An energy efficient embedded processor for hard real-time Java applications

Manish Tewary[1], Avinash Malik[2], Zoran Salcic[3], and Morteza Biglari-Abhari[4]

Dept. of Electrical and Computer Engineering, University of Auckland, Auckland 1010, NZ
mtew005@aucklanduni.ac.nz, avinash.malik@auckland.ac.nz,
z.salcic@auckland.ac.nz, m.abhari@auckland.ac.nz

Abstract. Energy management is very important and sometimes critical for certain classes of hard real-time systems. In this paper, we present effective energy reduction techniques for hard real-time systems developed in Java, which execute on bare metal and run on a time-predictable specialized Java processor. We modified traditional clock gating and dynamic frequency scaling methods to include the hardware-based run-time slack calculation in periodic tasks, thus reducing energy consumption in hard real-time systems. Two methods for energy reduction are employed leading to Energy Aware Java Optimized Processor (EAJOP). The first method includes task execution time monitoring and comparison with the estimated worst-case execution time to calculate the slack and bringing the processor to sleep for the slack duration upon task completion. The second method introduces real-time residual slack calculation at so-called checkpoints inside the periodic task, which are then used to lower the system frequency of the rest of the task dynamically, resulting in lower energy consumption. We compare EAJOP with baseline JOP when implemented on FPGA and demonstrate gains in energy consumption.

Keywords: Real-Time and Embedded Systems, Processor, Compiler, Energy Management.

1 Introduction

Java is a platform-independent object-oriented programming language used in the embedded and real-time world, especially after the introduction of Real Time Specification for Java (RTSJ) and Safety-Critical Java (SCJ). Using Java in embedded systems suffered from a few inherent issues like the use of an extra software layer in the form of the Java Virtual Machine (JVM) and unpredictability in execution time due to automatic garbage collection. Few attempts have been made to solve the extra software layer issue by architecting JVM directly in hardware. A prominent case is Java Optimized Processor (JOP) [1] which also offers timing predictability of execution of Java bytecodes and is open for research and modifications.

Energy consumption is an important concern in real-time embedded applications, especially for battery powered devices. The energy consumption reduction of any pro-

cessor is achieved by using different techniques, which are based on processor architecture, static analysis of programs and run-time control, which, however consume additional time and energy. We introduce an approach to energy management and reduction targeting hard real-time systems executing on bare metal processor, which relies on compiler additions, analysis of Worst-Case Execution Time (WCET) of a task and new hardware dedicated for run-time slack calculation, supported with small modifications (additions) of original time-predictable JOP processor in the form of energy management modes. The new processor called Energy Aware JOP (EAJOP) provides a hardware-based mechanism for slack measurement at different points (called checkpoints) in the program and it also implements energy reduction algorithms which use hardware-based run-time slack calculation (RTSC). EAJOP together with the Energy Aware Compiler Tool (EACT) constitutes the main contribution of this paper. Close affinity with JOP enables the use of all JOP compilation tools, while the modified toolchain enables evaluation of the results of energy saving algorithms implemented in EAJOP by comparing it with baseline programs which do not utilize energy management.

The rest of the paper is organized as follows: Section 2 introduces our motivations, task model and methods of energy consumption reduction. Section 3 explains the modifications that led to EAJOP. Section 4 explains EACT used to support energy consumption reduction. Section 5 presents analysis and validation of the approach containing experimental setup and results. Section 6 presents related works. Section 7 concludes the work and indicates some future research directions.

2 Preliminaries

2.1 Task Model and Energy Optimizations

A Java application may consist of one or more tasks which must execute with predefined dependencies, where a task is a unit of work which can be scheduled and executed on the processor. In this presentation, we focus on a single periodically executed task, represented by a Java program and its control flow graph (CFG). The time interval between two successive execution of the task is called the period of the task. A task has many paths from the beginning to the end of its execution. A task T will take time ET to execute with ET ∈ [LB, UB], where LB and UB are lower and upper bound on the execution time, respectively. In hard real-time systems, ET must not exceed the UB, which is considered as the Worst-Case Execution Time (WCET). This WCET can be found by using static timing analysis of programs assuming time-analyzable execution architecture [2].

