Positional Adaptation of Processors: Application to Energy Reduction *

Michael C. Huang
Dept. of Electrical and Computer Engineering
University of Rochester
michael.huang@ece.rochester.edu

Jose Renau and **Josep Torrellas**
Dept. of Computer Science
University of Illinois at Urbana-Champaign
{renau,torrellas}@cs.uiuc.edu

Abstract

Although adaptive processors can exploit application variability to improve performance or save energy, effectively managing their adaptivity is challenging. To address this problem, we introduce a new approach to adaptivity: the *Positional* approach. In this approach, both the *testing* of configurations and the *application* of the chosen configurations are associated with particular code sections. This is in contrast to the currently-used *Temporal* approach to adaptation, where both the testing and application of configurations are tied to successive intervals in *time*.

We propose to use *subroutines* as the granularity of code sections in positional adaptation. Moreover, we design three implementations of subroutine-based positional adaptation that target energy reduction in three different workload environments: embedded or specialized server, general purpose, and highly dynamic. All three implementations of positional adaptation are much more effective than temporal schemes. On average, they boost the energy savings of applications by 50% and 84% over temporal schemes in two experiments.

1 Introduction

Processor adaptation offers a major opportunity to the designers of modern processors. Currently, many proposed architectural enhancements have the desired effect (e.g. improve performance or save energy) *on average* for the whole program, but have the opposite effect during some periods of program execution. If the processor were able to adapt as the application executes by dynamically activating/deactivating the enhancement, the average performance improvement or energy savings would be higher.

Perhaps the area where adaptive processors have been studied the most is the low-power domain — this is why we focus the analysis in this paper on this area. In this case, researchers have proposed various architectural Low-Power Techniques (LPTs) that allow general-purpose processors to save energy, typically at the expense of performance (e.g. [1, 2, 4, 8, 17, 19]). Examples of such LPTs are cache reconfiguration and issue-width changes. By activating these LPTs dynamically, processors can be more effective. Some of the more advanced proposals for adaptive processors combine several LPTs [7, 12, 13, 15, 21].

*This work was supported in part by the National Science Foundation under grants EIA-0081307, EIA-0072102, and CHE-0121357; by DARPA under grant F30602-01-C-0078; and by gifts from IBM and Intel.

Unfortunately, controlling processor adaptation effectively is challenging. Indeed, an adaptive processor with multiple LPTs needs to make the twin decisions of when to adapt the hardware and what specific LPTs to activate. These decisions are usually based on testing a few different configurations of the LPTs and identifying which ones are best, and when.

Nearly all existing proposals for adaptive systems follow what we call a *Temporal* approach to adaptation [1, 2, 4, 6, 7, 8, 9, 12, 15, 19, 21]. In this case, both the testing (or exploration) for the best configuration and the application of the chosen configuration are tied to successive intervals in *time*. Specifically, to identify the best configuration, the available configurations are typically tested back-to-back one after another. Moreover, once the testing period is over, the adaptation decisions to be made at the beginning of every new interval are based on the behavior of the most recent interval(s). The rationale behind these schemes is that the behavior of the code is largely stable across successive intervals.

In this paper, we introduce a new approach to adaptation: the *Positional* approach. In this case, both the testing for the best configuration and the application of the chosen configuration are associated with *position*, namely with particular code sections. To identify the best configuration, the available configurations are tested on different invocations of the *same* code section. Once the best configuration for a code section is identified, it is kept for later use when that *same* code section is invoked again. The rationale behind this approach is that the behavior of the code is largely stable across invocations of the same section. Note also that, with this approach, we can optimize the adaptations *globally* across the whole program.

To combine ease of implementation and effectiveness, we propose to use *subroutines* as the granularity of code sections in positional adaptation. Moreover, we propose three implementations of subroutine-based positional adaptation that are generic, easy to implement, and effective. Each implementation targets a different workload environment: embedded or specialized server, general purpose, and highly dynamic. Our results show that all three implementations of subroutine-based positional adaptation are much more effective than temporal schemes. On average, they boost the energy savings of applications by 50% and 84% over temporal schemes in two experiments.

This paper is organized as follows: Section 2 describes in more detail subroutine-based positional adaptation; Section 3 presents our three different implementations; Section 4 discusses our evaluation environment; Section 5 evaluates the implementations; Section 6 discusses related work, and Section 7 concludes.

2 Subroutine-Based Positional Adaptation

This paper proposes the *Positional* approach to adaptation, in contrast to the currently-used *Temporal* approach [1, 2, 4, 6, 7, 8, 9, 12, 15, 19, 21]. The fundamental difference between them is their different approach to exploiting program behavior repetition: the temporal approach exploits the similarity between *successive intervals* of code in dynamic order, while the positional approach exploits the similarity between different *invocations* of the same code section.

These two approaches differ on how they test the configurations to identify the best one, and on how they apply the best configuration. Specifically, temporal schemes typically test several configurations in time succession. Consequently, each configuration is tested on a different section of the code, which may have a different behavior. This increases the inaccuracy of the calibration. Moreover, once the testing period is over, the adaptation decisions to be made at the beginning of every new interval are based on the behavior of the most recent interval(s). As a result, if the code behavior changes across intervals, the configuration applied will be non-optimal.

In positional schemes, instead, we associate both the testing of configurations and the application of the chosen one with *position*, that is, with a particular code section. To determine the best configuration for a code section, different configurations are tested on different executions of the same code section. Once the best configuration is determined, it is applied on future executions of the same code section.

Positional adaptation is based upon the intuition that program behavior at a given time is mostly related to the code that it is executing at that time. This intuition is also explored in [23]. Further, positional adaptation has the advantage that, if we can estimate the relative weight of each code section in the program, we can optimize the adaptations *globally* across the program: each configuration is activated in the code sections where it has the greatest benefit compared to all other sections and all other configurations available, all subject to a maximum cost (e.g. slowdown) for the whole program.

2.1 Granularity of Code Sections

Before applying configurations on code sections, we need to determine the granularity of these sections. Code sections used in positional adaptation should satisfy three conditions: capture homogeneous behavior within a section, capture heterogeneous behavior across contiguous sections, and be easy to support.

In this paper, we propose to use the major subroutines of the application as code sections for adaptation. Intuitively, choosing subroutines is likely to satisfy the first two conditions. Indeed, a subroutine often performs a single, logically distinct operation. As a result, it may well exhibit a behavior that is roughly homogeneous and different from the other subroutines. Later in the paper, we show data that suggest that, on average, code behavior is quite homogeneous within a subroutine (Section 5.2), and fairly heterogeneous across subroutines (Section 5.3.2).

Using subroutines also eases the implementation in many ways. First, most subroutine boundaries are trivial to find, as they are marked by call and return instructions. Second, most applications are structured with subroutines.

In our proposal, each code section is constructed with one of the major subroutines of the application plus all the minor subroutines that it dynamically calls. Code sections can nest one another. The remainder of the program, which is the root (*main*) subroutine and the minor subroutines called from *main*, also forms one code section. This section is usually unimportant: on average, it accounts for about 1% of the execution time in our applications. It is possible that, in some applications, this section may have a significant weight. In that case, we can extend our algorithm to subdivide this section to capture behavior variability. For the applications that we study, this is unnecessary.

It is possible that a given subroutine executes two logically distinct operations, or that it executes completely different code in different invocations. Our algorithms do not make any special provision for these cases and still obtain good results (Section 5).

Finally, there are other choices for code sections, such as fixed-sized code chunks (e.g. a page of instructions) or finer-grained entities such as loops. However, they all have some drawbacks. Specifically, for fixed-sized code chunks, the boundaries are arbitrary and do not naturally coincide with behavior changes. On the other hand, using fine-grained or sophisticated entities may involve higher time or energy overheads. Moreover, there is some evidence that using finer-grained entities only provides fairly limited improvements over using subroutine-based sections [11, 18].

3 Implementing Subroutine-Based Positional Adaptation

We present three different implementations of subroutine-based positional adaptation. They differ on how many of the adaptation decisions are made statically and how many are made at run time. Specifically, we call *Instrumentation* (I) the selection of *when* to adapt the processor, and *Decision* (D) the selection of *what* LPTs to activate or deactivate at those times. Then, each selection can be made *Statically* (S) before execution or *Dynamically* (D) at run time. Each of the three implementations targets a different workload environment, which we label as embedded or specialized-server, general purpose, and highly dynamic (Figure 1).

