Application-Level Speculative Processor Power Management

Alexandru E. Şuşu
Email: alex.susu@gmail.com
Aman Kansal
kansal@microsoft.com
Feng Zhao
zhao@microsoft.com

***Abstract*—In this paper we perform application-level energy minimization through the control of the CPU power states during the program execution. To attain this goal we model the program and the underlying architecture with Markov Decision Process (MDP) models.**

The MDP model augments the program CFG with probabilities of how likely is to take the conditional branches and costs expressing how long and how much energy it takes to execute the various regions of code. We learn the models through program profiling, obtain offline an optimal solution, apply the solution to the existing program and then run the program to validate that the theoretical results are met.

Using this method we are able to minimize the average total energy of the program, while keeping the average execution time below a given threshold. In certain cases we are also able to obtain better program Energy Delay Product (EDP) when following the standard optimal MDP policy than with other standard assignments of CPU power states. For example, the program EDP of the optimal solution can be 30% less than the EDP for the maximum performance power state. We also develop some new algorithms to tackle the problem of non-negligible CPU transition latencies with MDP optimization.

This is but an instance of quantitative analysis and optimization. We can apply this method as well for the analysis and control of the Quality of Results (QoR) or the memory used by the software.

Keywords: Markov Decision Processes, application-level energy management, CPU power states

Computer programs are formally analyzed for qualitative safety and liveness boolean properties. In such case, we express programs through transition systems such as control flow graphs or push down automata and logical formulas and. However, such techniques cannot capture interesting quantitative properties, expressible normally through real values, such as the execution time, energy, memory or quality of results.

In this paper, we perform energy and time analysis and optimization of computer applications, drawing from both program analysis and stochastic control.

Existing research in the field of quantitative program analysis computes worst case timing analysis [LLMR07], the expected number of function calls [Kwi09], expresses the QoR as a function of task failure rates [Rin06], measures through profiling the empirical computational complexity [GAW07] or analyzes the reliability analysis of software modules [KA07]. [HKH01], [ICM06] perform greedy (single-stage) application level energy minimization. We are aware only of [SG07], which performs entire program energy minimization using simplifying modeling assumptions such as the program has no variability in execution time.

We use stochastic stateful models to express the uncertainty in our system stemming from the program flow and the abstraction of the microarchitecture running the application. To the best of our knowledge we are the first to perform realistic quantitative program analysis and multi-stage control.

In this paper we bring the following contributions:

- we perform multi-stage program and microprocessor analysis and optimization, namely energy minimization under timing constraints. Note that we can similarly attack the dual problem, as well, execution time minimization under energy constraints.
- we analyze the average time and energy and optimize accounting for variation, as opposed to the worst case;
- we also cope with various non-idealities of the platform in the MDP model, more exactly, the realistic latencies when changing the power state. For this we develop an algorithm for making the MDP model comply with these latencies.
- we efficiently implement the policy resulting from the multi-stage optimization;

Our energy optimization method distinguishes from most of the existing work through the fact we express the performance as absolute time, obtained from program analysis and profiling, while the rest are doing energy level minimization such that they do not degrade almost at all the performance w.r.t. a specified baseline.

We build a stochastic model of the application with costs associated for the quantitative properties in discussion and compute optimal tradeoffs between energy and time. It is possible to use the method for QoR and time.

We use SimpleScalar with Wattch [BTM00], which we briefly modify, in order to assess the time and energy costs of executing the various program regions, similar to [SR08], [SG07]. We use the single issue, in-order standard SimpleScalar PISA processor architecture.

For a comparable analysis of the tradeoff between the energy and the time consumed by computer programs, [BBS$^+$00] defines the Energy Delay Product (EDP) metric. The optimally controlled programs have an EDP 36.6% smaller than the original, for state P_0.

One can argue that the profiling (we can call it also learning or training) and optimization phase have a big impact on energy consumption. But the method we propose is practical in a data center running the same tasks on many machines, in which we can delegate only one server to profile the program and perform MDP optimization and, then, communicate to the other machines the results.

I. Power Characteristics of CPUs

We introduce the notions of CPU power states used in x86, one of the most complex power-manageable processors.

The Intel x86 processor has two main types of power states: active and standby states.

The active states are named P (Performance) states, which vary both the voltage and frequency, and the less employed ones, T (Throttle) states, which vary only the frequency. Although the T-states can attain power savings, the energy consumed by the same task is basically the same, since higher CPU frequency implies

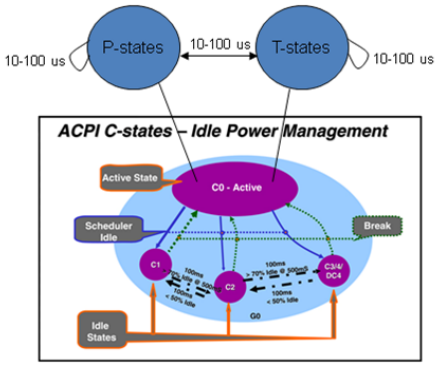

Fig. 1. The power states of an x86 CPU with transition latencies, from [NRM+06].

proportionally lower execution time. However, the P-states attain both power and energy savings, given the fact we can lower voltage when lowering frequency and the voltage has a quadratic impact on power. This is the reason CPU power managers employ normally only P-states.

The standby states are named C (Conserving) and are CPU standby modes, which put to sleep its various subsystems.

We depict these categories of power states in Figure 1.

The processor we use in our experiments has power states with the characteristics presented in Table I. Note that the DRAM access latency is of 67 ns, which imposes the presented DRAM latencies (read first, write second) in CPU cycles for each power state.

TABLE I
POWER STATE CHARACTERISTICS (THE POWER STATES ARE TAKEN FROM TABLE 2 FROM [ICM06], WITH THE EXCEPTION OF P_1 AND P_x).

P-state	Frequency [MHz]	Voltage [V]	DRAM latency
P_0	1500	1.484	100, 10
P_1	1400	1.4336	93, 9
P_2	1200	1.356	80, 8
P_3	1000	1.228	66, 7
P_4	800	1.116	53, 5
P_5	600	0.956	40, 4
P_x	1400	1.4	93, 9

A. CPU with Practically Zero P-state Transition Latencies

The Intel PXA255 embedded processor family has some active power states, called Turbo states, with very small transition latencies between them, of about one CPU cycle [LSP08]. These Turbo power states vary only frequency, since changing voltage would involve a bigger switching overhead.

II. The Stochastic Model of the Application and Runtime Architecture

To find optimal tradeoffs for the energy consumption and time of the program, we employ mathematical modeling.

The most detailed model would keep track of the full state of the machine: i) complete microarchitectural state, i.e., pipeline state, branch predictor state, register file state, L1/L2 cache, TLB and RAM states, maybe memory prefetcher, and so on; ii) program state, i.e., current instruction, global and local variables, heap data structures, etc.

Such a program model matches reality perfectly. It has deterministic transitions and would be able to tell us precisely how much time and energy consumes every instruction. Optimization is normally intractable in this case, due to the high dimensionality of the model. Therefore, we abstract away most of the state space variables, obtaining a randomized Markov model. In case we care about some of the unobserved state variables (e.g., some summarized state of the L1 cache) we can use Partially Observable MDP (POMDP).

We need to choose the simplest model representative for the application, which has a well developed and computationally efficient mathematical apparatus. We choose the Markov Decision Process (MDP) model, the reason being we need to capture the variability induced in execution time and energy due to:

- important program variables, such as symbolic loop bounds and program predicates, which affect execution of conditional branches;
- variable memory accesses caused by the different states of the L1/L2 caches and the RAM.

We decide not to use general stochastic programming [BL97], since it is too expressive and lacks a strong optimization apparatus.

We can see the MDP model as an alternative for the standard automata transition system representing the program. The difference is that we replace the conditional branches of the deterministic automata, with transition probabilities and add costs to the transitions. Our model is adequate for the type of average analysis and optimization we perform.

Others have used similar models for programs. [Ram65] is, to the best of our knowledge, the first to use a Markov model (Markov reward model, MRM, to be more exact) for a program, where every node is a subroutine, in order to compute its average execution time. [WMGH94] computes statically estimates of execution of the branches and basic blocks. Eeckhout et al. study in [EBS+04] higher-order Statistical Flow Graphs (SFGs), essentially CFGs augmented with probabilities obtained from profiling and found that a first-order SFG is enough to accurately capture program behavior and consequently to make accurate performance predictions.

A. Definition of the MDP Model

As already discussed, we formulate the control problem on a discrete time MDP model, $\mathcal{M}$, which is defined by: i) a discrete set of decision epochs, $\mathcal{T}$; ii) a finite set of reachable states (under any policy), $\mathcal{S}$; iii) a finite set of actions, specifying the CPU power states employed, $\mathcal{A}$; iv) an initial probability distribution over $\mathcal{S}$, p^1; v) a transition probability matrix for each action, represented by the function $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ denoting the probability to transition at a destination state from a source state if using a specific action; vi) two cost functions $c_{time}, c_{eng} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, representing the execution time and energy costs of the MDP states, when issuing the various actions.

