfrak{D}_s$ for the GSM pair of applications and Rijndael pair of applications. We show a small fraction of the computation curve for clarity. We used large value for d, 32 and 64, because the initialization part is a very small part of the execution time. It is notable that the maximum $\Delta\mathfrak{D}_s^2$ occurs at sharp local minima of the graphs. We choose the end of initialization interval at any point after the first maximum of $\Delta\mathfrak{D}_s^2$. Any point that follows the initialization can be considered a start for the periodic behavior. The end of the initialization interval can be taken as the first scheduling point for the application; that is why it is not advisable to excessively delay the choice for the end of initialization. For the following analysis, though, we need to make sure that enough cache warming has occurred before identifying a period representative to all other execution periods. In summary, we use early initialization point for scheduling

synchronization, to be introduced later in Section 4.5, while for the sake of analysis we consider a late initialization point.

4.2 Periodicity Detection

Detection of periodicity in experimental data has been studied by many researchers [16,17,15]. Autocorrelation (self-correlation) R is one of the mathematical formulas used to detect periodicity in a time series. The autocorrelation R is computed based on the autocovariance C_d, where d represent the time lag between the time series and its shifted version. The computation proceeds as follows:

Let $\bar{g}$ be the average of the time series g_k, then $C_d = \frac{1}{r} \sum_{k=d}^{r} (g_k - \bar{g}) \left(g_{(k-d)} - \bar{g} \right)$ and $R_d = \frac{C_d}{C_0}$. In this work, we adopted a methodology based on special form of autocorrelation called the *folded autocorrelation*. First, we define *folded covariance* as $FC_d = \frac{1}{r} \sum_{k=1}^{r} (g_k - \bar{g}) \left(g_{(k+d)} - \bar{g} \right)$. We assume $g_k = g_{k-r}$ for all $k > r$. The *folded autocorrelation* is then defined as $FR_d = \frac{FC_d}{FC_0}$.

Folded autocorrelation assumes virtual periodicity, thus simplify identifying periodicity. Figure 6 shows folded autocorrelation of the bus traffic after striping the initialization period. The periods between peaks of autocorrelation are candidates for defining periodicity. Although, the analysis introduced in Section 4.1 shows early prediction of possible periodicity, it does not precisely identify periodicity partly because of the inclusion of initialization period. The first few periods are usually affected by the cold start of the cache. Stripping the initialization from the bus traffic is needed to provide accurate estimate of the periodicity. Folded autocorrelation gives an accurate estimate for the periodicity. Precise identification of periodicity is needed to guarantee no drift in scheduling decision.

Except for GSM encode, periodicity can be detected easily both by inspection and mathematically. GSM encode has a large period that comprises five smaller periods with some similarity. We have chosen the larger period because this choice makes all periods almost similar for GSM encode.

4.3 Finding Common Periodicity

Based on the analysis introduced in the previous sections, we find the standalone periodicity for each process that needs coscheduling on the system with a shared bus. A process X is defined by the tuple $[x_k, i_x, p_x]$ where x_k is the percentage of bus busy cycles at interval k (ranging from 1 to p_x) after skipping i_x initial
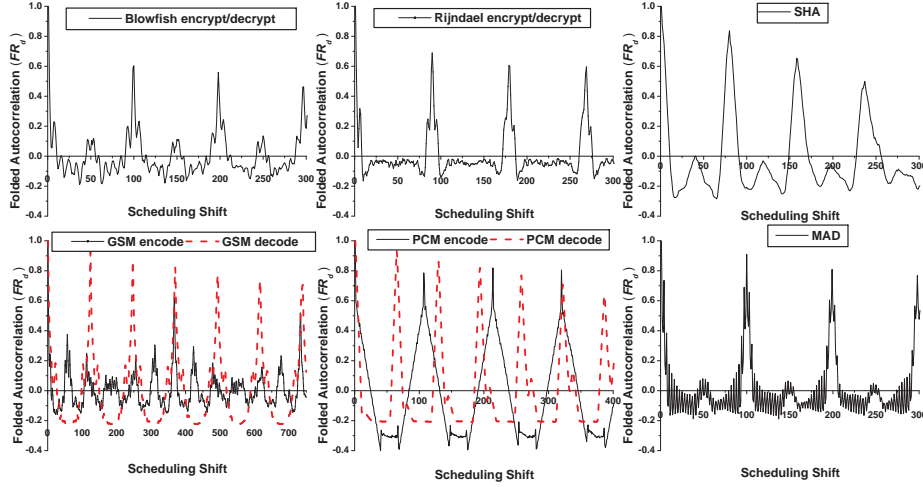
Fig. 6. Folded autocorrelation for periodicity detection.

intervals. Similarly, we define the tuple $[y_k, i_y, p_y]$ for process Y. The periodicity can vary from one process to another.

