Architectural and Compiler Strategies for Dynamic Power Management in the COPPER Project

Ana Azevedo, Radu Cornea, Ilya Issenin,
Rajesh Gupta, Nikil Dutt, Alex Nicolau, Alex Veidenbaum
Center for Embedded Computer Systems
University of California, Irvine
444 Computer Science Building
Irvine, CA 92697-3425
{aazevedo, radu, isse, rgupta, dutt, nicolau, alexv}@ics.uci.edu

Abstract

For a range of embedded system applications in mission critical and energy constrained scenarios it is important to be able to dynamically control power consumption in response to changing power availability. In this paper, we present our approach to dynamic adaptation of system power consumption and application performance through microarchitectural and software strategies. In particular, we discuss our techniques for compiler controlled dynamic register file reconfiguration and profile-driven dynamic clock frequency and voltage scaling. We evaluate the effectiveness of power scheduling heuristics based on these techniques in complying with desired power and performance constraints for a given application.

1 Introduction

Power and energy consumption have been focus of much attention in the design of microelectronics based embedded systems. While a range of circuit design and CAD techniques have been developed that yield significant reductions in power, lately the attention has been on system-level power management and on techniques under compiler, microarchitecture, application software and user control. Extensive reviews of power minimization techniques at various design abstraction levels are presented by [4, 18, 11, 9].

In this paper we present our approach to dynamic control of power and performance of an application that allows to adapt its power-performance profile to externally defined constraints. To control the power/performance profile, we examine use of register files, operating frequency and the supply voltage. We also explore their effectiveness in integration with compiler algorithms. In our scheme the compiler generates configuration code embedded in the application and produces different code versions to be selected at run-time to adapt the code to be run on different architectural organizations. For instance, in case of dynamic register file reconfiguration, the configuration code carries information on the number of registers needed by each function in the code. We also discuss our profile-driven technique for dynamically scaling the clock frequency and the supply voltage and how it can be combined with the former technique, taking into account both time and power constraints.

Related work on power-performance *modulation* comes from several research areas. The work in [5] formalizes a quantitative definition of power-aware systems. It proposes the *ensemble construction* technique for enhancing power-awareness based on composing a system as an assembly of blocks, each engineered to be energy-efficient in specific execution scenarios. One illustration of the technique applies it to the construction of a register file split into banks that can be activated/deactivated depending on the program needs.

The emergence of instruction level power models [28] contributed to the development of compiler techniques for power minimization. In [26, 21] instruction scheduling algorithms take into account per-instruction power and the inter-instruction power overhead when selecting instructions and reordering them to reduce switching activity in the control path. To reduce the switching activity of instruction operands and within registers shared by common data values, register allocation and binding algorithms have been designed [12, 10]. Discussions on the power effect of machine dependent optimizations and higher level compiler optimizations, such as loop unrolling and software pipelining, can be found in [20, 27]. Techniques for energy efficient memory systems are also being developed. [3] presents a compiler-driven code layout technique to exploit a specialized cache storing instructions from critical loops.

Emerging microarchitecture-driven power optimization techniques aim at dynamically reconfiguring the hardware resources, according to the parallelism in the program. In [23, 19] the information on code parallelism and the most efficient hardware configuration adapted to exploit it is either pre-computed and stored in code annotations or computed on-the-fly, using run-time profiling techniques.

Dynamic voltage scaling under operating system control using interval-based scheduling algorithms for non-real-time systems were introduced by [29, 14]. These strategies are limited by their poor understanding of application processing requirements. More recent approaches [15, 24, 16] scale frequency/voltage to adapt to the program's execution behavior and better account for the application computation workload and deadlines. The work in [8] shows how the compiler can automatically derive estimation of memory and CPU workload for an application and calculate the optimal frequency/voltage levels for executing the code. Another compile-time intra-task voltage scheduling technique is presented in [25]. Using a static program analysis technique on worst-case execution times, the algorithm inserts voltage and frequency scaling code in the original program to exploit all the slack time from runtime variations of different execution paths. A microarchitecture-driven approach to reduce power consumption by dynamic voltage scaling exploiting processor stalls during cache misses is presented in [22].

Closely related to our approach is the the hardware-controlled mechanism for energy and thermal management designed in [17]. The framework combines existing energy/power controlling techniques, like frequency/voltage scaling, memory system optimizations for low power and chip sleep mode. Techniques are prioritized by their estimated energy saving and slowdown. During program execution, the selected techniques are applied according to their priority, the application speed, estimated by calculating the average IPC, and the temperature limits to be respected. The focus of our effort is the compiler-guided combination of the techniques for low power that can provide tradeoff between power and performance.

This paper is organized as follows. Section 2 presents the COPPER (**co**mpiler-controlled continuous **p**ower-**per**formance) framework. Section 3 explains the techniques for compiler-controlled register file reconfiguration and profile-driven clock frequency/voltage scaling. In this section we also discuss how these orthogonal optimizations are combined to modulate applications power-performance profile under both power and time constraints. Finally we conclude in Section 4.