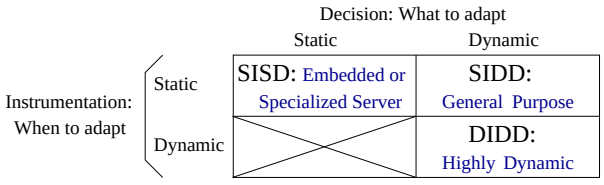

Figure 1. Different implementations of subroutine-based positional adaptation and workload environments targeted.

In general, as we go from *Static Instrumentation and Static Decision* (SISD) to SIDD, and then to DIDD, the adaptation process becomes increasingly automated and has more general applicability. However, it also requires more run-time support and has less global information. Note that there is no DISD environment because the decisions on LPT activation or deactivation cannot be made prior to deciding on the instrumentation points.

We want implementations that are generic, flexible to use, and simple. In particular, they should be able to manage any number of dynamic LPTs. Moreover, to trigger adaptations, we prefer not to use any LPT-specific metrics such as cache miss rate or functional unit utilization. There are two reasons for this. First, it is hard to

cross-compare the impact of two different LPTs using two different metrics. Second, such metrics need empirical thresholds that are often application-dependent.

While positional adaptation can be used for different purposes, here we will use it to minimize the energy consumed in the processor subject to a given tolerable application slowdown (*slack*). We assume that the processor provides support to measure energy consumption. While energy counters do not yet exist in modern processors, it has been shown that energy consumption could be estimated using existing performance counters [16]. In the following, we present each of our three implementations in turn.

3.1 Static Instrumentation & Decision (SISD)

In an embedded or specialized-server environment, we can use off-line profiling to identify the important subroutines in the application, and to decide what LPTs to activate or deactivate at their entry and exit points.

3.1.1 Instrumentation Algorithm

A single off-line profiling run is used to identify the major subroutines in the application. For a subroutine to qualify as major, its contribution to the total execution time has to be at least th_{weight}, and its average execution time per invocation at least th_{grain}. The reason for the latter is that adaptation always incurs overhead, and thus very frequent adaptation should be avoided. We instrument entry and exit points in major subroutines. At run time, minor subroutines will be dynamically included as part of the closest major subroutine up the call graph. Finally, recall that the remaining *main* code in the program also form one "major subroutine".

To reduce overhead, we use several optimizations. For example, we create a wrapper around a recursive subroutine and only instrument the wrapper. Also, if a subroutine is invoked inside a tight loop, we instrument the loop instead.

3.1.2 Decision Algorithm

We perform off-line profiling of the application to determine the impact of the LPTs. Consider first the case where each LPT only has two states (on and off), and LPTs do not interfere with each other. In this case, if the processor supports n LPTs, we perform $n+1$ profiling runs: one run with each LPT activated for the whole execution, and one run with no LPT activated. In each run, we record the execution time and energy consumed by each of the instrumented subroutines. Consider subroutine i and assume that E_i and D_i are the energy consumption and execution time (delay), respectively, of all combined invocations of the subroutine when no LPT is activated. Assume that when LPT_j is activated, the energy consumed and execution time of all invocations of the subroutine is E_{ij} and D_{ij}, respectively. Thus, the impact of LPT_j on subroutine i is ΔE_{ij} and ΔD_{ij}, where $\Delta E_{ij} = E_i - E_{ij}$ and $\Delta D_{ij} = D_{ij} - D_i$. These values are usually positive, since LPTs tend to save energy and slow down execution.

Once we have ΔE_{ij} and ΔD_{ij} for a subroutine-LPT pair, we compute the *Efficiency Score* of the pair as:

$$\begin{cases} -1 & \text{if } \Delta E_{ij} \leq 0 \quad \text{; increases energy consumed} \\ +\infty & \text{if } \Delta E_{ij} > 0 \ \& \ \Delta D_{ij} \leq 0 \quad \text{; saves energy, speeds up} \\ \frac{\Delta E_{ij}}{\Delta D_{ij}} & \textit{Otherwise} \quad \text{; saves energy, slows down} \end{cases}$$

The efficiency score indicates how much energy a pair can save per unit time increase, allowing direct comparisons between different pairs. High, positive values indicate more efficient tradeoffs. Subroutine-LPT pairs that both save energy and speed up the application are very desirable; pairs that increase the energy consumed are undesirable.

Once the results of all subroutine-LPT pairs are obtained, we sort them in a *Score Table* in order of decreasing efficiency score. Each row in the table includes the accumulated slowdown, which is the sum of the ΔD_{ij} of all the pairs up to (and including) this entry. This accumulated slowdown is stored as a percentage of total execution time. This table is then included in the binary of the application, and will be dynamically accessed at run time from the instrumentation points identified above. Note that, in each production run of the application, a tolerable slack for the application will be given. That slack will be compared at run time to the accumulated slowdown column of the table, and a cut-off line will be drawn in the table at the point where the slowdown equals the slack. Pairs in the table that are below the cut-off line are not activated in that run.

Simple extensions handle the case when an LPT has multiple states. Briefly, we perform a profile run for each configuration and record ΔE_{ij} and ΔD_{ij}. Since two such configurations cannot be activated concurrently on the same subroutine, if we select a second configuration, we need to "reverse" the impact of the first configuration on the score table. This effect is achieved by subtracting in the table the impact of one configuration from that of the next most efficient configuration of the same subroutine-LPT pair. In the case where an LPT has too many configurations, the algorithm chooses to profile only a representative subset of them. Alternatively, it could find the best configurations through statistical profiling [6].

When two LPTs interfere with each other or are incompatible in some ways, the algorithm simply combines them into a single LPT that takes multiple states. Some of these states may have ΔE_{ij} and ΔD_{ij} that are not the simple addition of its component LPTs'; other states may be missing due to incompatibility. The resulting multi-state LPT is treated as indicated above.

Finally, we assume that the effect of an LPT on a subroutine is largely independent of what LPTs are activated for *other* subroutines.

3.2 Static Instrum. & Dynamic Decision (SIDD)

In a general-purpose environment, it may be unreasonable to require so many profiling runs. Consequently, in SIDD, only the Instrumentation algorithm is executed off-line. It needs a single profiling run to identify the subroutines to instrument and their weight. The Decision algorithm is performed during execution, using code included in the binary of the application.

The Decision algorithm runs in the first few invocations of the subroutines marked by the Instrumentation algorithm. Consider one such subroutine. To warm up state, we ignore its first invocation in the program. In the second invocation, we record the number of instructions executed, the energy consumed, and the time taken. Then, in each of the n subsequent invocations, we activate one of the n LPTs, and record the same parameters. When a subroutine has gone through all these runs, our algorithm computes the efficiency scores for the subroutine with each of the LPTs, and inserts them in the sorted score table. With this information, and the weight of the

subroutine as given by the Instrumentation algorithm, the system recomputes the new cut-off line in the score table. At any time in the execution of a program, the entries in the score table that are above the cut-off line are used to trigger adaptations in the processor.

The fact that the Decision algorithm runs on-line requires that we change it a bit relative to that in Section 3.1.2. One difficulty is that the efficiency score for a subroutine-LPT pair is now computed based on a single invocation of the subroutine. To be able to compare across invocations of the same subroutine with different numbers of instructions, we normalize energy and execution time to the number of instructions executed in the invocation. Thus, we use Energy Per Instruction (EPI) and Cycles Per Instruction (CPI). Furthermore, the ratio of energy savings to time penalty used in the efficiency score (now $\frac{\Delta EPI_{ij}}{\Delta CPI_{ij}}$) is too sensitive to noise in the denominator that may occur across invocations of the same subroutine. Consequently, we use an efficiency score that is less subject to noise, namely $\frac{EPI_i * CPI_i}{EPI_{ij} * CPI_{ij}}$. The values in the numerator correspond to subroutine i when no LPT is activated, while the values in the denominator correspond to subroutine i when LPT_j is activated. As usual, high efficiency scores are better.