Finding optimal coschedule of bus traffic requires analyzing a common period that repeats for all processes sharing the bus. Common period makes the scheduling decision repeatable for all processes. A common period is composed of multiple basic periods of the coscheduled applications. Multiple criteria can be used in finding common period, as follows:

– For same realtime requirements for all processes: The shorter process is either appended with inactivity period to make all processes with the same period, or stretched assuming that it will run on a slower processor (heterogeneous system) such that all processes have similar periods.
– For different realtime requirements for considered processes: We need to define a common period p_c using the least common multiple of the two period count p_x, p_y. To avoid having a common period, p_c, that is as large as $p_x \cdot p_y$, it is sufficient to have p_c such that $(p_c \bmod p_x)/p_x < tol$ and $(p_c \bmod p_y)/p_y < tol$, where tol can be arbitrarily chosen, for instance less than 0.05. Increasing the common period (p_c) for coscheduling may reduce the effectiveness of the scheduling mechanism that is described in Section 4.5.

To compute p_c based on a certain tol, we start with an initial value for p_c of $p_x \cdot p_y$; we repeat decreasing p_c as long as the condition $(p_c \bmod p_i)/p_i < tol$ is satisfied for all processes. The minimum value for p_c while satisfying the condition is considered as a common period for coscheduling.

For Blowfish and Rijndael pairs of applications, we used the same realtime constraint for encrypt and decrypt. Both encrypt and decrypt have the same periodicity, which facilitates choosing a common period ($p_c = p_x = p_y$). Similarly,

we used the same periodicity for SHA and MAD because two identical copies are run for both applications.

For GSM pair of applications, GSM encode periodicity is almost three times the periodicity for GSM decode, assuming a $tol = 0.01$. While for most mobile computing applications the realtime constraint is the same for GSM encode and decode, we assumed an application where decode is needed for more frames than for encode, for instance in conference calls. Using $tol = 0.01$ for PCM pair, we find a common periodicity that coincides with three basic periods of PCM encode and with five basic periods of PCM decode.

In future work, we can consider a system with heterogeneous processor core to handle the same realtime constraint for different computational requirement.

4.4   Predicting Optimal Overlap of Coscheduled Processes

In this section, we aim at finding an overlap between two periodic processes such that the number of cycles needed to finish both processes is minimal. Simulation of all possible overlaps between coscheduled processes is extremely time consuming process, especially with large common periodicity of coscheduled processes. The computational requirement for simulation increases if we need to repeat the coscheduling search for different hardware configuration.

We like to narrow the search space for optimal coschedule based on the information we get from running each process as a standalone process. Formally, we need to find a scheduling shift l between the processes to be coscheduled given the bus-busy percentages x_k, y_k for process X and Y; respectively, where $k = 1, \ldots, p_c$. To achieve this objective, we propose the use of one of the following two metrics:

1. Find the minimum shift-variance of the sum of x_k and y_k shifted by $l = 1, \ldots, p_c - 1$. let $z_k^l = x_k + y_{(k+l) \bmod p_c}$

$$Var(l) = \frac{1}{p_c} \sum_{k=1}^{p_c} (z_k^l - \bar{z})^2 \tag{1}$$

where $\bar{z} = \frac{1}{p_c} \sum_{k=1}^{p_c} z_k^l = \frac{1}{p_c} \sum_{k=1}^{p_c} (x_k + y_k)$. Note that $\bar{z}$ is the same for all overlaps, which leads to the simple form on the right hand side of the equation for $\bar{z}$.

2. Find the minimum convolution of x_k, y_k for all scheduling shifts l.

$$Conv(l) = \frac{1}{p_c} \sum_{k=1}^{p_c} x_k \cdot y_{(k+l) \bmod p_c} \tag{2}$$

These two metrics give profiles of the effect of overlapping the bus traffic of the two processes. While these profiles help in knowing approximate area where minimum negative impact of overlap occurs, they do not provide a quantitative measure of the effect that optimal coschedule may introduce.

