Managing Multiple Low-Power Adaptation Techniques: The Positional Approach

Michael C. Huang, University of Rochester
Jose Renau and Josep Torrellas, University of Illinois at Urbana-Champaign

A major challenge in adaptive processing is to determine when to trigger an adaptation. To do this, we need to partition the program into phases that behave differently enough to warrant adaptation. Ideally, the behavior within each phase is homogeneous and predictable. The granularity of each phase should not be too fine or too coarse. If it is too fine, transient states and adaptation overheads can negate any gains. If it is too coarse, the behavior probably is not homogeneous.

In the context of improving energy efficiency, we use *low-power techniques* for adaptation. An LPT is a hardware structure that, if activated, typically saves energy at the expense of some performance. The processor activates LPTs at the beginning of a phase on the basis of their predicted effect.

We classify adaptation approaches based on how they exploit program behavior repetition. The conventional *temporal approach*[1] exploits the similarity between successive intervals of code in dynamic order; the newer *positional approach*[2] exploits the similarity between different invocations of the same code section.

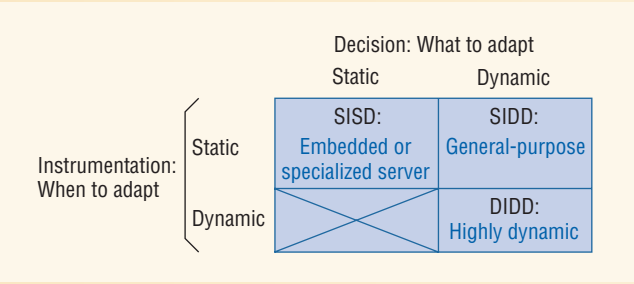

Figure A. Different implementations of positional adaptation and targeted workload environments.

The two approaches activate and deactivate LPTs based on different criteria. Specifically, temporal schemes divide the execution into time intervals and predict the upcoming interval's behavior based on previous intervals' behavior. Positional schemes, instead, associate program behavior with a particular code section. Thus, a positional scheme tests LPTs on different executions of the same code section. Once the positional scheme determines the best configuration, it applies that configuration on future executions of the same code section. This approach is based on the intuition that program behavior is largely determined by the code being executed. Experimental analysis shows that calibration is more accurate in the positional approach.[2]

Positional adaptation is also very flexible. We propose three

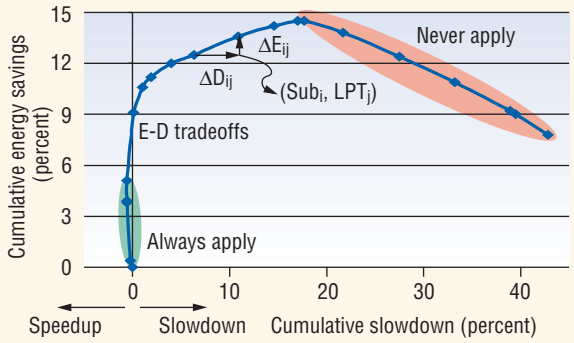

Figure B. Energy-delay tradeoff curve. Starting from left to right in the E-D tradeoffs region, the positional scheme applies pairs until the cumulative slowdown reaches the slack.

simple to implement in hardware, these two statistics can effectively guide reconfiguration decisions.

Daniele Folegnani and Antonio Gonzalez use program parallelism statistics, rather than issue queue usage, to guide reconfiguration decisions.[8] Specifically, if the processor rarely issues instructions from the back of the queue—that portion of the queue that holds the most recent instructions—the system assumes the queue to be larger than necessary and downsizes it. The system also periodically upsizes the queue at regular intervals to limit performance loss.

Researchers commonly use cache miss rate to guide cache configuration decisions. The DRI-cache, for example, measures the average miss rate over a set operation interval to determine whether to change the cache size. As miss rate information may already be available in microprocessor performance counters, the system can essentially acquire this statistic for free.

Compiler-based profiling offers an alternative to hardware monitoring. With this approach, developers either instrument the application and run it on the target machine to collect statistics, or they run it on a detailed simulator to gather the statistics. Michael Huang and his colleagues use the simulator approach to collect statistics about the execution length of subroutines for phase detection.[9]

The application behavior observed during the profiling run must be representative of the behavior encountered in production. Because this assumption may not hold for many general-purpose applications, and inexpensive hardware counters are readily available in modern microprocessors or can be added with modest overhead, hardware-based monitoring is more frequently used in adaptive processing.

