Managing Multi-Configuration Hardware via Dynamic Working Set Analysis

Ashutosh S. Dhodapkar and James E. Smith
Dept. of Electrical and Computer Engineering,
University of Wisconsin-Madison.
{dhodapka, jes}@ece.wisc.edu

Abstract

Microprocessors are designed to provide good average performance over a variety of workloads. This can lead to inefficiencies both in power and performance for individual programs and during individual phases within the same program. Microarchitectures with multi-configuration units (e.g. caches, predictors, instruction windows) are able to adapt dynamically to program behavior and enable/disable resources as needed. A key element of existing configuration algorithms is adjusting to program phase changes. This is typically done by "tuning" when a phase change is detected – i.e. sequencing through a series of trial configurations and selecting the best.

Algorithms that dynamically collect and analyze program working set information are studied. To make this practical, we propose working set signatures – highly compressed working set representations (e.g. 32-128 bytes total). Algorithms use working set signatures to 1) detect working set changes and trigger re-tuning; 2) identify recurring working sets and re-install saved optimal reconfigurations, thus avoiding the time-consuming tuning process; 3) estimate working set sizes to configure caches directly to the proper size, also avoiding the tuning process. Multi-configuration instruction caches are used to demonstrate the performance of the proposed algorithms. When applied to reconfigurable instruction caches, an algorithm that identifies recurring phases achieves power savings and performance similar to the best algorithm reported to date, but with orders-of-magnitude savings in the number of re-tunings.

1. Introduction

As microarchitecture and chip technology evolve, tradeoffs involving performance, power, and complexity become increasingly difficult, and optimization methods become increasingly sophisticated. One promising optimization method is to configure microarchitecture features dynamically to adapt to changing program characteristics [1-13]. As a program runs, it passes through phases of execution where its performance characteristics and, consequently, its hardware resource requirements may vary [14, 15]. Performance and/or power consumption can be optimized *on-the-fly* if significant phase changes can be detected and dynamic microarchitecture reconfiguration can be invoked in response to the phase changes.

In most proposed implementations, configurable units are designed to have a number of fixed configurations, e.g. four different cache sizes. Then, the runtime configuration algorithm selects from one of the multiple available configurations. Thus far, algorithms for determining the optimal hardware configuration have primarily been *ad hoc*, and consequently, there are about as many algorithms as there are proposals for multi-configuration units.

The research reported here is directed primarily toward development of configuration algorithms rather than developing new types of multi-configuration units. The goal is to find fundamental techniques that can be applied across a broad range of units. These algorithms will not only improve performance of individual multi-configuration units, but also permit unified control of several such units simultaneously. We envision these algorithms being implemented with co-designed virtual machine software [16], but that aspect is not essential to the research presented here; hardware or conventional software implementations could also be used.

As a basis for constructing reconfiguration algorithms, we are studying dynamic analysis of program working sets. There are three aspects of working sets that are of interest. Detection of a working set *change* indicates a program phase change, and can be used to trigger a search for an optimal configuration. Working set *size* can be used directly to choose the optimal configurations when performance is directly related to working set size (e.g. caches). Finally, the working set *identity* can be used to reduce re-optimization overhead: when a previously encountered working set can be identified, the optimal con-

Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)
1063-6897/02 $17.00 © 2002 IEEE

figuration for that working set can be stored and reinstated.

Working sets can be quite large, and it is likely impractical to work with full representations of working sets. Consequently, we propose a small hardware table (on the order of 32-128 bytes) to capture a working set "signature" that contains enough information to permit an estimation of the important working set characteristics. This working set information can be incorporated into a number of reconfiguration algorithms, and we demonstrate the use of working set signatures for multi-configuration instruction caches.

In the next three subsections, we summarize proposed methods for dynamically configuring hardware, describe reconfiguration algorithms, and discuss ways program working set behavior can be used in configuration algorithms.

1.1 Dynamically configurable hardware

A number of proposals have been made for adaptive/configurable hardware mechanisms targeted at performance and/or power optimization. A few important examples follow.

- *Configurable caches and TLBs* – line sizes and associativity are adjusted in response to program referencing behavior [2, 3, 5].
- *Allocation of memory hierarchy resources* – cache memory resources are divided among levels in the cache hierarchy [4] or configured for other uses, e.g. instruction reuse [6].
- *Allocation of memory buffer resources* – the same buffer resources are used for stream buffers or victim buffers, depending the current needs of the program [3].
- *Configurable branch predictors* – the length of the global history register [7] in a *gshare* (or related) predictor is varied.
- *Configurable instruction windows* – sections of the issue window are disabled when there is low instruction level parallelism [8, 9].
- *Configurable pipelines* – portions of clustered microarchitectures can be disabled [10], or a pipeline can vary between in order, out-of-order, and pipeline gating [11].

