Combining Edge Vector and Event Counter for Time-dependent Power Behavior Characterization

Chunling Hu Daniel A. Jiménez Ulrich Kremer
Department of Computer Science
{chunling, djimenez, uli}@cs.rutgers.edu

Rutgers University, Piscataway, NJ 08854 USA

Abstract. Fine-grained program power behavior is useful in both evaluating power optimizations and observing power optimization opportunities. Detailed power simulation is time consuming and often inaccurate. Physical power measurement is faster and objective. However, fine-grained measurement generates enormous amounts of data in which locating important features is difficult, while coarse-grained measurement sacrifices important detail.

We present a program power behavior characterization infrastructure that identifies program phases, selects a representative interval of execution for each phase, and instruments the program to enable precise power measurement of these intervals to get their time-dependent power behavior.

We show that the representative intervals accurately model the fine-grained time-dependent behavior of the program. They also accurately estimate the total energy of a program. Our compiler infrastructure allows for easy mapping between a measurement result and its corresponding source code. We improve the accuracy of our technique over previous work by using *edge vectors*, i.e., counts of traversals of control-flow edges, instead of basic block vectors, as well as incorporating event counters into our phase classification.

We validate our infrastructure through the physical power measurement of 10 SPEC CPU 2000 integer benchmarks on an Intel Pentium 4 system. We show that using edge vectors reduces the error of estimating total program energy by 35% over using basic block vectors, and using edge vectors plus event counters reduces the error of estimating the fine-grained time-dependent power profile by 22% over using basic block vectors.

1 Introduction

Research in power and energy optimizations focuses not only on reducing overall program energy consumption, but also on improving time-dependent power behavior. Evaluating such optimizations requires both accurate total energy consumption estimation and precise detailed time-dependent power behavior. Simulators are often used for power and performance evaluation, but detailed power simulation is very time-consuming and often inaccurate. While physical measurement is much faster, fine-grained power measurement requires proper measurement equipment and a large amount of space to store measurement results.

An example optimization that requires fine-grained, time-dependent power behavior information for its experimental evaluation is instruction scheduling for peak power and

step power (dI/dt problem) reduction, for instance in the context of VLIW architectures [1–3]. This previous work relies on simulation to evaluate the impact of the proposed optimizations. The dI/dt problem is caused by large variations of current in a short time. Such variations in CPU current may cause undesired oscillation in CPU supply voltage, which may results in timing problems and incorrect calculations [4]. In this paper, we introduce a new strategy to enable time-dependent power behavior characterizations based on physical measurements.

1.1 Characterizing Phases with Representative Slices

Program phase behavior shows that many program execution slices have similar behavior in several metrics, such as instructions-per-cycle (IPC), cache miss rate, and branch misprediction rate. Phase classification makes it easier to measure the fine-grained program behavior. A representative slice from each phase instead of the whole program execution is measured and analyzed, and then the whole program behavior can be characterized based on the analysis result. Using this whole program behavior characterization method in power behavior analysis, we can obtain fine-grained power behavior with significant savings in both time and storage space.

1.2 Illustrating Time-Dependent Power Behavior

Figure 1 shows the measured CPU current of *256.bzip2* from SPEC CPU 2000 measured using an oscilloscope. Figure 1(a) shows that the program execution can be roughly partitioned into 4 phases based on its power behavior. One representative slice from each phase can be measured to characterize the detailed power behavior of the benchmark. Figure 1(b) is the measured power behavior of half of a second in the first phase with a resolution that is 100 times higher than the one used for Figure 1(a). There is a repeated power behavior period of 300 milliseconds. Figure 1(c) shows the detailed power behavior of a piece of 0.05 second, from 0.1 second to 0.15 second in Figure 1(b). It shows repeated power behavior periods of less than 5 milliseconds, indicating possible finer phase classification than Figure 1(b). Also, finer measurement gives more information of time-dependent CPU power due to the resolution of the oscilloscope that we use for power measurement. The oscilloscope reports the average power for a given time granularity. This is the reason why the difference between the observed peak power (peak current) in Figure 1(a) and (c) is almost 6 Watts (0.5 amperes).

1.3 An Infrastructure for Characterizing Time-Dependent Power Behavior

In this paper, we present our infrastructure for program time-dependent power behavior characterization and optimization evaluation. Our *Camino* compiler statically instruments the assembly code of a program for profiling and physical measurement. A SimPoint-like [5] method is used for phase classification. SimPoint identifies several intervals, or *simpoints*, of program execution that characterize the behavior of the entire program execution. It is often used to speed up simulation by simulating only the simpoints and estimating, for instance, IPC, by taking a weighted average of the IPCs of each simpoint.

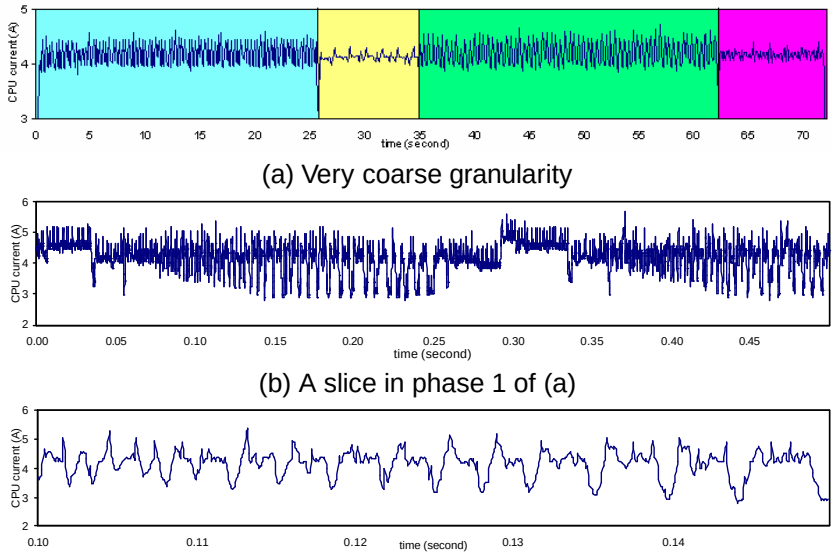


Fig. 1. Measured power behavior of bzip2 with different granularity.