A second difficulty in the on-line Decision algorithm occurs when a subroutine has only a few, long invocations. In this case, the algorithm may take too long to complete for that subroutine. To solve this problem, our system times out when a subroutine has been executing for too long. At that point, our system assumes that a new invocation of the same subroutine is starting and, therefore, it tests a new LPT.

The computation of efficiency scores and the updates to the table only occur in the first few invocations of the subroutines. In steady state, the overhead is the same as in SISD: at instrumentation points, the algorithm simply checks the table to decide what LPTs to activate. Appendix A briefly discusses the overheads involved.

3.3 Dynamic Instrumentation & Decision (DIDD)

We now consider an environment where the application binaries remain *unmodified*. In this case, both Instrumentation and Decision algorithms run on-line. In practice, DIDD is useful in highly-dynamic environments, such as internet workloads where programs are sometimes executed only once, or in just-in-time compilation frameworks. Moreover, it is also useful when it is too costly to modify the binary.

DIDD needs three micro-architectural features. The first one dynamically identifies the important subroutines in the application with low overhead. The second one automatically activates the correct set of LPTs for these subroutines. The third one automatically redirects execution from the first few invocations of these subroutines to a dynamically linked library that implements the Decision algorithm.

3.3.1 Identifying Important Subroutines

We propose a simple micro-architecture module called the *Call Stack* (Figure 2). On a subroutine call, the Call Stack pushes in an entry with the subroutine ID and the current readings of the time and energy counters. For ID, we use the block address of the first instruction of the subroutine. On a subroutine return, the Call Stack pops out an entry. If the difference between the current time and the entry's time is at least th_{invoc}, the subroutine is considered important. As a result, the hardware saves its ID in a fully-associative table of important subroutines called *Call Cache* (Figure 2).

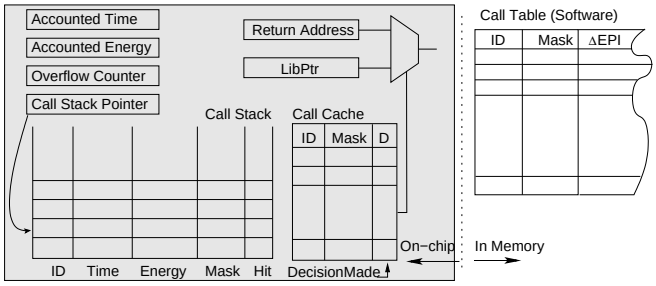

Figure 2. Support for DIDD. The shaded area corresponds to the proposed on-chip hardware. It occupies an insignificant area: about 0.29 mm^2 in 0.18 μm technology.

The Call Stack handles the nesting of important subroutines by subtracting the callee's execution time from the caller's. To support this case, we maintain an *Accounted Time* register that accumulates the cycles consumed by completed invocations of important subroutines up to the current time (Figure 2). With this support, when we push/pop an entry into/from the Call Stack, the current time is taken to be the wall clock time minus the Accounted Time. Moreover, when we finish processing the popping of an important subroutine, we add its time contribution to the Accounted Time. As a result, its contribution will not be erroneously assigned to its caller. Figure 2 also includes an *Accounted Energy* register that is used in the same way.

If the Call Stack overflows, we stop pushing entries. We maintain an *Overflow Counter* to count the number of overflow levels. When the counter falls back to zero, we resume operating the Call Stack.

3.3.2 Activating LPTs & Invoking Decision Algorithm

The hardware must perform two operations at the entry and exit points of these important subroutines. In steady state, it must activate or deactivate LPTs; in the first few invocations of these subroutines, it must redirect execution to the library that implements the Decision algorithm. We consider these two cases in turn.

In steady state, each of the subroutines in the Call Cache keeps a mask with the set of LPTs to activate on invocation (*Mask* field) and a *DecisionMade (D)* bit set. When an entry is pushed into the Call Stack, the hardware checks the Call Cache for a match. If the entry is found, the LPTs in the Mask field are activated (after saving the current mask of LPTs in the Call Stack) and the *Hit* bit in the Call Stack is set. The Hit bit will be useful later. Specifically, when the entry is popped from the Call Stack, if the Hit bit is set, the Call Cache is checked. If the corresponding D bit is set, the hardware simply restores the saved mask of activated LPTs.

In the first few invocations of an important subroutine, the subroutine must run under each of the LPTs in sequence, and the result must be analyzed by the Decision algorithm (Section 3.2). During this period, the corresponding entry in the Call Cache keeps its D bit clear and Mask indicates the single LPT to test in the next invocation. As usual, when an entry is pushed into the Call Stack, the hardware checks the Call Cache and, if the entry is found, the LPT

in the Mask is activated. When the entry is popped out of the Call Stack, if the Hit bit is set and the corresponding D bit is clear, the hardware redirects execution to the Decision algorithm library.

This redirection is done transparently. We modify the branch unit such that when a subroutine return instruction is executed, the hardware checks if the returning subroutine is in the Call Cache and its D bit is clear. If so, the return instruction is replaced by a jump to the entry point of the Decision library code. This entry point is stored in the $LibPtr$ special register (Figure 2). The library runs the Decision algorithm as in Section 3.2: it reads the time and energy consumed by this subroutine-LPT pair from the Call Stack, computes the efficiency score, and updates the score table. In its operation, the Decision algorithm keeps its state in a software data structure in memory called *Call Table* (Figure 2). Once finished, the library issues a return, which redirects execution back to the caller of the important subroutine. This is feasible because the original return address was kept in place, in its register or stack location. In addition, during the redirection to the library, the RAS (Return Address Stack) was prevented from adjusting the pointer and becoming misaligned. With this support, we have effectively delayed the return from the important subroutine by invoking the Decision algorithm seamlessly and with little overhead.

Before the Decision library returns, it updates the Mask for this subroutine in the Call Cache to prepare for the next invocation. If it finds that it has tested all the LPTs for this subroutine, it computes the steady state value for the Mask. Then, it sets the Mask to that value and sets the D bit. Future invocations of this subroutine will not invoke the library anymore.

It is possible that capacity limitations force the displacement of an entry from the Call Cache. There is no need to write back any data. When the corresponding subroutine is invoked again, the Call Cache will miss and, on return from the subroutine, the Decision library will be invoked. At that point, the Decision library will copy the Call Table entry for the subroutine to the Call Cache, effectively restoring it to its old value.

Overall, the proposed hardware is fairly modest. We use CACTI [24] to estimate that the hardware in Figure 2 takes 0.29 mm^2 in 0.18 μm technology. This estimate assumes 32 entries for the Call Cache and Call Stack.

3.3.3 Decision Algorithm

The Decision algorithm used is similar to the one for SIDD. The only difference is that subroutines are now identified on the fly and, therefore, their contribution to the total execution time of the program is unknown. Consequently, the algorithm needs to make a rough estimation. It assumes that all the important subroutines have the same weight: 10%. The inaccuracy of this assumption does not affect the ranking of the subroutine-LPT pairs in the score table. However, it affects the location of the cut-off line in the table. As a result, it is now more challenging to fine tune the total program slowdown to be close to the allowed slack.

We have attempted to use more accurate, yet more costly ways of estimating the contribution of each subroutine. Specifically, we have added support for the system to continuously record and accumulate the execution time of each subroutine. We can then recompute the weights of all the subroutines periodically and update the cut-off line in the score table. From the results of experiments not presented here, we find it hard to justify using these higher-overhead schemes.

3.4 Tradeoffs

Table 1 summarizes the tradeoffs between our three implementations. SISD is the choice when the off-line profiling effort can be amortized over many runs on the platform where profiling occurred. SISD has complete global information of the program and, therefore, can make well-informed adaptation decisions. The only source of inaccuracy is the difference between the profiling and production input sets. Finally, SISD has minimal run-time overhead.

	Pros	Cons	Domain
SISD	Global information of the program. Minimal run-time overhead	Requires many off-line profiling runs. Profiling has to be on target platform	Embedded systems and specialized servers
SIDD	Single performance-only profiling run. Profiling is partially platform independent	Run-time overhead at start-up. Partial information. Limited profiling	General purpose
DIDD	No off-line profiling	Same as SIDD. Extra micro-architectural support	Unavailable off-line profiling: e.g. dynamically generated binary

Table 1. Tradeoffs between the different implementations of subroutine-based positional adaptation.