2 The COPPER Framework

Figure 1 shows the COPPER dynamic power-performance management framework components and the software design flow. COPPER uses the *gcc* compiler, *Wattch* [6] power simulator and its underlying *SimpleScalar* performance simulator [7] to develop a complete path from code analysis and instrumentation to compilation and simulated execution. A new power profiler and a power scheduler modules were designed and integrated with the *Wattch* simulator.

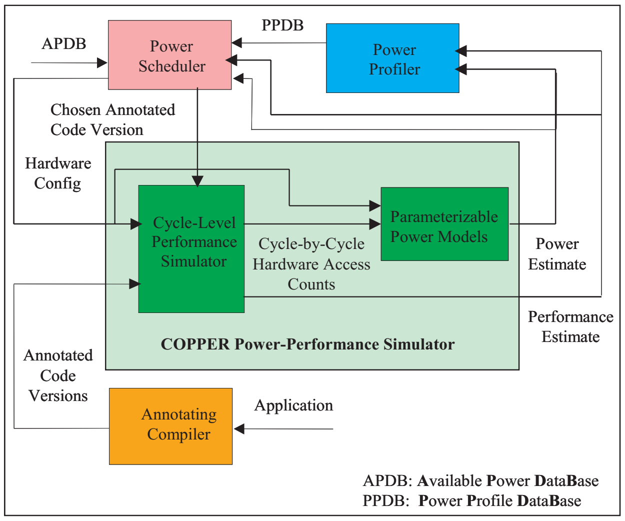

APDB
PPDB
Power Scheduler
Power Profiler
Chosen Annotated Code Version
Hardware Config
Cycle-Level Performance Simulator
Parameterizable Power Models
Cycle-by-Cycle Hardware Access Counts
Power Estimate
Performance Estimate
COPPER Power-Performance Simulator
Annotated Code Versions
Annotating Compiler
Application
APDB: Available Power DataBase
PPDB: Power Profile DataBase

Figure 1. The COPPER dynamic power-performance management framework.

The power profiler module together with *gcc* compiler form the prototype for a power-performance profiling compiler needed by our framework. The power profiler implements function-level profiling that collects total and individual hardware unit energy/power consumption levels and execution cycle counts for applications. This profiling information is summarized in a **p**ower **p**rofile **d**ata**b**ase (PPDB). Constraints on dynamic power consumption are specified in an **a**vailable **p**ower profile **d**ata**b**ase (APDB). The APDB can represent either readings from ever-changing external environmental conditions or pre-programmed power dissipation levels acceptable to accomplish specific tasks within the application.

COPPER uses a variety of architectural, compiler and technology “knobs” to control the power profile of the application. In this paper we focus on dynamic register file reconfiguration, frequency and voltage scaling. One way to control the profile is by creating multiple code versions that are selected by the runtime system. Besides differences in the register pressure, functional units utilization, amount of

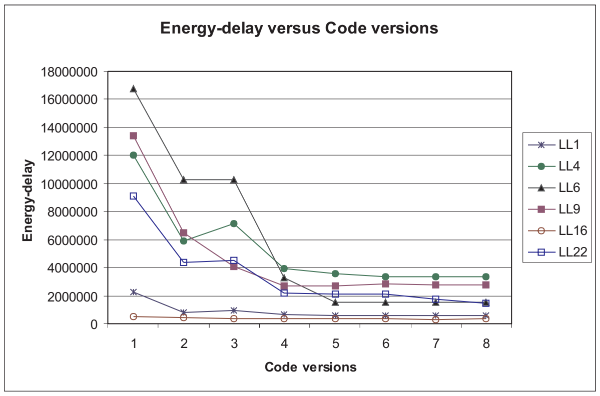


Figure 2. Energy-delay product variation for different code versions of Livermore loops.

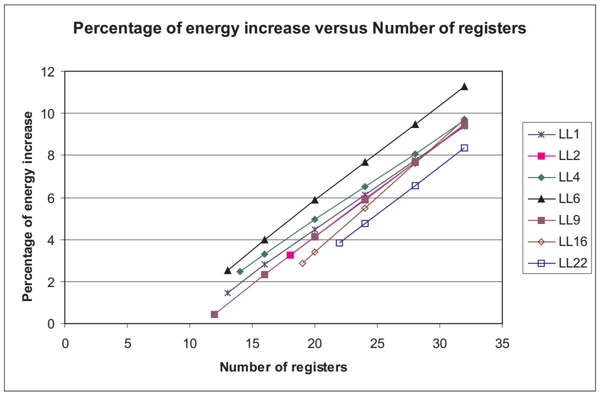


Figure 3. Percentage of total energy increase when varying the number of registers for Livermore loops.

memory accesses and instructions mix, the compiled code versions present different levels of instruction level parallelism. This provides additional flexibility in achieving performance goals within energy constraints.

However, code versioning is only a part of the mechanisms used to pass information from the compiler to the runtime system. The information computed by the compiler, such as time, energy profile and code characteristics, is also carried down to the run-time system using tables and code annotations. For example, the number of registers actually used by a function are annotated in the call operation.