SimPoint uses the Basic Block Vector (BBV), i.e., a vector of counts of basic block executions, as the feature for classification. We introduce the *edge vector* (EV), i.e., a vector of counts of control-flow-graph edge traversals, as the fingerprint of each interval of the program execution.

Instead of using a fixed number of instructions as interval length, we use infrequently executed basic blocks to demarcate intervals. This results in variable interval length, but much lower instrumentation overhead for physical power measurement of a representative interval. The selected simpoints are weighted based on the number of instructions executed in each phase, instead of number of intervals.

We show that our method enables us to do power measurement for simpoints with very low interference to program execution. To demonstrate the improved accuracy of using edge vectors for classification, we show that our infrastructure estimates the total energy of a program with an average error of 7.8%, compared with 12.0% using basic block vectors, an improvement of 35%. More importantly, we want to find representative intervals that represent the fine-grained time-dependent power profile of a phase. We develop a metric for measuring the accuracy of estimating a power profile and show that using edge vectors with event counter information improves accuracy by 22%.

Unlike simulation, physical measurement is sensitive to the overhead for identification of simpoints during program execution. So this low instrumentation overhead is very important. This infrastructure can be used to evaluate optimizations for energy consumption or time-dependent power behavior, for example, the impact on power behavior of pipeline gating [6] or dynamic voltage/frequency scaling [7].

We evaluate our infrastructure by measuring 10 SPEC CPU2000 integer benchmarks on a Pentium 4 machine, and we present the error rates in whole program energy consumption estimation as well as fine-grained power behavior estimation based on the measurement result of the selected simpoints.

This paper makes the following contributions: 1) We show that using edge vectors significantly improves accuracy over using basic block vectors for estimating total program energy as well as fine-grained power behavior. 2) We show that classification accuracy can further be improved by combining control-flow information such as edge vectors with event counter information. 3) We present our infrastructure that uses edge vectors and event counters to select representative intervals and to measure efficiently their power profiles with minimal perturbation of the running program.

2 Related Work

Several techniques have been proposed to identify program phases. Some of them use control-flow information [8, 5, 9–11], such as counts of executed instructions, basic blocks, loops, or functions, as the fingerprint of program execution. This fingerprint depends on the executed source code. Some methods depend on run-time event counters or other metrics [12–15], such as IPC, power, cache misses rate and branch misprediction, to identify phases. Our infrastructure uses the edge vector of each interval, a vector that gives a count for each control-flow edge in the program, along with the measured IPC. This set of features allows for a precise characterization of power phases.

SimPoint [8, 5] partitions a program execution into intervals with the same number of instructions and identifies the phases based on the BBV of each interval. One interval, called a *simpoint*, is selected as the representative of its phase. These simpoints are simulated or executed to estimate the behavior of the whole program execution. Sherwood *et al.* apply SimPoint to SPEC benchmarks to find simpoints and estimate the IPC, cache miss rate, and branch misprediction rate. The error rates are low and the simulation time saving is significant.

A new version of SimPoint supports variable length intervals. Lau *et al.* [11] shows a hierarchy of phase behavior in programs and the feasibility of variable length intervals in program phase classification. They break up variable length intervals based on procedure call and loop boundaries. We use infrequent basic blocks to break up intervals and at the same time use a pre-defined length to avoid too long or too short intervals. This satisfies our requirement for low-overhead instrumentation and accurate power behavior measurement. Besides phase classification, we also generate statically instrumented executables for physical measurement of simpoints and CPU peak power control on a dual core machine.

Shen *et al.* [9] propose a data locality phase identification method for run-time data locality phase prediction. A basic block that is always executed at the beginning of a phase is identified as the marker block of this phase, resulting in variable interval lengths. They introduce the notion of a phase hierarchy to identify composite phases.

We also use variable interval lengths, but the basic block that marks a phase is not necessary to uniquely mark the phase. It might be the mark for other phases. Phases

are identified by the execution times of the infrequent basic blocks that demarcate the intervals, such that we implement precise physical measurement.

PowerScope [16] maps energy consumption to program structure through runtime system power measurement and system activity sampling. System components responsible for the bulk of energy consumption are found and improved. The delay between power sampling and activity sampling results in possible imprecise attribution of energy consumption to program structure. Compared to the power measurement granularity used by PowerScope, which is 1.6ms, our infrastructure measures CPU current with much higher granularity. 1000 samples are collected for each 4ms. Precise mapping between power measurement and program structure is achieved through measuring the selected representative intervals.

Isci and Martonosi [17] show that program power behavior also falls into phases. Hu *et al.* propose using SimPoint to find representative program execution slices to simplify power behavior characterization, and validate the feasibility of SimPoint in power consumption estimation through power simulation of some Mediabench benchmarks [18]. Isci and Martonosi [19] compare two techniques of phase characterization for power and demonstrate that the event-counter-based technique offers a lower average power phase classification errors.

Our goal is to characterize the time-dependent power behavior, instead of power consumption, of programs. Our method causes negligible overhead for identification of an interval during program execution, and the measurement result is very close to the real time-dependent power behavior of the interval. Furthermore, through the combination of edge vector and event counters, we get better phase characterization than using only control flow information as well as the mapping between observed power behavior and the source code. The latter is difficult for an event-counter-based technique by itself.

3 Phase Classification Based on Edge Vectors and Event Counters

Our phase classification infrastructure is based on the ability to demarcate the start and end of a particular interval of execution with infrequently executed basic blocks. We instrument these infrequent basic blocks so that our instrumentation minimally perturbs the execution of the program.

Phase classification and power measurement of representative intervals for programs is implemented as an automatic process. The threshold for determining whether a basic block is infrequent, the minimum number of instructions in each interval, and the number of phases are the input to this process. The flowchart in Figure 2 illustrates its steps. The implementation of each step will be presented in the following sections.

3.1 Instrumentation Infrastructure for Profiling, Measurement, and Optimization

Camino [20] is a GCC post-processor developed in our lab. We use it to implement the static instrumentation for profiling and physical power measurement.