These measures will be highly accurate if the effect of overlapping a bus traffic from one process with the traffic from the other process is self-contained. The effect of overlap of two points is hopefully not biased toward extending the execution time for one process over the other. This requirement necessitates using fair arbitration policy on the bus. Coscheduling on bus with prioritization scheme may be less fruitful. A necessary condition is that the timing does not accidentally favor one process over the others, where one process always acquires the bus ahead of the other processes. Round-robin with preemption is one of those fair schemes but is unfortunately difficult to implement on multiprocessor bus. Simple round-robin, used in this study, provides a relatively fair arbitration mechanism.

Another inherent assumption is that criticality of cache misses are mostly the same for all cache misses, thus delaying a bus transactions impact performance the same. This is mostly true for simple cores with blocking cache misses that is modeled in this study. For systems with more complex cores, further investigation may be needed.

4.5   Coscheduling Enforcement Using Barriers

We propose to use barriers to define the overlap between coscheduled processes and to maintain this coschedule. We cannot use timing information collected by the standalone runs as a basis for scheduling synchronization. These timings are stretched due to the increase in memory latency because of the bus contention.

A robust technique would be to identify barrier locations on source code and to add barrier calls where necessary. Inserting software barrier has difficulties: First it requires changing the source code for each scheduling decision. Second, it cannot be inserted easily anywhere in the code; only specific locations are suitable for barrier insertion.

We adopted a simple approach based on the number of graduated instructions. The initialization and periodicity of an application are translated into instruction counts. The scheduling barriers are applied based on the number of graduated instructions. This simple mechanism can be applied easily in embedded environment where simple OS is used and program execution is repeatable and deterministic.

In this study, we considered hardware barrier [18,19] for synchronization, which is associated with little execution overheads. Hardware barrier can be implemented using a simple wired-and line(s). Triggering synchronization is
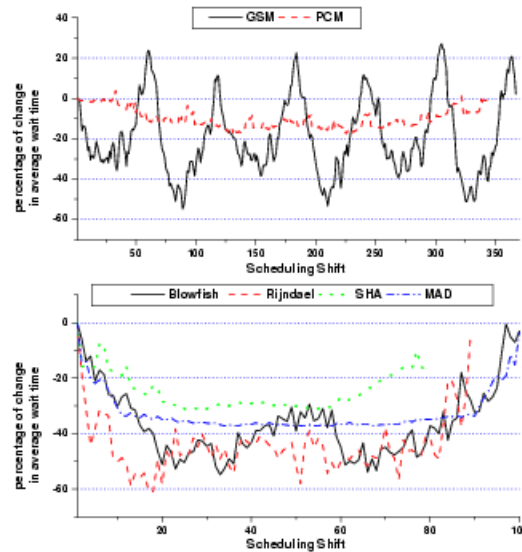
Fig. 7. Effect of coscheduling on bus wait time.

system dependent. A possible implementation that is explored in this study involves additional registers to hold the initial synchronization and the periodicity in terms of graduated instructions counts. These registers are part of the process context. An additional register is needed to hold the graduated instructions count. This register is initialized with the synchronization point. It is decremented each time an instruction is graduated. When the register value reach zero, the barrier synchronization is acquired. The barrier synchronization is released after all processes reach the barrier (the wired-and barrier line is asserted by all processors). During release, the register is reset to the periodicity instruction count.

Scheduling barriers do not represent any data dependency. They are not required for correct execution of coscheduled programs. They are used to ensure that no drift in scheduling occurs after executing many periods, and thus guaranteeing the persistence of the scheduling decision. With appropriate scheduling, this does not only maintain less execution cycles but also reduce the variability of execution because the memory traffic overlap will continuously repeat.

Using barriers usually causes period of idleness for the processors finishing work earlier, i.e. arriving earlier to the barrier. The waiting period on the barrier is quantified in Section 4.6.
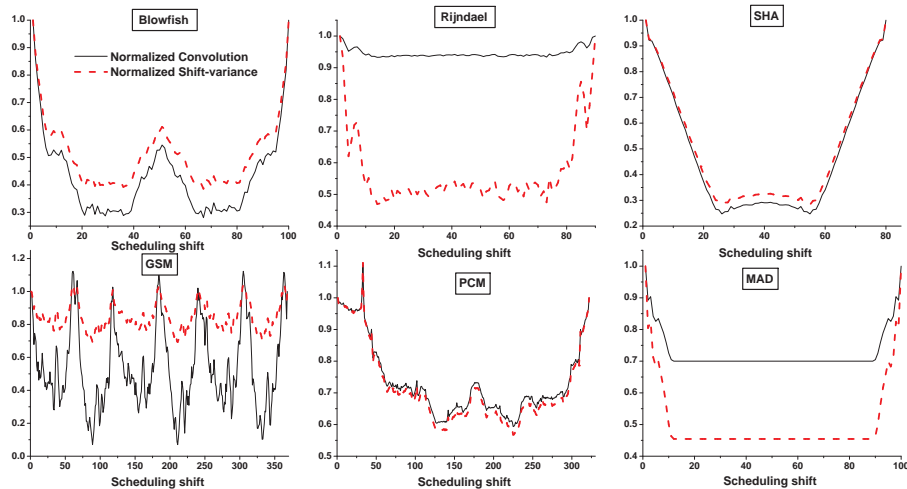
Fig. 8. Coscheduling effect prediction based on convolution and shift-variance.