The time difference between the measured total execution time for the task in clock cycles and WCET time in clock cycles is called slack. This slack can be utilized for energy reduction: (1) by bringing the processor to sleep for the duration of slack, where Sleep mode may be implemented using power gating or clock gating inside the processor or (2) by stretching the non-WCET path to WCET by continuously adjusting system frequency, thereby increasing processor utilization to close to 100% on every path in the task. Huang et al. [3] proposed a method for calculating intra-task DVFS schedules

called Checkpoint Insertion Method. Checkpoints are the points which serve as hints to the run-time system for taking power/energy management actions. Checkpoints are added to the program by the compiler after the static analysis of the program. In our approach, each checkpoint is also annotated with Worst-Case Remaining Cycles (WCRC) which is a static parameter computed by the compiler considering worst-case execution cycles required from the checkpoint to the end of the task. During run-time, processor calculates a dynamic parameter called Remaining Cycles (RC), by subtracting current execution time (CET) at the checkpoint from the WCET. New system frequency can be calculated at each checkpoint dynamically using the following equation:

$$F_{next} = (WCRC * F_{min})/RC \quad (1)$$

where F_{next} is new frequency and F_{min} is the minimum frequency at which worst-case execution times are satisfied. Since processor supports a finite set of frequencies, F_{next} is approximated to next higher frequency in the set of supported frequencies, which is closest to calculated frequency.

Checkpointing method can be implemented in software, which calculates frequency at the checkpoints and as explained in section 5.1, it results in a big overhead, motivating us to use a hardware solution. Processor changes frequency when it encounters checkpoints by using frequency change instructions. Since the time taken by checkpointing and frequency change instructions is fixed and pre-defined, the time-predictability of the original processor is preserved.

2.2 Hardware Based Run-time Slack Calculation and Energy Management

In this section, we introduce two energy saving methods for a real-time embedded processor, which use hardware-based run-time slack calculation (RTSC) either at the end of the task or at different checkpoints in the code during program execution. Figure-1 shows CFG of a small program with four paths from start to finish of the program. Execution times (ET_i) of these paths in increasing order can be shown by the following relation

$$ET_1 < ET_4 < ET_2 < ET_3$$

where Path3 (ET_3) is the WCET path and Path1 (ET_1) is the best-case execution time path. We will use CFG in Figure-1 for explaining the energy saving methods. Also, we assume that Path1 to Path4, if taken, are executed in a sequence respectively and then repeated ad-infinitum.

RTSC with Sleep Mode. Compiler's static analysis tool calculates the WCET in clock cycles for a task. This WCET is inserted in the program code at the start of the task by the compiler. Also, the start and end of each task are marked in the program code. The final slack (in clock cycles) is calculated by hardware at the end of the task execution and the processor goes to Sleep mode for the duration of slack. Figure-2 illustrates the RTSC-Sleep method for the CFG shown in Figure-1. When the processor completes Path1, Path2 or Path4, it has slack which can be used to put the processor to sleep,

whereas when the processor is executing WCET path (Path3) it has no slack, so the processor remains in Normal mode.

RTSC with DFS. In this method, the clock frequency for paths with slack is reduced so that they finish at WCET time. Since the frequency of operation is chosen from a finite set of frequencies, the program execution finishes at the nearest time to WCET that is feasible with the available system frequencies. The task program code is marked with checkpoints by compiler's static analysis tool based on the algorithm given in [3]. Checkpoints are inserted on two types of edges in the CFG:

1. Forward branches
2. Outgoing edges from loop body

An example of checkpoint insertion is shown in the CFG of Figure-1. Compiler calculates WCRC for every checkpoint. Each checkpoint in the program code is annotated by the compiler using a special instruction with the value of WCRC as an operand to the instruction. Compiler re-calculates WCET after insertion of checkpoints and it also inserts WCET at the start of the program code using a special instruction. During task execution, when the processor encounters any checkpoint, it calculates RC using WCET and CET and then calculates the new frequency of operation using Equation-1. Figure-3 shows that after the application of this method, the execution time for all paths is extended to a time closer to WCET.