SIDD has a wider applicability. It is best for environments where software is compiled for a range of adaptive architectures, each of which may even have a different set of LPTs. In this case, the ranking of adaptations is not included in the application code. It is obtained on-line, by measuring the impact of each LPT on the target platform, while the application is running. The only off-line profiling needed is to identify important subroutines and their execution time weight. This does not need to be carried out on the exact target platform. However, SIDD has several shortcomings. First, it incurs run-time overhead, partly due to the application of inefficient adaptations during the initial period of LPT testing. Second, some decisions on what adaptations to apply are necessarily sub-optimal, since they are made before testing all subroutine-LPT pairs. Finally, SIDD relies on the first few invocations of each subroutine to be representative of the steady state, which may not be fully accurate.

DIDD has the widest applicability. It works even when no off-line profiling is available. This is the case when binaries are dynamically generated, or in internet workloads where programs are often executed only once. The shortcomings of DIDD are the micro-architectural support required and all the shortcomings of SIDD with higher intensity. In particular, identifying the important subroutines on-line is challenging and, unless it is done carefully, may lead to high overheads.

4 Evaluation Environment

4.1 Architecture and Algorithm Parameters

To evaluate positional adaptation, we use detailed execution-driven simulations. The baseline machine architecture includes a 6-issue out-of-order processor, two levels of caches, and a Rambus-based main memory (Table 2). The processor can be adapted using three LPTs, which are described in Section 4.3. The simulation models resource contention in the entire system in detail, as well as all the overheads in our adaptation algorithms.

We compare our implementations of positional adaptation to three existing temporal adaptation schemes, which we call

Processor			
Frequency: 1GHz			Branch penalty: 8 cycles (min)
Technology: 0.18μm			Up to 1 taken branch/cycle
Voltage: 1.67V			RAS entries: 32
Fetch/issue width: 6/6			BTB: 2K entries, 4-way assoc
I-window entries: 96			Branch predictor:
Ld/St units: 2			gshare
Int/FP/branch units: 4/4/1			entries: 8K
MSHRs: 24			TLB: like MIPS R10000
Cache	L1	L2	Bus & Memory
Size:	32KB	512KB	FSB frequency: 333MHz
RT:	3 cycles	12 cycles	FSB width: 128bit
Assoc:	2-way	8-way	Memory: 2-channel Rambus
Line size:	32B	64B	DRAM bandwidth: 3.2GB/s
Ports:	2	1	Memory RT: 108ns

Table 2. Baseline architecture modeled. MSHR, RAS, FSB and RT stand for Miss Status Handling Register, Return Address Stack, Front-Side Bus, and Round-Trip time from the processor, respectively. Cycle counts are in processor cycles.

DEETM′, *Rochester*, and *Rochester′*. The parameter values used for all the schemes are shown in Table 3. th_{weight}, th_{grain}, and th_{invoc} are set empirically.

Algorithm	Parameter Values
SISD and SIDD	th_{weight} = 5%; th_{grain} = 1,000 cyc; LPT (de)activation overhead: 2-10 instr
DIDD	th_{invoc} = 256 cyc; Call Stack: 32 entries, 9B/entry, 56 pJ/access; Call Cache: 32 entries, full-assoc, 4B/entry, 66 pJ/access
DEETM′	Microcycle = 1, 10, 100 μs; Macrocycle = 1,000 microcycles
Rochester	Parameter values as in [3, 4], e.g. basic interval = 100 μs
Rochester′	Rochester with the tuning optimization in [6]

Table 3. Parameter values used for the positional and temporal adaptation schemes.

Consider the positional schemes first. Under static instrumentation (SISD and SIDD), we filter out subroutines whose average execution time per invocation is below th_{grain}; under dynamic instrumentation (DIDD), we filter out any invocation that takes less than th_{invoc}. These two thresholds have different values because they have slightly different meanings. The table also shows the values of the main instruction and energy overheads of the schemes; they are discussed in Appendix A. The energy numbers are obtained with the models of Section 4.2.

DEETM′ is an enhanced version of the DEETM Slack algorithm in [12]. In this algorithm, the set of active LPTs is re-assessed at constant-sized time intervals called *macrocycles*. At the beginning of a macrocycle, each different configuration is tested for one *microcycle*. After all configurations have been tested in sequence, the algorithm decides what configuration to keep for the remainder of the macrocycle. This algorithm is more flexible than the one in [12]: the latter assumes a fixed effectiveness rank of LPTs, which limits the set of configurations that it can apply. In [12], a microcycle is 1,000 cycles and a macrocycle is 1,000 microcycles. We examine three different microcycles, namely 1, 10, and 100 μs. We call the schemes DEETM′1, DEETM′10, and DEETM′100, respectively.

Rochester is the scheme in [4]. The algorithm uses a basic interval. Initially, each configuration is tested for one interval. After that, the best configuration is selected and applied. From then on, at the end of each interval, the algorithm compares the number of branches and cache misses in the interval against those in the previous interval. If the difference is within a threshold, the configuration is kept, therefore extending the effective interval. If the difference is over the threshold, the algorithm returns to testing the configurations. The algorithm uses several other thresholds. For our experiments, we start with the parameter values proposed by the authors [3], including a basic interval of 100,000 cycles. We then slightly tune them for better performance.

Rochester′ adds one enhancement proposed in [6] to the Rochester scheme. The enhancement appears when the difference between the branches and misses in one interval and those in the previous one is above the threshold. At that point, Rochester′ does not return to testing the configurations right away. The rationale is that it is best not to test configurations while the program goes through a phase change. The algorithm waits until the difference is below the threshold, which indicates that the change has stabilized. Then, the testing of configurations can proceed.

Overall, we consider temporal schemes with fixed-size intervals (DEETM′) and with variable-sized intervals (Rochester and Rochester′). Note that we do not choose the interval sizes so that all schemes have exactly the same average size, or they match the average size of the intervals in positional schemes. Instead, we use the parameter values as proposed by the authors (although we also slightly tune them to get better performance). With this approach, we hope to be fair and capture good design points for each scheme.

4.2 Energy Consumption Estimation

To estimate energy consumption, we incorporated Wattch [5] into our simulator. We enhanced Wattch in two ways. Recall that Wattch uses a modified version of CACTI [24] to model SRAM arrays. We have refined the modeling of such structures to address several limitations. Specifically, we enhanced the modeling of the sense amplifiers and the bitline swing for writes to make them more accurate. In addition, we always search for the SRAM array configuration that has the lowest energy consumption given the timing constraints.

For the energy consumption in the functional units, we used Spice models of the functional units of a simple superscalar core to derive the average energy consumed for each type of operation. We used results from [20] for more complicated functional units.

We compute the energy consumed in the *whole* machine, including processor, instruction and data caches, bus, and main memory. To model the energy consumed in the memory, we use Intel's white paper [14]. For example, from that paper, one memory channel operating at full bandwidth and its memory controller consume 1.2W.

4.3 Adaptive Low-Power Techniques (LPTs)

We model an adaptive processor with three LPTs that can be dynamically activated and deactivated (Figure 3). These LPTs are: a filter cache [17], a phased cache [10] mode for the L1 data cache, and a slave functional unit cluster that can be disabled. We choose these LPTs because they are well understood and target some major sources of energy consumption in processors. Note that our adaptation algorithms are very general and largely independent of the LPTs used – we simply choose these three LPTs as *examples*.

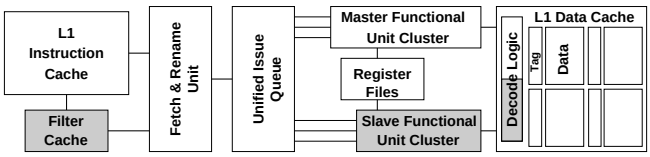

Figure 3. Pipeline of the adaptive processor considered. The shaded areas show the LPTs.

Instruction Filtering (IFilter). The instruction cache hierarchy has a 1-Kbyte filter cache [17]. If it is deactivated, instructions are fetched from the L1 instruction cache; otherwise, the processor checks the filter cache first. Using the filter cache usually saves energy because each hit consumes very little energy. The reason is that the filter cache is small, direct-mapped, and does not require a TLB check because it is virtually tagged. However, the code may run slower because of frequent misses in the small filter cache, which then access L1. Interestingly, this LPT may sometimes speed up the code: when the working set is small enough to fit in the filter cache, the faster access allows quicker recovery on branch mispredictions.

