Program Phase Detection Based Dynamic Control Mechanisms for Pipeline Stage Unification Adoption

Jun Yao Hajime Shimada Shinji Tomita
Grad. Sch. of Informatics, Kyoto Univ.
{yaojun, shimada, tomita}@lab3.kuis.kyoto-u.ac.jp

Yasuhiko Nakashima
Grad. Sch. of Information Science, NAIST
nakashima@is.naist.jp

Shin-ichiro Mori
Grad. Sch. of Engineering, Fukui Univ.
moris@fuis.fuis.fukui-u.ac.jp

Abstract

To reduce the power consumption in mobile processors, a method called Pipeline Stage Unification (PSU) is previously designed to work as an alternative for Dynamic Voltage Scaling (DVS). Based on PSU, we proposed two mechanisms which dynamically predict a suitable unification degree according to the knowledge of the program behaviors. Our results show that the mechanisms can achieve an average Energy Delay Product (EDP) decreasing of 15.1% and 19.2% for SPECint2000 benchmarks, compared to the processor without PSU.

1. Introduction

Recently, considering power consumption has shown its importance in the modern processor designing, especially for portable and mobile platforms such as mobile phone and laptop. To reduce the total energy, a method called dynamic voltage scaling (DVS) is currently employed. Basically, DVS decreases the supply voltage while the processor is experiencing low work load. This saves energy consumption for program execution.

However, Shimada et al. [1, 11, 12] and Koppanalil et al. [2] have presented us a different method to reduce the processor power consumption via inactivating and bypassing the pipeline register and using a shallow pipeline during the program execution, which is called pipeline stage unification (PSU). PSU can save power in the following ways:

1. Energy can be saved because of the clock gating of some pipeline registers.
2. After pipeline stage unification, a pipeline will become a shallow one with fewer stages. Usually, a shallow pipeline will have better IPC due to decreased branch misprediction penalties and functional unit latencies compared to the deep pipeline, as illustrated in [3] and [4].

Such designs make PSU still applicable when the efficiency of DVS is restricted by the process technology advancement, as described in paper [1].

Our research described in this paper is focusing on how to control PSU hardware to achieve a good power saving. Currently there is only one research related to PSU control [13] and it mentions about execution with predefined throughput. It did not consider the different program behaviors during the execution. In this paper, we propose some mechanisms to dynamically adjust the pipeline configuration to a suitable unification degree according to the program behavior change, so as to achieve better Energy Delay Product (EDP). By using the two different mechanisms described in this paper, we can get an average decreasing of 15.1% and 19.2% in EDP, respectively, compared with the EDP of processors under normal configuration. And compared these two mechanisms with unification degree 2, which usually have a good EDP efficiency, we can get a decreasing of 1.41% and 4.82%.

The rest of the paper is organized as follows. Section 2 describes the background techniques of this paper. Section 3 introduces the dynamic prediction mechanisms for a PSU enabled system. Simulation methodology and metrics to evaluate the efficiency of different unification degrees can be found in section 4. In section 5 we show the experiment results, together with some analysis. Section 6 concludes the paper.

2. Related works

This section describes the background techniques related to our research. Section 2.1 describes Pipeline

Stage Unification briefly and Section 2.2 introduces the working set signature method.

2.1. Pipeline stage unification

In paper [1, 11, 12], Shimada et al. proposes an energy consumption reduction method called Pipeline Stage Unification (PSU) to reduce the power consumption in mobile processors as an alternative for DVS. PSU is a pipeline reconfiguration method. Different from DVS, PSU unifies multiple pipeline stages by bypassing pipeline registers when the processor runs with low clock frequency, instead of scaling down the supply voltage.

Our work introduced in this paper is based on Shimada's previous architecture. Suppose the pipeline we are about to discuss will have 20 stages as shown in [1]. We assume 3 unification degrees in the latter part of this paper.