1) The State Variables of the MDP Model: Our MDP model uses a factored (or intensional) representation [BDH99], having a few state variables that the controller observes in order to decide which action to apply. We want the MDP model to be as abstract as possible for the problem given, in order to avoid the state space explosion.

Our controller requires to know the current region of the application the CPU executes. Programs are usually split into basic blocks, a contiguous set of instructions with only one entry and exit points. However, we prefer going for a coarser-grained region, which we call macroblock (MB), a group of basic blocks. For example an MB can be only one basic block or contain an entire loop.

To build the MDP model is straightforward: once we partitioned the CFG of the program in MBs, we replicate the MBs and associated transitions for each power state. This procedure is depicted in Figures 2, 3, 4.

Therefore, we propose the MDP program model have only the following two state variables: i) the current MB being executed by the program; ii) the power state of the currently executed MB (in fact, we can discard this variable, but it is useful, for example, for the solution with reconfiguration states).

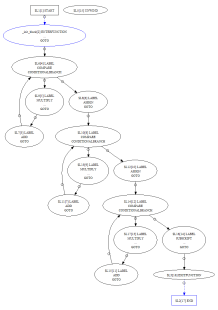

Fig. 2. The CFG of the program.

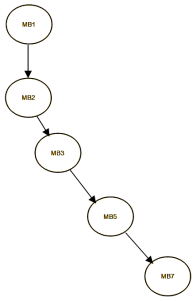

Fig. 3. The program represented through MBs.

We can add some other program state variables: important induction/loop variables and predicates, that affect considerably the program execution.

The reason we might want to observe the important induction variables is that doing so we can implement, if beneficial, an optimal policy assigning different power states to the basic blocks executed in different iterations of the loop. For example, suppose we have a loop with different behavior for the various iterations:

```
for (i = 0; i < n; i++)
    s += a[i][0];
    //can give cache misses
```

We can imagine that the access to $a[i][0]$ gives a cache miss for $i = 0$, but does not give for $i = 1..5$.

The important predicates are associated to the conditional branches with a lot of variability. We might want to observe these important predicates because we might be able to observe earlier what is the value of the predicate before the execution of the program reaches the conditional branches, so we might be able to take better informed decisions.

Note that while we observe the important program variables and predicates, we do not represent the microarchitectural state related to the memory system, branch predictor, etc.

2) The Transition Probabilities and the Costs of the MDP Model: The transition probabilities and the two types of MDP state-action costs, execution time and energy, are obtained through profiling. In principle, we can obtain estimates of the probabilities of the conditional branches through static analysis, similar to [WMGH94]. For the MDP costs we can use the average execution time and energy of each MB. However, if we want to represent more precisely an MB that has variable costs at different execution instances, we can have in the the MDP a different state for each MB instance.

We make a few more observations about the model. We need to have a self loop on the MDP sink state in order to model the infinite time horizon necessary for programs with variable execution time.

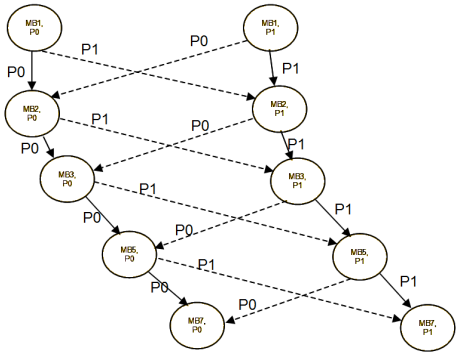

Fig. 4. The MDP model associated to the program. Note that we replicated the MBs for each power state.

B. Assumptions of the MDP Model

We make the assumption that for the system we model the transition probabilities are statistically independent.

Note that in certain cases the probabilities of the MDP transitions might not be independent given the fact the predicates used in different conditional branches might have dependencies. For example, suppose we have the following program:

```
if (p)
   X; //big computation
else
   Y; //small computation

if (p)
   Z; //small computation
else
   W; //big computation
```

The suboptimality incurred by the statistical dependence stems from the following aspect: if the predicate p is not an MDP state variable, the policy controller implemented does not know, for example, when executing block X (when p is true) that block Z follows to be executed. Such an uninformed MDP policy w.r.t. this dependence can behave poorly when assigning power states to the program blocks. However, we can easily take care of these situations with dependent conditional branches. In the above program example, we can merge the two if-then-else branches altogether into one by coalescing the blocks X, Z and Y, W, respectively. The other possibility is to have the controller observe the predicate p and act accordingly.

C. Semantics of the MDP Model

1) Pre-transition Power State Change Semantics: The semantics we use in this paper, which we suggestively call *pre-transition power state change* semantic, changes the power state to the one prescribed by the policy in the current MDP state. More exactly, this semantic prescribes the following behavior when we are in the MDP state $< MB, P >$ and the controller issues action P': the CPU changes its power state to P' and executes the code within the macroblock MB, associated to the MDP state. Note that before entering the MDP state MB we are already in the power state P.

For this semantic, the MDP costs depend on both the MDP state and the action taken as defined in II-A.

A variation of this semantic can consider the costs to be dependent only on the MDP state. For this to work, we need to replicate all MBs in all the other power domains and add additional edges between the MDP states representing the same MB, but with different power states. However, solving such MDP can lead to inconsistent optimal policies that cycle forever between two (or more) states associated to the same MB. This is easy to understand, if these states have very

small or zero costs. Since we have to avoid such infinite loops, we cannot consider this variation of the semantics.

2) Post-transition Power State Change Semantics: Another semantic, which we do not use in this report, has the following behavior suggested by its name: the power state P' prescribed by the action of the (optimal) MDP policy corresponding to the current MDP state $< MB, P >$ is applied only after the CPU executes the code associated to MB at power state P.

In this case, the MDP costs can be dependent only on the MDP states and not on the actions issued, i.e., $c_{time}, c_{eng} : \mathcal{S} \to \mathbb{R}$.

D. Incorporating Latencies when Changing Power States

As stated in Section I the transitions between various CPU power states take a non-negligible number of CPU cycles. We have to take these non-ideal transitions in consideration in our MDP model, which normally considers state transitions to happen instantaneously.

In this paper we assume that the transition latency is the same among all CPU states, with a value denoted by $T_{latency}$.

Let us take an example: we make a transition from P_0 (operating at 1500 MHz and 1.484 V) to P_2 (operating at 1200 MHz and 1.356 V). However, since the transition from power state P_0 to P_2 takes $T_{latency} = 10\mu s$, we try to factor into an ideal MDP model these transition times as extra costs in the MDP. We want to better understand what are the extra energy and time costs caused by a $T_{latency}$ transition from P_0 to P_2.

The extra energy cost is related to the fact that we execute at level P_0 instead of P_2, so we consume in reality more energy than the model. $E_{P_0,10\mu s} = 10 \cdot 10^{-6}s \cdot 1500 \cdot 10^6 Hz \cdot 1.484^2 = 33033.84\ units$, $E_{P_2,10\mu s} = 10 \cdot 10^{-6}s \cdot 1200 \cdot 10^6 Hz \cdot 1.356^2 = 22064.83\ units$. Therefore, the extra energy cost for executing P_0 instead of P_2 is the difference between the two values above, namely $10969.01\ units$.

The extra time cost is related to the fact that executing at P_0 instead of P_2, implies executing faster than modeled. Note that we assume an inorder, single-issue processor and no DRAM accesses. More exactly, the number of cycles in $10\mu s$ at 1500 MHz is $10 \cdot 10^{-6}s \cdot 1500 \cdot 10^6 Hz = 10 \cdot 1500$. The number of cycles in $10\mu s$ at 1200 MHz is $10 \cdot 10^{-6}s \cdot 1200 \cdot 10^6 Hz = 10 \cdot 1200$. This means that we execute 3000 $cycles$ more in P_0 than in P_2 in $10\mu s$. These 3000 cycles get executed in $3000/1200 = 2.5\mu s$ in the P_2 state, which represents the time cost gained by executing in P_0 instead of P_2 for $10\mu s$. Therefore, when issuing a power change from a P_0 state to a P_2 state, we put the transition $time\_cost = -2.5\mu s$.

In the general case, to account for the $T_{latency}$ when changing between power states P_x and P_y, we need to add to the corresponding MDP transition costs the following:
to the c_{time} cost we add $energy\_cost = T_{latency} \cdot (f_x \cdot V_x^2 - f_y \cdot V_y^2)[units]$ and to c_{eng} we add $time\_cost = T_{latency} \cdot (f_x - f_y)/f_y[\mu s]$.

III. The MDP Optimization Problem

As we have previously stated, we seek to minimize the total energy consumed, while meeting an average execution time constraint.