We do not maintain inclusion between the filter and L1 instruction caches. A read access to the filter cache takes 1 cycle and consumes 386 pJ, compared to 2 cycles and 2022 pJ to access the fully-pipelined L1. We model the filling of cold caches.

Phased Cache Mode (PCache). A phased cache is a set-associative cache where an access first activates only the tags [10]. If there is a match, only the matching data array is subsequently activated, reducing the amount of bitline activity and sense amplification in the data array. Consequently, a phased cache saves energy at the cost of extra delay.

In our processor, the 2-way set-associative L1 data cache can work as a normal or as a phased cache. Based on our analysis, in phased mode, a cache hit consumes 974 pJ, a 45% reduction over the 1763 pJ consumed in a normal mode hit. However, the hit takes two extra cycles to complete. Cache misses save even more energy and do not add latency. Note that there is overhead in switching between the two modes. Specifically, when the phased mode is activated, the cache buffers the signal to the data array for two cycles. When the cache is restored to normal mode, the cache blocks for two cycles to drain the pipeline. All these transition overheads are fully modeled.

Reduced Number of ALU Units (RALU). Wide-issue processors typically have many functional units (FUs). Since few applications need all the FUs all the time, processors typically clock-gate unused FUs. Our Wattch-based simulator models the normal clock-gating of unused FUs by reducing the energy consumed by FUs to 10% of their maximum consumption when they are unused.

With the RALU LPT, we go beyond this reduction. In our processor, the FUs are organized into a master and a slave cluster. Each cluster has two FP, two integer, and one load/store unit. The master cluster also has a branch unit. When this LPT is deactivated, both FU clusters can be used, and clock gating proceeds as indicated above. When this LPT is activated, the slave FU cluster is made inaccessible. This allows us to save all the clock distribution energy in the slave cluster. As a result, we save most of the remaining 10% dynamic power in the FUs of the cluster. For multi-cycle FUs, we can only activate this LPT after the FU pipeline is drained. This effect is modeled in our simulations. Overall, this LPT can only have a modest energy-savings effect.

4.4 Applications

To assess positional adaptation on different kinds of workloads, we run multimedia, integer, and floating-point applications. In selecting these applications, we try to include a diverse set of high-level behaviors. In particular, we include programs where the average dynamic subroutine is very short (30 instructions in MCF) or very long (35,000 instructions in HYDRO). The applications are compiled with the IRIX MIPSPro compiler version 7.3 with -O2.

Table 4 lists the applications. Each application has an input set used for the off-line profiling runs (*Profiling*), and one for all other experiments (*Production*). Recall from Section 2 that positional adaptation has the advantage that it optimizes the adaptations *globally*: each configuration is activated in the globally best section of the program. Therefore, to fully demonstrate the effectiveness of positional adaptation, we need to simulate the applications from the beginning to the end. Unfortunately, the full *ref* SPEC input sets are too large for this. Consequently, as the production input sets for the SPEC applications, we use a reduced reference input set (*reduced ref*), which enables us to run the simulations to completion. With these inputs, simulations take from several hundred million to over 2.5 billion cycles. For all applications, we have verified that these reduced input sets running on our simulator produce similar cache and TLB miss rates as the *ref* inputs running natively on a MIPS R12000 processor. We have also verified that the relative weight of each subroutine does not change much. For additional verification, one experiment in Section 5.3 compares executions with *reduced ref* and *ref* input sets.

Suite	Application	Profi ling Input	Production Input
SPECint2000	BZIP CRAFTY GZIP MCF PARSER	Test	Reduced ref
SPECfp95	HYDRO APSI	Test	Reduced ref
Multimedia	MP3D	128kbps joint	160kbps joint HQ
	MP3E	24kbps mono	128kbps joint

Table 4. Applications executed.

Due to space limitations, we do not show the breakdown of the energy consumed in the different components of our architecture as we run these applications. However, our results broadly agree with other reports [5]. As expected, energy consumption is widely spread over many components. Therefore, it is unlikely that a single LPT can save most of the energy.

5 Evaluation

To evaluate subroutine-based positional adaptation, we first characterize our algorithms (Section 5.1), then evaluate their impact (Section 5.2), and finally show why the subroutine is a good granularity (Section 5.3).

5.1 Characterization

Table 5 shows the result of running our static and dynamic Instrumentation algorithms. Recall that our algorithms identify the major subroutines in the code and instrument their entry and

exit points. At run time, non-major subroutines are automatically lumped in with their caller major subroutines. Also, the *main* code in the program plus any non-major subroutines that it dynamically calls form one other "major subroutine". For comparison, the table also shows data on all the subroutines in the applications.

Applic	Stat Instrum		Dyn Instrum		All Subroutines in Application		
						Size/Invocation	
	N	T (μs)	N	T (μs)	N	Time (ns)	Instruc
APSI	14	8.8	19	6.1	93	200.1	272.3
BZIP	4	2612.0	5	275.9	54	71.4	108.5
CRAFTY	5	2.6	10	11.6	113	49.6	58.9
GZIP	6	2368.0	11	955.1	69	152.7	202.2
HYDRO	8	2530.0	15	407.6	111	51349.4	34784.1
MCF	3	20.3	6	5.0	50	50.4	28.4
MP3D	5	3.5	9	5.4	65	928.9	1411.5
MP3E	7	35.4	24	8.5	151	178.5	280.3
PARSER	7	28.7	62	976.4	267	37.6	39.2
Average	6.5	845.5	17.9	294.6	108.1	5890.9	4131.7

Table 5. Characterizing the static and dynamic Instrumentation algorithms. In the table, N is the number of major subroutines, while T is the time between instrumentation points.

The data shows that our algorithms identify only a handful of major subroutines to drive LPT activation/deactivation. On average, the number is about 7 and 18 for the static and dynamic algorithms, respectively. This suggests that the structures needed to manage adaptation information are small (e.g. a 32-entry Call Cache in DIDD). Static and dynamic algorithms select a different number of subroutines because they work differently.

The table also shows the average time between instrumentation points, as they are found dynamically at run time. The time ranges from a few μs to thousands of μs. This is the average time between potential adaptations. Within one algorithm, this time varies a lot across applications, which indicates a range of application behavior. On average, these time values are roughly of the same order of magnitude as the intervals in temporal schemes (Table 3). They are long enough to render various overheads negligible (Appendix A).

We now characterize the activation of our LPTs. In a series of experiments, we activate each LPT_j on each subroutine i and record the resulting total energy saved in the program (ΔE_{ij}) and total program slowdown (ΔD_{ij}). With these values, we compute the efficiency score (Section 3.1.2) of each subroutine-LPT pair. We then *rank* the pairs from higher to lower efficiency score and accumulate the total energy and total delay. The result is the *Energy-Delay Tradeoff* curve of the application.

Figure 4-(a) shows such a curve for BZIP. The origin in the figure corresponds to a system with no activated LPT. As we follow the curve, we add the contribution of subroutine-LPT pairs from most to least efficient, accumulating energy reduction (Y-axis), and execution slowdown (X-axis). Finally, the last point of the curve has all the LPTs activated all the time. As an example, in Figure 4-(a), we show the contribution of a subroutine-LPT pair that saves ΔE_{ij} and slows down the program ΔD_{ij}.

The curve can be divided into three main regions. In the *Always apply* region, the curve travels left and up. This region contains subroutine-LPT pairs that both save energy and speed up the program. An example may be a filter cache in a small-footprint subroutine with many mispredicted branches. The filter cache satisfies the average access faster and with less energy than the ordinary cache. Overall, we always enable the pairs in this region.