4.6 Accuracy of Coscheduling Prediction for Two Processes Systems

In this section, we introduce the effect of coscheduling on memory access time and prediction accuracy of the effect of overlap. The memory access time affected by contention during the arbitration phase, the time to be granted the bus to start memory transfer, and the time to transfer data, especially that burst of data can be split into multiple non-contingent transfers.

Figure 7 shows the effect of scheduling shift on the average wait time on the bus for the six pairs of applications. The figure shows the percentage of change of wait time per memory transaction compared with the wait time for the initial scheduling decision (the *reference*). Every memory transaction faces the un-contended latency in addition to the additional wait time due to contention. The profiles for the bus-access wait-time follow the profiles predicted by equations 1 and equation 2. Both equations correlate to the wait on bus, as will be quantified later. The reference average wait-time per memory transaction is small (in the range of 2 to 8 cycles) for applications with low bus contentions; specifically for PCM, SHA and GSM benchmarks. For Blowfish, Rijndael, and MAD benchmarks, the reference average wait time per memory transaction ranges from 12 to 15 cycles. Percentage-wise the coscheduling decision impacts the wait time of some benchmarks, for instance GSM, more than others, for instance Rijndael, while scheduling decision impacts the latter's performance more. This is attributable to the higher miss rate and the higher wait time involved such that the overall performance is more sensitive to the memory system performance.

The execution cycles for different scheduling decisions (Figure 2) follow the profile of the bus performance introduced in Figure 8. To quantify the prediction

accuracy of our proposed scheme, we use the correlation coefficient; defined as follows: let $\bar{x}$ be the average value of a random variable X. The variance is defined as $E\left\{(X-\bar{x})^2\right\} = \sigma_{XX} \equiv \frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^2$. We also define $\sigma_X \equiv \sqrt{\sigma_{XX}}$. A good measure of dependence between the two random variables X and Y is the correlation coefficient [20], defined as

$$\rho_{XY} \equiv \frac{E\left\{(X-\bar{x})(Y-\bar{y})\right\}}{\sigma_X \cdot \sigma_Y} = \frac{\sigma_{XY}}{\sigma_X \cdot \sigma_Y} \tag{3}$$

The numerator of the right hand side is called the covariance σ_{XY} of X and Y. If X and Y are linearly dependent, then $|\rho_{XY}| = 1$. If $|\rho_{XY}| = 0$, then observing the value X has no value in estimating Y.

We use the correlation coefficient to study the relation between equation 1, equation 2, bus access wait-time, and execution cycles. Table 2 shows the correlation coefficient between these measurements. Both shift-variance (defined by equation 1) and convolution (defined by equation 2) are very strongly related for two processors system. One can replace the other, and preference is given to computational simplicity which favors convolution.

The correlation between the bus waits and the execution cycles ranges between a lowest of 0.72 for PCM and highest of 0.97 for GSM. These values can be classified as high to very high correlation according to Williams [21]. The bus performance does not perfectly correlate to the execution cycles (correlation of 1) because the effects of cache misses on the performance are not similar. Some bus transactions are more critical to performance than others, while all transactions contribute to the bus contention similarly. Additionally, the bus arbitration policy is not perfectly fair.

The bus wait is more correlated to convolution (and to shift-variance), compared with correlation to execution cycles, because we used the bus traffic only in the convolution computation. These correlations, between bus wait and convolution, range from a lowest of 0.71 for Rijndael to a highest of 0.93 for GSM. The total execution cycles depends on the interaction with other components on the system. It is apparent that correlation coefficients between execution cycles and convolution are lowered if the correlation between bus wait and execution cycles is low, or if the correlation between bus wait and shift variance is low.

These high correlation coefficients show that we can predict the effect of scheduling multiple processes sharing a bus. This prediction helps in identifying the best scheduling region. The exact performance difference, due to the scheduling, can be obtained through simulation of only few points of interest in the regions identified by the proposed scheme.