Of course, these various methods are not mutually exclusive, and in practice a combination of adaptive techniques will likely be used in the same processor. This leads to a fairly complex optimization problem, especially if the methods interact with one another. Huang et al. [12], describe a general framework and algorithms that are intended to deal with processors containing several configurable units.

1.2 Dynamic reconfiguration algorithms

Methods for controlling multi-configuration hardware generally involve a form of feedback where some performance characteristic (e.g. instructions per cycle (IPC) or miss rate) is measured and reconfiguration decisions are based on current and past measurements. The more sophisticated optimization schemes run for a fixed interval (also called a "window", "step", etc.) while monitoring some performance or program characteristic. This information is used to determine whether there has been a program phase change. If so, the configuration algorithm undertakes a *tuning* sequence, i.e. it systematically tries a number of configurations and measures the performance of each. It then selects the optimal one and continues, waiting for the next phase change.

The algorithm shown in Fig. 1 is proposed in [4]. This algorithm is both one of the better documented and the best performing we have found; henceforth, we use this algorithm for comparisons and refer to it as the *Rochester algorithm*. In [4], it is used to control a multi-configuration data cache hierarchy. That system repeatedly runs for a fixed number of instructions (100,000), and then makes a pass through the algorithm given in the figure. The system has two states: STABLE and UNSTABLE. As long as the configurable unit's performance, *unit_perf,* does not change more than *perf_noise* level and the number of branches does not change more than a *br_noise* level, the phase is STABLE and nothing is done. Otherwise, the phase is considered to be UNSTABLE, and the algorithm goes through a tuning sequence, looking for the best configuration. It begins with the smallest configuration and goes to the largest, unless the performance exceeds the *threshold*. Then, the algorithm selects the best performing configuration, makes the system state STABLE, and continues. If the tuning process selects the same configuration as in the previous phase, the noise levels are increased to prevent unnecessary tunings in the future. When stable, the noise thresholds are reduced until they reach a minimum level; in essence, the algorithm dynamically changes the threshold in order to detect major phase changes.

Reconfiguration algorithms have three basic properties that determine their applicability and effectiveness.

Detection efficiency – the ability of an algorithm to detect program phase changes. Low detection efficiency can lead to lost reconfiguration opportunities and non-optimal hardware configurations.

Reconfiguration overhead – the overhead associated with the transition from one configuration to another. The reconfiguration overhead depends on the amount of state contained in the structure. Flushing and/or re-learning the state can take 10's of cycles to 1000's of cycles (e.g. for reconfiguring a data cache).

Tuning overhead – the time spent searching for an optimal configuration. A high tuning overhead leads to

higher number of reconfigurations and more time spent in the non-optimal configurations. This is a more serious problem in microarchitectures with several multi-configuration units. For example, three units with three configurations each, can lead to up to 27 combinations to explore (depending on the degree to which they interact). In a proposed method for resizing global branch history [7], up to 16 different configurations are explored.

```
Initially:
unit_perf_noise = base_perf_noise;
br_noise = base_br_noise;

After each sampling interval:
if (state == STABLE)
   if (|unit_perf - last_unit_perf| < perf_noise   AND
       |num_br - last_num_br| < br_noise)
        perf_noise = max (perf_noise – perf_dec, base_perf_noise);
        br_noise = max(br_noise - br_dec, base_br_noise);
        last_num_br = num_br;
        last_unit_perf = unit_perf;
   else
        last_unit_size = unit_size;
        unit_size = SMALLEST;
        state = UNSTABLE;

else if (state == UNSTABLE)
    record overall_perf;
    if (unit_perf < threshold   AND  unit_size != LARGEST)
        unit_size ++ ;
   else
        unit_size = select that with best overall_perf;
        state = STABLE;
        last_num_br = num_br;
        last_unit_perf = unit_perf;
        if (unit_size == last_unit_size)
            br_noise += br_inc;
            perf_noise += perf_inc;
```

Figure 1. An algorithm that detects a phase change and then searches for the best configuration [4].

It is important to differentiate between number of *tunings* and number of *reconfigurations*. Each tuning can possibly be composed of multiple reconfigurations. Hence, reducing the number of tunings leads to significantly fewer reconfigurations, less time spent in non-optimal configurations, and better performance/power efficiency.