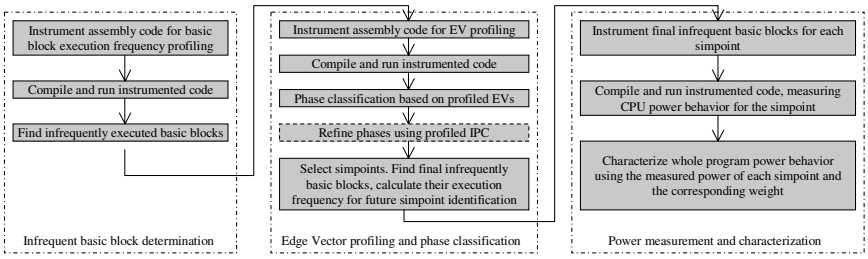


Fig. 2. Infrequent basic block-based phase classification and power measurement of simpoints.

Camino reads the assembly code generated by GCC, parses it into a control-flow-graph (CFG) intermediate representation, performs transformations including instrumentation, and then writes the modified assembly language to a file to be assembled and linked.

Instrumentation using *Camino* is simple and minimally intrusive. Only two routines are required: an instrumentation routine that inserts a call to the analysis routine, and and an analysis routine that does profiling or generates special signals.

Note that our infrastructure does two kinds of instrumentations: 1) profiling for all basic blocks to identify infrequent basic blocks and gathering features used to do phase classification, and 2) infrequent basic block instrumentation for signaling the start and end of a representative interval to our measurement apparatus. The first kind of instrumentation results in a moderate slowdown, but the second kind results in no slowdown so that the measured program's behavior is as close as possible to that of the uninstrumented program.

3.2 Infrequent Basic Blocks Selection

Instrumentation is done through *Camino* to collect the execution frequency of each basic block. Each basic block makes a call to an execution frequency counting library function. The distinct reference value of the basic block is passed to the function that increments the frequency of this basic block. During the first profiling pass, we collect counts for each basic block.

A threshold is needed to determine which basic blocks are infrequently executed and can be used to demarcate intervals. An absolute value is infeasible, since different program/input pairs execute different number of basic blocks. Instead, we consider a basic block to be infrequent if it account for less than a certain percentage of all executed basic blocks. Intuitively, when a low threshold is used, the selected infrequent basic blocks will be distributed sparsely in program execution and there is more variance in interval size than when a higher threshold is used. We investigate 4 different threshold values, 0.05%, 0.1%, 1%, and 5%, to explore the trade-off between interval size variance and instrumentation overhead.

3.3 Program Execution Interval Partitioning and Edge Vector Profiling

We use the edge vector (EV) of all edges as the fingerprint of an interval used for the clustering phase of our SimPoint-like phase classification method. This vector is the absolute count for each control-flow edge traversed during the execution of an interval. Compared to basic block vectors (BBV), EVs give us more information about the control behavior of the program at run-time. BBVs contain information about what parts of a program were executed, but EVs tell us what decisions were made in arriving at these parts of the program. This extra information allows a classification of phases that more accurately reflects program behavior. For the same BBV, it is possible that there are several EVs depending on the dynamic paths taken during program execution. An example is shown in Figure 3.

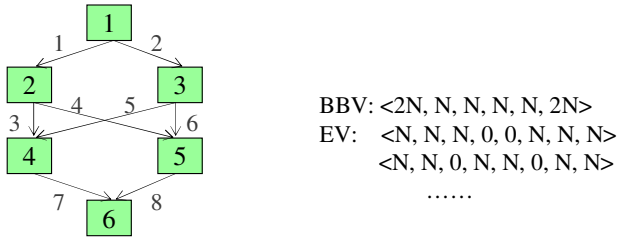


Fig. 3. Several EVs are possible for the same BBV.

Partitioning program execution just based on the execution times of infrequent basic blocks will results in intervals with a variable number of instructions. Acquiring detailed information from very large intervals to characterize program behavior is inefficient. Moreover, a large variance in interval size affects the accuracy of the phase classification result. In order to make use of our physical measurement infrastructure to characterize the whole program behavior more correctly and efficiently, we use a pre-specified interval size to avoid largely variable intervals.

Instrumentation for EV profiling is similar to that for basic block execution frequency profiling. An pre-defined interval size of 30 million instructions is used to avoid too large or too small intervals. All basic blocks are instrumented so that we can get the complete fingerprint of an interval. The library function remembers the last executed basic block and knows the taken edge based on the last and the current executed basic blocks. It counts each control flow edge originating in a basic block that ends in a conditional branch. It counts the total number of executed instructions for the current interval as well. When an infrequent basic block is encountered, if the count is larger than or equal to 30 million, this basic block indicates the end of the current interval and it is the first basic block of the next interval.

Note that, because we only have coarse control over where the demarcating infrequent basic blocks will occur, the actual interval might be somewhat longer than 30 million instructions; thus, the intervals are variable-length.

Figure 4 illustrates the interval partition using the combination of infrequent basic block and interval size. Here A, B, C, and D are basic blocks. C and D are infrequent and used to demarcate intervals. Since we use a pre-defined interval size, 30 million, only the occurrences of C and D in shadow mark intervals. Other occurrences do not mark intervals because the interval size is smaller than 30 million when they are encountered. We get intervals of similar size by using this method. An execution frequency counter of C and D can be used to identify the exact execution of an interval. For example, the fourth interval starts when the counter is 5 and ends when the counter is 8.

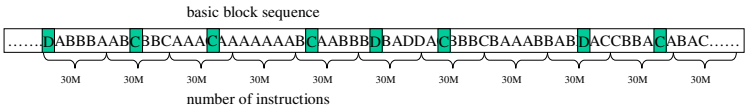


Fig. 4. Interval partitioning using infrequent basic blocks and interval length.

3.4 Phase Classification for Power Behavior Characterization

Intervals profiled in Section 3.3 are classified into phases based on their EVs. K-Means clustering is used to cluster the intervals with similar EVs and select a representative for each phase. The EV of each interval is projected to a vector with much smaller dimension. Then k initial cluster centers are selected. The distance between a vector and each center is calculated and each vector is classified into the cluster with the shortest distance. A cluster center is changed to the average of the current cluster members after each iteration. The iteration stops after the number of vectors in each cluster is stable. The simpoint of a phase is the one that is closest to the center of the cluster [5].

Since the intervals are demarcated by infrequently executed basic blocks and have variable number of instructions, weighting a simpoint with just the number of intervals in its phase cannot reflect the real proportion of this phase in whole program execution. In our method, each simpoint has two weights. One is based on the percentage of the number of executed instructions of the corresponding phase in that of the whole program, the other is based on the number of intervals in the corresponding phase as the one used in [5].