The power scheduler responds to changes in the available power profile. Run-time power scheduling heuristics predict application power dissipation based on the ahead-of-time power profile. In the current implementation, our power scheduler can dynamically change the size of the integer and floating point register files (by turning unused registers off) and change the operating clock frequency and supply voltage.

3 Dynamic Power Management

Identification of the right architectural features and compilation variables that have the most effect on power and performance profile while at the same time providing a good control through compiler algorithms is a challenging problem. A preliminary study of power consumption profiles across architectural components is presented in [2]. The register files provide a good potential for controlling power/performance profile since their use is often directly controlled by the compiler and these constitute an architectural feature present in virtually all types of processors. In addition to their impact on the operation delays and overall program delay, register files themselves have an impact on power consumption by varying sets or numbers of registers being used, and their use parameters such as the data width and the number of ports [5]. Scaling frequency and voltage are well-studied mechanisms for reducing energy dissipation and their impact on power and performance can be fairly accurately estimated [24].

3.1 Register File Reconfiguration

Register files provide two main sources of varying power/performance characteristics of an application at compile time, by changing heuristics to control register allocation and binding. Architectures that present multiple sets of register files provide an additional flexibility in the choice of variable bindings across register sets.

Figure 2 shows how the energy-delay product [13] varies for different code versions of Livermore loops. We chose this metric because it allows to compare code versions from both a power and performance standpoint. The code versions 1 to 8 were compiled for 4, 8, 12, 16, 20, 24, 28 and 32 available registers, for each Livermore loop. This means that the produced code versions use at most the number of available registers and code versions register demand vary for different loops.

For the LL6 loop, the most energy efficient code (determined by the minimum energy-delay product value) is the one generated by code version 5 that uses exactly 15 registers (13 integer and 2 floating-point registers). A power-performance aware compiler is able to find this optimal number of registers. Compiling the code for a register number below 15 increases register pressure and results in more memory usage with register spilling. As a consequence, the system energy consumption (and the energy-delay product) increases. Using more than 15 registers also leads to less energy-efficient code compared to the code performance

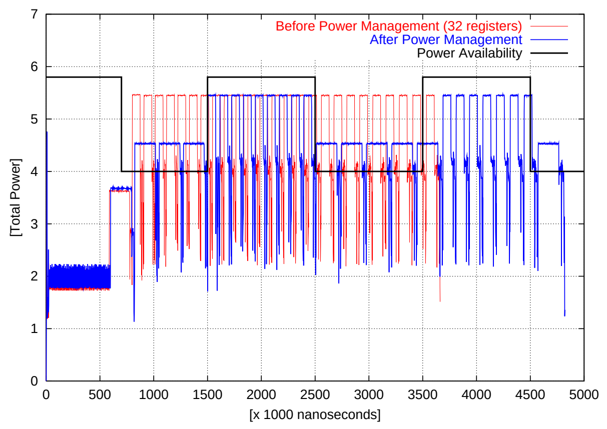


Figure 4. Power management via register file reconfiguration and code versioning for *compress* benchmark.

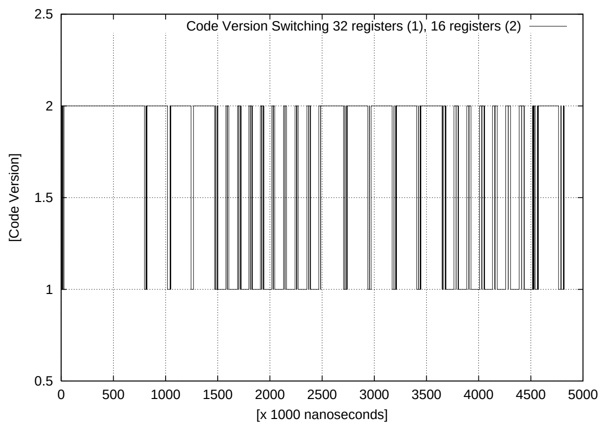


Figure 5. Code version (and register file reconfiguration) switching activity during dynamic power management for *compress* benchmark.

in the optimal configuration (15 registers) and the energy-delay product begins to increase again though at a slower rate. This is explained by the increase in register file power consumption. The above data suggests that register file size has an effect on code with both low and high register pressure.

Figure 3 highlights the contribution of register file units energy consumption in the total system energy. It shows, for the same set of Livermore loops, the percentage increase in total energy observed when running the best code version for each loop on register file architecture configurations supporting a number of registers higher than the code really needs. For LL6 loop example, when running the best code version (which uses 13 interger and 2 floating-point registers) on architectures with 13, 16, 20, 24, 28 and 32 registers of each type, we observed an increase in total energy from 2.5 to 11%.

In the COPPER framework we take advantage of the energy versus number of registers relation shown in Figures 2 and 3 to build our technique for dynamically throttling power/performance. We maintain multiple code versions produced under varying numbers of registers and exact information on the number of registers used in each function call. With code versioning it is possible to increase/decrease average power and expand/compress time. For example, running the code versions compiled for a number of registers higher than the optimal may lead to higher energy consumption and lower execution delay. On the other side, reconfiguring register files guarantees we do not dissipate more power than really needed to execute the code.