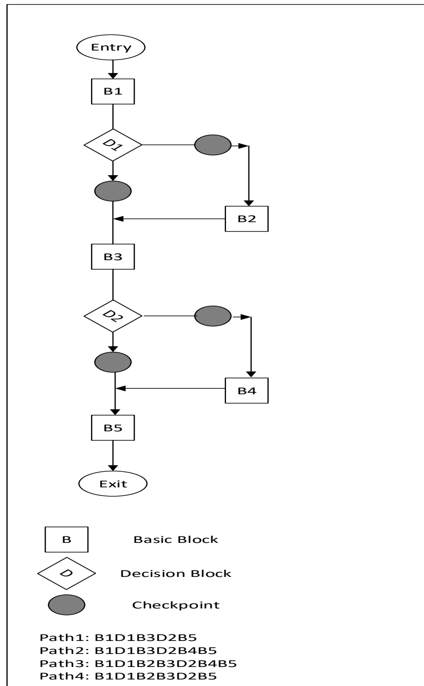

Fig. 1. CFG of a small program

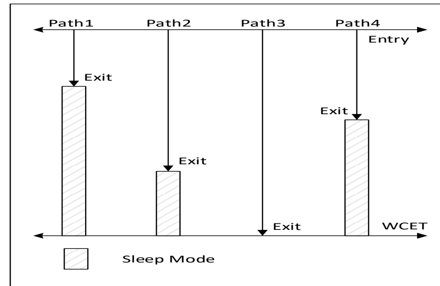

Fig. 2. RTSC with Sleep

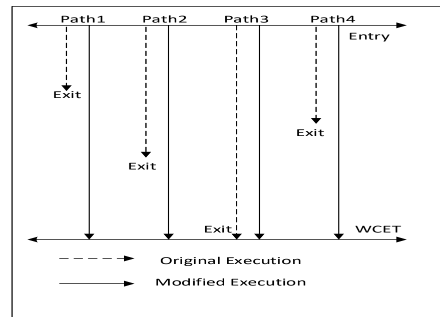

Fig. 3. RTSC with DFS

3 Energy Aware Java Optimized Processor (EAJOP)

EAJOP is a new energy-aware processor based on the JOP architecture, which introduces three modes of processor operation and five new bytecodes for energy management. Figure-4 shows the hardware layout of the EAJOP processor, with the following additions to the original JOP:

1. New bytecodes and microcodes to the core for energy management.
2. A Power Control Circuit (PCC) which implements processor modes (Normal, Sleep, DFS) for energy saving methods.
3. New logic in IO interface for holding (registering) the incoming external events when the core is in Sleep mode.
4. Additional logic and memory in IO interface to store experimental data used in energy measurements.

EAJOP saves energy by either adjusting the frequency or gating clocks of the following processor blocks: EAJOP core, memory interface, and other system components except for PCC and IO interface, which are always active.