1. U1: The normal mode without bypassing any pipeline registers.
2. U2: Merge every pair of two adjacent pipeline stages by inactivating and bypassing the pipeline register between them. The new pipeline will have 10 stages.
3. U4: Based on U2, merge the adjacent stages one step further. It becomes a 5-stage's pipeline.

2.2. Working set signature

Dhodapkar[5] and Sherwood[6] have shown that programs can be divided into phases in which program would have similar behaviors including the cache miss, IPC and power consumption. It is described as program phase, which may contain a set of instruction intervals, regardless of temporal adjacency. This theory gives us an opportunity to study the pipeline reconfiguration at a high level, i.e., from the view of the program behavior.

In order to detect the phase changes during the program execution, Dhodapkar designed a working set signature to work as the compacted representation for a program interval. The method to form a working set signature is shown in Figure 1. "b" in Figure 1 is the number of bits which are used to index a instruction in the cache block. If an instruction cache block contains 4 instructions, b is set to 2. During the program execution, Dhodapkar selected m bits from the program counter and used these bits to address 1 bit in the n-bit signature via a hash function. The signature is cleared at the beginning of an instruction interval. After the interval begins, a bit in signature is set if the corresponding instruction cache block is touched.

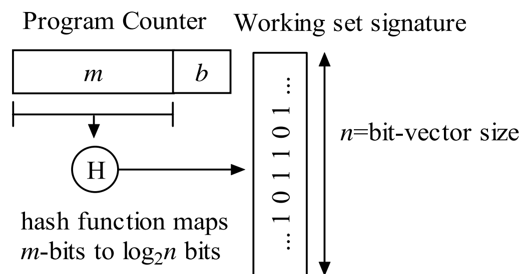

Figure 1. Mechanism for collecting working set signature [5].

Dhodapkar used a 1024-bit signature in his paper. The hash function he described is based on the C library srand and rand. He chose 100k instructions as the instruction interval length.

After collecting working set signatures, a method to calculate the distance between the two signatures S_1 and S_2 is given in [5] to classify intervals into groups. The distance δ is calculated as follows:

$$\delta = \frac{num_of_"1"bit_in(S_1 \oplus S_2)}{num_of_"1"bit_in(S_1 + S_2)} \quad (1)$$

Where num_of_"1"bit_in() represents the function that counts the number of "1" bits in the bit vector. If the distance δ is larger than a predefined threshold delta, the two instruction intervals are of different program phases. Dhodapkar used 0.5 as threshold in his paper.

3. Dynamic PSU control mechanisms

Based on the background in section 2, we can make such assumptions that since the energy consumption keeps nearly flat in a stable program phase, we can use the same pipeline stage unification degree in that phase and tune a new pipeline stage unification degree at the phase switching point. In the following sections, we design our algorithms as a framework of an interval-based loop. The word "interval" here refers to a large bundle of instructions. At each iteration, a calculation of the energy and performance over the current interval is made and passed to the algorithm core. It will be used to compare with the results of other configurations so that we can predict a suitable unification degree for the next interval based on the history result of comparison.

We applied 2 different control methods on the original PSU system. First one is basic phase detection method which needs only phase switching detection hardware. Second one is history table based method in which we add the additional table-structured hardware to store more history information for unification degree prediction.

```
After each interval I_k:
   δ =signature distance of I_k and I_k-1;
if (state == stable)
   if (δ> threshold)
      state = unstable;
      unification_degree = U1;
else if (state == unstable)
   if (δ<=threshold)
      state = tuning;
      unification_degree = U1;
else if (state == tuning)
   if (δ>threshold)
      state = unstable;
      unification_degree = U1;
   else if (unification_degree == U4)
     state = stable;
      unification_degree=best from tuning;
   else
     unification_degree
            =next tuning unification degree;
```

Figure 2. Algorithm of basic phase detection method

3.1. Basic phase detection method

We got the idea from Balasubramonian et al.[7] and Dhodapkar, et al.[5], and changed the algorithm a bit to work with the PSU system. The algorithm is shown in Figure 2. And Figure 3 outlines the execution of this algorithm.

