Using Program Phases as Meta-Data for Runtime Energy Optimization

Cristiano Pereira
cpereira@cs.ucsd.edu

Rajesh Gupta
gupta@cs.ucsd.edu

Department of Computer Science and Engineering
University of California, San Diego
http://www.cs.ucsd.edu

Abstract

Power consumption is a major concern in embedded systems design due to the portability and battery driven operation of such systems. The runtime optimization of embedded software applications for system-level power / performance tradeoffs requires ability of the runtime system to probe system and application status and utilize procedures that make these tradeoffs effective. To ensure efficiency of decision making, it is important that such decisions are made with the least overhead to system power. One way to achieve this capability is through systematic definition, and update of meta data that can be probed by the runtime system and given as input to the dynamic power management algorithms. In this paper, we use the concept of application reflection, a technique in which a program represents its own structure and behavior through the use of meta-data. Its use enables the ability of the runtime system to look at the program representation and make power management related decisions. We present a profiling scheme to build a reflexive data structure in which a program represents its own execution behavior, and use this information at run time to guide operating system power management decisions. Our scheme is inspired on *Simpoint*, a tool for automatic program phase classification and simulation points selection. We use main memory bank shutdown as an example of how our technique can be used and we show that we can achieve energy/delay savings comparable to the best known hardware based technique. We believe that our approach can also be used for efficient energy management of other resources such as processor and system peripherals.

1 Introduction

Power consumption is a major concern on embedded systems design due to the increase in mobility, complexity, and the ever increasing demand for performance and small form factors. Furthermore, the battery-driven nature of such systems requires a careful tradeoff between performance and power in order to maximize their lifetime and still satisfy the performance requirements.

Optimizations can be performed at various levels of the system, from architecture to applications. These optimizations can be carried out statically, at design time, or dynamically at run time, where an efficient infra-structure to enable the exchanging of information between applications demand and power manager is needed. The runtime optimization of embedded software applications for system-level power / performance tradeoffs requires ability of the runtime system to probe system and application status and utilize procedures that make these tradeoffs effective. Also, to ensure efficiency of decision making, it is important that such decisions are made with the least overhead to system power.

In this paper, we use the concept of application reflection, a technique in which a program represents its structure and dynamic behavior through the use of a resource demand meta-data. Its use enables the ability of the runtime system to look at the program representation and make power management related decisions. We present a scheme to build a reflexive data structure by using profiling. In the scheme proposed, the program represents its own execution behavior, and use this information at run time to guide operating system power management decisions. Our program representation exposes dynamic resource demand variation over time, enabling the runtime system to probe such information (resource demand) and manage the resources to minimize power without significant performance penalties. We use the concept of program *phases of execution*, which will be defined properly later in the paper. For now, a *phase of execution* are intervals of a program which behave in a similar fashion and therefore have similar resource needs. In order to validate our approach, we use main memory shutdown as an example.

The contributions of this paper are listed as follows. 1) The use application reflection for runtime optimization of power consumption; 2) The suggestion of a scheme to attach application meta-data representing different resource needs throughout the dynamic execution of a program. 3) A scheme to identify which *phase of execution* a program is running and access the related resource demand meta data associated with it.

The paper is organized as follows. Section 2 presents the related work. Section 3 describes our application reflexive approach for power management. In Section 4 we describe the experiments realized to validate our scheme, followed by Section 5, where we show our results. Finally we point out future research directions and conclude in Sections 6 and 7.

2 Related Work

Reflection is a programming language concept which allows a program to analyze, reason and modify its representation. Reflection enables inspection, in which either the program or the runtime environment access the representation of the program and adapt accordingly in order to optimize some aspect. Reflection is used in different contexts. For instance, distributed middleware implementations use reflection so that the middleware can adapt to applications/devices behavior. Each application or device represents its behavior by means of profiles (such as resource demands, QoS requirements, etc...) and the middleware looks for changes in the profile to adapt its behavior [5]. We use a similar concept in this paper. Each application carries a representation of its dynamic behavior (the meta-data). The runtime system (either the operating system or a power manager) monitors the application behavior to find out the current execution characteristics. Based on this information, the power manager adapts its behavior to optimize the energy consumption of the application.

Among the program characteristics that a power manager can use, application resource demand is important to help deciding the power mode in which a given resource should operate. It is therefore desirable that an application is able to identify and represent its resource demand throughout the execution. From this perspective, the behavior of a program varies significantly. In addition, the demand for a resource is related to the part of the program being executed at a given time. Furthermore, parts of programs that execute similar code (which execute the same instructions with approximately the same frequency) have similar resource demands [14]. Thus, being able to identify which part of the program is being executed and how much resources it requires is important to optimize their use. Sherwood *et al* [14] have devised a technique in which the dynamic execution of a program is divided in *phases of execution* using basic block frequency vectors (vectors representing how many instructions per basic block were executed). A *phase of execution* is group of program intervals with similar basic block vectors. The frequency vectors contain the number of times and the number of instructions each basic block of the program was executed in an interval. Therefore similar basic block vectors mean that two intervals executed similar instructions. The goal of their work is to find phase representative samples in order to speed up program simulation. The authors developed a tool called *Simpoint* [4] which classifies a program in phases. In a follow up work [15], they devised a technique for finding and predicting *phases of execution* online, and suggested that the processor can be adapted for each *phase of execution* to maximize performance and in some cases minimize energy consumption. Using very similar techniques, Lau *et al* [11] showed that not only basic blocks frequency vectors can be used for phase classification, but also loops, procedures, regions of memory accessed and types of instructions frequency vectors. Building upon this concept, we propose a technique to attach meta-data to each *phase of execution*, detect which phase of the program is executing, and utilize the meta-data for energy optimizations.

Energy optimizations of a computer system can be carried out in different ways. A well known method is dynamic power management (DPM) of devices. In DPM, devices are switched to low power modes for saving power. However, in order to bring them to active mode, a performance and power cost is associated. The decision on when and which mode devices should be switched to is not a trivial problem. [2] and [10] present good surveys on the subject. Among the various devices that support DPM, main memory banks are important due to their contribution for the overall power consumption of a typical computer system. In this context, Delaluz *et al* [6] proposed compiler level techniques and also self monitoring hardware based schemes for main memory bank shutdown. In particular he developed an approach called History Based Predictor (HBP), which will be explained in more details in Section 4. In a later work [7], the same author proposed an operating system scheduler driven memory shutdown approach, where the operating system periodically checks which memory banks were used in a last interval of execution. In their work, an interval of execution is the period between two operating system tick interrupts. The banks which were not used are switched to the deepest low power state.

Also related to memory banks shutdown, Lebeck *et al* [12] proposed a power aware page allocation scheme, where they suggest a memory allocation scheme that tries to maximize the idle time of memory banks and hence maximize the time these banks are in low power modes.

Finally, Park *et al* [13] proposed a scheme in which each process is loaded into contiguous memory space that is divided in banks or memory modules. Whenever a process is loaded into memory, its process control block (PCB) holds the information saying which banks are allocated to the process. Whenever a process is preempted, the banks allocated to it have a chance to be switched to lower power modes. To figure out which states the banks should be switched to they keep a history of the process waiting time. Depending on the predicted waiting time, the memory banks are switched to the appropriate modes.

3 Using Reflection for Energy Optimization

Runtime optimization of embedded system applications for system-level power / performance tradeoffs requires ability of the runtime system to probe the dynamic execution environment, application and system status. As an example, it is desirable for the system to be able to inspect the demand of the application for certain resources in order to better optimize their use. It has been shown in the literature that the behavior of programs varies significantly over its execution [14]. Therefore, being able to exploit specific behaviors or *phases of execution* enables different types of optimization (ranging from hardware adaptation to better suit the program demand, to multi-threading scheduling, which uses phase information to find out when the behavior of a specific thread changes). The optimization targets can be either performance, energy or a tradeoff between the two.