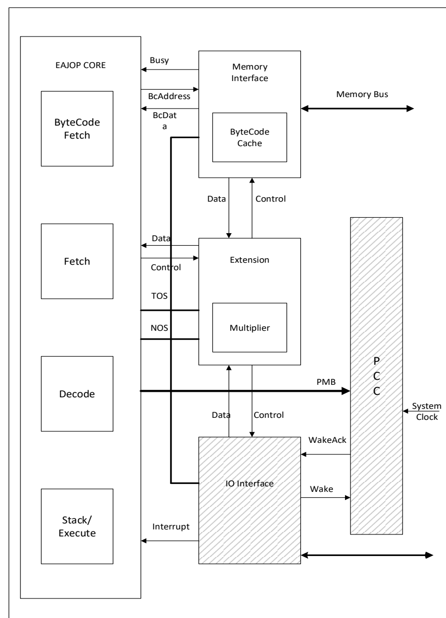

Fig. 4. EAJOP Architecture

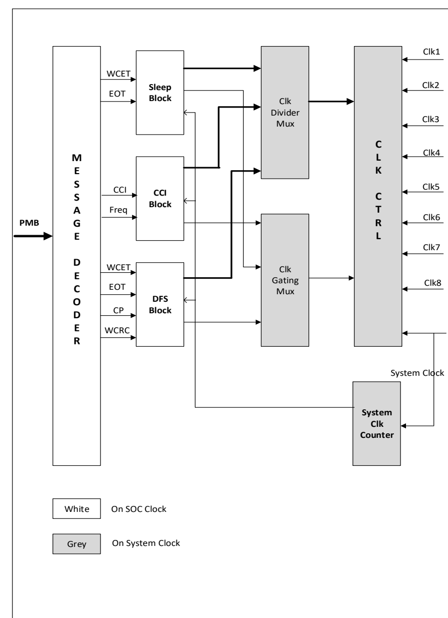

Fig. 5. PCC Architecture

EAJOP Core. Changes in JOP core were made to introduce energy management in the architecture without losing its time analyzability. EAJOP core includes five new bytecodes which can be used by the programmer and compiler for energy management actions on PCC. They are supported by five new microcodes. Each energy management bytecode is implemented using a sequence of microcodes. A new bus called power management bus (PMB) connecting the core to the power control circuit (PCC) was added, as shown in Figure-4. New bytecodes are explained below.

Mode Bytecode (Mode). The processor has three modes of operation, Sleep, DFS and Normal mode. Each mode can be chosen by executing Mode bytecode with a specific operand.
Clock Change Bytecode (CCI). CCI bytecode changes the frequency of the EAJOP processor where the frequency is given as an operand to the bytecode.
Worst Case Execution Time Bytecode (WCET). WCET bytecode is dual purpose, it marks the start of a task and it is also used to transmit WCET value (which it takes as an operand) to the PCC.
End of Task Bytecode (EOT). End of each task can be marked by EOT bytecode.
Check Point Bytecode (CP). CP bytecode is used to mark each checkpoint in the program code and it takes WCRC as an operand.
Checkpoint Minimum Distance Filter (CMDF). CP bytecode adds a time and energy overhead which can be bigger than the effects of lowering frequency, if the execution time between two checkpoints is too short. To circumvent this issue, we have included a CMDF in the Bytecode Fetch block to do a forced hardware abort of CP bytecode, if the last occurrence of CP bytecode was within a pre-defined number of cycles.
Power Control Circuit (PCC). PCC changes the processor's mode of operation and it also implements the energy management logic. The three modes of operation are Normal, Sleep and DFS modes. In Normal mode, no energy optimization technique is used. In Sleep mode, when EOT instruction is encountered then PCC deducts system clock counter value from WCET to calculate slack and then goes to sleep mode for the duration of slack. A counter in PCC decrements the slack value until it reaches zero, the clock gating is removed at this point and processor resumes normal operation. In DFS mode, when CP instruction is encountered then the new frequency is calculated and applied as per the logic given in section 2.2. Figure-5 shows the functional organization of the PCC block.

All the new energy management bytecodes add fixed execution time as shown in Table-1. Each bytecode is made up of new microcode instructions and a fixed sequence of nop's. New microcode instructions take fixed defined hardware actions (register transfers). This fact keeps EAJOP's WCET analyzable just like JOP. Though the real processor implementation would contain both the methods, but for comparison EAJOP was synthesized in two different flavors, the first one called EAJOP-Sleep implements RTSC-Sleep technique and the second one called EAJOP-DFS implements RTSC-DFS technique.

4 EACT – Energy Aware Compiler Tool for EAJOP

EACT is an EAJOP energy aware compiler tool written in Java, which takes standard Java compiled class file as input and generates an energy-aware memory and executable files for EAJOP as output. The baseline of EACT tool is the original JOP WCET tool (WCA) [2]. WCA uses freely available LpSolve Mixed Integer Linear Programming

(MILP) solver (available as a Java library) to get the worst-case execution time estimates. EACT extends WCA functionality by adding the capability to get WCET estimates for checkpoints (called WCRC). A CMDF was implemented in the EACT to reduce the number of checkpoints in the application. Figure-6 shows the compilation steps involved in the generation of Jop executable files for three different execution platforms.