A recent version of the SimPoint tool also supports variable-length phases [11].

Given the number of phases, K-Means clustering is performed for different number of clusters and different cluster seeds. The BIC (Bayesian Information Criterion) score of each clustering is calculated and used to choose the clustering with the best trade-off between BIC score and number of phases. The BIC score calculation in our method is changed to use the number of executed instructions in each phase such that phases with longer intervals have larger influence.

To identify an interval during program execution, we need to find the beginning and end of the interval. We use the execution frequencies of one or two infrequent basic

blocks that demarcate the interval. Infrequent basic blocks that mark the start or end of a desired representative interval are chosen as final infrequent basic blocks. Their execution frequencies in each interval are recorded, so that we know how many times a basic block has executed before the start of an interval. We instrument these final infrequent basic blocks with simple assembly code to increment a counter and trigger power measurement when the count indicates the beginning of the interval, or turn off measurement when the end of an interval is reached. The combination of infrequent basic blocks and static instrumentation enables us to identify the execution of an interval at run-time with negligible overhead.

3.5 Finer Phase Classification Using IPC

Two intervals that execute the same basic blocks may generate different time-dependent power behavior due to run-time events, such as cache misses and branch mispredictions. Phase classification only based on control flow information cannot precisely differentiate these intervals, so the resulting simpoints may not really be representative in terms of power behavior. Our infrastructure combines EV and instructions-per-cycle (IPC) as measured using performance counters provided by the architecture to take the run-time events into account.

IPC Profiling Profiling IPC is easy to do in our infrastructure. After the program execution is partitioned into intervals, all of the infrequent basic blocks that demarcate the resulting intervals are instrumented to collect the number of clock cycles taken by each interval. By running the instrumented program once, we can get the IPC values of all intervals by dividing the number of instructions by the number of cycles. We already have the number of instructions executed from the edge vector profiling. This technique very slightly underestimates IPC because of system activity that is not profiled, but we believe this has no impact on the accuracy of the classification since IPC tends to vary significantly between phases. Since we identify intervals based on infrequent basic block counts, the overhead is low and has a negligible impact on the accuracy of the profiling result.

Combining EV Clustering with IPC Clustering For a program execution, we first perform the phase classification in Section 3.4 to group intervals with similar EVs together. Then we do another phase classification based on the profiled IPC values. K-Means clustering is also used in the second phase classification. Then we combine the results from the two classifications and get a refined phase classification for power behavior characterization through refining the classification result of the first one using that of the second one. The mechanism in the next section performs more control on the number of the resulting phases without a significant loss in accuracy. Our experiment result shows that after applying the controlling mechanism, if the number of phases identified based on IPC is 10, the number of the resulting phases after the classification refinement is expanded to less than 3 times of the number after the first classification, instead of around 10 times.

Controlling Unnecessarily Fine Phase Classification Using a constant K value for the IPC-based phase classification of all programs results in unnecessarily fine partitioning and more simpoints to simulate or measure when the IPC values of the intervals in the same phase are already very close to each other. We control the number of resulting phases based on IPC in two steps.

The first step controls the selection of the initial centers based on the maximum and minimum IPC of the program. A percentage of the minimum IPC value is used as the distance d between the initial centers. This ensures that intervals with very close IPCs need no further partitioning and the final number of simpoints does not explode with little benefit. This percentage is adjustable in our infrastructure. The maximum value is divided by d. The value of quotient plus 1 is then compared with the given k. The smaller one is used as number of clusters. This value may be 1, meaning that the IPC values of all of the intervals are very close and no finer partitioning is necessary.

The second step maintains the distance between centers during the initialization of the centers in case there is a IPC much higher than others, but there are only two different IPC values during program execution. The first step does not know this and the number of clusters will be k which results in unnecessarily more simpoints. This step is similar to the construction of a minimum spanning tree except that we use the largest values in each step to choose the next initial center. The first initial center is selected randomly. During the generation of the other initial centers, each time the value with largest distance to the existing centers is the candidate. If this distance value is less than half of d, no more initial centers are generated. This prevents intervals with the similar EVs and very close IPCs from being partitioned into different clusters.

4 Experimental Setup

We validate our infrastructure through physical power measurement of the CPU of a Pentium 4 machine. This machine runs Linux 2.6.9, GCC 3.4.2 and GCC 2.95.4. Benchmarks are from the members of SPEC CPU2000 INT that can be compiled by *Camino* successfully. The back-end compiler for *gzip, vpr, mcf, parser* and *twolf* is GCC 3.4.2. The back-end compiler for the other benchmarks is GCC 2.95.4 because the combination of *Camino* and GCC 3.4.2 fails to compile these programs correctly. We measure the current on the separate power cable to the CPU using a Tektronix TCP202 DC current probe, which is connected to a Tektronix TDS3014 oscilloscope. The experimental setup is shown in Figure 5. The data acquisition machine is a Pentium 4 Linux machine that reads data from the oscilloscope when a benchmark is running on the measured system. Simultaneous benchmark execution and power data acquisition on different machines eliminates interference with the measured benchmark. The picture on the right of Figure 5 is our experimental setup, data acquisition machine is not shown in the picture.

The oscilloscope has a TDS3TRG advanced trigger module. When it is in trigger mode, it accepts trigger signals from one of its four channels. We use its edge trigger. It starts measurement only after the voltage or current on the trigger channel increases to some predefined threshold and stops when its window fills to its capacity. The data points stay in the buffer until the next trigger signal. We generate the trigger signal

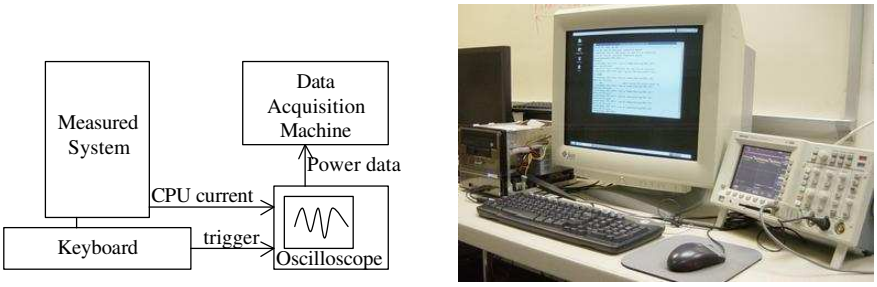


Fig. 5. The physical measurement infrastructure used in the experiments.

by controlling the *numlock* LED on the keyboard. A voltage probe is connected to the circuit of the keyboard to measure the voltage on the LED as shown in Figure 5. The voltage difference between when the light is on and off is more than 3.0V, which is enough to trigger the oscilloscope. The voltage on the trigger channel is set to high by instrumentation code to trigger the oscilloscope at the beginning of the program slice to measure. This voltage is consistently high until it is set to low at the end of this slice. It is easy to identify the power behavior of the measured slice.