We adopt the use of reflection, where applications carry meta data representing the dynamics of its phases of execution in order to optimize the run time energy consumption of the system. Accessing the meta data of the application, the runtime system (the operating system power manager) infers resource demand needs and make energy management decisions. We propose a scheme through which the run time system probes the application to find out the phase of the application being executed, the meta-data associated with it, and how it should be used to optimize the system energy consumption, the latter depending on which devices are the target of the optimization.

3.1 *Phases of execution* of a program

As noted before, programs execute different segments of code at different points in time. Looking at a dynamic trace of execution, programs have repetitive behavior in which the code executed and the frequency at which it is executed are the same. A phase of the program is defined as a group of execution intervals in which the execution of instructions is similar. An execution interval is a fixed number of dynamically executed instructions. Each phase needs different resources and executes different portion of the code. Sherwood *et al* [14] showed that intervals of execution which exercise similar portions of the applications code tend to have similar resource needs. They used basic block frequency vectors (BBV) to represent the code executed during certain intervals of execution, where two or more intervals, not necessarily contiguous but with similar BBVs, are considered part of the same *phase of execution* of the program. To classify BBVs as similar they use offline machine learning clustering techniques. Based on this principle, Lau *et al* [11] showed that not only BBVs are accurate in representing phase behavior, but also loop branches and procedure calls provide a good degree of accuracy for identifying phases of execution. Both works have as objective to identify simulation points which represent the program behavior. Instead of executing the whole program to evaluate the processor performance, only representative intervals from each phase are executed, speeding up simulation efficiency by orders of magnitude.

Our approach builds upon the principle of program *phases of execution*. It consists of two stages. An *offline* stage, where a program binary code is profiled in order to identify resource demands and phases of execution. From this, we build a reflexive data representation which stores resource needs of each phase, and also a signature which is used by the runtime system to match a program to a phase. We call it reflexive because it represents dynamic execution information which is inspected by the program itself or by the runtime system. The second stage is performed *online*. The runtime system monitors the program execution in order to identify the phase the program is running, and notify the power manager / operating system about the resource needs for the phase. The power manager makes the appropriate decisions to optimize the system power consumption.

Three important issues of the approach are: 1) which information is used as meta data; 2) how it is matched with actual program execution so that the right information is used; 3) how it is accessed. The approach we used is motivated by the approach described by Sherwood *et al* [15], in which the processor tracks and predicts program execution phase behavior using BBVs on the fly.

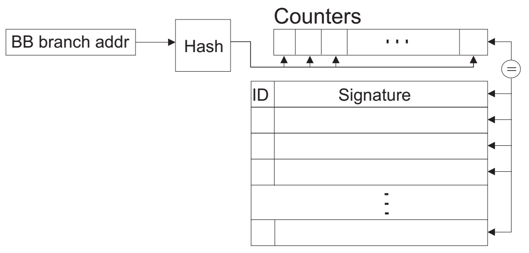


Figure 1: Phase tracker hardware for finding phases

The structure of the hardware to find and match phases used by Sherwood is illustrated in Figure 1. After the execution of a basic block, the basic block branch instruction address is hashed into a bucket of 32 counters. The counter is incremented by the number of instructions executed in the basic block. After an interval of execution, the vector of counters represents the signature of the interval, which is compared against a history of previously calculated signatures. If there is a match, the interval is classified as part of a phase already discovered previously. Otherwise the interval is considered a new phase of execution and is inserted in the history. A match is found if the Manhattan difference between two vectors a and b, represented by $Diff(a,b)$, is less than a threshold th. Hence a match is found if:

$$Diff(a,b) = (\sum_{i=1}^{32} \mid a_i - b_i \mid) \leq th$$

This is similar to the approach devised in their previous paper [14]. However it is executed at runtime and hence has to be simple enough to minimize overhead.

We use a similar approach to the one described, but we use it for offline phase classification as well as for online matching at run time. They key differences is that we do not execute the phase matching algorithm fully in hardware and we also do not use basic blocks to track phases, but branch loops. The reason is that basic blocks are too fine grain for the type of optimization we intend to perform. Therefore the overhead is greatly minimized if we keep track of a more course grain structure such as loop branches. Another difference is that the approach described in their paper is used for online phase matching whereas we use it for offline phase discovery and also for online phase matching. The details will be explained in the next subsection.

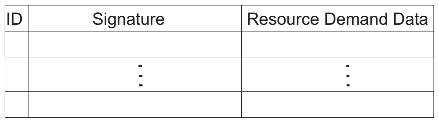
3.2 Using phases for meta-data representation and access

3.2.1 Offline stage

In the offline stage we use application profiling to classify the program execution in phases. Each phase is characterized by intervals with similar number of loop branch executions (all backward branches are considered loop branches). We run the program using an instruction set simulator and collect the number of times each loop branch is executed per interval. Each loop branch address is mapped into a position in a vector of counters and the appropriate counter is incremented each time the loop branch gets executed. After a fixed number of instructions (determined by the interval size) is executed, the loop branch vector (LBV) representing such interval is used to determine which phase the interval belongs to, in a similar manner as described by the hardware structure in Subsection 3.1. As in [15], we use vectors of 32 counters. We call the vectors of 32 counters compressed LBVs or CLBVs.

Intervals are classified as part of the same phase if their CLBVs are similar, which is determined by a match. Again, a match is found if the Manhattan distance is less than a threshold th. We show the tradeoffs of different thresholds in Section 5. The vectors are normalized before the comparison. The normalization is necessary for the subsequent online phase matching stage because the signatures calculated at run time are not complete and therefore only match with the full signatures if the normalized vectors are compared.

After the offline phase classification is finished, a set of phase signatures describe the loop branch frequencies for each phase of the program. These vectors are used at runtime to enable the program to identify the phase being executed. Along with the signatures, we also need to associate the resource demand data that is used for power management. This meta data will be used as guidance for the power manager in order to make its decisions. To be effective, the program has to detect as early as possible the phase in which it is executing. Another important requirement is that the phase detection along with operating system notification should have low overhead in terms of performance and energy consumption. Figure 2 represents the meta-data we associate with the program. This data can be added to the binary code using a tool such as ATOM [16].



ID	Signature	Resource Demand Data
	⋮	⋮

Figure 2: Meta data representation. Contains the phase ID, the phase signature and the resource demand information.

3.2.2 Online stage

During the online stage, the runtime system has to be able to identify which phase it is running at a given moment. For that we use a mix of hardware and software. We collect CLBVs dynamically using hardware. We assume that the hardware has a performance counter which counts the number of branch loops executed. Such performance counter can be found in modern processors such as Intel Xscale [9]. Usually they can be programmed to issue an interrupt on overflow. After a given number of loop branches executed, the processor calls an interrupt service routine, (ISR), which executes as software, to compare the CLBV dynamically computed with the CLBVs attached in the program as meta-data. The dynamic CLBV for the interval being executed is also computed in hardware. The hardware necessary to do so has to detect a loop branch, map it into one of the 32 vector positions and increment the position mapped. The vectors are not normalized at this point.

Algorithm 1 shows the ISR code to carry out the comparison. After the interrupt is issued, the interrupt service routine has to read the content of the vector stored in the processor memory, represented by the variable *part_sig*, normalize it (lines 1–7) and compare it with the vectors found in the application meta-data structure (lines 8–20), represented by the variable *phase[][]*. The matching ISR has time complexity of $O(n)$ where n is the number of phases. The overhead of the ISR will depend on how often it is issued, how many phases the program has and how early a phase can be detected. For comparison purposes, 32 subtractions and additions have to be carried out. Part of the overhead will be fixed regardless of how many phase. For instance, for every execution of the ISR, 32 divisions have to be executed for normalizing the CLBV.