Simulating the system without scheduling barriers, we found a drift in the execution overlap that leads to performance change from one scheduling period

Table 2. Correlation coefficient (ρ_{XY}) between bus wait time(Wait), execution cycles (Cycles), convolution(Conv), and shift-variance (SVar).

| Bench	Conv/SVar	Cycles/SVar	Wait/SVar	Wait/Cycles
Blowfish	$\cong 1.000$	0.7675	0.8043	0.8610
Rijndael	0.9974	0.7238	0.7085	0.7956
SHA	$\cong 1.000$	0.7087	0.8707	0.8827
GSM	$\cong 1.000$	0.8580	0.9266	0.9764
PCM	$\cong 1.000$	0.5537	0.8538	0.7243
MAD	$\cong 1.000$	0.8610	0.8858	0.9380

to the other. As discussed earlier, we propose using scheduling barrier to circumvent this problem. We show the effect on performance of scheduling barrier in Figure 9, which shows the percentage of barrier wait for different scheduling decision. The barrier overhead is reported as average and the variation in the wait time is reported as the 99th percentile around the average. Using scheduling barriers incurs a small overhead for all applications studied. The barrier synchronization time relative to the total execution time does not exceed 1.3% for Rijndael and Blowfish. We noticed that the variation of barrier wait time is largest for Rijndael and Blowfish, although almost two identical processes are overlapping. This shows that the variation is mostly caused by the changes on overlap of bus transactions and not the difference in the amount of work, which is adjusted by the choice of common periodicity.

4.7 Proposed Scheme Applicability

Dynamic Voltage Scaling (DVS) and the proposed technique tackle the same problem of adjusting driving frequency/voltage for the sake of reducing the power of applications with varying execution time. For hard realtime applications, DVS increase clock/voltage of the system to meet worst case execution time and then tries to reclaim slack time by reducing voltage/frequency if it arises during execution [22,23].

The advantage of DVS is that this technique can be applied to all kind of tasks, but faces the following challenges: a) estimating the worst case execution time, needed for hard realtime applications, is not always feasible for multiprocessor machines; b) changing voltage/frequency usually involves complexity in design and delayed response that reduces the amount of saving in power. The dynamic adjustment may overclock frequency thus wasting power, or under-clock the system frequency thus not meeting real time deadlines.

The proposed technique in this paper is applicable for special class of applications with repeating pattern of accessing the memory. For these applications, a static schedule is selected that minimize the execution cycles and thus the
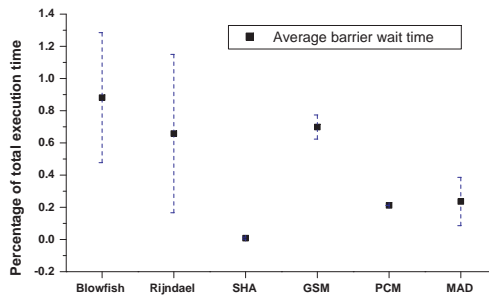
Fig. 9. Scheduling barrier wait-time percentage with the 99th percentile around the average.

frequency/voltage needed to drive the system. This schedule is maintained by low-overhead hardware barrier, thus reducing the variability in execution time.

The proposed scheme exploits repetitiveness of bus traffic. We show that the design space exploration can be surrogated by static analysis that alleviates the need of seemingly infeasible simulations. We studied different embedded applications from three different categories of benchmarks included in MiBench suite. The sources of bus traffic repetitiveness are as follows: a) algorithmic repetition of processing, for instance, applying the same processing to multiple frames of data; b) control flow of the application that is not dependent on the data processed.

Certainly these conditions do not apply for all embedded application. We found that some applications, from MiBench suite, may not benefit from the proposed scheme as summarized below: a) Applications that have amount and type of processing dependent on the data in the frame processed. The traffic generated on the bus is thus not cleanly periodic, although the processing of some frames can be similar. Lame application, GPL'd MP3 encoder, and JPEG application, a standard lossy compression standard, are example applications that exhibit this behavior. Fortunately, many of these applications are usually not hard realtime applications; b) Applications with constant bus traffic on the bus, for instance CRC32, 32-bit Cyclic Redundancy Check. Applications with constant behavior does not benefit from DVS, as well as our technique, because there is no variability in execution time; c) Applications with no periodic behavior, for instance FFT (Fast Fourier Transform). Some of these benchmarks are kernel codes that are called by higher level codes, and may be called in a repetitive way. In this case they can benefit from the proposed scheme.