A. The Execution Time Constraint

In our energy minimization problem we want the execution of the program to take at most a given expected total time constraint, ETTC. This ETTC expresses the Quality of Service (QoS) of the application.

We argue that a non-symbolic $ETTC$ is usually desirable since (real-time) schedulers take decisions using normally constant (Worst Case) Execution Time, independent of the variables of the program.

We define $ETTC_{min}$ as the expected time for running the program at the highest frequency power state and $ETTC_{max}$ as the expected time for running the program at the lowest frequency power state. The calculation of these values is done, in principle, by multiplying the time costs of the MDP states for the fastest and slowest P-states (in the case of Table I) with the expected number of visits of these states. We claim that the feasible and interesting actual expected program total time values that can result from the MDP optimization are in the interval $[ETTC_{min}, ETTC_{max}]$. Therefore, if we specify an $ETTC < ETTC_{min}$ the optimization will not yield a feasible solution; if we specify an $ETTC > ETTC_{max}$, the optimization yields the same policy as for $ETTC_{max}$.

We treat in detail these aspects in Section .

B. The Optimization Problem

The solution to our problem is, in general, a Markov randomized stationary policy $\pi : \mathcal{S} \to \triangle(\mathcal{A})$, which prescribes for each state the optimal probability distribution over the actions the controller has to use in order to attain the problem objectives.

The mathematical formulation of our problem is given in (1).

$$\max_{\pi} \lim_{N\to\infty} E^{\pi}\{\sum_{i=0}^{N-1} c_{time}(i)\} \quad \text{s.t.} \lim_{N\to\infty} E^{\pi}\{\sum_{i=0}^{N-1} c_{eng}(i)\} \geq ETTC \tag{1}$$

To obtain an optimal policy π that meets all the specified conditions, we can employ total discounted reward CMDP optimization. We have the choice of using the finite or the infinite time horizon MDP framework. The infinite time horizon has the advantage that the optimal solution policy π is stationary w.r.t. the decision epoch. Therefore we can represent the strategy as a simple table: $(\pi_{s,a})$, for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$, with no dependency on the epoch; this keeps the space storage requirement of the policy very low.

Our total discounted MDP optimization is formulated in (2): the first equation expresses the total discounted energy cost of the program, with discount factor β, while the second equation expresses the constraint on the total discounted execution time.

$$\max_{\pi} \sum_{i=0}^{\infty} p^1 p_{\pi}^i \beta^i c_{eng}(i) \quad \text{s.t.} \sum_{i=0}^{\infty} \beta^i p^1 p_{\pi}^i c_{time}(i) \geq ETTC \tag{2}$$

The standard way to solve the CMDP optimization formulation in (2) is to reduce it to the LP in (3) [DD05], [Alt99], [Kal]. The unknowns of the LP are called state-action frequency (or occupation measure) variables, $x_{s,a}$ and can be interpreted as the total expected discounted number of times action a is executed in state s [Kal], [Put94].

$$\begin{aligned} \max \ & \sum_{s\in\mathcal{S}} \sum_{a\in\mathcal{A}} x_{s,a} c_{eng}(s,a) \\ \text{s.t.} \ & \sum_{a\in\mathcal{A}} x_{s,a} - \beta \sum_{s'\in\mathcal{S}} \sum_{a\in\mathcal{A}} p(s'|s,a) x_{s',a} = p^1(s), \forall s \in \mathcal{S}, \\ & \sum_{s\in\mathcal{S}} \sum_{a\in\mathcal{A}} x_{s,a} c_{time}(s,a) \geq ETTC, \\ & x_{s,a} \geq 0, \forall s \in \mathcal{S}, a \in \mathcal{A} \end{aligned} \tag{3}$$

Let S_x be the subset of reachable MDP states for which $\sum_a x_{s,a} > 0$ (this implies that the states $s \in S \setminus S_x$ satisfy $\sum_a x_{s,a} = 0$). We compute the policy π for the states $s \in S_x$ by making $\pi_{s,a} = x_{s,a} / \sum_{a'\in\mathcal{A}} x_{s,a'}$. For the other states we assign arbitrary actions - in fact, we discover they are not reachable in the DTMC induced by the optimal policy.

Given the fact we have only one constraint in the CMDP, the optimal policy randomizes only once, if the contraint is binding.!!!!

C. Implementing the MDP Policy

We implement the MDP policy resulting from the MDP optimization in the C application. Another possibility, which we do not explore, being more complex and less flexible, is to implement the policy inside the runtime, be it a VM, a simulator like SimpleScalar or even in a synthesizable processor.

The overhead for implementing the Markov policy π depends mostly on: i) the observation of the various state space variables; ii) in case the policy randomizes, the call to the *rand()* function, which is in the range 53-70 cycles on PISA in SimpleScalar. Clearly, for the MDP states where policy π is deterministic, we implement the associated actions without calling *rand()*.

We want to take into account the overhead of the implementation of policy π in the optimization itself. For this, we can instrument the C program with a dummy MDP policy that will bring the results of the profiling closer to the application running the optimal policy. A detailed example is given in Section VI-C.

Given the fact the optimal policy can randomize once, we are interested in making it randomize in a particular MDP state, s_r, in order to instrument the code with randomization only for the associated macroblock. To accomplish this, we add the constraints (4) to the LP in (3), where we have constant $X \geq x_{s,a}, \forall s \in \mathcal{S}, a \in \mathcal{A}$ and the set of binary variables $\Delta_{s,a} \in \{0,1\}, \forall s \in \mathcal{S}, a \in \mathcal{A}$. This is similar to the constraints added in [DD05] to make the entire policy deterministic.

$$\sum_{a \in \mathcal{A}} \Delta_{s,a} \leq 1, \forall s \in \mathcal{S} \setminus \{s_r\} \tag{4}$$
$$\frac{1}{X} \cdot x_{s,a} \leq \Delta_{s,a}, \forall s \in \mathcal{S} \setminus \{s_r\}, a \in \mathcal{A}$$

Note that if the only state space variable we observe is the PC, then we do not need to explicitly implement the verification of the PC, since we can simply insert before runtime in the code the corresponding action of the policy π for each macroblock.

Note that the execution of the program on an in-order (e.g., 1-wide) SimpleScalar PISA architecture takes considerably more (e.g., twice longer) than when running on an Out-of-order (OoO) SimpleScalar PISA architecture.

D. Coping with Physical Timing "Reconfiguration" Constraints

In real implementations, microprocessors do not allow instantaneous power state changes. An x86 processor is able to change the power state in about $10\mu s$, because it needs to adapt the frequency and the voltage is decreased in several steps in order to keep the reliability of the computation. The Intel PXA255 embedded processor is able to change the power state much faster, in the order of 1 cycle.

Therefore, we need to capture this latency of power state reconfiguration in our MDP optimization problem (i.e., issue a different MDP action).

As discussed already, we assume that the P-state switch latency has the same value among all CPU states, namely $T_{latency}$.

Definition III.1 *For a given MDP, a (physically) $T_{latency}$-feasible power state change is an MDP action that is issued at least $T_{latency}$ time units after the last power "reconfiguration".*

We define the following problem: Given the MDP defined in Section II and the constant $T_{latency}$, we want to find the policy that minimizes the expected total running energy cost under a given timing constraint, $ETTC$, and issues only $T_{latency}$-feasible power state changes.

To solve this problem (offline), we have the following possibilities we can:

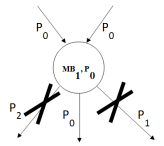

Fig. 5. Non reconfigurable state.

- i) allow the power state change only in certain MDP states. More exactly, we need to choose a *reconfiguration set*, i.e., the set of MDP states that allow only feasible power state transitions.
- ii) make the MDP states have the same costs, by expanding each MDP state into several nodes, each with the same time cost (this time cost needs to be carefully chosen taking into consideration all time costs and choosing a greatest common divisor among them) and proportional energy costs. This expansion can also be done probabilistically, which avoids increasing much the state space, at most tripling it.

For the moment, we pursue the first alternative above. This is useful in case $T_{latency}$ is big (e.g., more than 10 CPU cycles).

1) Solution with MDP Reconfigurable States: In order to handle the multi-epoch action optimization, we pursue the first alternative presented above. As already stated, a reconfigurable state is one that allows to choose a different action than the one that is associated to it. We depict in Figure 5 a non reconfigurable state, which disallows changing the current P-state, let us call it P_0. This constraint is propagated in the LP associated to the CMDP optimization problem. Namely, the LP bound constraints for the state-action frequencies are adapted in the following way to transform state s into a non reconfigurable state:

$$x_{s,P_n} = 0, \text{ for any action } P_n \neq P_0,$$
$$\text{while } x_{s,P_0} \geq 0 \text{ remains unchanged.}$$

Definition III.2 *The power reconfiguration compatibility graph of a given MDP with the feasibility constraint $T_{latency}$ is an undirected graph with the same set of vertices as the MDP graph. Two arbitrary nodes x and y are adjacent iff performing a $T_{latency}$-power reconfiguration at state x allows to perform $T_{latency}$-feasible reconfiguration at state y, as well, and viceversa.*

In terms of the MDP graph, we have an edge between two vertices in the power reconfiguration compatibility graph if the shortest paths between the corresponding MDP states in both directions are at least $T_{latency}$ time units.