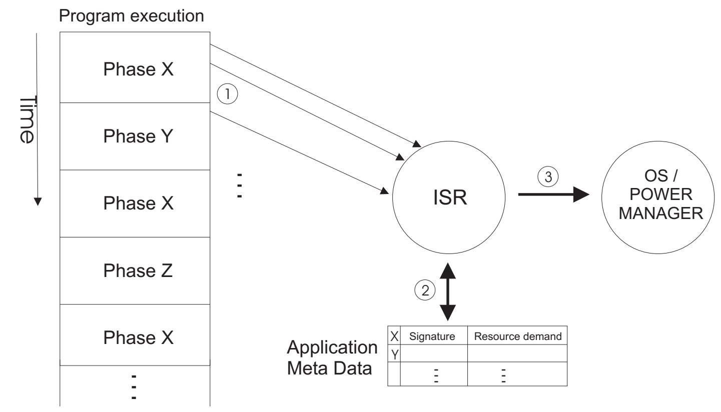


Figure 3: Phase detection diagram. 1) Every N loops the ISR is called; 2) The ISR tries to match the CLBV computed by the hardware with the ones stored along with the application code; 3) If a match happens the ISR transfer to the power manager the meta data associated with the phase.

Figure 3 shows our approach for power management using application reflection. In the figure, we assume that the application has been profiled and the resource demand data along with phase signatures (identified by the box "Application Meta Data") are stored at a pre-determined location in the application binary code. At every N loops a match is attempted. If it is found the ISR notifies the operating system power manager to use the new resource demand estimates. We note that the phase matching can also be executed in hardware as proposed by Sherwood in case of prohibitive overhead for the software implementation. In this case an interrupt is raised to notify the application that a phase was detected, then the application would notify the power management and pass it the resource demand meta-data.

We note again that we chose to use loop branches to minimize the overhead of phase detection. For detecting phases using BBVs we would have to keep track of the number of times each basic block gets executed at run time and try a phase match every N BB executions. Such operation would be executed more often than using LBVs. Nevertheless, Lau *et al* [11] showed that the homogeneity of phases classified by using LBVs is comparable to the homogeneity of phases classified by BBVs. A diagram showing the flow of our generic profiling and meta data gathering process is shown in Figure 4.

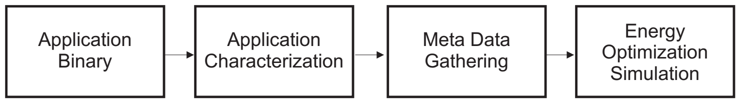


Figure 4: Generic Flow for Meta Data Construction

Algorithm 1 Match signature ISR

```
total = 0;
for $i$ = 1 to 32 do
    total += part_sig[i];
end for
for $i$ = 1 to 32 do
    norm_part_sig[i] = part_sig[i] / total;
end for
min_diff = 2.0;
for $i$ = 1 to numphases do
    diff = 0;
    for $j$ = 1 to 32 do
        partial_diff = abs(phase[i][j] - norm_part_sig[j]);
        diff += partial_diff;
    end for
    if (diff < threshold) and (diff < min_diff) then
        min_diff = diff, phase_id = i;
    end if
end for
if min_diff < 2.0 then
    return phase_id;
end if
```

Given the application binary code, the stages are as follows:

Application Characterization The application is profiled and the phases are identified. Here different thresholds for phase classification can be tried out.

Meta Data Gathering Based on the phase classification, the meta data of interest is collected.

Energy Optimization Simulation Given the meta-data, which includes the phase signatures and the attached power management information, the energy optimization scheme is simulated for different phase matching thresholds, phase intervals and matching attempt frequencies.

After the simulation is carried out, the best number of phases and frequency of phase match attempts (every how many loop branch we try to match a phase) is chosen for the application in question. To validate our approach we used main memory bank shutdown as an example.

3.3 Handling Multiple Data Sets

The description presented so far has no mention to how different data set influences the phase detection and therefore the technique. The influence of the input data on a phase of execution is to create different signatures and therefore different *phases of execution* with different characteristics and resource demands. We assume that in general program inputs can be classified in different categories which in turn yield different behavior. In order to handle different inputs, the applications have to be profiled with one input sample from each category. The interval signatures (CLBVs) generated by each run should then be classified in phases altogether so that each phase will be identified by a "global" signature among all different runs of the same program, one for each input category. In this way, different program behaviors along with different resource demands (which are generated by a different inputs) will still be present in the metadata and correcly identified. We have not realized experiments to analyze the influence of different data inputs on the phase classification but we intend to do so in future work.

4 Experiments

In order to show how our approach can be used, we chose main memory banks shutdown as an example. Typically, the main memory subsystem of a computer system is composed by memory chips. These memory chips are organized into banks in such a way that whenever a memory access is requested, the memory controller has to map the address requested to one of the memory banks and read/write the data from/to it. We assume that the memory controller does not reschedule requests coming from the processor. It is responsible for the translation of read and write requests of the CPU into the control signals of the DRAM. We also assume a linear translation of physical addresses into the banks. For the sake of simplicity we assume that the requests to memory are also non-overlapping. A summary for different organization of embedded systems memory controllers is presented by Gries and Romer [8]. We assume that the memory is divided in small modules and each memory can operate in different modes with different power consumption and costs associated, in conformity to [1]. In this way, unused modules can be switched into different modes of execution based on how long they are idle, hence reducing energy consumption. Figure 5 shows the representation of the memory model assumed. The processor sends commands to the memory controller to access data as well as to switch memory banks to the desired low power state.

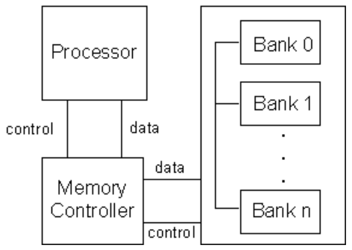


Figure 5: Memory model diagram

4.1 Memory banks shutdown

Memory banks can operate in the active mode, where the time to access the information is minimum, or in low-power modes, where there is a resynchronization time to bring the memory back to the active mode and then serve the request. The deeper the low-power mode, the lesser the power consumption of the memory bank and the higher the resynchronization penalty. In the offline stage, we profile the demand for each memory bank on different phases of an application. In the online stage, the runtime system finds out which phase is being executed and tell the operating system power manager the estimated demand for the main memory banks. The operating system then uses the resource demand information to make decisions of which state each memory bank should be changed to in order to optimize the energy / performance tradeoff.

The low power modes memory model used is the same as used by Delaluz *et al* [6], which is based on data sheets for RAMBUS [1]. Table 1 shows the low power modes along with the energy consumed per cycle and the resynchronization delays of each mode of a DRAM memory bank. We use normalized values when calculating energy dissipation in our experiments.

We assume that the system does not have virtual memory so we do not have to handle the effects of paging. During the execution of a program the demand for main memory changes over time. Being able to track the phases of high main memory demand and which banks are mostly used as early as possible is vital for efficient energy management unless specialized hardware is provided. If the latter is true, the task of shutting down memory banks can be delegated to the hardware. For shutting down memory banks, the operating system power manager has to be able to send a command to the memory banks requesting a transition to a low power mode. We assume that the operating system can interface with the memory controller in order to send such

Low Power Mode	Energy per cycle	Resynchronization delay (cycles)
Active	3.570nJ	0
Standby	0.830nJ	2
Napping	0.320nJ	30
Power-Down	0.005nJ	9,000

Table 1: Energy consumption per low power state and resynchronization delays

commands. In the next subsection we explain the experimentation process used for memory bank shutdown, as well as show the meta-data used.

4.2 Memory bank shutdown resource demand profiling

Given the application binary code, we used Simplescalar [3] for CLBV (i.e., intervals signatures) collection. Next we classify the application in phases using different thresholds th. Since each CLBV is normalized, after computing the difference between two vectors, the maximum difference is less or equal to 2. We use thresholds of 1.5, 1.0 and 0.5. If the threshold is set to 0.5, this means that as long as at least 75% of the CLBVs are the same, the two intervals will be classified as in the same phase. Due to loop branch address aliasing, it does not mean that 75% of the loops are the same.