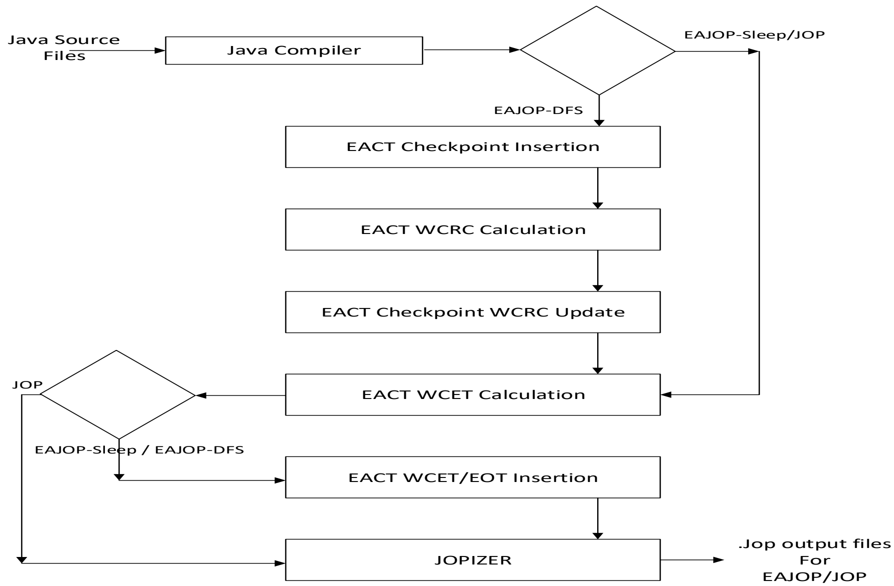

Fig. 6. Compilation steps for three processor platforms

5 Analysis and Validation

5.1 Comparison with software-based implementation of bytecodes

Hardware actions implemented for the bytecodes and algorithms explained in section 3 may be implemented in software (Java), which can calculate frequency at the checkpoints and switch to new frequency using clock change instruction with current execution time calculated by a software routine. A comparison between software implementation and EAJOP-DFS bytecode implementation is shown in Table-1, it shows that hardware implementation for checkpointing is around a thousand times less expensive than the software implementation.

5.2 EAJOP – Two execution platforms

As mentioned before, we synthesized EAJOP in two flavors, thus developing two different energy aware platforms. The results of synthesis for a Cyclone-V Altera (Intel) chip are shown in Table-2.

Table 1. Comparison between EAJOP and software-based implementation

Bytecode	EAJOP Hardware Clock Cycles	Software Based Implementation
Mode	7	2
CCI	16	28
WCET	7	17
EOT	16	17350
CP	19	22784

Table 2. Resource utilization across platforms

Platform	JOP	EAJOP-Sleep (%inc)	EAJOP-DFS (%inc)
Total Memory Bits	3265536	3265536 (0%)	3265536 (0%)
ALMs	3034	3241 (6%)	3732 (23%)
Registers	1680	1817 (8%)	2288 (36.2%)
PLLs	1	1	3

EAJOP-DFS implementation increases the resource utilization by more than 20% as it uses many adders, counters, and frequency selection logics.

5.3 EAJOP – Power Measurement and Energy Estimation

Vijay et. al. [4] give a dynamic power estimation technique for FPGA based designs. We use a reference value for JOP in Nanowatts per MHz for single ALM in targeted FPGA. To get the estimated value for complete design, we multiply reference value with the number of ALM used by the design. This total value can be multiplied by the frequency in MHz to get the power consumption estimates at different frequencies of operation. We use the same technique to estimate value for EAJOP-Sleep and EAJOP-DFS platforms. Maximum frequency was constrained to 50MHz in our current implementation. Table-3 gives the estimated average dynamic power consumption for three platforms, where P is average dynamic power consumption for one ALM.

Different power and energy estimation techniques have been explained in [5]. We chose the instruction level power/energy estimation technique for our research. To calculate the energy consumed by the program of a task, program was converted to a sequence of Java bytecodes and then each bytecode was converted to a sequence of microcodes. Since each microcode takes a fixed number of clock cycles to execute, we could calculate the total time taken for execution at different frequencies and the dynamic component of energy consumed then is simply found by multiplication of average dynamic power consumed per unit of time with the total time taken by application to execute.