To generate code versions we use the *gcc* compiler, varying *gcc* compilation flags. Each code version is analyzed and annotated with information on register file utilization. When power profiling each code version, registers are reconfigured at each function call as the program runs, using the annotated values for register use. The power scheduler is invoked at each function call. Based on the APDB instantaneous power level and the PPDB information for the code versions, the scheduler decides on the new code version using heuristics. For example, in our experiments we use a heuristic that selects the code version that dissipates total system power below and closest to the instantaneous power limit derived from the available power profile. The scheduler then reconfigures the architecture as suggested by the code annotations and program simulation resumes.

We illustrate our power scheduling heuristic using SPEC95 *compress* benchmark. In this example, code versions were compiled for 32 and 16 integer and floating-point registers. The experiments were simulated at 600MHz-2.2V. Figure 4 shows that the resultant power profile *follows* the available power profile. However, the consumed power is not always under the imposed limit. The latter condition will be enforced by the technique in Section 3.2. The code version (and register reconfiguration) switching activity is shown in Figure 5. It is a difficult design problem to choose the right time granularity at which the code should be instrumented to yield an effective and useful modification of the power profile. Our choice of changing code version and reconfiguring registers at function calls is not always effective in controlling the power profile as shown by Figure 4. We also notice that the power management on the *compress* example increases the execution delay by 32% compared to the performance of the code compiled for 32 registers. Our technique tries to satisfy peak power constraints at the expense of overall system energy consumption increase,

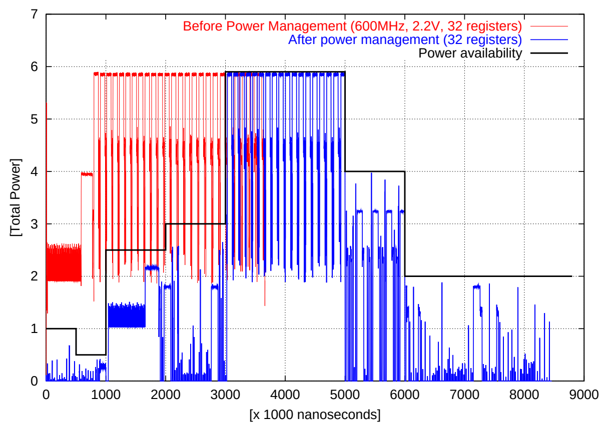


Figure 6. Power management via clock frequency/voltage scaling for *compress* benchmark.

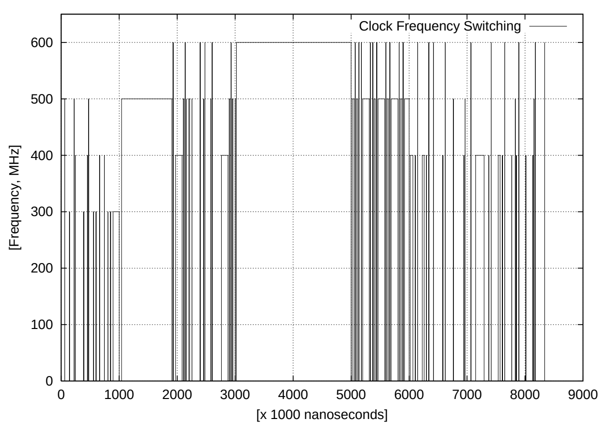


Figure 7. Clock frequency switching activity during dynamic power management for *compress* benchmark.

which might be an acceptable side-effect depending on the underlying power source. In the example in Figure 4 we have not fully accounted for the hardware complexity nor the overhead to turn on or off groups of registers.

3.2 Clock Frequency and Voltage Scaling

Power management through clock frequency and voltage scaling is divided in three phases: compilation, profiling and scheduling. For a given register file configuration, the code is compiled and profiled as before. The new power scheduler heuristic takes into account the frequency/voltage choices in determination of the expected power profile. In the current implementation we support scaling to four different clock frequency and voltage configurations: 600MHz-2.2V, 500MHz-1.8V, 400MHz-1.5V and 300MHz-1.1V. We simulate 20μs of time overhead for complete 0-100MHz clock frequency transition and during this period the processor cannot operate [24]. The scheduler controls and modulates power by following the available power profile, checking it at fixed time intervals of 100ns, changing clock frequency and voltage if needed. In order to select a new configuration the scheduler checks the profiled average system power dissipated by the function currently in execution. It chooses a combination of clock frequency and voltage that yields power dissipation *closest* and *below* the available power limit.

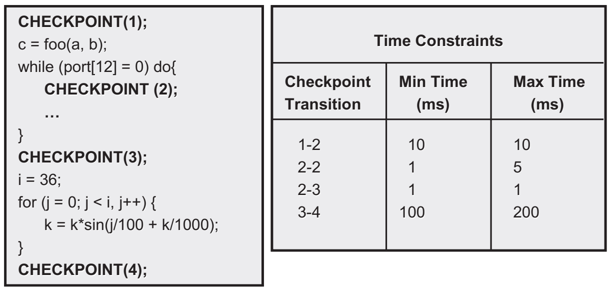
Figure 6 shows the modulated profile generated by the power scheduler when running *compress* benchmark. The rate at which frequency (and voltage) is switched throughout the program execution is shown in Figure 7. We observe a significant increase of 130% in the execution delay when compared to the performance of the same code running at 600MHz-2.2V.