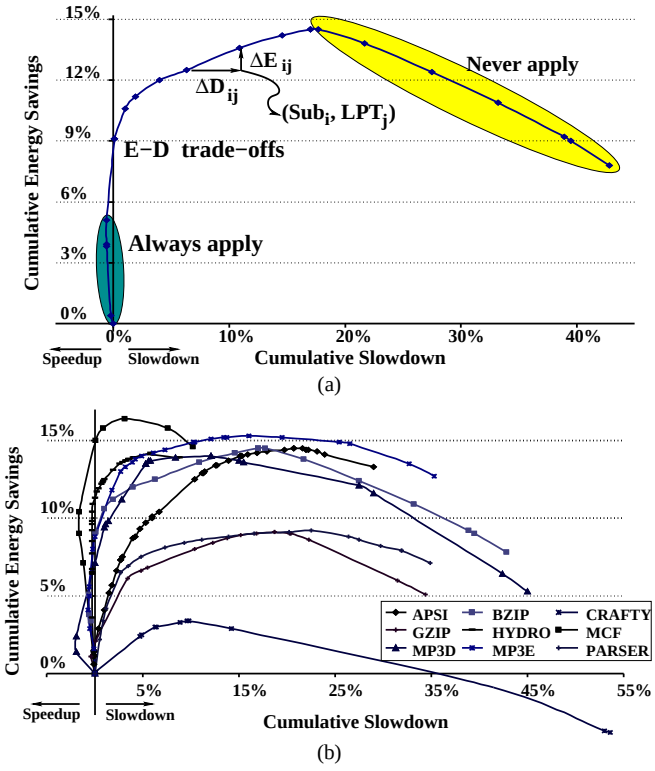

Figure 4. Energy-delay tradeoff curve for BZIP (a) and for all the applications (b).

In the *E-D trade-offs* region, the curve travels right and up. This region contains pairs that save energy at the cost of slowing down the program. This is the most common case. Starting from the left, we apply the pairs in this region until the accumulated application slowdown reaches the allowed slowdown (slack).

In the *Never apply* region, the curve travels right and down. This region contains pairs that increase energy consumption and slow down the program. These pairs should not be applied.

Figure 4-(b) shows the curves for all the applications. We see that all the applications exhibit a similar behavior. The figure also shows that if all LPTs are activated indiscriminately all the time (rightmost point), the result is a very sub-optimal operating point.

Finally, we characterize how our algorithms use the three LPTs. Figure 5 shows the percentage of time that each LPT is activated for the different applications. Due to space constraints, we only show data for SISD. The figure shows the results for a target application slack set to 0.5% and 5% of the application execution time. Overall, the figure shows that our algorithm activates all three LPTs for a significant portion of the time in many applications. Moreover, LPT selection varies across applications.

5.2 Effectiveness of Positional Adaptation

To evaluate the effectiveness of positional adaptation, we perform two experiments, where we want to save as much energy as possible while trying to limit the performance penalty to no more than 0.5% or 5.0%. We compare our three positional schemes (SISD, SIDD, and DIDD) to the temporal algorithms in Table 3 (DEETM′1, DEETM′10, DEETM′100, Rochester, and Rochester′).

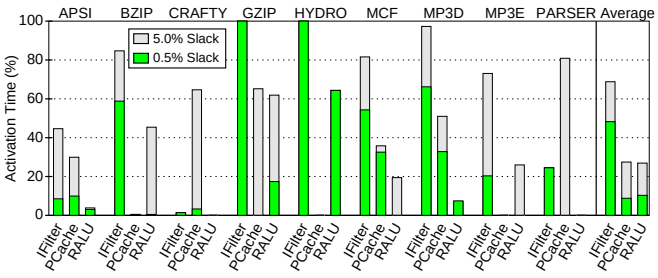

Figure 5. Percentage of time that each LPT is activated in each application under SISD. The data corresponds to two experiments with different slacks.

For each algorithm, Figure 6 shows the reduction in energy in the machine (upper bars) and the increase in execution time of the application (lower bars). The energy includes the contribution of the processor, caches, bus, and memory. The bars are separated into two groups, corresponding to the 0.5% and 5.0% slack experiments. Each bar is the average of our nine applications.

Note that the energy reduction bars are normalized to the energy reduced by an *ideal* adaptation algorithm that serves as an upper bound for a given slack. This algorithm adapts the processor every 1,000 instructions based on *perfect* knowledge of the impact of each of our LPTs on these upcoming instructions. Moreover, the adaptation is overhead-free. We choose to show the bars relative to this ideal scheme rather than to simply show the fraction of energy reduced by each scheme. The reason is that the latter depends on how good *our LPTs* are as much as how good *our algorithms* are. Recall that our algorithms are general and largely independent of the LPTs used. With these LPTs, the ideal algorithm reduces energy use in the machine by 8.7% and 11.6% for the 0.5% and 5.0% slack experiments, respectively. Our bars show how close we get to this ideal reduction.

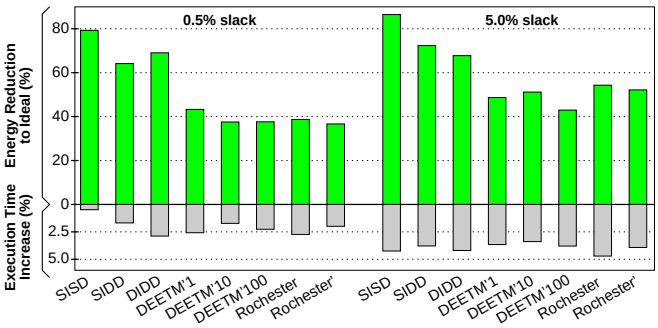

Figure 6. Energy reduction in the machine and program execution time increase for different algorithms.

Consider the 0.5% slack experiments first. In energy savings, the positional schemes are significantly more effective than the temporal ones. On average, positional schemes are 70% as effective as the ideal scheme, while temporal schemes are only 38% as effective. Positional schemes are more effective because they are able to predict code behavior more accurately. As discussed in Section 2, the accuracy is greater for two reasons: (1) in the testing period, they test all configurations on different invocations of the *same* code, and (2) in steady state, they make the adaptation decisions for an upcoming interval based on the behavior of a previous instance of the *same* interval.

Among positional schemes, SISD saves about 80% of the energy that the ideal scheme saves, and does not slow down the program much beyond the target slack. It is, therefore, the preferred scheme if it is possible to use it. Both SIDD and DIDD save less energy and mispredict past the 0.5% slack. In particular, DIDD mispredicts significantly, mostly because of lack of global information at run time. With such a slowdown, DIDD manages higher energy savings than SIDD. In normal conditions, we would expect the opposite.

For the 5% slack case, the positional schemes are again more effective than the temporal ones: on average, they save 75% of the energy that the ideal scheme saves, while temporal schemes save 50%. Among the positional schemes, there is a more gradual change in behavior. The smoother shape appears because positional schemes can now identify good subroutine-LPT pairs to apply more easily than in the 0.5% slack experiment. To see why, note that we want the pairs in Figure 4-(b) that are to the left of *X=5%* (instead of those to the left of *X=0.5%* in the 0.5% slack experiment). The wider range available lessens the impact of measurement inaccuracies, causing fewer selections of pairs beyond the target range, as it happened for DIDD in the 0.5% slack experiment. Overall, the differences in the resulting SISD, SIDD, and DIDD bars now broadly reflect the difference in accuracy between the schemes.

As for the temporal schemes, the differences in energy and slowdown between them appear to be modest. We note that each scheme has its own strengths. Specifically, Rochester and Rochester$'$ can vary the size of the interval between adaptations dynamically, which improves their effectiveness. On the other hand, the DEETM$'$ schemes have the ability to apply any given LPT for only a fraction of a macrocycle [12], if they estimate that full application would result in exceeding the slack. The result is that all the schemes have roughly similar effectiveness.

Overall, we derive two main conclusions. First, positional schemes are more effective than temporal ones. They boost the energy savings over temporal schemes by an average of 84% and 50% in the two experiments performed. Moreover, they are relatively more effective in the small slack experiment, where accurately selecting the best adaptation is harder.

The second conclusion results from the observation that the energy savings of the ideal and SISD schemes are quite close (on average, SISD saves 83% of ideal). Recall that the ideal scheme selects the best LPTs every 1,000 dynamic instructions without overhead, while SISD can only attempt to select at major subroutine boundaries. Such boundaries occur every several hundred μs on average (Table 5). Consequently, we infer that the behavior of the code executed inside each of these subroutines appears quite homogeneous to our LPTs.