During the execution of Simplescalar for CLBV collection, we also collect a trace of main memory accesses and store it in a separate file. After the application intervals are classified in phases, we build the resource demand meta-data that will be used to guide the power manager decisions. In the case of memory banks, we chose the average inter arrival time for memory access requests for each memory bank. We compute the average inter arrival time per bank per phase as the number of instructions in the interval divided by the average number of memory accesses to each of the banks added to the standard deviation. We add the standard deviation so that we have a conservative estimate when the variance is too high. The following formula shows the inter-arrival time estimate used for each memory bank:

$$IA_{pred} = \frac{IntervalSize \times CPI_{avg}}{(average + stddev) \times CpuMemClkRatio}$$

where $IntervalSize$ is the number of instruction in a interval, CPI_{avg} is the average number of cycles per instructions for the phase in which the particular interval was classified. The terms *average* and *stddev* are the average number of main memory accesses and standard deviation for the interval in question, and finally $CpuMemClkRatio$ is the ratio between the CPU clock period and the memory clock period.

The power management policy uses the resource demand per phase for guiding the shutdown of memory banks. The model we assumed for memory banks shutdown considers the energy cost of resynchronization to be a fraction F_a, where $0 \leq F_a \leq 1$, of the energy cost of being active per cycle. The resynchronization energy cost per cycle is denoted as E_r ($E_r = F_a \times 3.57$nJ). Depending on the number of main memory accesses per interval, a given bank will be switched to one of the low power states previously described.

Given the estimated number of memory accesses N for an interval of size I, the energy dissipation of operating on state i is given by:

$$(N \times E_r \times T_{r_i}) + (NumMemCycles \times E_i)$$

E_i is the energy per cycle for the low power state i, where $0 \leq i < M$ and M is the number of low power states. T_{r_i} is the resynchronization time for low power state i, and $NumMemCycles = (IntervalSize \times CPI)/CpuMemClkRatio$ the number of memory cycles for the given interval. E_0 and T_{r_0} denote the energy consumption and resynchronization delay for the active state.

Given a number of memory accesses N for a given memory bank b, we want to switch the bank to the power state which minimizes energy. We first consider the number of memory accesses N_{i+1}, which makes the energy consumption of staying in state i the same as staying in state $i+1$ where $E_i > E_{i+1}$. Such value is found in the following equation:

$$(N_{i+1} \times E_r \times T_{r_i}) + (NumMemCycles \times E_i) = (N_{i+1} \times E_r \times T_{r_{i+1}}) + (NumMemCycles \times E_{i+1})$$

From the equation above we find that:

$$N_{i+1} = NumMemCycles \times \frac{1}{E_r} \times \frac{E_{i+1} - E_i}{T_{r_i} - T_{r_{i+1}}}$$

Let the number of estimated memory accesses for a given interval be N_{pred}, and let $N_0 = \infty$. If $N_{i+1} \leq N_{pred} < N_i$ the memory bank in question is switched to the low power state i.

We can also calculate the inter arrival time values and use them to decide which low power state to switch to. The inter arrival time is given by:

$$IA_{i+1} = \frac{NumMemCycles}{N_{i+1}} = E_r \times \frac{T_{r_i} - T_{r_{i+1}}}{E_{i+1} - E_i}$$

Similarly, we set $IA_0 = 0$. Let $IAPred_b^{ID}$ be the estimated inter arrival request time for bank b in a given set of intervals classified as in phase ID. If $IA_i < IAPred_b^{ID} \leq IA_{i+1}$ the memory bank b is switched to the low power state i when executing an interval in phase ID. We note that this conditions to switch to a lower power state i are the same as in the Lower Envelope Algorithm (LEA) presented in [10].

Putting everything together, all intervals within a given phase will have a estimated memory access inter arrival time per bank denoted as $IAPred_b$. Whenever an interval belonging to a phase starts executing the banks are set to the appropriate low power states. Figure 6 represents the resource demand meta-data as a 2 dimensional array of application phases and memory banks estimated inter arrival time. The meta data represents the demand for each memory bank per phase.

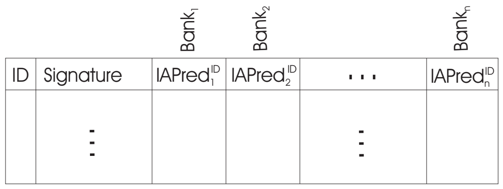


Figure 6: Memory bank meta-data associated with phase signatures

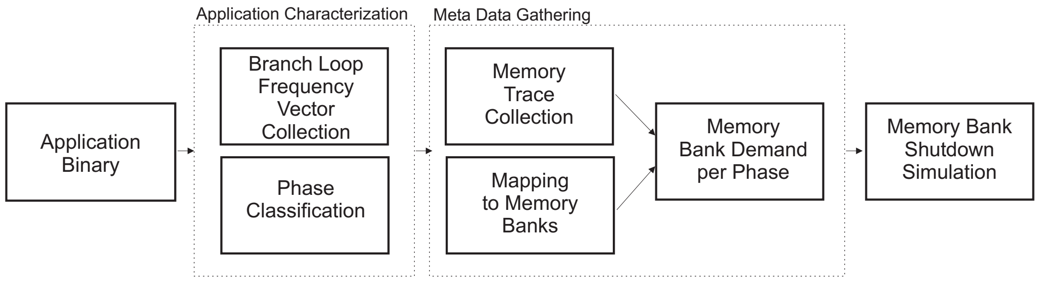
Figure 7 summarizes the simulation flow for our memory bank shutdown scheme using application reflection. The same flow as illustrated in Figure 4 is present. First we perform application characterization, followed by meta data gathering and simulation of the memory bank shutdown policy. The results are presented in the next section.

Note that the technique presented can be used for other purposes other than shutting down memory banks. It can be used for shutting down any device in the system as well as to scale processor frequency and voltage using processor IPC and other relevant parameters.

The next section presents the results for the experiments using the formulation developed in this section.

5 Results

For evaluating the performance of the approach, we chose 5 benchmark applications: *bzip*, *gzip*, *ghostscript*, *adpcm* and *mpegdecode*. The first two are from SPEC2000 benchmarks, the next two from mediabench benchmarks and the last one the Berkeley MPEG-2 decoder. We chose pairs application/input with at least 1



Application Characterization
Meta Data Gathering
Application Binary
Branch Loop Frequency Vector Collection
Phase Classification
Memory Trace Collection
Mapping to Memory Banks
Memory Bank Demand per Phase
Memory Bank Shutdown Simulation

Figure 7: Memory bank shutdown for meta data construction

billion instructions of execution. For the SPEC2000 benchmarks applications, we only execute the first 5 billion instructions due to the long time to execute the experiments had we done otherwise. Note that we have to execute the benchmarks to collect the memory accesses trace in order to simulate the memory shutdown algorithms later on. For the phase classification we used intervals of 5 and 10 million instructions.

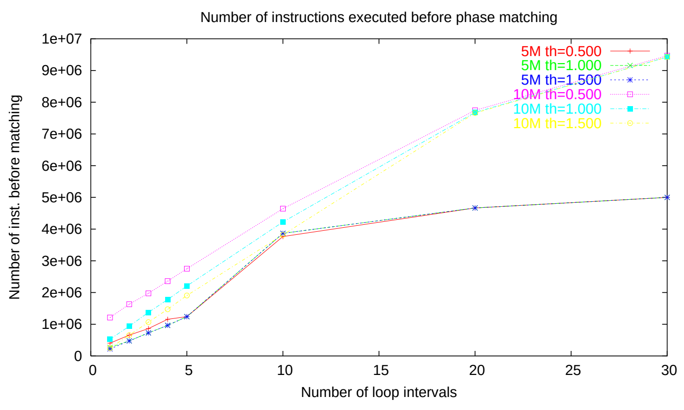
5.1 Phase detection accuracy

In order to be more effective when using the application meta-data, the earlier the runtime system detects which phase the program is running, the better. We try to detect the program phase every N loop branch instructions executed. How frequent it happens will depend on how many loop branch instructions are executed compared to the total number of instructions executed in the program. For every match attempt, an interrupt will be served. Therefore, we need to find a tradeoff between how often it is issue and how early the phase will be accurately detected. We run a set of experiments where we analyze those factors. Table 2 shows the number of phases identified according to the thresholds used. The higher the threshold the fewer phases are identified.