We do not view these as limitations of applicability because the design of embedded systems does not involve generalized design rules. Special techniques are needed for different classes of systems to achieve certain design objectives, for instance ultra low-power systems. The proposed scheme can be thought of as Static Voltage Scaling (SVS) technique that suites special applications that

shows periodicity. For this class of applications, SVS does not require complex mechanism for detecting and changing the driving frequency and voltage. Additionally, SVS reduce the variability in execution time by forcing repetitive overlap of contending processes through barrier synchronization.

5 Multidimensional Coscheduling

Applying the proposed coscheduling techniques gain importance if the number of scheduled processes increases because the search space for scheduling increases exponentially. For instance, if we would like to explore one hundred scheduling decision for a certain application, then scheduling two processes of this application will require one hundred simulation runs, while scheduling four processes will require 10^6 simulation runs. For the applications considered in this study, GSM would have 49,836,032 scheduling decisions for four processes. Certainly, exploring such design spaces is not feasible through simulations; and a tool to predict points of interest in the design space is critically needed. For multiprocessor system, estimating WCET is very challenging because of the numerous contention scenarios that a process may encounter. Conventionally, a simple approach involves overestimating the clock frequency to guarantee probabilistic meet of deadlines.

In this section, we introduce the following generalization of the formulation introduced earlier in Section 4.4 to predict the effect of coscheduling, as follows:

Let the periodic time series for bus-busy percentages for the m applications that need to be coscheduled in the system $x_k^1, x_k^2, \cdots, x_k^m$; The common periodicity p_c is then computed such that $(p_c \bmod p_{\mathbf{x}^i})/p_{\mathbf{x}^i} < tol$.

Extending the definition for the shift-variance will be as follows: We define scheduling-shift vector as $\mathbf{L} : (l_1, l_2, ..., l_{m-1})$, where $l_i = 1, ..., p_c - 1$. $z_k^{\mathbf{L}} \triangleq x_k^1 + \sum_{i=2}^m x_{(k+l_{i-1}) \bmod p_c}^i$

$$Var(\mathbf{L}) = \frac{1}{p_c} \sum_{k=1}^{p_c} (z_k^{\mathbf{L}} - \bar{z})^2 \tag{4}$$

where $\bar{z} = \frac{1}{p_c} \sum_{k=1}^{p_c} \sum_{i=1}^m x_k^i$. The convolution definition can be formulated as follows:

$$Conv(\mathbf{L}) = \frac{1}{p_c} \sum_{k=1}^{p_c} x_k^1 \prod_{i=2}^m x_{(k+l_{i-1}) \bmod p_c}^i \tag{5}$$

Traditionally, shared bus systems are limited to few processors on the system bus. We limited system exploration to a system of four processors. Unfortunately, we cannot perform full verification of the prediction of coscheduling

Table 3. Results summary for four processors system; Maximum difference of execution cycles (Cycles), and correlation with Convolution (Conv) and shift-variance (SVar).

benchmark	Conv/SVar	Cycles/SVar	Cycles/Conv	%Cycles Diff.
Blowfish	0.9001	0.6900	0.6334	9.35
Rijndael	0.9675	0.4705	0.3875	23.84
SHA	0.9049	0.7074	0.4863	5.02
GSM	0.6427	0.8241	0.5015	13.74
PCM	0.9070	0.6592	0.4696	4.17
MAD	0.8691	0.9002	0.7368	20.96

effect on performance, because we cannot simulate all the points of the scheduling space. Instead, we simulated three hundreds randomly selected scheduling points of the design space for each pair of applications. We also conducted simulation of the best and worst scheduling points that were predicted by the multidimensional shift-variance defined by equation 4.

To evaluate the correlation between the prediction function and the simulated point; the multidimensional space of scheduling is projected into a single dimensional space, and then the formulation defined by equation 3 is used.

Table 3 summarizes the results for the same set of applications studied in Section 4.6. We explored systems of four processes. We doubled the processes by repeating the pairs introduced in Section 4.6. Table 3 shows percentage of change in the execution cycles. The percentage of change increases with increasing of the number of processes contending on the bus. The main reason is that the difference between coincidence of high traffic burst of four processes and the optimal distribution of bus traffic gets larger. Certainly, this trend happens because of the ability of bus to absorb the bandwidth required by all running processes. If the applications bandwidth demands exceeded the bus ability, we expect that all processes will be slowed down and the difference in execution cycles will become smaller. The execution cycles difference is computed for the randomly selected scheduling points that were simulated, including points predicted by the shift variance as candidate for minimum and maximum cycles. The difference reaches a peak of 24% for Rijndael. Even if these differences in cycles are not proved global minimum and global maximum, they show the impact of scheduling on performance, and the need to enforce a scheduling to avoid missing deadlines for realtime systems.