5.3 Insights into Subroutine-Based Adaptation

Finally, we present data to help understand why subroutine-based positional adaptation is effective. Specifically, we discuss its accuracy in the testing (Section 5.3.1) and steady-state (Section 5.3.2) periods. We also discuss the influence of different input sets (Section 5.3.3). Appendix A discusses its overheads.

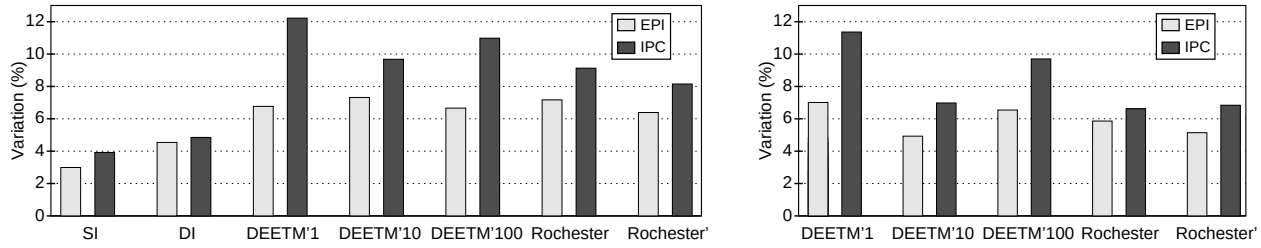

Figure 7. Variation of Energy Per Instruction (EPI) and IPC across testing intervals without actually applying any LPTs. Chart (a) uses the usual input sets for the applications, while Chart (b) changes the input sets of the SPEC applications to be *ref*.

5.3.1 Accuracy in the Testing Period

The accuracy of an adaptation algorithm is affected by how accurately the different configurations are calibrated during the testing period. Recall that, in that period, each configuration is activated for one interval. Then, the impact of the different configurations are compared to each other. Since each configuration is tested on a different interval, the more stable the code behavior is across these intervals, the more accurate this comparison is.

To estimate code stability across testing intervals, we measure the average energy per instruction (EPI) and IPC of each testing interval without actually applying any LPTs. Then, we compute the variation of these metrics in the testing period.

Figure 7 shows the resulting variation of the EPI and IPC for different algorithms. For positional adaptation, we consider the static (*SI*) and dynamic (*DI*) Instrumentation algorithms. For these algorithms, the testing intervals are the first few invocations of each major subroutine. For DEETM$'$, the testing intervals are the first few microcycles in each macrocycle. We compute the average on a macrocycle basis and then average out for all macrocycles. Finally, for Rochester and Rochester$'$, every phase change is followed by several testing intervals. Consequently, we compute the average on a phase-change basis and then average out for all phase changes.

The figure is divided in two parts. Consider first Figure 7-(a), which uses default parameters. We see that the subroutine-based positional algorithms have lower EPI and IPC variations. This suggests that they test configurations during more stable execution conditions and, therefore, achieve a higher accuracy in calibrating LPTs. The reason is that they test all configurations on the *same* code section.

Recall that to fully evaluate our positional schemes, we need to run applications to completion and, therefore, had to reduce the input sets for the SPEC applications. To see if using a bigger input set affects the results, Figure 7-(b) repeats the experiments using the *ref* input sets for the SPEC applications. The programs run for a window of 4.1 billion cycles after the initialization, and then stop. In this case, only the temporal schemes can be evaluated fairly. From the figure, we see that using the bigger input sets reduces the variation in the temporal schemes only slightly.

5.3.2 Accuracy in the Steady-State Period

Adaptation algorithms identify steady-state periods where a configuration is kept unchanged from one interval to the next. Code conditions are stable, and the algorithm predicts that the current configuration will have a similar impact in the next interval. Consequently, the steady-state accuracy of an algorithm will be greatest when the impacts of a configuration on two intervals that belong to the same steady state are most similar.

To estimate the accuracy, we identify, for each algorithm, the set of intervals that it considers to be in a given steady state. For positional algorithms, these are successive invocations of the same subroutine in steady state; for temporal algorithms, they are contiguous intervals not separated by phase changes. We then apply one LPT to all these intervals and record the changes in energy consumption and execution speed. Then, we compute the variation of this change across all these intervals. The smaller this variation is, the more accurate the scheme is in steady state. Finally, we average out all the steady states.

Figure 8 shows the variation for different algorithms. For brevity, we only show the average of all three LPTs. From the figure, we see that the subroutine-based positional algorithms have a lower variation. The impact of LPTs across intervals in steady state is more stable. Consequently, these algorithms can more accurately predict the impact of an LPT on an upcoming interval in steady state. This low variability is a result of executing the same code section.

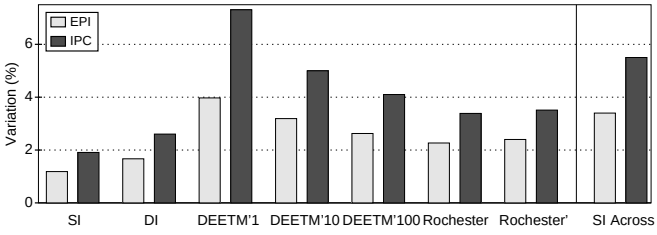

Figure 8. Variation of the impact induced by an LPT on steady-state intervals. The "SI Across" bars on the right show, for SI, the variation of the impact across different subroutines. In all cases, the data is the average for IFilter, PCache, and RALU.

For comparison, the "SI Across" bars on the right of Figure 8 show, for SI, the variation of the impact across different subroutines. We can see that, for SI, the variation across different subroutines is much higher than that across different invocations of the same subroutine. This data shows that code behavior across subroutines is relatively heterogeneous.

5.3.3 Influence of Different Input Sets

To gain insights into the influence of using different input sets for profiling and production runs in SI, we perform the following experiment. We run an application and measure the average impact of a given LPT on a given subroutine. We perform this experiment for four different input sets: *test*, *train*, *ref*, and *reduced ref*. Figure 9 shows the variation observed across these four runs. In the figure, the data is grouped by application, averaging out all the subroutines and all the LPTs.

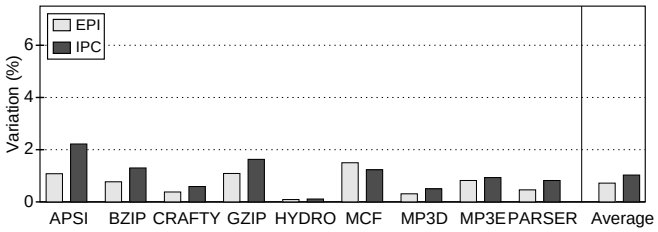

Figure 9. Variation of the average impact of an LPT on a subroutine across different input sets.

Overall, the average impact of an LPT on a subroutine is quite stable across different input sets. In fact, if we compare the figure to the SI bars in Figure 8, we see that the variation across different input sets is even smaller than the variation across invocations of the subroutine inside a single program execution. This suggests that, to predict the average impact of an LPT on a subroutine, it can be more accurate to use the *average* impact measured with a *training* input than to use the impact measured on *one* invocation using the *same* input. Therefore, using profiles is a viable solution for subroutine-based positional adaptation.

6 Related Work

There are many proposals for adaptive hardware mechanisms targeted at performance or energy optimization [1, 2, 4, 6, 7, 8, 9, 12, 13, 15, 19, 21]. They adapt a variety of aspects of the processor, including cache organization, issue width, issue window size, or voltage and frequency. The majority of these schemes use *Temporal* adaptation [1, 2, 4, 7, 8, 9, 12, 19, 21]: the testing of the LPTs and the activation of the chosen LPTs are related to time. In our paper, we introduced *Positional* adaptation, where both the testing and the application are tied to a code section. Our approach exploits the fact that the behavior of the program at a given time is directly related to the code it is executing. This idea is also exploited in [23].

The scheme in [13, 22] performs a form of positional adaptation for multimedia applications composed of repeating frames. The scheme uses the best adaptation for the past frames to predict the adaptation to perform in the next frame. The adaptation is positional because each frame simply uses a different data set to execute the same code. Overall, this work is different than ours in that it requires user knowledge of the frame-based structure of the code. Furthermore, it is specialized for the multimedia domain. Our work applies to general-purpose code and exploits its subroutine structure.