We have three states in this algorithm:

1. stable: The adjacent intervals are of same phases;

2. unstable: A phase switching in program occurs and current interval is of different phase with last interval;

3. tuning: The period when the adjacent intervals become stable again and different unification degrees are being explored.

Figure 3 is a sample of execution. Firstly we suppose that the program starts from stable. After each program interval, we compare the signature of current interval with the signature of the previous interval. If the distance is larger than the threshold, we change the state to unstable. For simplicity, we use U1 as the unification degree for the unstable phase. The next intervals are unstable until the distance becomes smaller than the threshold again. Then we change the state to tuning, which tries different unification degrees in the following three intervals and collect the corresponding EDP. After tuning, if the interval is still under the same program phase, we can choose a best unification degree for this phase and set the state to

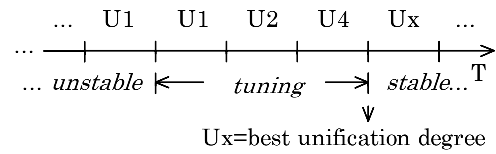

Figure 3. Outline of execution under basic phase detection method

stable. This algorithm is based on the assumption that program will show same behavior including energy, performance and so on in the same program phase.

Because we only compare the signatures of each consecutive interval pair, this method is of low cost. The corresponding control hardware will also show the advantage of simplicity.

3.2. History table based method

In order to use the feature that a phase will recur during the program execution, we designed the table based method to keep the phase information in a history table. If the program comes into a phase that has appeared in the past, we can choose a suitable unification degree from the cached history information without starting a new tuning procedure.

Figure 4 is the diagram of the hardware approach of this table method; Figure 5 shows the detailed algorithm of history table based method.

The table that we are using in this algorithm is constructed in the following way:

1. The signature field: Each different signature occupies one table entry so that we can use this field to index the table items. It has a same storage size as the signature.

2. The state field: It denotes the state of the table entry. We define two states here: tuned and tuning. A state of tuning means that this entry has just been added into the table and which one is the best unification degree is still not figured out. After all three unification degrees have been tried, we select a best unification degree from the tuned EDP results (another field in the table) and set the state as tuned. 1 bit is used for this field.

3. The EDP field occupies three fixed point storage units for each entry. It holds the EDP information for the interval represented by this signature. We keep the tuning information of different unification degrees in the three fields denoted as U1, U2 and U4, respectively. They are updated when the entry is under the tuning state.

4. The bestU field: It holds the best unification degree for this signature. This field is set after the tuning finishes. If this phase occurs again, we can predict the suitable unification degree from this field.

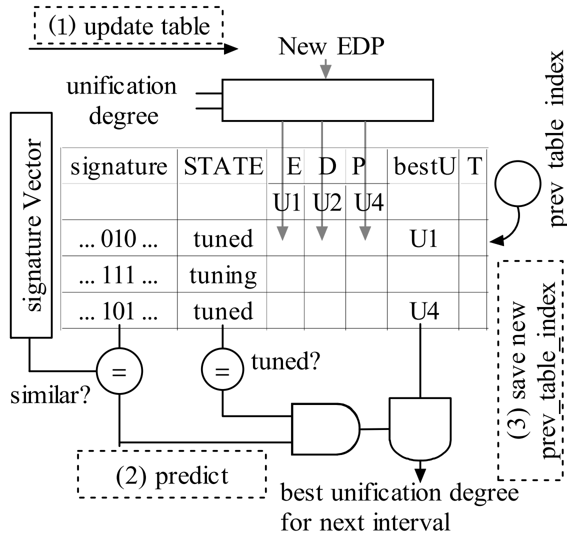

Figure 4. hardware approach for history table based method

Two bits are used for this field.