Table 3 shows that the correlation between the convolution and shift variance is not perfect for multidimensional scheduling, as we have seen earlier for one-dimensional scheduling. The shift variance gives better correlation with the performance observed through simulations. This behavior is observed for all ap-

plications because the shift variance predicts performance based on the effect of the sum of all traffic instead of multiplying them.

Another observation is that the correlation of cycles with shift-variance decreased with four processors systems compared with two processors system because the fairness of the bus gets stressed. Although there is a need for further improvements in this direction, simulations can not be seen as a feasible alternative.

6   Related and Future Works

Recently numerous research proposals targeted optimizing communication architecture for efficient use of energy [24,25]. These proposals adapt communication architecture to the application need or more specifically to the traffic generated by the application.

Adapting the system frequency/voltage to the application need is intensively studied for uniprocessor and multiprocessor systems. For uniprocessor machines, different proposals [6,26] address the problem of adapting frequency-voltage to meet realtime constraint and optimize for the energy consumption.

For multiprocessor, Yang et al. [27] propose dividing power-aware scheduling of processes into two phases: The first phase, done offline, summarizes the possible schedule decisions. During runtime, the second phase, the actual scheduling decision is taken based on the predetermined scheduling options. Static Power Management is utilized by Gruian [1] to adjust the best supply voltage/frequency for each processor. Zhu et al. [8] adjust the voltage/frequency level of the system to reclaim the slack time [28] in executing tasks by reducing the frequency of future tasks.

In contrast, our work addresses one of the main causes of variability in shared memory multiprocessor which is the contention for memory on a shared bus. Our proposed technique finds a scheduling decision that reduces the number of cycles needed to execute a task by reducing the effect of bus contention. We predict a good offline static schedule for the applications. We verified our technique for six pairs of applications. We explored extending the coschedule to any number of coscheduled applications. One constraint to this work is that it is applicable for applications that exhibit periodic bus behavior.

Future work includes exploring different bus arbitration policies and studying the best scheduling decision under these policies. Differentiating bus traffic into critical and less critical to performance need to be augmented to our formulation to reach a better estimate of performance and serve the needs for more complex processors systems. We believe that studying the effect of contention on shared resources should gain more attention from system designers.

7 Conclusions

Minimization of execution cycles for a given periodic task is essential for power saving. For a given realtime deadline, the frequency increases linearly with the execution cycles. The power consumption varies cubically with the frequency. One source of increasing the execution cycles is the contention on the shared bus in multiprocessor system. Through cycle simulation, we show that the execution cycles can vary by up to 13% for benchmarks running on two processors system, and 24% for four processors systems. These executions cycles varies the system power requirements greatly, because of the need to adjust the clock frequency to meet the system realtime constraints. This dynamic power saving can reach 57% for Rijndael executed on a quad-core MPSoC.

To alleviate the high cost of searching best scheduling overlap using simulation, we propose a scheme based on shift variance of the bus traffic profiles obtained while running these applications individually. We outlined the steps needed to strip application's initialization period and to detect the application periodicity based on bus traffic. Using shift-variance, we show that we can predict the effect of coscheduling under multiple scheduling overlaps. We also propose the use of scheduling barrier to maintain scheduling decision, which incurs very little overhead. We show that the prediction of scheduling effect using shift variance is highly correlated to the results obtained using simulations. We also extended our performance prediction mechanism to systems of larger number of processors with acceptable prediction accuracy.

References

1. Gruian, F.: System-Level Design Methods for Low-Energy Architectures Containing Variable Voltage Processors. The First Int'l Workshop on Power-Aware Computer Systems-Revised Papers (PACS'00) (2000) 1–12
2. Shin, Y., Choi, K., Sakurai, T.: Power Optimization of Real-time Embedded Systems on Variable Speed Processors. The 2000 IEEE/ACM Int'l Conf. on Computer-Aided Design (ICCAD'00) (2000) 365–368
3. Weiser, M., Welch, B., Demers, A., Shenker, S.: Scheduling for Reduced CPU Energy. The First USENIX Symp. on Operating Systems Design and Implementation (OSDI'94) (1994) 13–23
4. Nguyen, T.D., Vaswani, R., Zahorjan, J.: Using Runtime Measured Workload Characteristics in Parallel Processor Scheduling. Workshop on Job Scheduling Strategies for Parallel Processing (IPPS'96) (1996) 93–104
5. Snavely, A., Tullsen, D.M.: Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. The 9th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX) (2000) 234–244
6. Rotenberg, E.: Using Variable-MHz Microprocessors to Efficiently Handle Uncertainty in Real-time Systems. The 34th annual ACM/IEEE Int'l Symp. on Microarchitecture (MICRO 34) (2001) 28–39