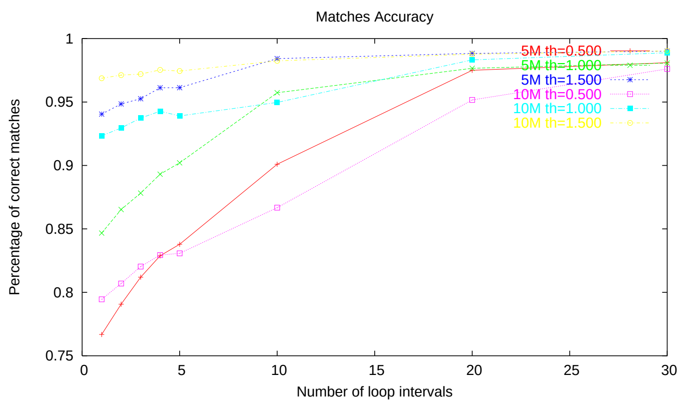
Program (input)	5 million			10 million		
	Threshold			Threshold		
	0.5	1.0	1.5	0.5	1.0	1.5
gzip (input.graphic)	9	6	4	5	3	3
bzip (input.graphic)	31	16	10	23	12	7
mpegdecode (lion)	3	2	1	3	1	1
ghostscript (tiger)	8	4	1	6	4	1
adpcm (clinton)	1	1	1	1	1	1

Table 2: Number of phases for different thresholds for each benchmark with interval size of 5 and 10 million instructions

Figure 8-(a) shows the average number of instructions executed per interval before a phase is matched as a function of how many sets S of N loops are executed. N is set to 10,000 in these experiments. S varies as $1, 2, 3, 4, 5, 10, 20$ and 30. The more sets S the higher the number of instructions executed before the first match. Note that for every 50,000 loops, a bit more than 2 million instructions are executed in the average for intervals of 10 million and approximately 1 milion instructions for intervals of 5 millions. Section 5.2 shows what is the impact in terms of energy savings of recognizing the phase every 50,000 loop branch instructions. S determines how early the power manager can start using the resource demand meta-data. Before a phase of execution is recognized the power manager can either use the resource demand meta-data from the previous phase or do not use any information until a phase is detected. Figure 8-(b) shows the accuracy of phase detections as a function of S. Every time the Algorithm 1 runs, if a phase is detected, we verify whether the phase detected is correct. If no phase is detected (the Manhatan difference is not smaller than the threshold) or if the wrong phase is detected we have a mismatch. The bigger the S the more accurate because more instructions are gathered and the chances of a correct match are higher. Also, the higher the threshold the higher the accuracy because less information is needed for a phase match.



(a)



(b)

Figure 8: (a) - Average number of instructions executed per interval before a phase match is detected for 5 and 10 million instructions and different thresholds; (b) - Phase matching accuracy - percentage of time the phase detected was correct for 5 and 10 million instructions and different thresholds. Benchmark *adpcm* is not used in this average because it has only one phase of execution.

5.2 Memory banks shutdown energy/delay results

For collecting the memory traces and loop branches data we used Simplescalar configured as an inorder StrongARM processor with 32Kb of instruction and data L1 cache and no L2 cache. Since the delays occurred by the resynchronization of memory banks before accesses can cause other components other than the memory itself to spend energy due to the waiting, we set $F_a = 1$. We also ran experiments with $F_a = 0.4$ to evaluate the effects of changing the energy cost of resynchronization. The CpuMemClkRatio is set to 2 so the processor clock is twice as fast as the memory clock. The number of phases per application depends on the size of the intervals, the thresholds and obviously on the application behavior characteristics.

In order to evaluate the benefits of using phase analysis to perform memory bank shutdown, we compare our reflexive approach with a static policy where all the memory banks are kept in the same low power state throughout the whole program execution, with *only profiling* of memory banks, where a single profile for the whole program is used to shutdown the banks, and with a self-monitored hardware based approach, developed by Delaluz *et al* [6], called HBP (Hardware Based Predictor). The policies compared are desribed as follows:

static NAP This is the most energy*delay efficient static policy. The memory is kept in the NAP low power state for the entire program execution.

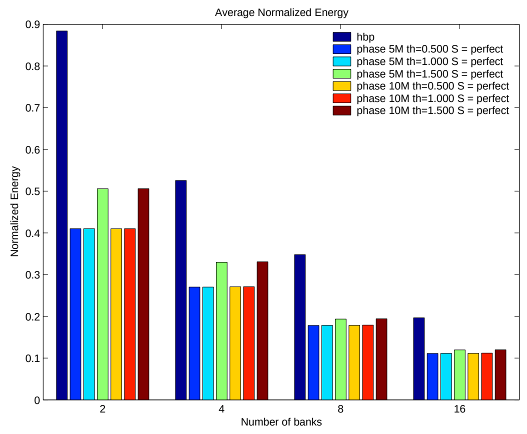
only profiling A profile of memory bank demand for the entire program is collected and used to choose which low power states each memory bank will be switched too. This corresponds to setting $th = 2.0$ when using the phase classification.

HBP A hardware based self-monitored approach to shutdown the banks. Dedicated hardware for each bank is used to predict and keep track of idle time and shutdown the banks accordingly. Using the last idle time as the next, an idle time prediction is calculated and the memory banks are set to the low power state in which the idle time prediction is higher than the synchronization delay. Also a pre-wakeup time is schedule so that the memory banks is ready for use when the request arrives and no resynchronization penalty is paid. When the memory bank is woken up too early, a constant threshold scheme is implemented which shuts down the memory banks based on LEA [10]. If the idle time is higher than thresholds for low power state i, the memory is switched to that state.

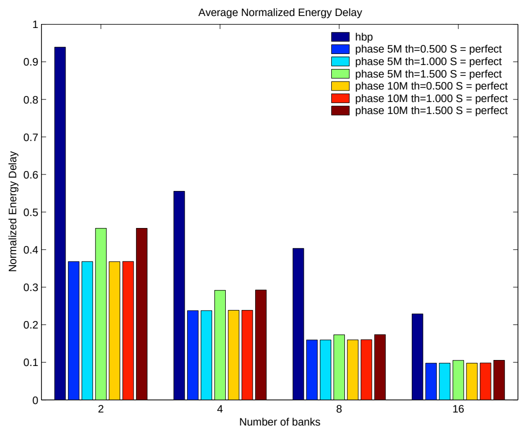
phase The phase policy is the policy proposed in this paper. We classify program intervals into phases offline and use the resource demand information from each phase online to guide switching the memory banks the appropriate power state.