5. The T field: It records the time that the entry is touched. We use it when replacing old entries. Several bits are used according to the table size.

At the point that we are about to predict a suitable unification degree for next interval "I_{k+1}", it does not really start so that we do not know the signature. Hence we have no index to look up the history table and can hardly predict the best unification degree. To solve this problem, we engage a specific register named prev_table_index to store the table index of the previous interval. After each interval, we calculate the EDP of current interval and store it in the entry which prev_table_index refers to ((1) in Figure 4). Therefore the EDP field and best unification degree field of each current entry hold the information for the next interval. After current interval finishes, we can look for the current signature in the history table. If there is a hit, the corresponding entry will probably carry the best unification degree for the next interval. And we can predict the best unification degree based on this entry ((2) in Figure 4). The register prev_table_index will be updated to current table index before we start the next interval ((3) in Figure 4).

In Figure 5, "prev" denotes prev_table_index and "v" denotes a temporary table index. Also the syntax like "prev->state" denotes the "state" field of the entry point by prev. unif_degree is the current unification degree.

There are two main actions which will be performed on the table:

1. Find the nearest signature. We simply look up the table, comparing the new signature with all cached

```
After each interval Ik:
if (prev && prev->state==tuning)
   prev->EDP[unif_degree]=EDP for Ik;
   if (unif_degree==U4)
      prev->bestU
            =best(prev->EDP[U1, U2, U4]);
      prev->state = tuned;
v = find_nearest_signature();
δ = signature distance between v->sig and Ik;
if (!v || δ>threshold) /* miss */
   v=new_table_entry();
   v->sig=signature of Ik;
   unif_degree = U1;
   v->state = tuning;
else if (v->state == tuned)
   unif_degree = v->bestU;
else /* v->state == tuning */
   unif_degree = next unif_degree for v;
prev = v;
```

Figure 5. Algorithm of history table mode

signatures, in order to find a smallest distance. If this smallest distance is larger than the threshold, we call it a table miss and insert the new signature for the late tuning. Otherwise we say there is a table hit;

2. Replace the least recently used table entry when there is no sufficient place for the coming new signature, while we call new_table_entry() in Figure 5.

The performance of these two actions will greatly depend on the size of the table. As indicated in paper [5], a program will not show many different signatures during execution if the interval is set to be 100k instructions. We can set the table size at a small level, for example, 16 entries. Hence the overhead introduced by the looking up and replacing can be negligible. We will discuss this more detailedly in section 5.4.

In this method we have an assumption that if interval Ik+1 once happens after interval Ik and Ik occurs again, the next interval will probably be Ik+1. It is a bit like a simple history branch predictor. We can efficiently predict the best unification level for Ik+1 if the next interval for Ik is always Ik+1, while we must endure some misprediction penalty if the next interval for Ik is variable. We will show the efficiency of this method in section 5.

4. Simulation methodology

We use a detailed cycle-accurate out-of-order execution simulator, Simplscalor Tool Set [8], to measure energy and performance of different unification degrees. Table 1 lists the processor

Table 1: processor configuration

Processor	8-way out-of-order issue, 128-entry RUU, 64-entry LSQ, 8 int ALU, 4 int mult/div, 8 fp ALU, 4 fp mult/div 8 memory ports
Branch prediction	8K-entry gshare, 6-bit history, 2K-entry BTB,16-entry RAS
L1 I cache	64KB/32B line/2 way
L1 Dcache	64KB/32B line/2 way
L2 unified cache	2MB/64B line/4-way
Memory	64 cycles first hit, 2 cycles burst interval
TLB	16-entry I-TLB, 32-entry D-TLB, 128 cycles miss latency

Table 2: Assumptions of latencies and penalty

unification degree	U1	U2	U4
clock frequency rate	100%	50%	25%
branch misprediction penalty	20	10	5
L1 Icache hit latency	4	2	1
L1 Dcache hit latency	4	2	1
L2 cache hit latency	16	8	4
int Mult latency	3	2	1
fp ALU latency	2	1	1
fp Mult latency	4	2	1

configuration. We assume a deep pipeline similar to the current processors. Table 2 summarizes the latencies and penalties in pipeline configuration of U1, U2 and U4, respectively.