1.3 Configuration algorithms using working set analysis

Because phase changes are manifestations of working set changes [17], we consider algorithms based on analysis of explicit working set information. In Section 2, we define a working set *signature*, a lossy-compressed representation of the true working set. By using working set signatures to detect phase changes, very accurate configuration algorithms can be developed. In Section 3, we apply the working set detection method to variations of the Rochester algorithm and show that similar average cache sizes and miss-rates can be achieved with fewer reconfigurations in some cases.

For some multi-configuration units, the optimal configuration is directly related to working set size. In Section 4, we show that the working set signature can be used for estimating size and develop a simple algorithm for finding an optimal cache configuration. This algorithm significantly reduces reconfigurations.

Finally, working sets can be used to identify recurring phases. Re-tuning is done only when a program phase change actually occurs. If the phase has occurred in the past, the optimum configuration is looked up in a table thereby eliminating the tuning overhead. As far as we know, none of the reconfiguration algorithms reported in literature exploit knowledge of recurring phases. In Section 5, we propose such an algorithm and show that reuse of configuration information can lead to a 95% reduction in number of tunings on average for integer benchmarks. Section 6 describes the implementation of hardware and software required to enable our reconfiguration scheme.

2. Working with working sets

For decades, operating system researchers have studied working set behavior to optimize memory hierarchy usage, and they have shown that working sets are the cause of phase behavior.

2.1 Basic definitions

Classically, a *working set* $W(t_i,\tau)$ for i=1,2…, is a set of distinct segments $\{s_1, s_2.., s_\omega\}$ touched over the i^{th} *window* of size τ [16]. The working set size is ω, the cardinality of the set. The segments are typically memory regions of some fixed size, such as a page.

Following some initial studies of working sets, researchers focused on more general models of program behavior and developed the *phase transition model* [17, 18]. Batson and Madison defined a *phase* as a maximal interval during which a given set of segments stay on top of the LRU stack [18]. In other words, a phase is defined as the maximum interval over which the working set remains more or less constant. The phase transition model states that programs follow a series of steady state phases with rather abrupt transitions in between. Phase transition studies have shown that programs have a marked phase behavior and bigger phases are composed of several smaller phases.

Most of the early working set research was directed at program paging behavior, but as one would expect, similar behavior occurs with smaller, cache line-size addressing units that are more in line with applications to configurable hardware. Also, early work tended to lump instructions and data together. We distinguish instruction and

data working sets, and in this paper we focus on the instruction working set.

As defined, capturing a working set requires a *window*. The window size determines the finest granularity at which phases can be resolved. In this paper, we consider fine grain working sets containing cache line sized elements (32-256 bytes) because we primarily deal with multi-configuration units (e.g. caches and predictors) that work at this granularity. Also, for design simplicity, a series of non-overlapping windows is used, rather than a sliding window as is often used in paging studies.

The method of *sampling* information is another important parameter. In this paper, we assume that sampling occurs at every committed instruction. One could, however, resort to periodic sampling or random sampling to reduce sampling overhead. This will be an area of future research.

We are interested in identifying working sets, measuring sizes and detecting changes in working sets. In order to do this, we need a measure of similarity because the same phase may not always touch *exactly* the same segments in each working set window. There is some level of noise in the measurements partially due to mismatch in the phase and window boundaries and partially due to small differences in execution. We define the *relative working set distance*

$$\delta = \frac{\left|W(t_i,\tau) \cup W(t_j,\tau)\right| - \left|W(t_i,\tau) \cap W(t_j,\tau)\right|}{\left|W(t_i,\tau) \cup W(t_j,\tau)\right|}, \qquad (1)$$

to compare two phases with working sets $W(t_i,\tau)$ and $W(t_j,\tau)$. A large δ value indicates a working set change whereas a small δ indicates no change. At the extreme ends, $\delta = 0$ when the sets are identical, and $\delta = 1$ when the working sets are totally different. We define a *threshold* δ_{th} and say there is a *working set change* if $\delta > \delta_{th}$.

2.2 Working set signatures

Representing and manipulating complete working sets is probably impractical for our application. Consequently, we propose a lossy-compressed working set representation that we call the *working set signature*.

The working set signature is an n-bit vector formed by mapping working set elements into n-buckets using a randomizing hash function (see Fig. 2). As mentioned before, the working set elements are of cache line granularity and hence the low-order b address bits are ignored when hashing. The size of the bit-vector is in the range of 32 – 128 bytes. One could consider varying size dynamically to suit the application; this however, is a topic of future research. The bit-vector is cleared at the beginning of every interval (window) to remove stale working set information.