4.1 Instrumentation Overhead

In order to get the power behavior close to the real program power behavior, the instrumentation overhead should be as low as possible to reduce its impact on the measured power behavior. We instrument all of the infrequent basic blocks that demarcate the final simpoints to evaluate the overhead. The instrumented code does the same thing as it does to generate signals before and after each simpoint, but controls another LED. Thus, we get the same overhead as when the CPU power of a simpoint is measured, and still can use the *numlock* to generate signals to get the precise measurement of each program. If we measure the simpoints one by one, the overhead is even lower than the one measured in this experiment, since only one or two basic blocks are instrumented.

We use the auto mode of the oscilloscope to measure the power behavior of the whole benchmark execution and still identify the exact power data points for the benchmark by setting the voltage on the trigger channel to high and low before and after the execution of each benchmark. However, no instrumentation is needed to generate signals during program execution. The oscilloscope records power data points continuously, and the data acquisition program running on another machine collects the data points. We adjust the data acquisition to read the data in each window without losing data points or reading duplicated data points due to a data reading period that is too long or too short, respectively. This is validated through the comparison of the real benchmark execution time and the one obtained from the measurement result. To evaluate the instrumentation overhead, we also measure the power consumption of the 10 benchmarks without any instrumentation.

4.2 Energy Consumption Estimation Based on Simpoints

The first step to verify that this infrastructure is useful in power behavior characterization is to calculate the error rate when the measurement result of the selected simpoints is used to estimate the power consumption of the whole program. Although we use EVs as the fingerprint of an interval in our infrastructure, we also measured the CPU power of the simpoints using BBVs for comparison.

The energy consumption of each simpoint is measured using the trigger mode of the oscilloscope. We generate an executable for each simpoint and measure the simpoints one by one so we can get very high resolution as well as the lowest possible instrumentation overhead. Program execution and data acquisition are on the same machine. Reading data from the oscilloscope is scheduled after the measurement of a simpoint is done. Data acquisition does not interfere with the running program. We implement an automatic measurement and data acquisition process to measure any number of simpoints as a single task.

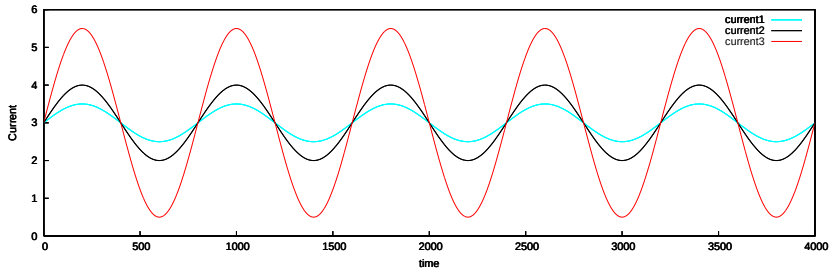
4.3 Power Behavior Similarity Evaluation

Even though we can get low error rates in estimating whole program energy consumption, energy consumption is the average behavior of an interval. Intervals that are classified into the same phase may have different time-dependent power behavior. If intervals in the same phase have largely different power behavior, we cannot characterize the time-dependent power behavior of the whole program execution using the measurement result of the simpoints.

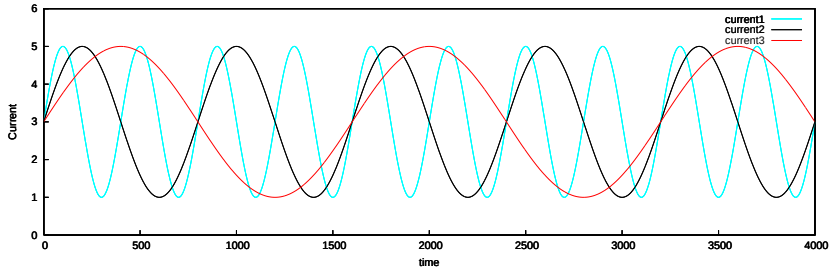
Comparing in the Frequency Domain Our power measurements come in the form of discrete samples in the time domain. Power behavior is characterized by periodic activity, so a comparison in the frequency domain is more appropriate for determining whether two intervals are similar. Fast Fourier Transform (FFT)is a computationally fast way to calculate the frequency, amplitude and phase of each sine wave component of a signal. Thus, we compare the power behavior similarity of two intervals by comparing their discrete Fourier transforms computed using FFT. After the FFT calculation of a power curve, each frequency is represented by a complex number. In power curve similarity comparison, the phase offset of the same frequency should not affect the similarity of two curves. For instance, two power curves might be slightly out of phase with one another, but have exactly the same impact on the system because they exhibit the same periodic behavior. So when we compare two power curves, we calculate the absolute value of the complex number for each frequency, the distance between two corresponding absolute values, and the Root Mean Square (RMS) of the distances for all frequencies. The equation is given in a following section.