It is easy to see that a feasible reconfiguration set is a clique in the power reconfiguration compatibility graph.

The underlying graph of the DTMC induced by a "uniformly" distributed policy on our MDP $\mathcal{M}$ is $\mathcal{G} = (V, E)$, where V is the set of vertices of the MDP and E is the set of edges.

Suppose we compute the shortest paths between all pairs of MDP states. For any two MDP states s_1 and s_2, let $w(s_1, s_2)$ be the weight of the shortest path between these two nodes in the induced graph of the MDP. We compute the matrix w with a standard Floyd-Warshall algorithm, with time complexity $\mathcal{O}(|\mathcal{S}|^3)$.

We define a pair (i, j) of *incompatible reconfigurable states* as being the vertices in the induced graph of the MDP with $w(s_1, s_2) \leq T_{latency}$.

Let MAX be a real constant equal to the maximum conceivable value of $x_{s,a}$, for all $s \in \mathcal{S}, a \in \mathcal{A}$.

We can extend the LP in (3), by adding the binary variable rs_s for each s, which can be 1 (or 0) if $s \in \mathcal{S}$ is considered (or not) a reconfigurable state. We add the following constraints to the LP in (3):

For each $s \in \mathcal{S}$ and $b \in \mathcal{A} \backslash \{\text{action associated to } s\}$: (5)

$x_{s,b} \leq MAX \cdot rs_s$,

For each pair (i, j) of incompatible reconfigurable states:

$rs_i + rs_j \leq 1$,

$rs_i \in \{0, 1\}$

The first set of inequalities expresses the fact that if we make state s reconfigurable ($rs_s = 1$), then we make $x_{s,b} \leq 0$, which basically makes $x_{s,b} = 0$, given the fact we also have in the LP $x_{s,b} \geq 0$; otherwise, we basically obtain $x_{s,b} \leq MAX$.

To obtain the reconfiguration set that yields the optimal MDP objective, we need to solve the MILP (Mixed Integer LP; note that the integer variables are actually binary) formed by (3, 5).

E. Programs with Symbolic Loop Bounds

Programs can have symbolically bounded loops, which can influence to a great extent the program execution. We call these loops important symbolically bounded loops. The values of the symbolic loop bounds are important for our energy-delay optimization, and, therefore, we make this a state variable.

These loops can be identified to a certain extent through static analysis [GMC09].

There are cases where static analysis cannot provide any extra information about the values of the symbolic loop bounds. On the other hand, profiling can learn these values and their frequencies, but the learning can take a long period for us to converge to a meaningful model.

In order to optimize for such situation we can replicate the loop for all meaningful values of the symbolic bound and perform MDP optimization as such. The problem we face here is the curse of dimensionality, which results in optimal MDP policies requiring a lot of storage: even for a few symbolically bounded loops with just a few possible values for each bound, the resulting MDP state space can be very big. To handle the MDP state space explosion, we can use *approximate dynamic programming* techniques [Pow07].

Another solution to avoid the state space explosion is to perform analysis and optimization for symbolic Markov models [HHZ09].

Such important symbolically bounded loops can occur at any place in the program. Ideally, we can learn the values of the symbolic bounds at the beginning of the program, as it is the case in Figure 6. However, there are programs where these values are defined only late during the execution, for example, when reading from big data streams values that affect the control flow (e.g., when unarchiving compressed RLE data). This is why we require a good model/belief on the possible values of the bounds to perform a meaningful optimization. We call these late assigned loop bounds. For example, assume we have the program in Figure 7. Here, we do not know the value of $numf1s$ until the moment we call the $fscanf()$, and, in order to take informed decisions before executing the $fscanf()$, we need to be given a probability distribution over $numf1s$. In case we do not have apriori information about this distribution, we can conservatively assume a uniform distribution over all possible values of $numf1s$.

We have two alternatives of handling such symbolic variables in the MDP optimization:

- one policy for each (*OPFE*) value of the important program variable, which has the advantage the optimization is performed specifically for each value, yielding the best results for these conditions. As discussed above, we can use this alternative only if the variable is defined at the very beginning of the program. The disadvantage is that we require storage for all the policies resulting from each value of the bound.
- one policy for all (*OPFA*) values of the important program variable. The disadvantage is that the optimization is less precise, in the sense that the expected value matters for all possible values, so for a particular one the actual expected total time might be far away from the specified timing constraint. The optimization can be regarded as more "expressive", in the sense we can define a probability distribution over the possible values of the meaningful variables of the program. For OPFA we can perform state aggregation, where an MDP state can represent an interval of values of the variable, for example.

We compare the two alternatives in Section VI-B.

We can reduce the number of states of the MDP, and thus the space taken by the policy, by ignoring in certain program regions the variables that are not relevant for the investigated quantitative properties. For example, assume we have the program from Figure 6.

In order to save space, since $numf1s$ is not used until the *for* loop, we do not need to observe the $numf1s$ up to this variable. From the first use in the program of $numf1s$, we need to observe $numf1s$. Even where $numf1s$ is no longer a live variable, we need to apply the policy keeping track of $numf1s$, since $numf1s$ produced an impact before on time/energy/etc and is still impacting these current states as well.

```
//beginning of the program
numf1s = param;

... //this part of code does not use numf1s
...

for (i = 0; i < numf1s; i++) {
    ...
}

...
... //this part of code does not use numf1s
...
```

Fig. 6. Program fragment with a symbolically bounded loop, with bound known from the beginning.

```
... //this part of code does not use numf1s

fscanf(fin, "%d", &numf1s);

for (i = 0; i < numf1s; i++) {
    ...
}

...
... //this part of code does not use numf1s
...
```

Fig. 7. Program fragment with a late assigned symbolic loop bound.

IV. Methodology Tool Flow

The method we use in this paper is described in Figure 8.

Note that with the *MDP_opt* tool, available for download from http://sites.google.com/site/alexsusu/home/mdp_prism_opt, we

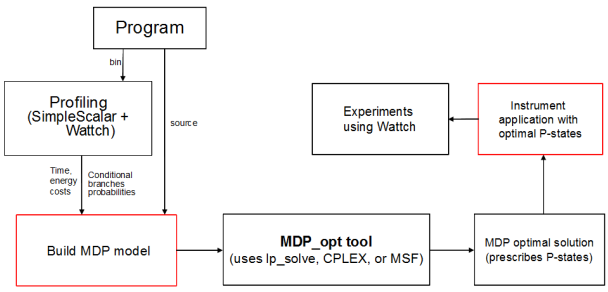

Fig. 8. The tool flow of our method.

are able to solve complex CMDPs with up to 40,000 reachable states and 6 actions [§10]. In fact, the bottleneck is the associated LP, which, in the mentioned example, uses six times more, i.e., 240,000, variables. Note that the time for solving an LP and finding an optimal solution depends mostly on the condition number of the basis matrix and, in part, on the number of LP vars. When this condition number is big, or as they say, the LP is ill-conditioned, LP solvers cannot reliably solve the problem. In one instance, solving this LP took more than six days to be solved exactly on a standard 1 GHz x86 computer platform. However, in [§10] we use MDP state aggregation [Pow07], [BPSW08], which reduces the search time to a couple of hours, at the expense of obtaining a suboptimal solution.

The *MDP-opt* tool writes also a specification of the implementation of the optimal MDP policy. We use this specification, represented in Figure 15 and change manually the target P-state values of the placeholder DVFS instructions in the C program from Listing 1.

To profile the characteristics of the program in discussion with the optimal policy, we do the following: i) we copy to a separate folder the C application, *prof_simplescalar.sh* and the *Makefile*; ii) in this new folder we run *make*, which creates the binary from the application; iii) then, we run *./prof_simplescalar.sh binary_file_application*.

V. The Experimental Platform: SimpleScalar with Wattch

The SimpleScalar tool simulates by default the PISA architecture, which is similar to the MIPS architecture. SimpleScalar was extended by the community to simulate Alpha, ARM, PowerPC (http://www.simplescalar.com/v4test.html; http://www.cse.ohio-state.edu/ dinan/dsswatch_guide.pdf) and XScale [CMP+07] architectures. There is also a SimpleScalar spin-off called Zesto, which simulates x86 code [LSX09].

For us, the biggest advantage of SimpleScalar is we can use the Wattch model extension [BTM00] to assess the power consumption of programs and their energy, through extension. Also, we can relatively easily add CPU power states with specifiable latencies.