We used 8 integer benchmarks (gzip2, gcc, gzip, mcf, parser, perlbmk, vortex and vpr) from SPECint2000, with train inputs. 1.5 billion Instructions are simulated after skipping the first billion instructions.

To evaluate the energy and performance together in the tuning procedure, we can use PDP, EDP and EDDP as the metric, which can be calculated as W/MIPS, $W/(MIPS)^2$ and $W/(MIPS)^3$, respectively [10]. Since these equations put different emphasis on energy and performance, it will show different efficiency according to the evaluated platforms. Basically, PDP is suitable for portable systems and EDP is for some high end systems such as workstation and laptop, while EDDP is good for server families. For simplicity, we apply one single metric during one program execution. The experiments and analysis in Section 5 are based on EDP because our PSU is targeted on high-performance mobile computer. Our mechanisms can easily change to the metric of PDP or EDDP to fit for different platforms.

In this paper, we are considering the energy saving in the processor. Energy saving in U2 and U4 contains two parts (1) Energy saved by stopping clock drivers of some pipeline registers in order to inactivate and bypass them. (2) Execution time decreased by better IPC due to small latencies and penalties. We get eq.2 from paper [1, 9, 10] to calculate the energy saving under different unification degrees.

$$\frac{E_{Ux}}{E_{normal}} = \frac{IPC_{normal}}{IPC_{Ux}} \times (1-\beta) \qquad (2)$$

Where E_{Ux} is energy in unification degree Ux and Enormal is energy in normal execution mode; IPCnormal is IPC in normal execution while IPC_{Ux} is IPC in Ux; β is the power saving part from inactivated pipeline registers. Since half of the pipeline registers are inactivated in U2, we can get a β of 15%. Furthermore, for U4, an extra half of pipeline registers are inactivated, we can get a new β of 22.5%, as described in [1].

5. Results and analysis

5.1. Two non-phase based methods for comparison

Before we apply our algorithms on the PSU controller, we run the benchmarks under single unification degree method and optimal method. These two methods are used to measure the efficiency of the phase detection based algorithms.

(1) Single unification degree method

Use a fixed unification degree U1, U2 or U4 in the whole program execution and collect EDP data of each interval.

(2) Optimal method

Based on the data collected from single unification degree method, we can find a best unification degree for each interval. By using such profiling data we can set the unification degree to the best one at the beginning of each instruction interval. This method is a theoretical optimal one and can not be achieved in real execution because it is based on the post-simulated trace analysis. It will have a smallest EDP result among all the mechanisms we have mentioned. And if the EDP result of another mechanism is close to this optimal one, we can say that mechanism is efficient.

5.2. General analysis via comparing average EDP

We chose the signature size to be 1024 bits and the threshold delta to be 0.5. Each interval has 100k instructions. A simple hash function based on division is used to lower the signature collection cost. Figure 6

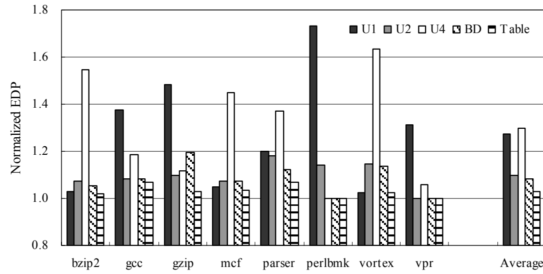

Figure 6. Normalized EDP for SPECint2000 benchmarks.

shows the EDP results for all 8 benchmarks. In Figure 6, the horizontal axis denotes benchmarks and the average value, and the vertical axis denotes EDP value normalized by EDP of the optimal method for each benchmark. The columns in one benchmark represent the normalized EDPs of U1, U2, U4, basic phase detection and history table based method, from left to right. Also the average results of all benchmarks are listed. The method of smaller EDP result is more efficient.