Figure 6 shows the FFT distance between the *sine* curves with different values in amplitude, frequency and phase offset, calculated using our method mentioned above. We generate 4096 samples for each curve. Ideally, there is only one frequency in the FFT output of each *sine* curve. But we get multiple frequencies due to the discrete data samples. This is the reason why the calculated distance values are not 0's in Figure 6 (c). The three curves in Figure 6 (a) have the same frequency and phase offset, but different

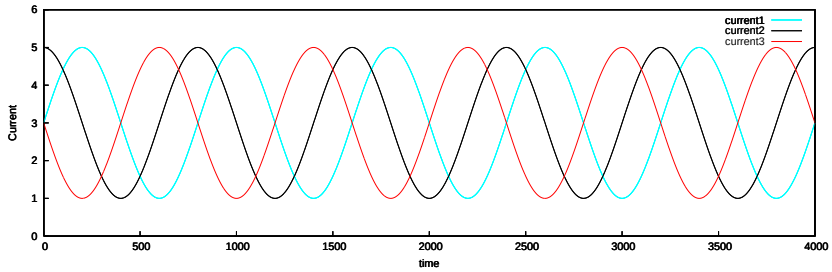
amplitude, which determines the simmilarity of two curves. Figure 6 (b) shows the effect of frequency in our similarity calculation. The small(compared to the values in (a) and (b)) distance between the curves in Figure 6 (c) demonstrate that the effect of phase offset is eliminated.



(a) same frequency and phase offset, different amplitude. dist(1,2)=22.4, dist(1,3)=89.8, dist(2,3)=67.3



(b) same amplitude and phase offset, different frequency. dist(1,2)=119.6, dist(1,3)=115.6, dist(2,3)=120.8



(c) same amplitude and frequency, different phase offset. dist(1,2)=6.5, dist(1,3)=8.1, dist(2,3)=3.0

Fig. 6. Power curve distances calculated using our similarity calculation method

A More Robust Sampling Approach for Verification Measuring every interval in a long-running program is infeasible because of time and space constraints (indeed,

this fact motivates our research). Thus, we use a more robust sampling methodology to verify that power behavior is consistent within a phase. We choose 20 intervals at random for each phase of each program to compare the FFT results of their curves. If the number of intervals in some phase is less than 20, all of the intervals are selected. The selected intervals for each phase are selected from a uniformly random distribution among all the intervals in the phase.

Instrumenting for Verification Infrequent basic blocks demarcating the intervals from the same phase are instrumented to measure each interval in the same way we measure a simpoint. Each selected interval is measured separately. Then the FFT is performed on the measured power curve of each interval. The Root Mean Square (RMS) error of the FFT results is used to evaluate the variation of the power behavior of the intervals in this phase. For each phase, we calculate the arithmetic average over the frequencies in the FFT result of all measured intervals as the expected FFT of the phase. The distance between an interval i and the expected FFT is:

$$D_i = \sqrt{\frac{\sum_{j=1}^{N} (\sqrt{{c_j}^2 + {d_j}^2} - \sqrt{{a_j}^2 + {b_j}^2})^2}{N}}$$

c_j and d_j are the real and imaginary part of the jth frequency of interval i, respectively. a_j and b_j are the real and imaginary part of the jth frequency of the expected FFT respectively. N is the number of frequencies in the output of Fast Fourier Transform. Then the FFT RMS of a phase is calculated as:

$$FFT_{RMS} = \sqrt{\frac{\sum_{i=1}^{M} {D_i}^2}{M}}$$

M is the number of measured intervals in the phase. The lower FFT_{RMS} is, the high the similarity among the time-dependent power behavior of the intervals in the phase.

The FFT_{RMS} for each phase is then weighted by the weight of the corresponding phase to get the RMS for the whole benchmark. We evaluated the weighted FFT_{RMS} for all of the 10 benchmarks in two cases: when phase classification is based on EV only, and when IPC is used to refine phase classification.

4.4 Interval Length Variance

Using infrequent basic blocks to partition program execution into intervals results in variable interval length. We use a pre-specified interval size to avoid intervals that are too small. Intervals of large size are still possible due to the distribution of the infrequent basic blocks during program execution. We analyze the resulting size for each interval of each benchmark to show the distribution of the interval sizes.

We evaluate the interval length variance of a benchmark as the weighted RMS of the interval lengths in each phase. If this value is high, intervals that are of largely different

number of instructions are classified into the same phase, the simpoint for the phase can not be representative of the phase in terms of power behavior.

5 Experimental Results and Evaluation

Using the power measurement infrastructure described in Section 4, we measured the CPU power curves for the instrumented benchmarks, the ones with all final infrequent basic blocks instrumented, the simpoints, and the selected intervals from each phase.

5.1 Instrumentation Overhead

Figure 7 shows the overhead of the instrumentation using different thresholds. It is normalized to the measured energy consumption of the uninstrumented benchmarks. A positive value means the measured energy consumption for this configuration is larger than that of the uninstrumented one. A negative value means the opposite. For some benchmarks, for example, *perlbmk* and *gap*, the energy consumption of the instrumented program is slightly lower than the uninstrumented program. One possible reason is that inserting instructions somewhere might accidentally improve the performance or power consumption, possibly due to a reduction in conflict misses in the cache because of different code placement. Overhead in execution time when different thresholds are used follow the same trend. Instrumentation overhead for power measurement of a single simpoint is even lower because only one or two of the final infrequent basic blocks are instrumented.

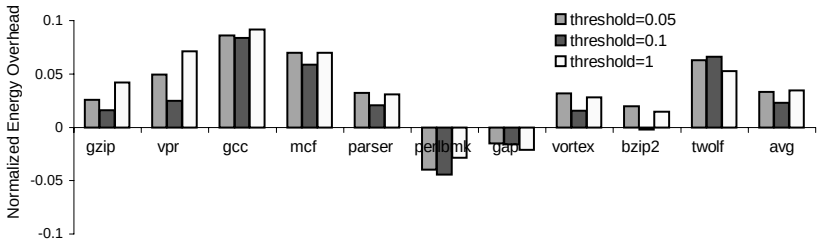


Fig. 7. Normalized instrumentation overhead in energy consumption. The difference between the energy consumption of the instrumented and uninstrumented benchmark divided by the energy consumption of the latter.

5.2 Total Energy Consumption Estimation

We investigate both BBV and EV as the fingerprint of intervals in phase classification. A maximum number of clusters, 30, is used to find the best clustering in both cases.

Simpoints are measured and the whole program energy consumption is estimated as

$$E_{est} = \sum_{i=1}^{k} E_i \times W_i$$

E_i is the measured energy consumption of the ith simpoint, W_i is its weight, and k is the number of phases. Although intervals have variable sizes, we estimate the total energy consumption using the weight based on the number of intervals in each phase.