We use Wattch 1.0, which is built on top of SimpleScalar version 3.0d. SimpleScalar with Wattch performs cycle-accurate simulation with a speed in the order of hundreds of KIPS, which is much smaller than that of a native profiler. This means one second real-time execution on a 1 GHz processor becomes about 1 hour of simulation in SimpleScalar with Wattch running on a 2 GHz x86 computer. However, if we discard the power measurements we can use the vanilla SimpleScalar in faster functional simulation modes, such as sim-cache and sim-profile - see Table 2-2 from [Bur98], which are at least five times faster than the cycle-accurate ones.

In order to avoid big simulation times with the SPEC CPU 2000 benchmarks, we create input data sets for some of the benchmarks, which take very little time to execute. For example, we create two input data sets for the 179.ART benchmark: i) *./sim-outorder scanner -scanfile c756hel.in -trainfile1 Alex_small.img -stride 2 -startx 134 -starty 220 -endx 135 -endy 225 -objects 1*, for which the simulation takes 2m49.154s; ii) *./sim-outorder scanner -scanfile c756hel.in -trainfile1 Alex_small_15_15.img -stride 2 -startx 134 -starty 220 -endx 135 -endy 225 -objects 1*, for which the simulation takes 6m.

One can use the *std* input data sets created by Doug Burger for SPEC CPU 95 [Bur98].

Following are other energy profilers, written in decreasing order of relevance: SimpleScalar-Arm (available at www.eecs.umich.edu/ panalyzer/), Xtrem [CMP+07], SimplePower, this last one not relying on SimpleScalar.

SimpleScalar provides microarchitectural profiling information through simulation. One of its advantages is that the simulation observes the microarchitectural parameters without being intrusive, while executing the application. SimpleScalar is an application-level simulator and normally does not run a multitasking OS.

A. Adding Extra Logging Profile Information in SimpleScalar

The notions of number of cycles, time and energy it takes to execute an instruction need to be carefully defined, since the SimpleScalar simulates a pipelined and possibly out-of-order architecture. The pipelining implies that we can have several instructions executing at the same time in the various pipeline stages. This also implies that trying to measure, for example, the number of cycles a basic block takes might be inexact, since instructions from a basic block might be slowed down, for example, because of a cache miss generated by an instruction in a previous basic block.

SimpleScalar's standard profiling, obtained by running the command `sim-outorder -pcstat sim_cycle ...` provides us the `sim_cycle` value for each machine code instruction, which informs us at which CPU cycle the instruction gets executed for the first time. Therefore, we have to extend the existing profiling infrastructure to get the right number of cycles of each execution per machine code instruction, from the moment of fetch to the commit.

We are interested in logging only fully executed instructions, since a mispredicted fetched instruction gets recovered before the commit phase. Therefore, we log at the commit time of each instruction the cycle, the (physical) time and the energy consumed by the time the instruction was fetched, and the same values for the moment of commit. For doing so: i) we add the fields *fetch_cycle*, *fetchTotalEnergy*, *fetchTime* to the C structs *RUU_station* and *fetch_rec* defined in *sim-outorder.c*; ii) we add corresponding code for logging in *ruu_commit()*.

Initially, the extra profiling code was written such that we output a log line during *ruu_commit()*. When the simulation ended, we ran a C application *getCostsMDP.c*, which parses the log file in order to collect the statistics. In order to avoid the big log files, which can easily get in the order of GBs for decent program execution (e.g., 7 GB of data on HDD for 30 billion cycles executed), we add the functionality of *getCostsMDP.c* in SimpleScalar itself. The current logging implementation stores in a table the profiling information for each code region of interest, at the end of the commit phase of each delimiting instruction, to avoid mispredictions. This information is written in the files *mystats* and *MDPCosts_PowerState.h*. In order to create these two files, we add at the end of function *stat_print_stats()* in *stats.c* a call to *PrintAlexStats()*.

This impacts the performance in the following way: although now the simulation is more CPU intensive, since we check during the simulation if the current instruction simulated is part of one of the observed groups of instructions, we however reduce dramatically the I/O (MB per minute of log data). Therefore, we take less time overall.

B. Adding Multiple CPU Power States to Wattch

To test the optimal policies, our processor needs to support the various P-states described in Table I.

To implement in SimpleScalar P-states we have two options: i) add to the processor ISA a new instruction that allows us to change the power state of the CPU from within our application, similar to the work in [HKH01]; ii) specify in the simulator itself (or runtime environment, in general, for that matter: processor, even VM), outside the executed application, the program addresses where the P-state needs to be changed and the prescribed values.

We choose the first alternative, being closer to what current processor power managers do. This solution is also more modular, separating the CPU from the software together with the policy.

The P-state assigning assembly instruction, which we call *dvfs*, has the op-code *0x000000a9* and is followed by a 32-bit operand specifying the target P-state.

We specify in SimpleScalar the properties of the P-states, as given in the Table I, in the machine definition file, *machine.def*.

To change the CPU power state, we write directly in the C application assembly code with the *dvfs* instruction. More exactly, we define the macro given in Figure 9 and write, for example

```
#define DVFS(NEW_PSTATE_ASM_STRING) \
 /*Following is the opcode of the dvfs instruction.*/ \
   asm (".byte 0xa9"); \
   asm (".byte 0x00"); \
   asm (".byte 0x00"); \
   asm (".byte 0x00"); \
 /*Following is the target P-state (1st byte = LSB).*/ \
   asm (NEW_PSTATE_ASM_STRING); \
   asm (".byte 0x00"); \
   asm (".byte 0x00"); \
   asm (".byte 0x00");
```

Fig. 9. C preprocessor code that changes the CPU power state to the value in *NEW_PSTATE_ASM_STRING*.

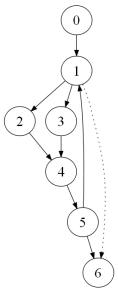

Fig. 10. The detailed MDP model for the loop, only for one power state; we need to replicate this structure for each P-state, add the respective P-state changing edges and add the appropriate state costs.

DVFS(".byte 0x01"); to change to P-state P_1. Note that in this macro-definition we specify the *dvfs* instruction directly through its eight hexadecimal byte values - the PISA architecture is little endian and takes machine code instructions of 8 bytes - since we did not define in the *GNU AS* tool an assembly mnemonic for it.

VI. Experiments

In this section we apply the proposed method. We introduce a series of benchmarks, on which we apply the optimization techniques discussed above.

The first three subsections are using similar simple programs composed of one main loop containing an *if-then-else* statement. Then, we introduce real life benchmarks. Then we run these programs instrumented with the optimal policies on Wattch and discuss the results we obtain.

Note that we assume the transitions between the CPU power states happen instantaneously. Remember that in Section I-A we presented a CPU performing almost instantaneously the P-state transition, i.e., with $T_{latency} = 0$. We also might be witnessing in the near future the advent of (multi-core) CPUs targeting server and desktop computers with such transition latencies.

A. Loops with Constant Bounds

Programs that have only loops with constant bounds are easier to model, observe and control than when having symbolic bounds.

In this experiment, we give a rather simple example, with an *if-then-else* construct executed in a loop for ten iterations. Note that we use only two power states, P_0 and P_2.

First, we model the constant bounded loop either with an MDP with a loop with a few states and a backedge, replicated for both power domains. We specify a time constraint of $ETTC = 0.0002830434$ seconds.

The detailed model for the loop is given in Figure 10. The value of the objective function resulted from the MDP optimization is 187.621 units.

Next, we can abstract further and summarize the loop iteration in one MB state, as depicted in Figure 11. The value of the optimal total energy resulted from the MDP optimization is 191.154 units.

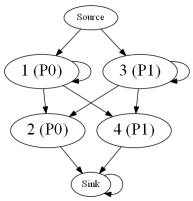

Fig. 11. The MDP model with one state for a loop iteration - notice the self-loops.

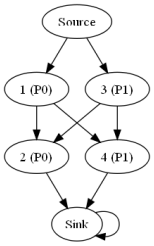

Fig. 12. The MDP model with one state for the whole loop.

We can go even further and abstract the whole for loop in one state, as shown in Figure 12. The value of the optimal total energy resulted from the MDP optimization in this case is 192.874 units.

As we can see the relative difference in the total energy cost objective function between the three models is of at most 2.79%. This is due to the model abstraction.

B. Symbolically Bounded Loops

Symbolically bounded loops pose great challenge for the timing analysis in general and, similarly, to our methodology.

We extend the example in Section VI-A by making the loop bound non-constant, with domain set $\{2, 5, 10\}$. In this setting we use all seven power states: we do profiling only for power states P_0 and P_2 and the costs associated to the other power states are scaled w.r.t. P_0. We set for all experiments $ETTC = 0.0002830434$ seconds.

If we generate an MDP policy for each variable value, i.e., use the OPFE alternative described in Section III-E, we have: for two iterations of the loop, the value of the objective function is 1,255,120; for five iterations, the value of the total energy is 2,022,830 units; for ten iterations, the value is 3,594,710 units. The uniformly weighted average of the optimal total energy for the three different cases is 2,290,886.