Let ET be execution time, ST be slack time, P_f be power at frequency f, P_{cg} the power consumed at clock gated state, f_{max} be the maximum frequency supported by processor and P_{fmax} be average power consumed at f_{max}. Let f_s be the set of feasible frequencies

$$f_s = \{f_1, f_2, f_3, \ldots\ldots, f_{max}\}$$

Then consumed energy can be defined by following equations:

$$Energy_EAJOP_Sleep = (ET*P_{fmax}) + (ST*P_{cg})$$

$$Energy_EAJOP_DFS = \sum_{f \epsilon f_s} ET_f * P_f$$

Table 3. Average Dynamic Power Consumption across platforms

Platform	Operational Freq (MHz)	Average Power Consumption (NanoWatts)
JOP	50	(3034 x 50) x P
EAJOP-Sleep (Pf_{max})	50	(3241 x 50) x P
EAJOP-Sleep (P_{cg})	Clock Gated	(184 x 50) x P
EAJOP-DFS (P_f)	f	(3732 x f) x P

5.4 Experimental Setup and Case Study

We used four applications (Bubble-Sort, Matrix-Multiplication, Sieve, and Lift-Control) from Jembench suite from Java embedded benchmarks [6] for proof of concept. Sixty input patterns were used for each application, each pattern chosen randomly by the test program which calls the applications. WCET was calculated by the EACT tool. Java compiled program was processed by EACT and JOP tools to produce energy aware JOP executable files for experiments on all three platforms as shown in Figure-6. EAJOP stores the execution time (in clock cycles) of each code section with its operation frequency inside a RAM. Java programs were written as wrappers around applications to read data from EAJOP data RAM and calculate energy consumed for every data point which can then be either stored in a text file or displayed on standard output.

5.5 Experimental Results

Baseline energy consumption, calculated for the specific WCET of the program, remains constant for every input pattern as the program is made to run for WCET cycles irrespective of its ET. Each program has a different WCET, resulting in different baseline energy consumption. Output file size across different platforms is shown in Table-4. WCET comparisons for the four experimental applications normalized to JOP are

shown in Table-5. Normalized energy consumption for the four experimental applications is shown in Table-6. RTSC Sleep method tends to save more energy than RTSC DFS on an average execution path but on some paths with extremely high slacks, RTSC DFS saves more energy.

Table 4. Output file size across platforms normalized with JOP output file Size

Platform	Lift	Bubble	Sieve	Matrix
EAJOP-Sleep	1.001	1.006	1.005	1.005
EAJOP-DFS	1.010	1.076	1.041	1.037

Table 5. WCET across platforms normalized with JOP WCET

Platform	Lift	Bubble	Sieve	Matrix
EAJOP-Sleep	1.0023	1.0023	1.0023	1.0023
EAJOP-DFS	1.32	1.31	1.41	1.19

Table 6. Energy Consumption across platforms normalized with JOP Energy

Platform	Lift	Bubble	Sieve	Matrix
Min Energy (EAJOP-Sleep)	0.59	0.12	0.04	0.06
Max Energy (EAJOP-Sleep)	0.66	0.52	0.21	1.018
Avg Energy (EAJOP-Sleep)	0.64	0.40	0.14	0.36
Min Energy (EAJOP-DFS)	0.782	0.09	0.024	0.009
Max Energy (EAJOP-DFS)	0.785	0.58	0.228	1.184
Avg Energy (EAJOP-DFS)	0.783	0.43	0.123	0.378

Saving in the RTSC Sleep is nearly directly proportional to the ratio of slack to WCET, this is due to near negligible time overhead in the RTSC Sleep which is fixed at 25 clocks irrespective of complexity and length of the program code. We observed that average time overhead added by the RTSC DFS was 1100, 1800, 1500, 5000 cycles in Bubble, Lift, Sieve, Matrix respectively, which on an average is about 1.5% to 13% of WCET. Savings in the RTSC DFS was found to be proportional to left-over slack in the execution path as some slack is consumed by time overheads. For paths with negligible slack, EAJOP consumes more energy than JOP, this is due to higher average power in EAJOP as compared to JOP.