For BBV-based phase classification, we use three percentage values 0.1%, 1%, and 5% to get the threshold for infrequent basic blocks. The measured energy consumption of simpoints are used to estimate the whole program energy consumption. The error rate is the lowest when threshold is 1% due to the trade-off between uniform interval size and instrumentation overhead. Then we use 1%, 0.1% and 0.05% as threshold in EV-based phase classification. Energy consumption of a measured benchmark or simpoint is calculated as:

$$E = U \times \sum (I \times t)$$

where U is the voltage of the measured CPU power cable, I is the measured current on the CPU power cable, t is the time resolution of the power data points. The sum is over all of the data points for one benchmark or simpoint.

Energy estimation error rate is calculated as:

$$error = \frac{|energy\_estimated - energy\_measured|}{energy\_measured}$$

Execution time estimation is similar to energy estimation.

Figure 8 shows the error rates of the infrequent basic block-based phase classification method using different program execution fingerprints. The error reported is that of the estimate using the threshold that delivered the minimum overall error for each method: 1% for BBVs, and 0.1% for EVs. The figure shows that EV performs better than BBV for almost all of the benchmarks. EV improves the estimation accuracy on average by 35%. One possible reason for the higher error rate of EV for some benchmarks is that we only record conditional edges taken during program execution. Some benchmarks have many unconditional edges, such as *jmp*, so it is possible that some information is lost in EV, although we significantly reduce the edge vector size. For example, method *sort_basket* of *mcf* is called 14683023 times and many of its edges are non-conditional edges. We can improve the phase classification accuracy through recording executiion frequency of all edges, at the cost of largger edge vectors and slower phase classification. All of the following analysis and evaluation are for the experimental results of EV-based phase classification if there is no specification.

5.3 Time-dependent Power Behavior Similarity

As mentioned in Section 4.3, we use the distance between the FFT results of their power curves to evaluate the similarity of two intervals in terms of power behavior. We use 4096 points in the Fast Fourier Transform. The maximum number of data points for a curve is 10,000 when the oscilloscope is in trigger mode. If the measured data points

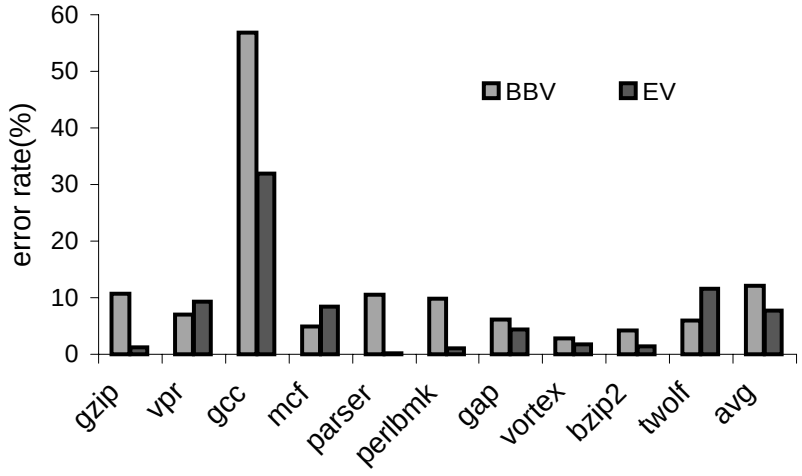


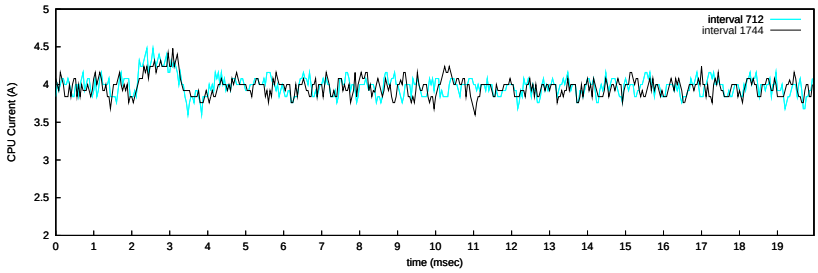
Fig. 8. Error rates of energy consumption estimation using different fingerprints.

for the curve of an intervals is less than 4096, the curve is repeated to reach the number of frequencies. Figure 9 (a) shows the measured CPU current curves of two intervals from the same identified phase, while (b) shows that of two intervals from two different phases. Distance between the FFT values is included to show the relation between time-dependent power behavior similarity and FFT distance. In Figure 9 (a), the upper curve uses the left *y* axis, while the other one use the right *y* axis, to avoid overlapping curves. The second column of each group in Figure 10 is the weighted FFT_{RMS} for each benchmark when EV is used for phase classification.

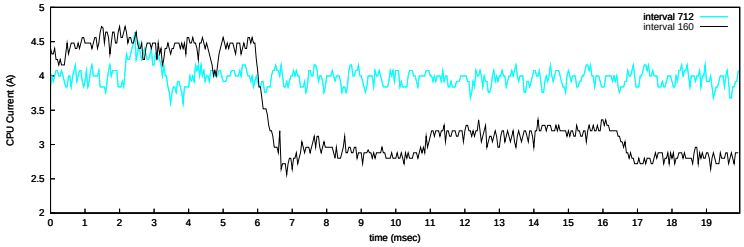
We measure the IPC using performance counters for each interval and do phase classification based on IPC to refine the EV-based phase classification. The third column in each group in Figure 10 is the weighted FFT_{RMS} for each benchmark when EV+IPC is used for phase classification. The similarity among the intervals is improved by 22% over using BBVs. Compared to the FFT distance between an interval and another interval from a different phase, the distance inside a phase is much smaller. This shows that the combination of EV and IPC enables us to classify intervals into phases in which the intervals have similar power behavior. Thus the power behavior of the whole program can be characterized by the measured behavior of the simpoints.

5.4 Interval Length Variance

Figure 11 shows the weighted average of interval length variance of each phase for each benchmark when BBV and EV is used in phase classification respectively. A smaller number means the intervals of the same phase have very close interval size. Again it shows that EV is better for our infrastructure because, on average, it causes much lower interval length variance than BBV no matter which threshold is used. Again One



(a) Power curves of intervals from the same phase(distance=5.4).



(b) Power curves of intervals from different phases(distance=55.1).

Fig. 9. Similarity between measured CPU current of intervals.