The scheme in [6] is based on a temporal scheme with variable-sized intervals [4]. Calibrations and applications of adaptations are performed in intervals tied to time (number of instructions). However, the scheme collects working-set signatures for the code executed in each interval and saves the configuration used. When the algorithm sees a signature similar to one seen before, it applies the saved configuration. This is done to eliminate the overhead of another testing period. Reusing adaptations when the code may be similar gives the algorithm an interesting positional aspect.

The temporal scheme in [15] also has an aspect of positional adaptation: reconfiguration is only attempted in some known sections of the code. In these sections, the system tests several adaptations *in time sequence* to identify the best configuration — making the scheme intrinsically temporal.

The temporal scheme in [7] improves the accuracy of the testing period for temporal schemes by using "mimic" counters that can predict the effect of multiple configurations without trying them out one by one. However, like other temporal schemes, the algorithm exploits the similarity of behavior across consecutive time intervals. Moreover, it is not clear that this approach of using counters can be exploited for all LPTs.

Performing adaptations at subroutine boundaries was considered in [4]. However, after comparing it to performing adaptations at periodic intervals, the latter was selected, largely due to simplicity.

7 Conclusions and Future Work

This paper has presented Positional adaptation, a new approach where both the testing of configurations and the application of the chosen configurations are associated with particular code sections. We use subroutines as the granularity for such code sections. We have also designed three very general implementations of subroutine-based positional adaptation, which correspond to different choices in the tradeoff between general applicability and effectiveness. To evaluate these implementations, we selected several example dynamic LPTs. Overall, all three implementations are more effective than temporal adaptation schemes. On average, they boost the energy savings of applications by 50% and 84% over temporal schemes in two experiments. In general, of course, the absolute energy savings are highly dependent on the LPTs used. Intuitively, subroutine-based positional adaptation is effective because the system becomes highly predictable: different invocations of the same subroutine usually have similar code behavior, and react similarly to the same adaptation.

While we have used positional adaptation in a low-power environment, we can also apply it to performance-centric designs. In this case, the designer's toolkit could include a set of High Performance Techniques (HPTs) instead of LPTs. The goal would be to adapt the processor by activating each HPT at the best point in the program, such that the program runs as fast as possible without increasing energy consumption beyond a certain limit. Other environments can also use positional adaptation.

Our finding that the behavior of different invocations of the same subroutine is very predictable can be exploited in many ways. Specifically, it can be used to reduce the overhead of costly operations by intelligently applying them to only one of several executions that we expect to behave similarly. These costly operations can be dynamic optimization, cycle-by-cycle architectural simulation, or evaluation of various time-consuming optimizations.

References

[1] D. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. In *International Symposium on Microarchitecture*, pages 248–259, November 1999.

[2] R. Bahar and S. Manne. Power and Energy Reduction Via Pipeline Balancing. In *International Symposium on Computer Architecture*, pages 218–229, May 2001.

[3] R. Balasubramonian. Personal communication. October 2002.

[4] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. In *International Symposium on Microarchitecture*, pages 245–257, December 2000.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *International Symposium on Computer Architecture*, pages 83–94, June 2000.

[6] A. Dhodapkar and J. Smith. Managing Multi-Configuration Hardware via Dynamic Working Set Analysis. In *International Symposium on Computer Architecture*, pages 233–244, May 2002.

[7] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. Scott. Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 141–152, September 2002.

[8] D. Folegnani and A. González. Energy-Effective Issue Logic. In *International Symposium on Computer Architecture*, pages 230–239, May 2001.

[9] T. Halfhill. Transmeta Breaks x86 Low-Power Barrier. *Microprocessor Report*, 14(2):1,9–18, February 2000.

[10] A. Hasegawa, I. Kawasaki, K. Yamada, S. Yoshioka, S. Kawasaki, and P. Biswas. SH3: High Code Density, Low Power. *IEEE Micro*, 15(6):11–19, December 1995.

[11] M. Huang, J. Renau, and J. Torrellas. Profile Based Energy Reduction for High-Performance Processors. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 2001.

[12] M. Huang, J. Renau, S. Yoo, and J. Torrellas. A Framework for Dynamic Energy Efficiency and Temperature Management. In *International Symposium on Microarchitecture*, December 2000.

[13] C. Hughes, J. Srinivasan, and S. Adve. Saving Energy with Architectural and Frequency Adaptations for Multimedia Applications. In *International Symposium on Microarchitecture*, pages 250–261, December 2001.

[14] Intel Corporation. *Mobile Power Guidelines 2000, Rev 1.0*, 1998.

[15] A. Iyer and D. Marculescu. Power Aware Microarchitecture Resource Scaling. In *Design, Automation and Test in Europe*, pages 190–196, March 2001.

[16] R. Joseph and M. Martonosi. Run-Time Power Estimation in High Performance Microprocessors. In *International Symposium on Low Power Electronics and Design*, August 2001.

[17] J. Kin, M. Gupta, and W. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. *International Symposium on Microarchitecture*, pages 184–193, December 1997.

[18] G. Magklis, M. Scott, G. Semeraro, D. Albonesi, and S. Dropsho. Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Processor. In *International Symposium on Computer Architecture*, June 2003.

[19] S. Manne, A. Klauser, and D. Grunwald. Pipeline Gating: Speculation Control for Energy Reduction. In *International Symposium on Computer Architecture*, pages 132–141, July 1998.

[20] A. Nannarelli. *Low Power Division and Square Root*. PhD thesis, University of California, Irvine, Department of Electrical and Computer Engineering, June 1999.

[21] D. Ponomarev, G. Kucuk, and K Ghose. Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources. In *International Symposium on Microarchitecture*, pages 90–101, December 2001.

[22] R. Sasanka, C. Hughes, and S. Adve. Joint Local and Global Hardware Adaptations for Energy. In *International Conference on Architectural Support for Programming Language and Operating Systems*, pages 144–155, October 2002.

[23] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, October 2002.

[24] S. Wilton and N. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal on Solid-State Circuits*, 31(5):677–688, May 1996.

Appendix A: Positional Adaptation Overheads

Our algorithms have both steady-state and initialization overheads. The former occur when activating/deactivating LPTs in steady state, and have three sources.

The first source is extra instructions to activate/deactivate LPTs in SISD and SIDD. Upon entering an instrumented subroutine, we save the current LPT mask and indirectly load the mask to apply from a table. Upon exiting the subroutine, we restore the mask and check that we are not in initialization mode. Thus, depending on the implementation, entering or exiting an instrumented subroutine adds 2-10 instructions. The frequency of such an operation is shown in Table 5. In the worst-case application (CRAFTY), it occurs once every 2.6 μs, although on average it occurs once every 845 μs or 295 μs depending on the Instrumentation algorithm used. Consequently, the resulting energy and time overhead is negligible.

The second source of overhead is the accesses to the Call Stack and Call Cache in DIDD. Upon any subroutine entry, the Call Stack is updated and the Call Cache is read. If this is a major subroutine, its LPT mask is activated and, on subroutine return, the old mask is restored. On subroutine return, the Call Stack is updated. In all these operations, the energy consumed is modeled. For example, Table 3 shows that a Call Stack and a Cache Cache access consume 56 pJ and 66 pJ, respectively. The energy consumed is included in our simulations, and can be shown to be insignificant. Since these structures consume very little energy, even in an environment where subroutine calls are very frequent, this overhead will not become significant.

The third overhead of LPT activation/deactivation is some architectural state transitions that depend on the particular LPT. These overheads are discussed in Section 4.3, and include buffering the signal to the data array (or blocking the cache) for 2 cycles in PCache and filter cache misses in IFilter. The impact of these overheads on performance and energy consumption is included in our simulations.

Finally, the dynamic execution of the Instrumentation algorithm in DIDD and the Decision one in SIDD and DIDD are initialization overheads. Such overheads are in practice negligible because they occur only during the beginning stages of the program. For example, the Decision algorithm invokes a library $n+1$ times for each instrumented subroutine, where n is the number of LPTs. Although many such invocations involve little more than reading and saving the energy and performance counters, we estimate that the average invocation takes 100 instructions. Given that there are on average 7 or 18 instrumented subroutines per application (Table 5), the total cost of the algorithm is less than 10,000 instructions. This is minuscule compared to the program execution time. An analysis of the Instrumentation algorithm shows that its overhead is also minuscule.