When generating one MDP policy for all possible values, i.e., use the OPFA alternative, with uniform probability distribution, the value of the objective function is 4,186,520.

We can notice that the optimal total energy obtained through the OPFA method is bigger than the average of the objectives of the three policies under the OPFE alternative. We conclude that, in this case, OPFE (specialization) performs better than OPFA (generalization), of course at the expense of a bigger overhead required to check the loop bound.

C. Conditional Branches

In this experiment we use the simple C program depicted in Listing 1, similar to the ones in the above paragraphs. The program executes in a loop for ten times an *if-then-else* block, which calls functions *BigComputationFunction* and *SmallComputationFunction*, respectively, on the two branches. These functions implement a bigger CPU workload of 10,000 iterations, respectively a smaller one of 1,000 iterations. The transition probabilities of the *then* and *else* branches are fixed statically in the program at 0.2 and 0.8, in the definition of the vec vector.

We use in the optimization for this program all seven P-states from the Table I.

[!!!!Should I maybe take out the C program. Should I take out the ASM code with the code regions?]

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include "powerState.h"

#define NVALS 10
int vec[NVALS] = {0, 1, 0, 0, 0, 0, 0, 1, 0, 0};

void BigComputationFunction() {
  int i;
  for (i = 0; i < 10000; i++);
}

void SmallComputationFunction() {
  int i;
  for (i = 0; i < 1000; i++);
}

int main() {
  int i;
  float r;

  //MDP state 1

  /* We will change this and the following DVFS
     instructions with the optimal P-state values
     obtained from the MDP optimization. */
  DVFS(PROFILE_PSTATE_ASM_STRING);

  srand(time(NULL));

  //MDP state 2
  for (i = 0; i < NVALS; i++) {
    DVFS(0x00);

    //MDP state 3
    /* We cannot power manage the if from the C code. */
    if (vec[i]) {
      //MDP state 4
      DVFS(PROFILE_PSTATE_ASM_STRING);

      BigComputationFunction();
    }
    else {
      //MDP state 5

      /* Place-holder for randomized choice of the MDP policy. */
      r = ((float)rand()) / RAND_MAX;
      if (r < 0.508439) {
        DVFS(PROFILE_PSTATE_ASM_STRING);
      }
      else {
        DVFS(PROFILE_PSTATE_ASM_STRING);
      }

      SmallComputationFunction();
    }

    //MDP state 6
    DVFS(PROFILE_PSTATE_ASM_STRING);
  }

  //MDP state 7
  DVFS(PROFILE_PSTATE_ASM_STRING);

  return 0;
}
```

Listing 1. The profiled C application instrumented with DVFS instructions, used for profiling, but also as place-holders for the implementation of the optimal policy that we will compute.

The *main()* function of this program gets translated to the assembly code from Listing 2.

```
//MDP state 1
00400310 <main> addiu $sp[29],$sp[29],-32
00400318 <main+8> sw $ra[31],28($sp[29])
00400320 <main+10> sw $s8[30],24($sp[29])
00400328 <main+18> addu $s8[30],$zero[0],$sp[29]
00400330 <main+20> jal 00400678 <__main>
00400338 <main+28> 0x000000a9:00000006 //DVFS instr.: go to Pstate Px
00400340 <main+30> addu $a0[4],$zero[0],$zero[0]
00400348 <main+38> jal 00400990 <time>
00400350 <main+40> addu $a0[4],$zero[0],$v0[2]
00400358 <main+48> jal 00400a20 <srand>
00400360 <main+50> sw $zero[0],16($s8[30])

//MDP state 2
//Note that we cannot implement here a DVFS instruction in C
00400368 <main+58> lw $v0[2],16($s8[30])
00400370 <main+60> slti $v1[3],$v0[2],10
00400378 <main+68> bne $v1[3],$zero[0],00400388 <main+78>
00400380 <main+70> j 00400490 <main+180>

//MDP state 3
00400388 <main+78> 0x000000a9:00000006 //DVFS instr.: go to Pstate Px
00400390 <main+80> lw $v0[2],16($s8[30])
00400398 <main+88> addu $v1[3],$zero[0],$v0[2]
```

```
Energy costs:
       MB1         MB2         MB3         MB4         MB5         MB6         MB7
P0 .0009514684 .0000003978 .0000021304 .0057551302 .0005836180 .0000011050 .0000082210
P1 .0008778587 .0000003715 .0000018742 .0053746742 .0005448578 .0000009742 .0000071004
P2 .0007722984 .0000003326 .0000015300 .0048108348 .0004873670 .0000007983 .0000056174
P3 .0006241695 .0000002734 .0000011302 .0039543156 .0004003572 .0000005924 .0000039800
P4 .0005048169 .0000002262 .0000007994 .0032710242 .0003309251 .0000004223 .0000026142
P5 .0003661838 .0000001661 .0000005129 .0024005293 .0002426881 .0000002731 .0000015498
Px .0008379836 .0000003544 .0000017879 .0051273261 .0005197435 .0000009294 .0000067736

Time costs:
       MB1         MB2         MB3         MB4         MB5         MB6         MB7
P0 .0000143513 .0000000060 .0000000321 .0000868067 .0000088029 .0000000167 .0000001240
P1 .0000151641 .0000000064 .0000000324 .0000929971 .0000094276 .0000000169 .0000001229
P2 .0000173295 .0000000075 .0000000345 .0001084800 .0000109897 .0000000180 .0000001267
P3 .0000203410 .0000000090 .0000000372 .0001301550 .0000131776 .0000000195 .0000001310
P4 .0000247082 .0000000113 .0000000398 .0001626600 .0000164561 .0000000210 .0000001300
P5 .0000322203 .0000000150 .0000000463 .0002168467 .0000219227 .0000000247 .0000001400
Px .0000151641 .0000000064 .0000000324 .0000929971 .0000094269 .0000000169 .0000001229
```

Fig. 13. The energy and time costs of the macroblocks of the program, for the seven P-states.

```
004003a0 <main+90> sll $v0[2],$v1[3],0x2
004003a8 <main+98> lui $v1[3],4096
004003b0 <main+a0> addiu $v1[3],$v1[3],96
004003b8 <main+a8> addu $v0[2],$v0[2],$v1[3]
004003c0 <main+b0> lw $v1[3],0($v0[2])
004003c8 <main+b8> beq $v1[3],$zero[0],004003e8 <main+d8>

//MDP state 4
004003d0 <main+c0> 0x000000a9:00000006 //DVFS instr.: go to Pstate Px
004003d8 <main+c8> jal 004001f0 <BigComputationFunction>
004003e0 <main+d0> j 00400460 <main+150>

//MDP state 5
004003e8 <main+d8> jal 00400a50 <rand>
004003f0 <main+e0> mtc1 $v0[2],$f0
004003f8 <main+e8> cvt.s.w $f0,$f0
00400400 <main+f0> l.s $f2,-32768($gp[28])
00400408 <main+f8> div.s $f0,$f0,$f2
00400410 <main+100> s.s $f0,20($s8[30])
00400418 <main+108> l.s $f2,20($s8[30])
00400420 <main+110> cvt.d.s $f0,$f2
00400428 <main+118> l.d $f2,-32760($gp[28])
00400430 <main+120> c.lt.d $f0,$f2
00400438 <main+128> bc1f 00400450 <main+140>
00400440 <main+130> 0x000000a9:00000006 //DVFS instr. for randomized policy
00400448 <main+138> j 00400458 <main+148>
00400450 <main+140> 0x000000a9:00000006 //DVFS instr. for randomized policy
00400458 <main+148> jal 00400280 <SmallComputationFunction>
//This next instruction is the NOP-like instruction.
00400460 <main+150> addiu $v0[2],$v0[2],0

//MDP state 6
00400468 <main+158> 0x000000a9:00000006 //DVFS instr.: go to Pstate Px
00400470 <main+160> lw $v1[3],16($s8[30])
00400478 <main+168> addiu $v0[2],$v1[3],1
00400480 <main+170> addu $v1[3],$zero[0],$v0[2]
00400488 <main+178> sw $v1[3],16($s8[30])
00400490 <main+180> j 00400368 <main+58>