7. Seth, K., Anantaraman, A., Mueller, F., Rotenberg, E.: FAST: Frequency-Aware Static Timing Analysis. The 24th IEEE Real-Time Systems Symp. (RTSS'03) (2003) 40–51

8. Zhu, D., Melhem, R., Childers, B.R.: Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multiprocessor Real-Time Systems. IEEE Trans. on Parallel and Distributed Systems **14**(7) (2003) 686–700

9. Butts, J.A., Sohi, G.S.: A Static Power Model for Architects. (2000) 191–201

10. Brandolese, C., Salice, F., Fornaciari, W., Sciuto, D.: Static Power Modeling of 32-bit Microprocessors. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems **21**(11) (2002) 1306–1316

11. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: A Free, Commercially Representative Embedded Benchmark Suite. The IEEE 4th Annual Workshop on Workload Characterization (2001)

12. Benini, L., Bertozzi, D., Bogliolo, A., Menichelli, F., Olivieri, M.: MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. Journal of VLSI Signal Processing **41** (2005) 169–182

13. Dales, M.: SWARM – Software ARM. (http://www.cl.cam.ac.uk/ mwd24/phd/swarm.html)

14. : ARM, AMBA Bus. (http://www.arm.com/products/solutions/AMBA_Spec.html)

15. Sherwood, T., Perelman, E., Calder, B.: Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. Int'l Conf. on Parallel Architectures and Compilation Techniques (2001)

16. Small, M., Judd, K.: Detecting Periodicity in Experimental Data Using Linear Modeling Techniques. Physical Review E **59**(2) (1999) 1379–1385

17. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically Characterizing Large Scale Program Behavior. The 10th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X) (2002) 45–57

18. Beckmann, C.J., Polychronopoulos, C.D.: Fast Barrier Synchronization Hardware. The 1990 ACM/IEEE conference on Supercomputing (Supercomputing'90) (1990) 180–189

19. Sivaram, R., Stunkel, C.B., Panda, D.K.: A Reliable Hardware Barrier Synchronization Scheme. The 11th Int'l Symp. on Parallel Processing (IPPS'97) (1997) 274–280

20. Shanmugan, K.S., Breipohl, A.M.: Random Signals: Detection, Estimation and Data Analysis. Wiley (1988)

21. Williams, F., Monge, P.: Reasoning with statistics: How to read quantitative research (5th ed.). London: Harcourt College Publishers (2001)

22. Lee, Y.H., Krishna, C.M.: Voltage-Clock Scaling for Low Energy Consumption in Fixed-Priority Real-Time Systems. Real-Time Systems **24**(3) (2003) 303–317

23. Krishna, C.M., Lee, Y.H.: Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time systems. IEEE Trans. on Computers **52**(12) (2003) 1586–1593

24. Nicoud, J.D., Skala, K.: REYSM, a High Performance, Low Power Multi-processor Bus. The 13th Int'l Symp. on Computer Architecture (1986) 169–174

25. Lahiri, K., Dey, S., Raghunathan, A.: Communication Architecture Based Power Management for Battery Efficient System Design. The 39th Conf. on Design Automation (DAC'02) (2002) 691–696

26. Anantaraman, A., Seth, K., Patil, K., Rotenberg, E., Mueller, F.: Virtual Simple Architecture (VISA): Exceeding the Complexity Limit in Safe Real-time Systems. The 30th annual int'l Symp. on Computer Architecture (ISCA '03) (2003) 350–361

27. Yang, P., Wong, C., Marchal, P., Catthoor, F., Desmet, D., Verkest, D., , Lauwereins, R.: Energy-Aware Runtime Scheduling for Embedded-Multiprocessor SOCs. IEEE Design and Test of Computers **18**(5) (2001) 46–58

28. Hua, S., Qu, G.: Power Minimization Techniques on Distributed Real-time Systems by Global and Local Slack Management. The 2005 Conf. on Asia South Pacific Design Automation (ASP-DAC'05) (2005) 830–835