As shown in Figure 6, we can see that not all of the benchmarks will show the smallest EDP results under a single unification degree. For benchmarks like bzip2, mcf and vortex, U1 is the most energy efficient unification degree, and for benchmarks like perlbmk, degree U4 has the smallest EDP result. For other benchmarks, including gcc, gzip, parser and vpr, U2 is better than U1 or U4. These results confirm our assumption that there is no fixed pipeline configuration which can always have best energy performance efficiency for all the programs, and reconfigurations during the execution are necessary.

For the efficiency of our mechanisms, Figure 6 shows that the basic phase detection method can achieve an average EDP of 108%, compared with the optimal method. And it obtains a decreasing of 15.1%, 1.41% and 16.4% when compared with single U1, U2 and U4, respectively.

The history table based method shows better average results, as compared with the basic phase detection method. It can achieve an average EDP of 103% of the optimal method. Compared with single U1, U2 and U4, it can gain a total EDP decreasing of 19.2%, 4.82% and 20.5%.

We can see from these results that both basic detection method and table based method can have some efficiency in reducing the processor energy consumption by prediction the next suitable pipeline unification degree. And table based method is a bit more effective since it caches more history information which can reduce the tuning cost, as we have expected.

Table 3. Prediction accuracy of each benchmark, together with benchmark characteristics.

Bench-mark	Stable Rate (%)	nSigs	Avg. ST_Len.	Pred. Acc.(%)	
				BD	Table
bzip2	86.80	12	28.93	85.78	94.73
gcc	89.73	55	53.84	60.77	51.95
gzip	59.18	3	9.575	66.58	84.68
mcf	32.98	6	2.382	41.30	49.80
parser	67.63	33	11.75	49.24	60.12
perl.	99.97	1	14995	99.98	99.98
vortex	51.74	6	4.720	46.16	87.41
vpr	99.97	1	14995	99.98	99.98

5.3. Prediction accuracy

Since we are designing the dynamic mechanisms to predict a suitable unification degree for the next interval, the prediction accuracy is very important to the final energy saving result. To study the efficiency of the design methods more detailedly, we list the prediction accuracy of the unification degrees in table 3, together with some benchmark characteristics.

In table 3, the column of stable rate stands for the percentage of the total intervals that are in stable time. The “nSigs” column denotes the number of different signatures when the programs are under the stable time. We obtain this value by comparing the signatures of two stable phases. If the distance is larger than the predefined threshold, we increase this count by 1. It can be roughly used to represent the complexity of the benchmark. A higher value shows that the programs can be classified into more different stable phase groups and may require more tunings. It may potentially increase the complexity for dynamical prediction. The column of “Avg. ST_len” represents the average interval length of the stable phase for each benchmark. These three columns are the statistical results we got from the basic detection method. The accuracy of using working set signature to identify the program phase is important for further reconfiguration on processor. It has been valued in paper [5] by Dhodapkar.

Another column named “Pred. Acc.” in table 3 is the ratio of precise prediction of the unification degree for basic detection method and table based method, respectively. We got these two columns by calculating the similarity of predicted unification degrees with those theoretical precise unification degrees from optimal method. In order to show the efficiency of history table based method optimally, we simply choose an infinite table size in table 3. Fixed table size will be discussed in section 5.4.

Basically, from table 3, we can see that the prediction accuracy of table based method is better than the basic detection based method. This is similar with the conclusion we have obtained in section 5.2.

Also from this table, we can see that the prediction accuracy changes due to the program characteristics. For some simple benchmarks like perlbmk and vpr, most intervals are of the same stable phase. For these two benchmarks, the prediction accuracy of both dynamical methods can reach nearly 100%. The prediction accuracy of the basic detection based method drops visibly when the program becomes less stable. This may related with the simple design of the basic detection method. We only compare the signatures of consecutive intervals and start a tuning at each point the program goes toward stable. If the stability of program is low, the basic detection method will get hurt because we can hardly save energy in unstable and tuning phase.