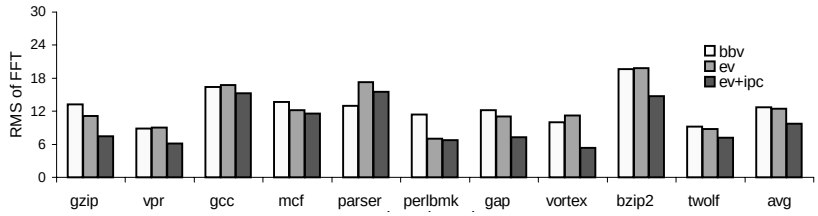


Fig. 10. Root Mean Squared error of the FFT calculated based on RMS of FFT and the weight of each phase.

possible reason for the higher RMS of EV for some benchmarks is that we only record conditional edges taken during program execution, which results in information loss. Although the possible reason is the same as in Section 5.2, the higher error rate or RMS happens to different benchmarks in these two set of experiments. The reason is that total power consumption is an average metric, if the energy consumption of the selected representatiove interval is close to the average energy consumption of all of the intervals in the same phase, the error rate should be low. While RMS of interval length is used to evaluate the similarity among intervals in the same phase, low error rate in total energy consumption does not mean this RMS value is small. This also applies to time-dependent power behavior and is also one of the motivation to use FFT

to evaluate the time-dependent power behavior similarity among intervals in the same phase.

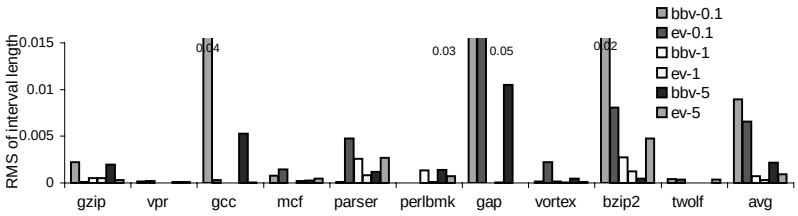


Fig. 11. Weighted average of the RMS error of interval length in the same phase.

6 Conclusion

This paper introduced our infrastructure for efficient program power behavior characterization and evaluation. We presented a new phase classification method based on edge vectors combined with event counters. We described the physical measurement setup for precise power measurement. By demarcating intervals using infrequently executed basic blocks, we find intervals with variable lengths and negligible instrumentation overhead for physical measurement of simpoints. Through experiments on a real system, we demonstrated that our new phase classification method can find representative intervals for energy consumption with an accuracy superior to using basic block vectors. More importantly, we demonstrated the ability of our infrastructure to characterize the fine-grained time-dependent power behavior of each phase in the program using a single representative interval per phase. The ability of instrumenting programs on various levels, identifying phases, and obtaining detailed power behavior of program execution slices makes this infrastructure useful in power behavior characterization and optimization evaluation.

References

1. Yun, H.S., Kim, J.: Power-aware modulo scheduling for high-performance VLIW. In: International Symposium on Low Power Electronics and Design (ISLPED'01), Huntington Beach, CA (August 2001)
2. Toburen, M., Conte, T., Reilly, M.: Instruction scheduling for low power dissipation in high performance microprocessors. In: Power Driven Microarchitecture Workshop, Barcelona, Spain (June 1998)
3. Su, C.L., Tsui, C.Y., Despain, A.: Low power architecture and compilation techniques for high-performance processors. In: IEEE COMPCON, San Francisco, CA (February 1994) 489–498

4. Hazelwood, K., Brooks, D.: Eliminating voltage emergencies via microarchitectural voltage control feedback and dynamic optimization. In: International Symposium on Low-Power Electronics and Design, Newport Beach, CA (August 2004)
5. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically characterizing large scale program behavior. In: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02). (2002)
6. Manne, S., Klauser, A., Grunwald, D.: Pipeline gating: speculation control for energy reduction. Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA'98) (1998) 132–141
7. Hsu, C.H., Kremer, U.: The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI'03) (2003) 38–48
8. Sherwood, T., Perelman, E., Calder, B.: Basic block distribution analysis to find periodic behavior and simulation points in applications. Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT'01) (2001) 3–14
9. Shen, X., Zhong, Y., Ding, C.: Locality phase prediction. Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04) (2004) 165–176
10. Iyer, A., Marculescu, D.: Power aware microarchitecture resource scaling. Proceedings of the conference on Design, Automation and Test in Europe (DATE'01) (2001) 190–196
11. Lau, J., Perelman, E., Hamerly, G., Sherwood, T., Calder, B.: Motivation for variable length intervals and hierarchical phase behavior. In the Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05) (2005) 135–146
12. Chi, E., Salem, A.M., Bahar, R.I.: Combining software and hardware monitoring for improved power and performance tuning. Proceedings of the Seventh Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT'03) (2003)
13. Duesterwald, E., Cascaval, C., Dwarkadas, S.: Characterizing and predicting program behavior and its variability. Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03) (2003) 220
14. Srinivasan, R., Cook, J., Cooper, S.: Fast, accurate microarchitecture simulation using statistical phase detection. Proceedings of The 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05) (2005)
15. Isci, C., Martonosi, M.: Identifying program power phase behavior using power vectors. Proceedings of the IEEE International Workshop on Workload Characterization (WWC-6) (2003)
16. Flinn, J., Satyanarayanan, M.: Powerscope: A tool for profiling the energy usage of mobile applications. Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications (1999) 2
17. Canturk Isci, Margaret Martonosi: Runtime power monitoring in high-end processors: Methodology and empirical data. Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'03) (2003) 93
18. Hu, C., Jiménez, D.A., Kremer, U.: Toward an evaluation infrastructure for power and energy optimizations. In: 19th International Parallel and Distributed Processing Symposium (IPDPS'05, Workshop 11), CD-ROM / Abstracts Proceedings. (April 2005)
19. Isci, C., Martonosi, M.: Phase characterization for power: Evaluating control-flow-based and event-counter-based techniques. In 12th International Symposium on High-Performance Computer Architecture (HPCA-12) (Febrary 2006)
20. Hu, C., McCabe, J., Jiménez, D.A., Kremer, U.: The camino compiler infrastructure. SIGARCH Comput. Archit. News **33**(5) (2005) 3–8