//MDP state 7
00400498 <main+188> 0x000000a9:00000006 //DVFS instr.: go to Pstate Px
004004a0 <main+190> addu $v0[2],$zero[0],$zero[0]
004004a8 <main+198> j 004004b0 <main+1a0>
004004b0 <main+1a0> addu $sp[29],$zero[0],$s8[30]
004004b8 <main+1a8> lw $ra[31],28($sp[29])
004004c0 <main+1b0> lw $s8[30],24($sp[29])
004004c8 <main+1b8> addiu $sp[29],$sp[29],32
004004d0 <main+1c0> jr $ra[31]
```

Listing 2. Disassembled code of the *main()* function from the C Listing 1, instrumented with place-holder DVFS instructions for the implementation of the optimal policy that we will compute.

Given the fact we need to act differently on the two branches, we build a more refined model than in the previous sections. The MDP states in Figure 14 represent the following parts of the program: the *Source* is the node without code associated to it, which allows going in any desired P-state; the MDP state 1 for P_0 (respectively, state 8 for P_1) represents the code region from the entry point of the *main()* function to the beginning of the *for* loop; the MDP state 2 for P_0 (respectively, state 9 for P_1) represents the code region with the *for* loop header, which checks the termination condition; the state 3 (respectively, 10) checks the value of the *if* predicate and directs the control flow on the right branch; states 4 and 5 (respectively, 11 and 12) represent the code regions for the *true* (with the bigger computation) and the *false* (with the smaller computation) branches; state 6 (respectively, 13) represents the end part of the *for* loop (the footer); state 7 (respectively, 14) is the code region until the end of the *main()* function. Note that this model does not take into consideration the code before entering and after returning from the *main()* function.

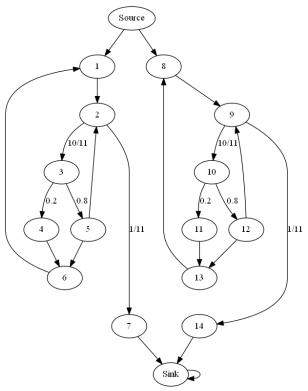

Fig. 14. The MDP model built from the program, for only two P-states, P_0 and P_1, without the P-state changing edges.

```
Macroblock 1:
    if power_state=0 then change to power state p5 with probability 1.000.

Macroblock 2:
    if power_state=5 then change to power state p5 with probability 1.000.

Macroblock 3:
    if power_state=5 then change to power state p5 with probability 1.000.

Macroblock 4:
    if power_state=5 then change to power state p4 with probability 1.000.

Macroblock 5:
    if power_state=5 then change to power state p4 with probability 0.775534.
    if power_state=5 then change to power state p5 with probability 0.224466.

Macroblock 6:
    if power_state=4 then change to power state p5 with probability 1.000.
    if power_state=5 then change to power state p5 with probability 1.000.
    //We reduce the above to: change to power state p5 with probability 1.000.

Macroblock 7:
    if power_state=5 then change to power state p5 with probability 1.000.
```

Fig. 15. The detailed specification of the optimal MDP policy.

These extra portions take about 140,000 cycles to execute, which represents about one third of the total execution of the program. However, we do not optimize this code, since it is not accessible from our C program, this code being contained in the standard C library. Since we do not have access to the standard C library code, we cannot easily change the power state from the application and choose to disregard these portions of code.

The energy and time costs of the of the MDP states are depicted in Figure 13. We notice that P-states P_0 and P_5 are invariably the fastest and slowest for all MDP-states - there could have been some issues given the associated P-state DRAM latencies.

For this MDP model, we have $ETTC_{min} = 0.0002590593$ and $ETTC_{max} = 0.0006422943$ seconds. The calculation of these values is done, as discussed, by multiplying the time costs of the MDP states for the fastest and slowest P-states with the expected number of visits of these nodes. More exactly, for $ETTC_{min}$ we use the costs for the P-state P_0: $ETTC_{min} = 0.0000143513 + 10 \cdot (0.0000000060 + 0.0000000321 + 0.2 \cdot 0.0000868067 + 0.8 \cdot 0.0000088029 + 0.0000000167) + 0.0000001240 = 0.0002590593$, while for $ETTC_{max}$ we use P_5: $ETTC_{max} = 0.0000322203 + 10 \cdot (0.0000000150 + 0.0000000463 + 0.2 \cdot 0.0002168467 + 0.8 \cdot 0.0000219227 + 0.0000000247) + 0.0000001400 = 0.0006422943$.

When computing the optimal policy for $ETTC = ETTC_{min}$ we obtain an (almost) deterministic policy prescribing state P_0, as expected. Setting $ETTC < ETTC_{min}$ makes the optimization problem unfeasible. For $ETTC = ETTC_{max}$ we obtain a deterministic policy with P_5, again as expected. Making $ETTC > ETTC_{max}$ does not improve anything: we obtain the same policy as for $ETTC_{max}$.

To make the optimization interesting, we now put a time constraint of 0.0005 seconds, which makes the optimal policy randomize once. To make the policy randomize on the state associated with the *else* branch, we add the constraints (4) to the LP in (3). The theoretical energy consumption of the obtained optimal policy is of 0.00940802 units. Therefore, the theoretical optimal EDP is of 0.00000470401 units, which is an increase of 6% over the EDP when running only in P_0 and of 3% w.r.t. P_5.

We use the specification of the optimal MDP policy written by the *MDP-opt* tool, represented in Figure 15, and change manually the target P-state values of the place-holder DVFS instructions in the C program from Listing 1.

To measure the average total time and energy values of the optimized program, since the policy is randomized, we make several runs of the instrumented program to trigger the two different behaviours of the policy. In one run, SimpleScalar reports it spent 0.0098778175 units of energy for a total time of 0.0005298401 seconds. In the other scenario, it spent 0.0097016615 units in 0.0005407380 seconds. The average program quantities are 0.0097897395 units of energy for 0.00053528905 seconds.

We notice that the experimental results match closely the predicted theoretical value: the experimental average time is 4% bigger than the expected value, and the energy 7% bigger. The differences come from the various practical issues of the implementation of the policy such as the fact the *dvfs* instruction is in some cases not placed at the very beginning of the MB.

We now make the optimization problem more realistic: we set $T_{latency} = 10$ ns. As discussed in III-D1, solving the MILP formed by (3, 5) we obtain that the optimal reconfiguration set is {34, 39, 40} and the optimal expected total energy of 0.00940802 units for an $ETTC = 0.0005$ seconds. When making $T_{latency} = 10$ ms we obtain that the optimal reconfiguration set is {35, 39}, the optimal energy of 0.00941373 units for the same $ETTC$. As expected, the bigger we make $T_{latency}$ the smaller the optimal expected energy becomes, when keeping $ETTC$ constant.

D. SPEC CPU Benchmarks

We now use our methodology on a couple of SPEC CPU 2000 benchmarks to show its benefits also on complex programs. We choose the 179.art benchmark, since we can simulate them in SimpleScalar in little time, in the order of a few minutes. To reach a short simulation time we change the input data set, similar to the *std* data sets developed by Doug Burger [Bur98], [HKH01] for SPEC 95.

1) 179.art: The ART (Adaptive Resonance Theory) benchmark performs thermal image recognition using neural networks. The program does the following: it first reads a few training images to adapt the neural network to the objects to be found (a helicopter and an airplane) and then it reads another image where to look for these objects. When it finishes the recognition, ART outputs the part of the image that is most likely to contain the object, with the confidence of the match.

When running the ART benchmark on certain data sets we notice two hotspots, the *match()* and *train_match()* functions, which run together for more than 75% of the total time of the application. Therefore, we choose to apply our optimization method only on these two functions, in the idea that this yields a big benefit.

We build an MDP model with MBs containing more than one basic block. In fact, the model has states that contain code regions with one or more loops. The MDP model we use for the *match()* function is depicted in Figure 16. We build the model for two different input sizes that can be given to the ART benchmark with equal probability. The input size is represented by the *numf1s* program variable, the number of image pixels to analyze. One is an image of 10x10 pixels, the other is of 15x15. The solution policy handles both cases, as in OPFA. We put $ETTC = 0.0004532075$ seconds, for which the expected total energy consumed by the program is 4,356,620 units.

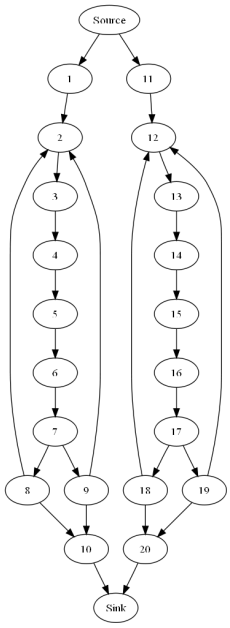

Fig. 16. The MDP model for the *match()* function for two input data sets, for only one power state.