3.3 Power Profile Control under Timing Constraints

In general, the dynamic power management techniques from Sections 3.1 and 3.2 result in execution delay increase. For effective power and performance control, it is, therefore, essential to impose constraints on the timing performance as well and take them into account in power scheduling decisions. While time constraints are expected to be met closely, the available power profile only imposes an upper bound on run-time power consumption. Another power scheduling challenge is to minimize this power consumption as much as possible. In this section we show how the previously studied techniques can be combined to address the issues mentioned above.

To specify time constraints we use program *checkpoints*. A checkpoint c represents a specific location in the code. Figure 8 shows an example of code with four checkpoints and four possible checkpoint transitions: transition (1-2) comprises the instructions from the code beginning up to the instructions in the first iteration of the while-loop; transition (2-2) includes the code fragment for one loop iteration; transition (2-3) represents the control flow taken to exit the while-loop; and transition (3-4) is composed of all the remaining code instructions. In our framework, user-defined checkpoints are inserted in the source code and compiled into special $break$ instructions by using *SimpleScalar* instruction set annotation bits.

Time constraints are set for the execution time of the piece of code between checkpoints, in terms of acceptable lower and upper bounds (l_c, u_c) This information is



```
CHECKPOINT(1);
c = foo(a, b);
while (port[12] = 0) do{
    CHECKPOINT (2);
    ...
}
CHECKPOINT(3);
i = 36;
for (j = 0; j < i, j++) {
    k = k*sin(j/100 + k/1000);
}
CHECKPOINT(4);
```

Time Constraints		
Checkpoint Transition	**Min Time (ms)**	**Max Time (ms)**
1-2	10	10
2-2	1	5
2-3	1	1
3-4	100	200

Figure 8. Code example with checkpoints and imposed time constraints.

stored in a **c**heckpoint **d**ata**b**ase (CDB), along with the possible checkpoint transitions derived from the program control flow. Figure 8 shows an example of how minimum and maximum time constraints for checkpoint transitions and the description of the possible transitions are specified in a CDB.

To estimate the *quality* of the time-constrained scheduling we define an error measure σ_c

$$\sigma_c = \begin{cases} |t_c - l_c|, & t_c < l_c \\ |t_c - u_c|, & t_c > u_c \\ 0 & otherwise \end{cases}$$

as the absolute value of the difference between the actual checkpoint execution time t_c and the closest time constraint bound. The average relative error in satisfying the time constraints is the sum of the error values σ_c for all checkpoint transitions divided by the program execution time, or

$$\varepsilon = \frac{\sum_c \sigma_c}{executiontime}$$

Our heuristic for scheduling with power and time constraints is divided in three phases: two ahead of time profiling phases and the final run-time power scheduling phase. Details about each phase follow below.

Profiling Phase 1: This phase selects the best code version for each function in the program. We compile code versions for 16 and 32 registers, in the same way as in Section 3.1. Code selection is based on the energy delay metric $energy * delay^2$, where $energy$ is the total profiled system energy consumed and $delay$ is the total profiled cycle count per function. We chose this metric because dynamic voltage scaling leads to the relation $energy \propto frequency^2$. Finally, we produce code annotations carrying the best code version as chosen by the metric along with its register demand.

Profiling Phase 2: The goal in phase 2 is to generate a **c**heckpoint **p**ower **p**rofile **d**ata**b**ase (CPDB). This second profiling phase collects energy and maximum cycle count for checkpoint transitions. Using the code annotations from Phase 1, we run the program adjusting register file size and changing the code version to be run.

Run-Time Power Scheduling: The power scheduler dynamically changes configuration to the annotated code version and register file size at function calls and, if necessary, dynamically adjusts the frequency at program checkpoints. Other times at which the scheduler might adjust the frequency include points of abrupt changes in the APDB and times the scheduler realizes an expected checkpoint transition did not take place. The latter happens when an alternative control flow path is executed and therefore frequency must be adjusted. To reduce overall energy consumption, when reducing the clock frequency the voltage is scaled proportionally to changes in the clock speed [24]. In our experiments we use configuration 600MHz-2.2V as baseline and clock frequency can vary in 20MHz increments. Figure 9 outlines the algorithm used at each checkpoint. The main phases of the algorithm are described below.

Create list of events: Step 1 in Figure 9.
The scheduler creates a list of all possible events from the current time marked by the current checkpoint to the farthest deadline of all possible future checkpoints. The in between events might include some other checkpoint transitions and available power profile change points.

Calculate frequency limit: Step 2 in Figure 9.
To calculate the run-time frequency limit (f_{limit}) values the scheduler uses the APDB, CPDB and CDB information, checking the available power ($power\_limit$) and the maximum profiled power ($max\_power\_per\_cycle$) consumed from the checkpoint in consideration to all its possible transitions to future checkpoints. The frequency limit is calculated by a formula which is based on the fact that dynamic voltage scaling combines two CMOS design equations: $energy \propto voltage^2$ and $frequency \propto (voltage - V_t)/voltage$. Figure 10 illustrates a list of events and the conversion of available power constraints into frequency limit constraints.