Working set signatures can be used to estimate the size, change, and identity attributes of the full working set. The *size* (number of ones) of the signature is probabilistically

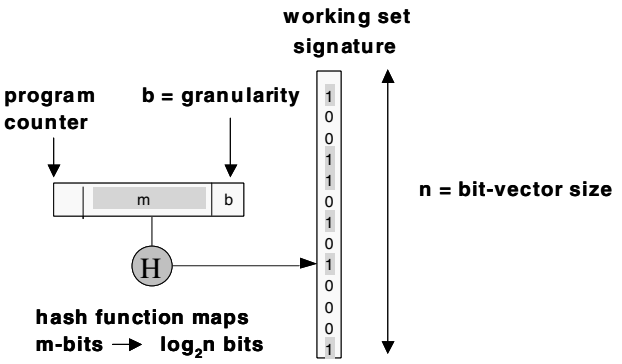

Figure 2. Mechanism for collecting working set signatures. *m* bits selected from the program counter are used to address a table containing *n* bits. The table is cleared at the beginning of each window, and a bit is set if the corresponding instruction block is touched.

related to the true working set size. When *K* random keys are hashed into *n* buckets, the fraction of buckets filled, *f*, is given by

$$f = 1 - (1 - \frac{1}{n})^K. \qquad (2a)$$

Given the fraction of the signature filled, the working set size can be estimated using the relation

$$K = \log(1 - f) / \log(1 - \frac{1}{n}). \qquad (2b)$$

Using this relation, we find that a 90% filled table corresponds to a working set size about 2.5 times larger than the number of filled entries. In Section 3 this relationship will be experimentally validated.

To detect working set changes and identities, we use a measure of similarity analogous to the one defined above for working sets. For two signatures S_1 and S_2, the *relative signature distance* is defined as

$$\Delta = \frac{\left|S_1 \oplus S_2\right|}{\left|S_1 + S_2\right|}, \qquad (3)$$

i.e., (ones count of exclusive OR)/(ones count of inclusive OR). As with full signatures, we will use a threshold value Δ_{th} to detect phase changes.

3. Measuring working set changes

In this section we use instruction working set signatures to detect phase changes (working set changes) and then incorporate this mechanism in an example configuration algorithm.

3.1 Methodology

To evaluate the properties of working set signatures, we used a modified version of the SimpleScalar toolset [19] and a subset of benchmarks from the SPEC 2000 suite. The benchmarks were compiled using the base level optimizations. The choice of benchmarks was based on the presence of 1) long and short term phases with differing performance, 2) recurring phases, to test our working set identification scheme, and 3) different working sets in the same benchmark that led to similar behavior for certain cache/predictor configurations and completely different behavior for others – to show variable effectiveness of reconfiguration.

For collecting working set signatures, a window of 100K instructions is used (unless stated otherwise), and all benchmarks are run for 20,000 such intervals or 2 billion instructions. The signature bit vector size for most of the experiments is 1024 bits (128 bytes); in Section 6.3, we show that signatures as small as 32 bytes perform nearly as well. The hash function used during simulation is based on the C library functions `srand` and `rand`.

3.2 Signature accuracy

In order to evaluate the accuracy of working set signature distances (as compared with full working sets), we measured the relative distances between pairs of consecutive windows. Fig. 3a is a plot of the relative working set distance (y-axis) versus the relative distance for the corresponding signatures (x-axis). This particular graph is for *gzip*, but all the benchmarks display very similar behavior. That these distances are highly correlated is evident. There is some slight dispersion due to hash collisions when forming signatures. It is clear that using signatures for detecting phase changes will be nearly as accurate as using full working sets.

For comparison, the Rochester algorithm uses the dynamic count of conditional branches to measure working set changes. We define a relative distance metric for conditional branch counts in the same way as signature distances i.e.,

$$\Delta = \frac{BR\_CNT_i - BR\_CNT_{i-1}}{BR\_CNT_{i-1}}, \qquad (4)$$

where, BR_CNT_i is the conditional branch count for the i^{th} window. A plot of full working set distances versus the branch count distances shows some correlation, but with a high level of dispersion (Fig. 3b). More importantly, there are several significant working set changes that are associated with very small relative branch distances.

In order to detect a phase change, we need to define the value of *threshold* – Δ_{th}. The threshold is defined empirically. Thresholds that are powers of two (0.125, 0.25, 0.5…) are used because the implied division for forming the relative distance becomes a matter of shifting and comparing. Experiments showed that the ability to detect phase changes is relatively insensitive to the threshold, because, as was noted in [17], a phase change tends to be abrupt and very pronounced. Consequently, a threshold of 0.5 is used, which filters out most of the noise and detects only the significant phase changes.