Different with basic detection method, history table base method is less sensitive to the program stability. It is well illustrated from benchmarks like gzip and vortex. Although the stability ratios for these two benchmarks are lower than 60%, the prediction accuracy can still reach 84.68% and 87.41%, respectively. This is because the table based method records the historical tuned information in extra structures. If the jump direction from one signature to another signature is stable, the prediction will be accurate. But on the other hand, this method is sensitive to the number of phase groups. For example, gcc is quite stable but the number of different signatures during stable phase is large, which lead to the uncertainty in the jump directions. More detailed results of table based method for gcc will be listed in section 5.4.

Some simulation configuration like the threshold and the signature size will affect the efficiency of the phase detection so as to have final impact on the energy saving results obtained by our two dynamical methods based on the signature. We have tried several threshold values such as 0.5, 0.25 and 0.1, and found that the value of 50% was the most efficient one. Also, different signature size like 1024-bit and 256-bit have been tested. The results of 1024-bit are slightly better than 256-bit but the difference is not dominant. Due to the paper length, we are not going to list the detailed results in this paper.

5.4. Table size

The size of the history table is another important parameter for the table based method. We can see from table 3 that the numbers of different signatures in

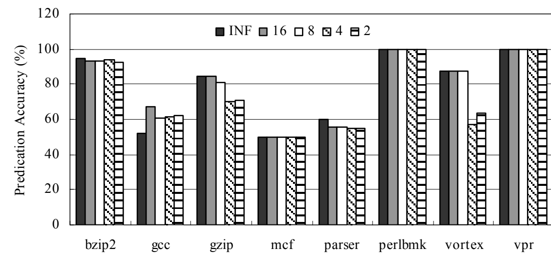

Figure 7. Prediction accuracy of different table size for table based method

stable phases are relatively small for most of the benchmarks. Also most of the benchmarks are quite stable. Therefore it is possible to set a small fixed table size without degrading the ratio of prediction accuracy.

In this serial of experiments, we set the table size as a fixed number, from 16 entries to 2 entries. We use the LRU mechanism to replace the table entry when there is no sufficient place for new signature. The results are shown in Figure 7. A signature size of 1024 bit and a threshold of 0.5 are used for the configuration.

In Figure 7, the columns for each benchmark represent the results of infinite table size, 16-entry, 8-entry, 4-entry and 2-entry, respectively. We can see from the results that there is almost no degradation between the infinite-entry, 16-entry and 8-entry for all benchmarks. A sharp decreasing of prediction accuracy occurs on 4-entry table size for gzip and vortex. Other benchmarks like bzip2, perlbmk and vpr show no loss of accuracy even when the size shrinks to 2-entry. And for benchmark gcc, the accuracy even increases after we reduce the table size from infinite to 16 entries. The results of 8-entry and 4-entry are also better than the infinite one. It seems that for gcc, the old history information may sometimes have a bad impact in helping the prediction.

From these results, we can assume that an 8-entry table size will be sufficient for SPECint2000 benchmarks. With a small table size, we can look up the table faster so as to introduce less overhead into the PSU control system.

6. Conclusions and future work

In this paper, we have designed two dynamic control mechanisms for PSU enabled processors in order to achieve good EDP. These two mechanisms are based on phase detection via working set signature. By using these two methods, we can dynamically reconfigure the unification degree during the program execution due to the program behavior change. Our simulation show that the two methods can achieve an average EDP decreasing of 15.1% and 19.2%, compared to the original system without PSU enabling.

Such results are about 8.34% and 3.02% larger than the optimal mode. Both methods can reduce energy consumption in processor via dynamically predicting a unification degree for the coming interval. Either of the dynamical methods shows some advantages. The basic detection method is simple and introduces less hardware complexity, while the history table based method shows better efficiency in predicting.