```
#         10x10              #       15x15

1    2 P0  24     46         12 P0     34     56
2    3 P1  25     47         13 P0     35     57
3    4 P0  26 P0  48         14 P0     36     58
4    5 P0  27     49         15 P0     37     59
5    6 P0  28     50         16 P0     38     60
6    7 P0  29     51         17 P0     39     61
7    8 P1  30     52         18 P0,P1  40     62
8    9     31 P3  53         19 P0     41 P0  63
9   10     32     54         20 P3     42 P3  64
10  11     33     55 P3      21 P3     43     65 P3
```

Fig. 17. The resulting policy for the model in Figure 16.

For the 10x10 input image, the simulation with Wattch takes about 3 minutes on a 2.2 GHz Dual-core x86 laptop. For the 15x15 input image, it takes about 7 minutes. For a 100x100 image, it takes about about 8 hours.

The MDP state costs for the various P-states are computed for the *match()* function instrumented with "place-holders" for a deterministic MDP policy, in order to account, as well, for its implementation overhead.

The resulting policy is depicted numerically in Figure 17.

Notice that the policy is entirely deterministic for the input data image of 10x10. This is because the constraint is not binding. The policy for the input image of 15x15 randomizes, as expected, for only one MDP state, 18, where we choose P_0 and P_2 with probability distribution (0.404304, 0.595696). To implement the policy, we insert code in the application that observes the variable *numf1s*, to determine if the program was fed the 10x10 or 15x15 input data set. When implementing the policy, we generate the minimal code: for example, since the code regions 3, 4, 5 and 6 are in a linear sequence, we do not need to change the power state to P_0 every time, so we only implement the power change at state 4.

We present now the experimental results when running the optimal policies.

For the 10x10 input data set, we have a total execution time of 0.0001956738 seconds and total energy of 18,082,562 units.

For the 15x15 input data set, because the policy prescribes a randomized choice for the MDP state 18, we perform three experiments: for the first experiment we obtain the time of execution of function *match()* of 0.0007413127 seconds, the energy of 70,554,376 units; for the second experiment we obtain an execution time of 0.0007389353 seconds and energy of 70,867,384 units; for the third experiment we obtain a time of 0.0007436897 seconds, and energy of 70,241,000 units. Averaging these values, we obtain that the average total execution time is 0.00074131257 seconds and the average total energy is 70,554,253 units.

Now, if we average for both 10x10 and 15x15 input sets, the execution time of *match()* is 0.000468493185 seconds and the associated energy is 44,318,408 units. This means, that for both 10x10 and 15x15 image inputs, the actual average energy is 101.7% of the theoretical expected and the actual average time is 103.4% of the theoretical expected values of the optimal MDP policy, which had an objective minimum expected energy of 43,566,200 units for an $ETTC$ of 0.0004532075 s. Also, the actual EDP is 20,762.87, while the theoretical expected EDP is 19,744.53. This means the actual EDP is bigger with 5.2% than the theoretical one.

The average EDP for the input images of 10x10 and 15x15, for the P_0 P-state is 31,713.79. So, the EDP for the optimal policy is with 52.7% smaller from the EDP for P_0.

We present in Table II the above mentioned results.

TABLE II
RESULTS OF THE EXPERIMENTS WITH THE 179.ART BENCHMARK.

Scenario	Time [s]	Energy [units]	EDP
Average optimal	0.000468493185	44,318,408	20,762.87
Theoretical optimal	0.0004532075	43,566,200	19,744.53
P_0, 10x10	0.0001980854	6,352,153	3,902.1
P_0, 15x15	0.0007736680	76,939,324	59,525.49
P_0, avg	0.0008727107	41,645,738.5	31,713.79

The small differences between the theoretical values and the empirical ones are caused by the inaccuracies of the model, in the sense that: i) the exact overhead of the MDP policy was not well captured at the profiling phase, since we did not know what will be the exact policy; ii) each profiling is done for a certain power state. However, when we change the power state in the application instrumented with the policy, the transitory phenomena cannot be perfectly captured by the MDP model.

ACKNOWLEDGMENTS

The authors wish to thank Sumit Gulwani, Guy Shani, Kalyan Basu, ... for a few interesting discussions.

REFERENCES

[Alt99] Eitan Altman. *Constrained Markov Decision Processes*. Chapman and Hall/CRC, 1999.

[BBS+00] David M. Brooks, Pradip Bose, Stanley E. Schuster, Hans Jacobson, Prabhakar N. Kudva, Alper Buyuktosunoglu, John-David Wellman, Victor Zyuban, Manish Gupta, and Peter W. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6):26–44, 2000.

[BDH99] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *J. Artif. Intell. Res. (JAIR)*, 11:1–94, 1999.

[BL97] J.R. Birge and F. Louveaux. *Introduction to Stochastic Programming*. Holden-Day, Incorporated, 1997.

[BPSW08] Craig Boutilier, David C. Parkes, Tuomas Sandholm, and William E. Walsh. Expressive banner ad auctions and model-based online optimization for clearing. In *AAAI*, pages 30–37, 2008.

[BTM00] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. *International Symposium on Computer Architecture*, 0:83, 2000.

[Bur98] Douglas Christopher Burger. *Hardware techniques to improve the performance of the processor/memory interface*. PhD thesis, The University of Wisconsin - Madison, 1998. Supervisor-Goodman, James Richard.

[CMP+07] Gilberto Contreras, Margaret Martonosi, Jinzhang Peng, Guei-Yuan Lueh, and Roy Ju. The XTREM power and performance simulator for the Intel XScale core: Design and experiences. *ACM Trans. Embed. Comput. Syst.*, 6(1), February 2007.

[Ş10] Alexandru E. Şuşu. Stochastic Optimization for Environmentally Powered WSNs Using MDP Models with Multi-Epoch Actions. *ICINCO - International Conference on Informatics in Control*, 2010.

[DD05] Dmitri A. Dolgov and Edmund H. Durfee. Stationary Deterministic Policies for Constrained MDPs with Multiple Rewards, Costs, and Discount factors. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 1326–1332, Edinburgh, Scotland, August 2005.

[EBS+04] L. Eeckhout, Jr. Bell, R.H., B. Stougie, K. De Bosschere, and L.K. John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 350–361, June 2004.

[GAW07] Simon F. Goldsmith, Alex S. Aiken, and Daniel S. Wilkerson. Measuring empirical computational complexity. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 395–404, New York, NY, USA, 2007. ACM.

[GMC09] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 127–139, New York, NY, USA, 2009. ACM.

[HHZ09] Ernst Moritz Hahn, Holger Hermanns, and Lijun Zhang. Probabilistic Reachability for Parametric Markov Models. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*, pages 88–106, Berlin, Heidelberg, 2009. Springer-Verlag.

[HKH01] Chung-Hsing Hsu, Ulrich Kremer, and Michael Hsiao. Compiler-directed dynamic voltage/frequency scheduling for energy reduction in microprocessors. In *ISLPED '01: Proceedings of the 2001 international symposium on Low power electronics and design*, pages 275–278, New York, NY, USA, 2001. ACM.

[ICM06] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 359–370, Washington, DC, USA, 2006. IEEE Computer Society.

[KA07] YoungMin Kwon and Gul Agha. A Markov Reward Model for Software Reliability. *Parallel and Distributed Processing Symposium, International*, 0:335, 2007.

[Kal] Lodewijk Kallenberg. Markov Decision Processes - lecture notes, available at www.lnmb.nl/courses/lecture-notes-mdp.pdf. *2009*.

[Kwi09] Marta Z. Kwiatkowska. On Quantitative Software Verification. In *SPIN*, pages 2–3, 2009.

[LLMR07] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Sci. Comput. Program.*, 69(1-3):56–67, 2007.

[LSP08] Martin P. Lawitzky, David C. Snowdon, and Stefan M. Petters. Integrating real time and power management in a real system. In *Proceedings of the 4th Workshop on Operating System Platforms for Embedded Real-Time Applications*, Prague, Czech Republic, Jul 2008.

[LSX09] Gabriel H. Loh, Samantika Subramaniam, and Yuejian Xie. Zesto: A cycle-level simulator for highly detailed microarchitecture exploration. In *ISPASS*, pages 53–64, 2009.

[NRM+06] Alon Naveh, Efraim Rotem, Avi Mendelson, Simcha Gochman, Rajshree Chabukswar, Karthik Krishnan, and Arun Kumar. Power and thermal management in the intel core duo processor. *Intel Technology Journal*, 10(2), 2006.

[Pow07] Warren B. Powell. *Approximate Dynamic Programming: Solving the Curses of Dimensionality (Wiley Series in Probability and Statistics)*. Wiley-Interscience, 2007.

[Put94] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley-Interscience, April 1994.

[Ram65] C. V. Ramamoorthy. Analysis of computational systems: Discrete Markov analysis of computer programs. In *Proceedings of the 1965 20th National Conference*, pages 386–392, New York, NY, USA, 1965. ACM.

[Rin06] Martin Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 324–334, New York, NY, USA, 2006. ACM.

[SG07] K. Shyam and R. Govindarajan. Compiler-directed dynamic voltage scaling using program phases. In *HiPC*, pages 233–244, 2007.

[SR08] Sanjit A. Seshia and Alexander Rakhlin. Game-theoretic timing analysis. *International Conference on Computer-Aided Design*, 0:575–582, 2008.

[WMGH94] Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. Accurate static estimators for program optimization. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 85–96, New York, NY, USA, 1994. ACM.