Calculate optimal frequency: Step 3 in Figure 9.
Upon obtaining the frequency limit, the scheduler calculates the range of frequencies the code can be run to satisfy the time constraint upper bounds. In this calculation it uses the CDB information on imposed time constraints for all possible checkpoint transitions from the checkpoint in consideration. At this phase three situations may arise. In the case there is only one future checkpoint transition (Case 1 in Figure 9, graphically illustrated in Figure 11), we calcu-

1. At a checkpoint execution, create a new list of events with the time intervals the events occur.
2. Calculate frequency limit f_{limit} for each of the time intervals in the list of events.

$$f_{limit} = \sqrt[3]{power\_limit*base\_freq^2 / max\_power\_per\_cycle}$$

3. Calculate optimal frequency and create a schedule for adjusting frequency values.

Case 1: Only two checkpoints, frequency limit is high.

$$Optimal\_frequency = \frac{profiled\_number\_of\_cycles}{max\_time\_constraint\_between\_checkpoints}$$

Case 2: Only two checkpoints, frequency limit is *not* constantly high.

$$Optimal\_frequency = \begin{cases} freq\_limit \\ (profiled\_number\_of\_cycles - limited\_number\_of\_cycles)/(max\_time\_constraint - limited\_time) \end{cases}$$

Case 3: Several possible future checkpoints, each checkpoint transition handled as Case 1 or Case 2.

$$Resulting\_optimal\_freq(first\_interv) = \max(opt\_freq1,\ opt\_freq2)$$

$$Optimal\_freq(second\_interval) = \frac{remaining\_number\_of\_cycles\_for\_the\_second\_interval}{remaining\_time\_for\_the\_secong\_interval}$$

4. Execute the program, adjusting optimal frequency values according to the created schedule.
5. As soon as the next checkpoint is executed, discard the list of events. Go to step 1.

Figure 9. Outline of the algorithm for scaling frequency and voltage under power and time constraints.

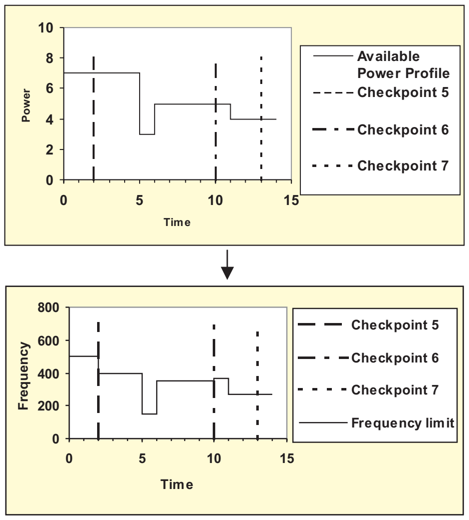


Figure 10. Converting available power constraints into frequency limit constraints.

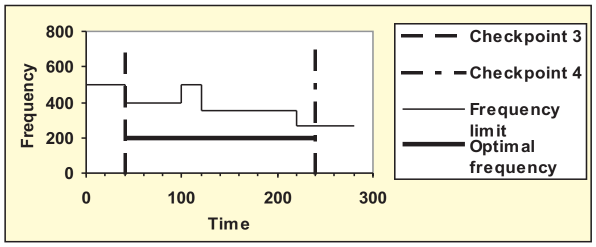


Figure 11. Calculate optimal frequency algorithm, case 1.

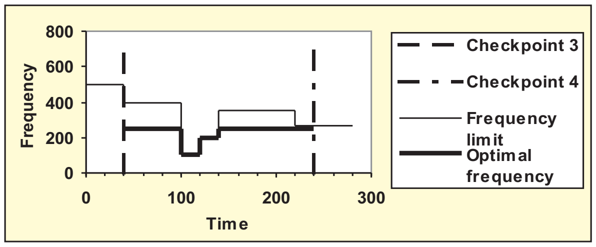


Figure 12. Calculate optimal frequency algorithm, case 2.

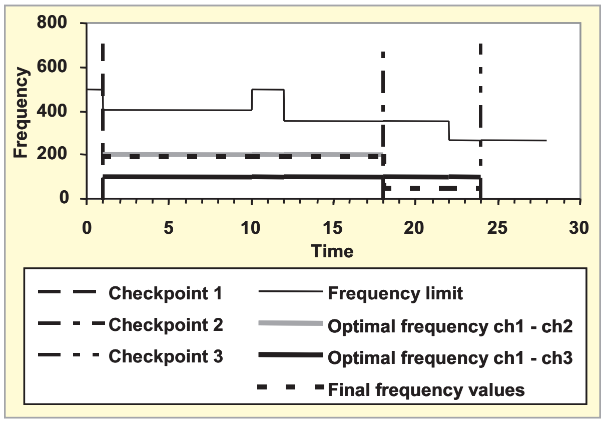


Figure 13. Calculate optimal frequency algorithm, case 3.