Figure 9 shows the average energy (a) and energy*delay savings (b) for the six benchmarks executed normalized to *static NAP* policy and for $F_a = 1.0$ and $F_a = 0.4$. We used intervals of execution of 5 and 10 million instructions and varied the threshold as 0.5, 1.0 and 1.5. When $S = perfect$, at the beginning of each phase the correct profile information related to the phase is used, resulting in maximum benefit in terms of using the meta-data for deciding the best memory bank low power transition. The Figure also shows the energy consumption of HBP when also normalized to *static NAP*. The goal of this experiment is to shows the energy gains as well as the energy*delay gains when using the phase information throughout the program execution. We use energy*delay to evaluate which schemees yields the best tradeoff performance /energy. By using it, memory banks which are rarely used in some intervals can be put in the deepest sleep state with little energy and delay penalties. Static NAP cannot take advantadge of it because there is no notion of intervals and phases. HBP tries to keep track of this information by prediction the next arrival time based on the last. The extra savings in terms of energy*delay are in the average 65%, 77%, 85% and 90% for 2, 4, 8 and 16 banks respectively when comparing to static NAP. Considering only the energy, the savings follow the same trend with a little bit less savings. The more memory banks the more chances to turn them off and therefore the higher the gains when compared to a static policy. When comparing with HBP, *phase* performs better for all memory bank configurations by saving about 55%, 22%, 25% and 13% more for 2, 4, 8 and 16 memory banks respectively and $F_a = 1$. The gains decrease because the savings when using HBP also increase with the number of memory banks. For $F_a = 0.4$, in comparison with HBP the savings are reduced because the penalty paid by HBP reduces as well. The other important point about the figure is to show that for thresholds 0.5 and 1.0 the savings are very similar. However, when using the threshold as 1.5 there is a decrease in the gains, specially for the 2 and 4 bank configurations. Figure 10 will point out more clearly the reasons for this behavior.

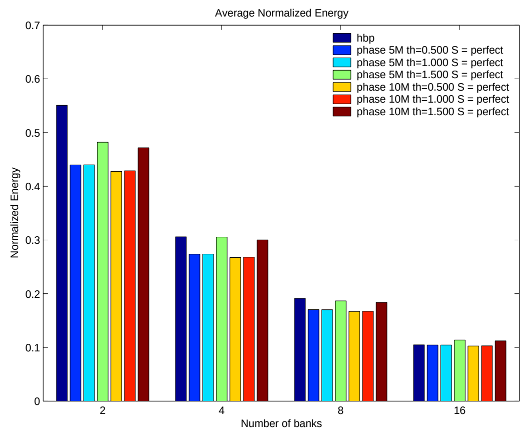
The charts in Figure 10 presents the results of the *phase* policy normalized to the *only profiling*. The goal of this experiment is to show that if the threshold for phase classification is too high the similarity of the intervals classified as in the same phase is poor and as a consequence the resource demand for each interval is not accurately represented by the average among all intervals of the phase. Therefore less chances to switch memory banks to lower power states are discovered. This once more proves that splitting the program into phases is beneficial because it uncover portions of the code which do not use certain memory banks at all or which use them very rarely. This allows extra energy savings by being able to put these banks in low power



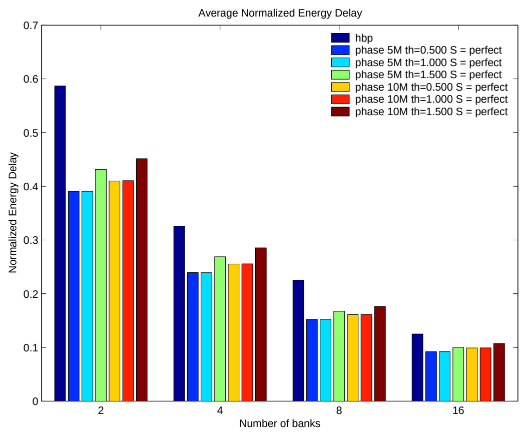
(a)



(b)



(c)



(d)

Figure 9: Normalized Energy (a), (c) and Energy*delay (b), (d) w.r.t. static NAP policy for $F_a = 1.0$ (a), (b) and $F_a = 0.4$ (c), (d).

modes. The chart shows that in the average using thresholds of 0.5 and 1.0 present extra savings of 16% for 2 and 4 banks when compared to *only profiling* and 10% for 8 and 16 banks. Another point is that the savings are very similar for 0.5 and 1.0 and for intervals of 5 and 10 million instructions, indicating that using $th = 1.0$ and intervals of instructions as 10 million instructions is a better choice since it reduces the number of phases and hence the overhead.

All the experiments presented so far assume a perfect matching when using the *phases of execution* information during runtime. In reality though, it is hard to detect the correct phase right in the beginning of each interval. Figure 11 shows the impact of online phase detection on the energy delay savings. When $S = perfect$ at the beginning of each phase the correct profile information related to the phase is used, resulting in maximum benefit. For S varying as 1 and 5, a phase match is tried every 10,000 and 50,000 branch loops respectively. Only after the match the correct meta-data is used for memory bank low power transition. Note that for all phases detected except the first, the meta data used in the previous phase is used until the new phase is detected by our technique. This means that the wrong profile information might be used until the correct phase is detected. This experiment shows the effects of matching the phases online in the energy savings. For intervals of execution of 10 million instructions the effect of matching is negligible regardless of the thresholds used. The same is not true for intervals of 5 million instructions. Figure 8 shows that the

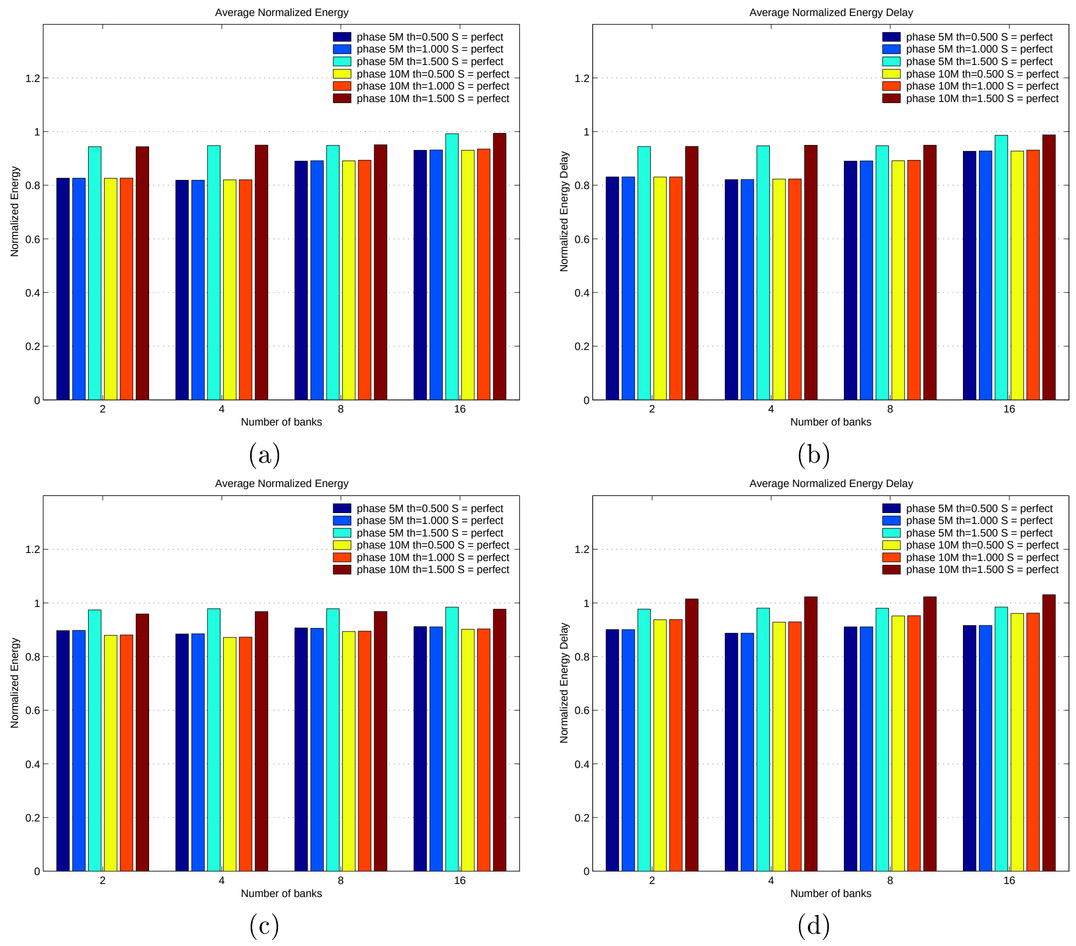
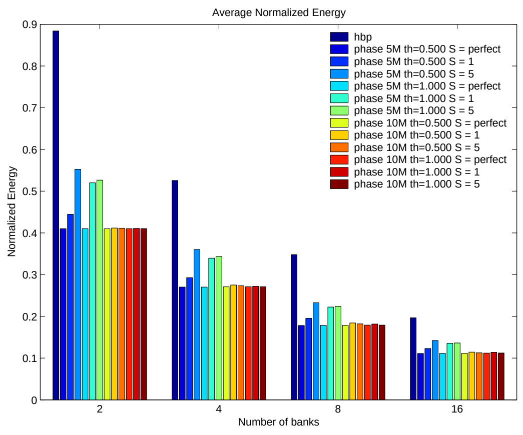