6 Related Works

A survey of system level power-aware design techniques was done in [7]; this work is directed towards real-time systems and provides foundations for our research. The work

in [8] was one of the seminal works utilizing compiler and OS assisted power management in the program code to hint the processor on choosing the correct frequency for a real-time system. Their work utilizes slack in the intra-task optimizations, but their treatment uses OS to control and apply frequency changes. The work in [3] defined the checkpoint insertion method for generating a Dynamic Voltage and Frequency Scaling (DVFS) schedule. This method improves average energy savings by 3% to 15% depending upon the process technology. The research in [9] covers both inter-task and intra-task power optimizations using both static and run-time optimizations applied to real-time systems. The work in [10] presents the improvements upon the checkpoint insertion method by implementing a technique to find the correct place to insert checkpoints in the code. They proposed a method for estimating remaining worst-case execution cycles at checkpoints by using execution trace mining for applications. The work in [11] proposes a new online voltage scaling (VS) technique for battery-powered embedded systems with real-time constraints and a novel rescheduling/remapping technique for DVFS schedules. All the above-mentioned research works used OS or supervisory-task controlled energy management whereas our current research focuses on energy management in real-time systems running on bare metal without any controlling software task or OS.

7 Conclusions

In this paper, we presented a brief overview of real-time power aware Java processor called EAJOP and discussed techniques of energy management for a hard-real-time system based on EAJOP architecture. We compared the energy savings of RTSC Sleep and RTSC DFS with baseline JOP platform and found that higher the slack in the programs, more energy is saved by energy management methods implemented in EAJOP. For current implementation and experiments, RTSC Sleep gives better savings than RTSC DFS on low slack paths but on the path with very high slacks RTSC DFS performs better. For any real-time application, selection of the correct platform could be made based on desired WCET, FPGA size, and energy constraints. Our future efforts are directed towards using EAJOP architecture and EACT tool for developing new algorithms for energy optimizations, some of these are:

1. Extending the current work (on single task applications) to systems with multiple periodic tasks.
2. Using the design space exploration technique to find the optimum hardware and compiler solution considering energy, cost and performance for an application.
3. Using the methods in multicore NoC-based architecture context.

References

[1] M. Schoeberl, "A Java Processor Architecture for Embedded Real-Time Systems," *Journal of System Architectures,* vol. 54, no. 1-2, pp. 265-286, Jan-Feb 2008.

[2] M. Schoeberl and R. Pedersen, "WCET analysis for a Java processor," in *Proceedings of JTRES 4th international workshop on java technologies for real time and embedded systems*, Paris, 2006.

[3] P. K. Huang and S. Ghiasi, "Efficient and scalable compiler directed energy optimizations for real time applications," *ACM Transactions on Design Automation of Electronic Systems,* vol. 12, no. 3, 2007.

[4] D. Vijay and T. Tim, "Methodology for High Level Estimation of FPGA Power Consumption," in *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, Shanghai, 2005.

[5] H. Sultan, G. Ananthanarayanan and S. R. Sarangi, "Processor power estimation techniques: a survey," *International journal of high performnace system architecture,* vol. 5, no. 2, 2014.

[6] M. Schoeberl, T. B. Preusser and S. Uhrig, "The embedded Java benchmark suite JemBench," in *JTRES '10 Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, Prague, 2010.

[7] O. S. Unsal and I. Koren, "System-level power-aware design techniques in real-time systems," in *Proceedings of the IEEE*, 2003.

[8] N. Aboughazaleh, D. Mosse, B. R. Childers and R. Melhem, "Collaborative operating system and compiler power management for real-time applications," *ACM Transactions on Embedded Computing Systems,* vol. 5, no. 1, pp. 82-115, 2006.

[9] H. Takase, G. Zeng, L. Gautheir and H. Kawashima, "An integrated optimization framework for reducing the energy consumption of embedded real-time applications," in *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*, 2011.

[10] T. Tatematsu, H. Takase, J. Gang and H. Tomiyama, "Checkpoint extraction using execution traces for intra-task DVFS in embedded systems," in *Sixth IEEE International Symposium on Electronic Design, Test and Application*, 2011.

[11] C. Yuan, M. T. Schmitz, B. M. Al-hashimi and S. M. Reddy, "Workload-ahead-driven online energy minimization techniques for battery-powered embedded systems with time-constraints," *ACM Transactions on Design Automation of Electronic Systems (TODAES),* vol. 12, no. 1, 2007.