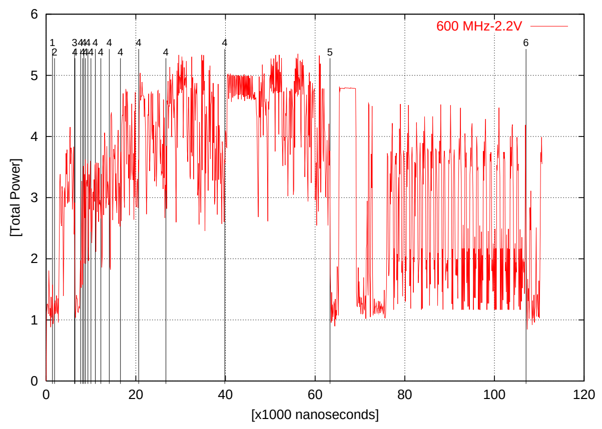


Figure 14. Power consumption highlighting checkpoint execution times for *paraffins* benchmark.

late a potential $Optimal\_freq$ value by dividing the profiled number of cycles for the checkpoint transition by the maximum time allowed for this transition in the CDB. If this value is less or equal to f_{limit}, than $Optimal\_freq$ is set to this value. The second case (Case 2 in Figure 9, sketched in Figure 12) still handles the situation in which there is only one possible future checkpoint transition. However, in this case, the frequency limit values calculated in the previous phase, for each event interval that happens within the checkpoint transition interval, are lower than the potential $Optimal\_freq$ values calculated by Case 1. For such intervals the $Optimal\_freq$ is fixed to f_{limit}. The $Optimal\_freq$'s for the remaining intervals are calculated just like in Case 1, after counting off the cycles (and time) spent executing at f_{limit} speed. In the third case (Case 3 in Figure 9, depicted in Figure 13), several possible checkpoints can be reached from the checkpoint in consideration. Each checkpoint transition is then handled as either Case 1 or Case 2. After calculating the $Optimal\_freq$ values for each checkpoint transition, the scheduler selects the maximum $Optimal\_freq$ among the values. It keeps adjusting the $Optimal\_freq$ as program paths are executed, always selecting an $Optimal\_freq$ value that makes the code run as slow as possible within the time constraints. For example, in Figure 13, after checking that there is no checkpoint occurrence after time 18, the scheduler assumes the transition (1-2) did not occur and lowers the speed for the execution of the remaining code until checkpoint 3 is executed.

We illustrate the combined heuristic using *paraffins* benchmark [1]. Figure 14 shows the power consumption profile for *paraffins* when executing the code switching to the best code version (compiled for 32 or 16 registers) and

start checkpoint	end checkpoint	MinTime (ns)	MaxTime (ns)
1	2	4000	4000
2	3	8000	8000
3	4	1000	1000
4	4	6000	32000
4	5	40000	60000
5	6	70000	70000

Figure 15. Checkpoint database (CDB) for *paraffins* benchmark.

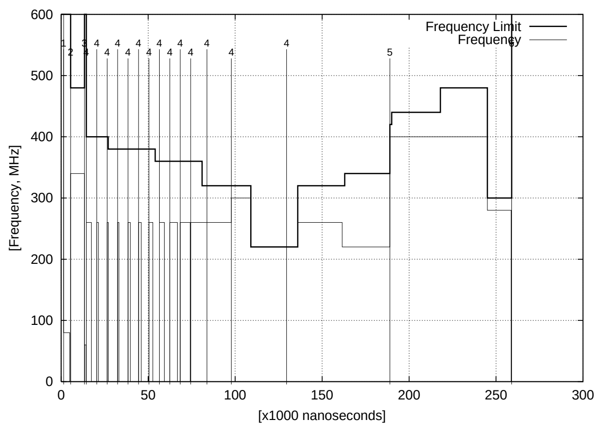


Figure 16. Calculated target frequencies satisfying time and power constraints for *paraffins* benchmark.

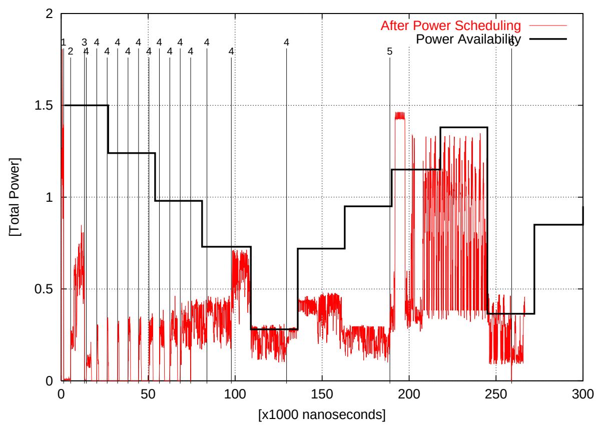


Figure 17. Power management using combined technique for *paraffins* benchmark.

reconfiguring register files at function calls. The frequency does not vary continuously, but is adjusted using 20MHz increments with values in the range 0-600MHz. For the time constraints defined in Figure 15 and the available power profile shown in Figure 17, we obtain the frequency limit and optimal frequency values plotted in Figure 16. The moments in which the optimal frequency values reach 0 represent situations in which the execution of the code between checkpoints finishes earlier than expected (this happens because the target frequency calculation uses the profiled maximum cycle count for checkpoint transitions and accounts for checkpoint transitions that might not take place). In such cases, we stop the simulated processor to satisfy the minimum time constraints between checkpoints. The final result of dynamic power scheduling is shown in Figure 17. The power dissipated is modulated throughout the program execution unlike in Figure 14, where the available power profile was not met. Although peaks of higher power consumption do occur (as the heuristic uses average power between checkpoints), the average power consumption is kept below the available power limit. For this experiment the average error ε without power scheduling is 102% and 65% after power scheduling, indicating a good match with the power and time constraints.