Figure 10: Normalized Energy (a), (c) and Energy*delay (b), (d) w.r.t. only profiling policy for $F_a = 1.0$ (a), (b) and $F_a = 0.4$ (c), (d).

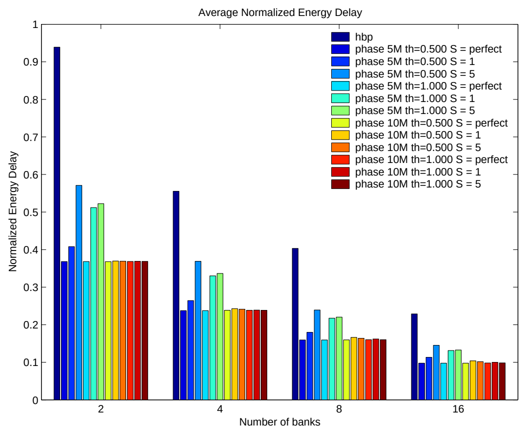
average number of instructions executed before a match for $S = 1$ and $S = 5$ is 0.5 million and a little over 1 million respectively, which in a 5 million instructions interval represents 10% to 20% percent of the whole interval. Besides, the accuracy for $th = 0.500$ and $th = 1.000$ is under 80% and 85% respectively. Note that the effect of using $S = 1$ when $th = 0.5$ for intervals of 5 million instructions is not very significant though, indicating that if necessary this configuration is an attractive option.

5.3 Phase detection threshold and energy savings correlation

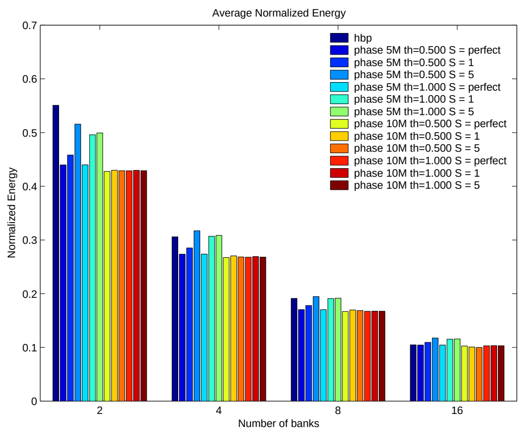
The results presented in the subsection 5.2 show that there is a correlation between the phase detection threshold and the energy savings when using phases to guide memory banks shutdown. In this subsection we explain this relation in more details. The higher the thresholds, the less phases will be detected because smaller is the homogeneity among the intervals in the same phase. Therefore the variance in the predicted number of memory accesses should also be higher. We define two metrics to show how the variance and the prediction for memory accesses change with the threshold. We call them $metric1$ and $metric2$. Let $Intervals_{phase}$ denote the number of intervals classified as in the same phase, $TotalIntervals$ the total number of intervals for the program, avg_{phase}^{bank} and $stddev_{phase}^{bank}$ the average number or memory accesses and the standard deviation for a



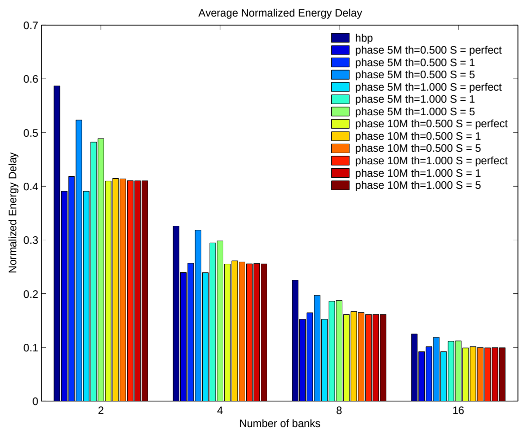
(a)



(b)



(c)



(d)

Figure 11: Normalized Energy (a), (c) and Energy*delay (b), (d) w.r.t static NAP policy and using online phase matching with $S = 1$ and $S = 5$ for $F_a = 1.0$ (a), (b) and $F_a = 0.4$ (c), (d).

given phase and memory bank combination respectively.

$$metric1 = \sum_{\forall phases} \sum_{\forall banks} \frac{Intervals_{phase}}{TotalIntervals} \times (avg_{phase}^{bank} + stddev_{phase}^{bank}) \tag{1}$$

$$metric2 = \sum_{\forall phases} \sum_{\forall banks} \frac{Intervals_{phase}}{TotalIntervals} \times \frac{stddev_{phase}^{bank}}{avg_{phase}^{bank}} \tag{2}$$

$metric1$ (equation 1) summarizes the estimation for the number of memory accesses changes as the threshold varies by giving the sum of the estimations for all memory banks and phases. $metric2$ (equation 2) shows how the variance changes as the phase detection threshold varies. The second term of equation 2 is known as the *coefficient of variance*. It gives the percentage of the average the standard deviation represents. The first term of both equations represent the weight of a phase in the whole program execution. In equation 1 smaller values indicate that smaller number of memory access predictions are calculated. In equation 2 smaller values indicate the less variation is present in the estimations. Both metrics with low values indicate that the variance is low and the number of memory accesses estimates are also smaller, an ideal situation for minimizing energy consumption in the memory banks.

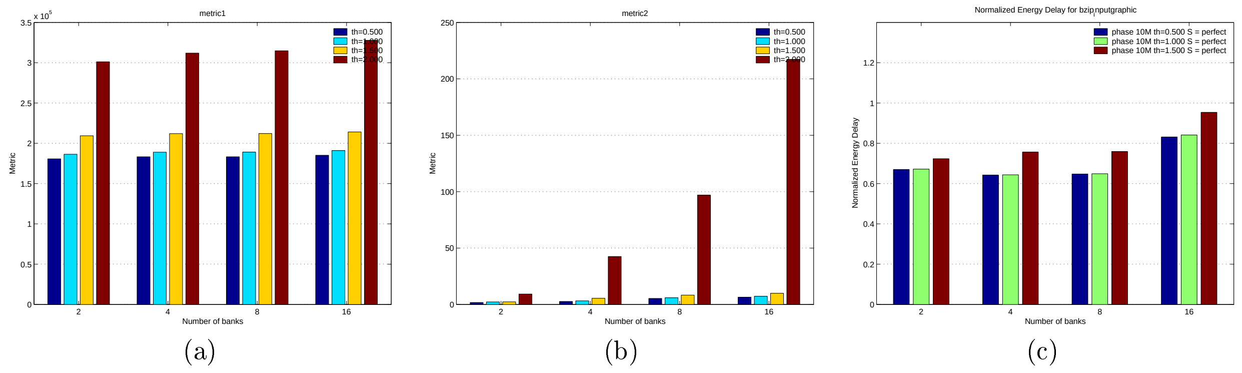


Figure 12: $metric1$ (a), $metric2$ (b) and energy*delay (c) for *bzip* with intervals of 10 million instructions and varying the phase detection threshold