Currently the energy consumption model in this paper is still very rough. We are planning to study the hardware approach so as to build a more accurate model, including the detailed overhead introduced by the dynamical prediction mechanisms. Also, different program phase detection methods other than the working set signature will be tried on the PSU system.

Acknowledgement

This research is partially supported by Grant-in-Aid for Fundamental Scientific Research (S) #16100001 from Ministry of Education, Culture, Sports, Science and Technology Japan.

7. References

[1] H. Shimada, H. Ando and T. Shimada, "Pipeline Stage Unification: A Low-Energy Consumption Technique for Future Mobile Processors", *International Symposium On Low Power Electronics And Design 2003,* Aug. 2003, Seoul, Korea, pp. 326-329.

[2] J. Koppanalil, P. Ramrakhyani, S. Desai, A. Vaidyanathan and E. Rotenberg, "A Case for Dynamic Pipeline Scaling", *International Conference on Compilers, Architecture and Synthesis for Embedded Systems 2002,* Oct. 2002, Grenoble, France, pp. 1-8.

[3] M.S. Hrishikesh, N.P. Jouppi, K.I. Farkas, D. Burger, S.W. Keckler and P. Shivakumar. "The Optimal Logic Depth Per Pipeline Stage is 6 to 8 FO4 Inverter Delays", *29th Annual International Symposium on Computer Architecture*, May 2002, Alaska, U.S., pp. 14-24.

[4] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P.N. Strenski and P.G. Emma, "Optimizing Pipelines for Power and Performance", *35th Anuual International Symposium on Microarchitecture*, Nov. 2002, Istanbul, Turkey, pp. 333-344.

[5] A.S. Dhodapkar and J.E. Smith, "Managing Multi-Configuration Hardware via Dynamic Working Set Analysis", *29th Annual International Symposium on Computer Architecture*, May 2002, Alaska U.S., pp. 233-244.

[6] T. Sherwood, E. Perelman, G. Hamerly, S. Sair and B. Calder, "Discovering and Exploiting Program Phases", *IEEE Micro*, Vol. 23, No. 6, Nov.-Dec. 2003, pp. 84-93.

[7] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu and S. Dwarkadas, "Memory Hierarchy Reconfiguration for Energy and Performance in General Purpose Architectures", *33rd annual International Symposium on Microarchitecture*, Dec. 2000, California, U.S., pp. 245-257.

[8] D. Burger and T.M. Austin. "The SimpleScalar Tool Set, version 2.0. Technical Report", CS-TR-97-1342, Univ. of Wisconsin-Madisoon Computer Sciences Dept., 1997.

[9] M.K. Gowan, L.L. Biro and D.B. Jackson, "Power considerations in the Design of the Alpha 21264 Microprocessor", *35th Conference on Design Automation Conference*, June, 1998, San Francisco, U.S., pp. 726-731.

[10] R. Gonzalez and M. Horowitz, "Energy Dissipation in General Purpose Microprocessors", *IEEE JSSC*, Vol. 31, No. 9, Sep. 1996, pp. 1277-1284.

[11] H. Shimada, H. Ando and T. Shimada, "Pipeline with Variable Depth for Low Power Consumption (in Japanese)", *IPSJ Technical Report*, 2001-ARC-145, 2001, pp. 57-62.

[12] H. Shimada, H. Ando and T. Shimada, "Pipeline Stage Unification for Low-Power Consumption," *Cool Chips V*, Apr. 2002, Tyoko, Japan, pp. 194-200.

[13] H. Shimada, H. Ando, T. Shimada, "A Hybrid Power Reduction Mechanism Using Pipeline Stage Unification and Dynamic Voltage Scaling (in Japanese)", *Symposium on Advanced Computing Systems and Infrastructures 2004,* May 2004, Sappro Japan, pp. 11-18.