Triggering. A microprocessor can use several approaches to trigger a reconfiguration decision. The first approach reacts to particular characteristics of the monitored statistics. For example, Ponomarev's adaptive-issue queue scheme upsizes the queue when the average number of valid queue entries over the interval period is low enough that

different implementations that target different workload environments. They differ on which adaptation decisions they make statically and which decisions they make at runtime.[2] Specifically, as Figure A shows, *instrumentation* (I) is the selection of when to adapt the processor, and *decision* (D) is the selection of what LPTs to activate or deactivate at that time. The system can make each selection *statically* (S) before execution or *dynamically* (D) at runtime. For example, an implementation can produce static instrumentation and static decision (SISD).

The targeted environments are labeled as embedded or specialized server, general-purpose, and highly dynamic. In these implementations, we use the program's major subroutines as the code section. The core control algorithm for all the implementations is essentially the same.

Tests of the different LPTs on different subroutines record the impact on energy and performance for comparison with other LPT and subroutine combinations. Specifically, we rank the pairs in decreasing order of energy savings per unit slowdown.

Figure B shows this ranking for a sample application in a system with several LPTs. The difference between the three implementations is how much information they provide. The more static schemes are more accurate because they have more information thanks to offline profiling.

The origin in the figure corresponds to the system with no activated LPT. As we follow the curve, we add the contribution of all subroutine-LPT pairs from most to least efficient, accumulating energy reduction (*y*-axis) and execution slowdown (*x*-axis). As an example, Figure B shows the contribution of a pair (subroutine$_i$, LPT$_j$) that saves ΔE_{ij} and slows down the program ΔD_{ij}.

We divide the curve into three main regions based on the results for each pair: improving both performance and energy (*Always apply*), saving energy at the cost of performance degradation (*E-D tradeoffs*), and degrading both energy and performance (*Never apply*). As the names suggest, we apply all the pairs in the first region and no pairs in the third region. Given a slack—or tolerable performance degradation—we start from left to right in the E-D tradeoffs region and apply pairs until the cumulative slowdown reaches the slack. In two experiments, the positional schemes boosted the energy savings by an average of 84 percent and 50 percent over several temporal schemes.[2]

References

1. R. Balasubramonian et al., "Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures," *Proc. Int'l Symp. Microarchitecture*, IEEE CS Press, 2000, pp. 245-257.
2. M. Huang, J. Renau, and J. Torrellas, "Positional Adaptation of Processors: Application to Energy Reduction," *Proc. Int'l Symp. Computer Architecture*, IEEE CS Press, 2003, pp. 157-168.

Michael C. Huang *is an assistant professor in the Department of Electrical and Computer Engineering, University of Rochester. Contact him at michael.huang@ece.rochester.edu.*

Jose Renau *is a PhD candidate in the Computer Science Department at the University of Illinois at Urbana-Champaign. Contact him at renau@cs.uiuc.edu.*

Josep Torrellas *is a professor and Willett Faculty Scholar in the Computer Science Department at the University of Illinois at Urbana-Champaign. Contact him at torrellas@cs.uiuc.edu.*

a smaller configuration could have held the average number of instructions. The system can easily determine this condition from the sampled average occupancy statistics, the queue's current size, and the possible queue configurations. To prevent nonnegligible performance loss, the system upsizes the queue immediately when the overflow counter exceeds a preset threshold.

Another approach detects phase changes to trigger reconfiguration decisions. Balasubramonian's adaptive memory hierarchy[10] compares cache miss rates and the branch counts of the last two intervals. If the system detects a significant change in either, it assumes that a phase change has occurred. Ashutosh S. Dhodapkar and James E. Smith[11] improve on this approach by triggering a phase change in response to differences in working-set signatures—compact approximations that represent the set of distinct memory elements accessed over a given period. A significant difference in working-set signatures constitutes a phase change.

Still another approach triggers a resource upsizing only when a large enough increase in performance would be expected. This technique can be used to upsize an adaptive-issue queue.[12] A larger instruction window permits the stall time of instructions waiting in the window to be overlapped with the execution of additional ready instructions in the larger window. However, if this overlap time is not sufficiently large, upsizing the queue will provide little performance benefit. The system estimates the overlap time and uses it to trigger upsizing decisions.

Huang and colleagues proposed positional adaptation,[9] which uses the program structure to identify major program phases. Specifically, as the "Managing Multiple Low-Power Adaptation Techniques: The Positional Approach" sidebar describes, this approach uses either compile-time or runtime profiling to select an appropriate configuration for long-running subroutines. In the static approach, a profiling run measures the total execution time and the average execution time per invocation of each subroutine. Developers identify phases as subroutines with values for those quan-