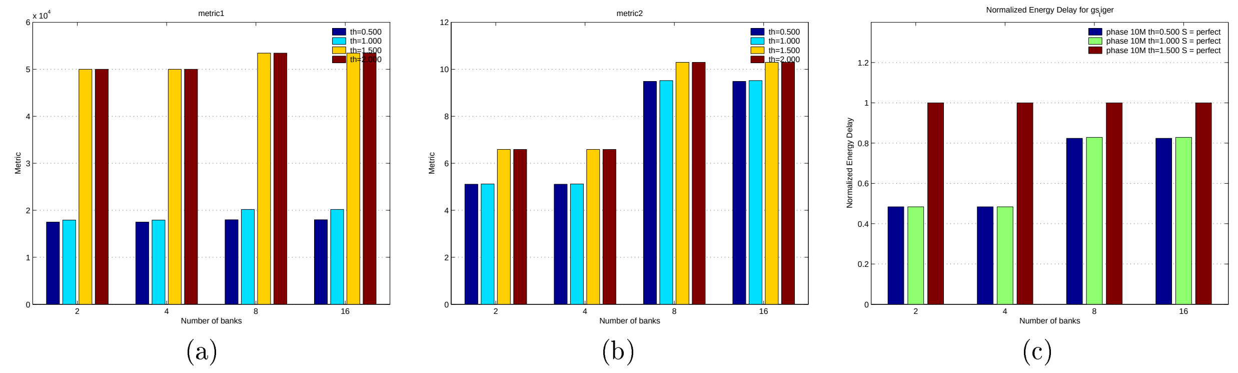


Figure 13: $metric1$ (a), $metric2$ (b) and energy*delay (c) for *ghostscript* with intervals of 10 million instructions and varying the phase detection threshold

Figures 12 and 13 shows both metrics (a,b) and the energy*delay (c) for *bzip* and *gs* with intervals of 10 million instructions and thresholds $0.500, 1.000, 1.500$ and 2.000. The energy*delay chart (c) is normalized to the energy consumption for $th = 2.000$. In Figure 12 we see that the both metrics increase as the threshold increases, showing a direct correlation between the variance and the number of memory accesses estimates as the threshold increases. Similarly, the energy*delay also increases with the threshold since there is more variations and therefore more more penalties are paid for wrong estimates. In Figure 12-(b) we see a sharp increase for $th = 2.0$. This means that the variance is very high. The energy*delay doest not suffer the same sharp increase because the estimates values do not suffer such increase increase as seen in Figure 12-(b). In Figure 13 the opposite happens. The variance does not increase very much, but the estimates do, resulting on a more noticeable increase in the energy*delay product. This happens because the benchmark has a few intervals with lots of memory accesses and lots of intervals with few memory accesses. With low thresholds these intervals are classified as in the same phase ramping up the estimates and therefore avoiding greater energy*delay savings. This indicates that $metric1$ is a better gauge to predict the energy savings.

5.4 Overheads

For assessing the execution overhead of our approach, we run all the benchmarks and calculated the average number of instructions between every 10,000 loop branch instructions to be about 350,000 instructions. We also executed the routine to match the phase signatures at run time on simplescalar and calculated the number of instructions executed by the match algorithm at every 10,000 loop branches to match a partial signatures. Table 3 summarizes these results. The routing executes the algorithm presented in Algorithm 1.

# of phases	# of instructions	overhead
5	2,580	0.7%
10	4,500	1%
20	8,280	2%
30	12,060	3%

Table 3: Execution time overhead of online phase matching

The overead for 10 phases is about 1% of the number of instructions executed. For 30 phases this overhead is about 3%. For a full characterization overhead it remains to calculate the energy overhead for either the fully hardware based approach or for the mixed hardware and software approach. This will be done in future work.

Another overhead that has to be taken into account is the size of the meta-data. For inter arrival time estimate, we assume that 4 bytes per bank / phase are used. Assuming 16 banks and 10 *phases of execution*, $16 \times 10 \times 4 = 640$ bytes are needed. For the signatures, we also assume 4 bytes per bucket. Since we use 32 buckets, 128 bytes multiplied by the number of phases are needed. Therefore, assuming 16 banks and 10 phases, a total of 1920 bytes of meta data is attached to the binary code. Note that we do not really need to store the inter arrival time estimate in the meta-data if the memory energy consumption characteristics are fixed. In this case, we only need 4 bits per memory bank / phase to denote which state the bank should be switched to, reducing the size of the inter arrival data from 640 to 80 bytes and the total size of the meta-data from 1920 to 1360 bytes. Another possible optimization to reduce the overhead is to only consider phases with intervals which represent only a significant percentage of the program execution therefore reducing the number of phases. The signature and resource demand of the eliminated phases could then be removed from meta data. An analysis of which phases are not significant has not been carried out in this paper though.

6 Future Work

We believe that the use of application reflection is useful for helping minimizing energy consumption of applications. To fully validate our memory shutdown example we need to charaterize the energy overhead of collecting the phase signatures and matching them online. We believe the overhead is not significant due to the simplicity of the operations. Furthermore, we also believe that the reflection approach can be used to minimize energy of other components such as the processor itself, by the use of dynamic voltage scaling (DVS), and other processor peripherals such as co-processors, network interfaces, flash memory and others.

7 Conclusions

In this paper we presented a scheme in which an application is classified in phases, meta-data representing the phases of the application along with resource demands is attached to the code, and during the execution either the run time system or the application probes the application to find out which phase is being executed. We call this approach an application reflection because the application carries a representation of its own dynamic behavior along with resource demand meta-data and uses this information to guide the power manager on decision making.

We used main memory bank shutdown as an example of how the technique can be used. The results showed significant energy$*$delay gains are obtained when comparing the scheme with a static policy, with only profiling (without using phase information th=2.0) and with the best known hardware based scheme.

The average savings are 80%, 13% and 28% respectively. We believe that the same scheme can be used for optimizing other resources. One example is performing processor frequency and voltage scaling using IPC estimations from the application phases and integrating such scheme with memory shutdown.

References

[1] Rambus inc. http://www.rambus.com.

[2] L. Benini and G. De Micheli. *Dynamic Power Management: Design Techniques and CAD Tools*. Kluwer Academic Publishers, 1997.

[3] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. technical report cs-tr-97-1342, university of wisconsin, madison, june 1997.

[4] B. Calder, T. Sherwood, E. Perelman, and G. Hamerly. Simpoint web page, http://www.cs.ucsd.edu/users/calder/simpoint.

[5] L. Capra, W. Emmerich, and C. Mascolo. Reflective middleware solutions for context-aware applications. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 126–133. Springer-Verlag, 2001.

[6] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. Hardware and software techniques for controlling dram power modes. *IEEE Transactions on Computers*, 50(11):1154–1173, 2001.

[7] V. Delaluz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Scheduler-based dram energy management. In *Proceedings of the 39th conference on Design automation*, pages 697–702. ACM Press, 2002.

[8] M. Gries and A. Romer. Performance evaluation of recent dram architectures for embedded systems, TIK-Report No. 82, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Switzerland, 1999.

[9] Intel. Intel 80200 processor based on intel xscale microarchitecture developers manual, 2000.

[10] S. Irani, S. Shukla, and R. Gupta. Algorithms for power savings. In *Proceedings of the 14th Symposium on Discrete Algorithms*, 2003.

[11] J. Lau, S. Schoenmackers, and B. Calder. Structures for phase classification. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2004.

[12] A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power aware page allocation. In *Architectural Support for Programming Languages and Operating Systems*, pages 105–116, 2000.

[13] J. H. Park, S. Wu, and B. A. Izadi. Coarse-grained dram power management. In *Embedded Systems and Applications*, pages 248–254, 2003.

[14] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[15] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *30th International Symposium on Computer Architecture*, June 2003.

[16] A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205. ACM Press, 1994.