4 Conclusions and future work

This paper outlines a framework for exploration of compiler and run-time strategies to control a power consumption profile under performance constraints. We have shown how code versioning, dynamic register reconfiguration and frequency/voltage scaling can be used to achieve this goal. Our future work includes finding other knobs for modulating power; further exploiting code versioning as a mechanism for modulating power by experimenting with other code optimizations; and improving the quality of dynamic power scheduling, by minimizing the average error produced by the heuristic for meeting time constraints and the average error for meeting power constraints. We believe that combining simpler power scheduling techniques with different impact on the power and performance profile is a promising way to build flexible dynamic power management heuristics.

Acknowledgments

This research is supported by DARPA PAC/C program under contract number F336-15-00-C-1632.

References

[1] http://www.trimaran.org, Trimaran Project.

[2] A. Azevedo, R. Cornea, I. Issenin, R. Gupta, N. Dutt, and A. Nicolau. Power-performance mangement in the COPPER project. In *UCI Technical Report 01-02*, January 2001.

[3] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis. Architectural and compiler support for energy reduction in memory hierarchy of high performance microprocessors. In *ISLPED*, August 1998.

[4] L. Benini and G. D. Micheli. System-level power optimization: techniques and tools. *ACM Trans. on Design Automation of Electronic Systems*, 5(2):115–192, April 2000.

[5] M. Bhardwaj, R. Min, and A. Chandrakasan. Power-aware systems. In *Proceedings of 34th Asilomar Conference on Signals, Systems and Computers*, November 2000.

[6] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA*, June 2000.

[7] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. In *Technical Report 1342, University of Wisconsin-Madison, CS Department*, June 1997.

[8] U. K. C. Hsu and M. Hsiao. Compiler-directed dynamic frequency and voltage scheduling. In *Workshop on Power-Driven Microarchiteture*, June 1998.

[9] A. Chandrakasan and R. Brodersen. *Low Power Digital CMOS Design*. Kluwer Academic Publishers, 1995.

[10] J. Chang and M. Pedram. Register allocation and binding for low power. In *DAC*, 1995.

[11] S. Devadas and S. Malik. A survey of optimization techniques testing low power VLSI circuits. *DAC*, pages 242–247, December 1995.

[12] C. H. Gebotys. Low energy memory and register allocation using network flow. In *DAC*, June 1997.

[13] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, September 1996.

[14] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting on a low-power CPU. In *Proceedings of the First Annual Int'l Conf. on Mobile Computing and Networking*, 1995.

[15] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. Srivastava. Power optimization of variable-voltage core-based systems. *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, 18(12), December 1999.

[16] http://www.transmeta.com Transmeta corporation.

[17] M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas. A framework for dynamic energy efficiency and temperature management. *International Symposium on Microarchitecture*, December 2000.

[18] M. J. Irwin. Low power design for systems on a chip - a tutorial. In *12th Annual IEEE ASIC/SoC Workshop*, September 2000.

[19] A. Iyer and D. Marculescu. Power aware microarchitecture resource scaling. In *DATE*, March 2001.

[20] M. Kandemir, N. Vijaykrishan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. *DAC*, pages 304–307, June 2000.

[21] M.-C. Lee, V. Tiwari, S. Malik, and M. Fujita. Power analysis and minimization techniques for embedded DSP software. *IEEE Trans. on VLSI Systems*, 5(1):123–135, January 1997.

[22] D. Marculescu. On the use of microarchitecture-driven dynamic voltage scaling. In *ISCA*, June 2000.

[23] D. Marculescu. Profile-driven code execution for low power dissipation. In *ISLPED*, July 2000.

[24] T. Pering, T. Burd, and R. Brodersen. Voltage scheduling in the lpARM microprocessor system. In *ISLPED*, July 2000.

[25] D. Shin, J. Kim, and S. Lee. Intra-task voltage scheduling for low-energy hard real-time applications. *DT*, pages 20–30, March 2001.

[26] C. L. Su, C.-Y. Tsui, and A. M. Despain. Saving power in the control path of embedded processors. *IEEE Design and Test of Computers*, 11(4), December 1994.

[27] V. Tiwari, S. Malik, and A. Wolfe. Compilation techniques for low energy. In *ISLPED*, October 1994.

[28] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step toward sofware power minimization. *IEEE Trans. on VLSI Systems*, 2(4):437–445, April 1994.

[29] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 13–23, Novermber 1994.