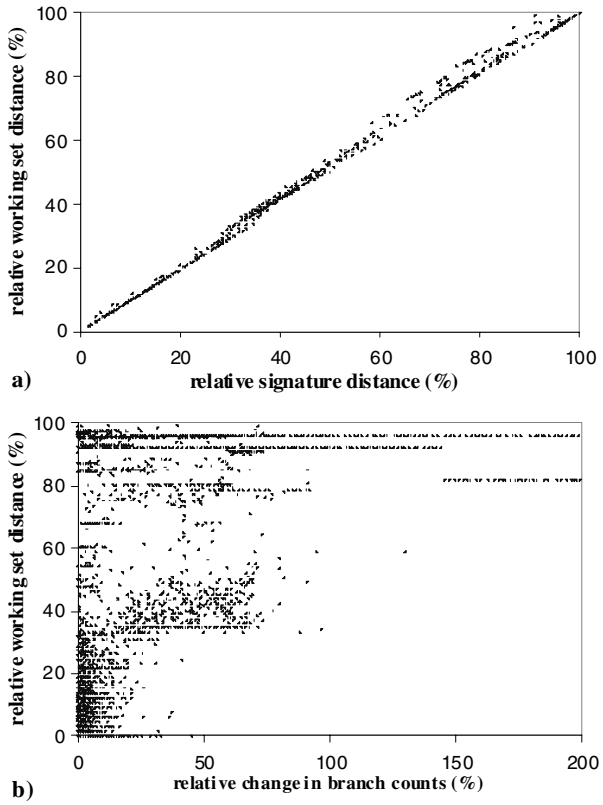

Figure 3. a) Relative working set distance vs. relative signature distance for benchmark *gzip* using a 32-byte signature. b) Relative working set distance vs. relative branch distance (Eq. 4).

3.3 Evaluation: managing configurable hardware

In this subsection, we use working set signatures for detecting phase changes, and incorporate phase change detection into a reconfiguration algorithm. To illustrate its performance, it is applied to a multi-configuration instruction cache.

The algorithm we propose is given in Fig. 4 and will be referred to as the *signature based* algorithm. The signature size is 128 bytes. This algorithm has three states: STABLE - when the program working set is stable and the configuration is optimal, UNSTABLE – when the working set is in transition and TUNING – when the working set is stable and different configurations are being explored.

At the end of each window (100K instructions), the relative signature distance with respect to the previous signature is computed. Assuming the system is initially

```
if (state == STABLE)
  if ( working_set # last_working_set > DELTAMAX )
    state = UNSTABLE;

else if (state == UNSTABLE)
  if ( working_set # last_working_set <= DELTAMAX )
    unit_size = SMALLEST;
    state = TUNING;

else if (state == TUNING)
  if ( working_set # last_working_set < DELTAMAX )
    record overall_performance;
    if (unit_perf > THRESHOLD  AND  unit_size != LARGEST)
      unit_size ++ ;
    else
      unit_size = select best from among those tried;
      state = STABLE;
  else
    state = UNSTABLE;
```

Figure 4. Basic algorithm based on working set signatures. The algorithm uses relative signature distances (represented with # operator) to detect phase changes and then performs tuning when the phase transition completes.

STABLE, if the distance is greater than the threshold (0.5), the state becomes UNSTABLE and subsequent intervals wait for the distance to go below the threshold, indicating stability has been restored. When this happens, the state transitions to TUNING, and the algorithm begins searching for the optimal configuration. Once the optimal configuration is found, the state transitions to STABLE. On the other hand, the state transitions back to UNSTABLE if the signature distance exceeds the threshold while TUNING is in progress.

The Rochester algorithm and the signature-based algorithm are similar in overall structure, but one difference is that the signature-based algorithm does not tune while the working set is in transition; it waits for the phase to stabilize.

To illustrate the algorithm's performance, we consider an instruction cache that can be reconfigured to 2KB, 8KB, 32KB or 128KB, depending on the requirements of the program. The goal is to save power by using the smallest cache that gives good performance. We use the cache miss rate as a measure of performance, the number of reconfigurations/tunings as a measure of overhead, and the average cache size as a measure of power consumption.

For comparison we use the Rochester algorithm given in Fig. 1, adapted to instruction cache configuration. As noted earlier, this algorithm detects phase changes using dynamic branch counts. The parameters used for the algorithm [4, 23] are *base_br_noise* = 4500, *br_dec* = 50, *br_inc* = 1000, *base_perf_noise* = 450, *perf_dec* = 5, *perf_inc* = 100 and *threshold* = 2%.

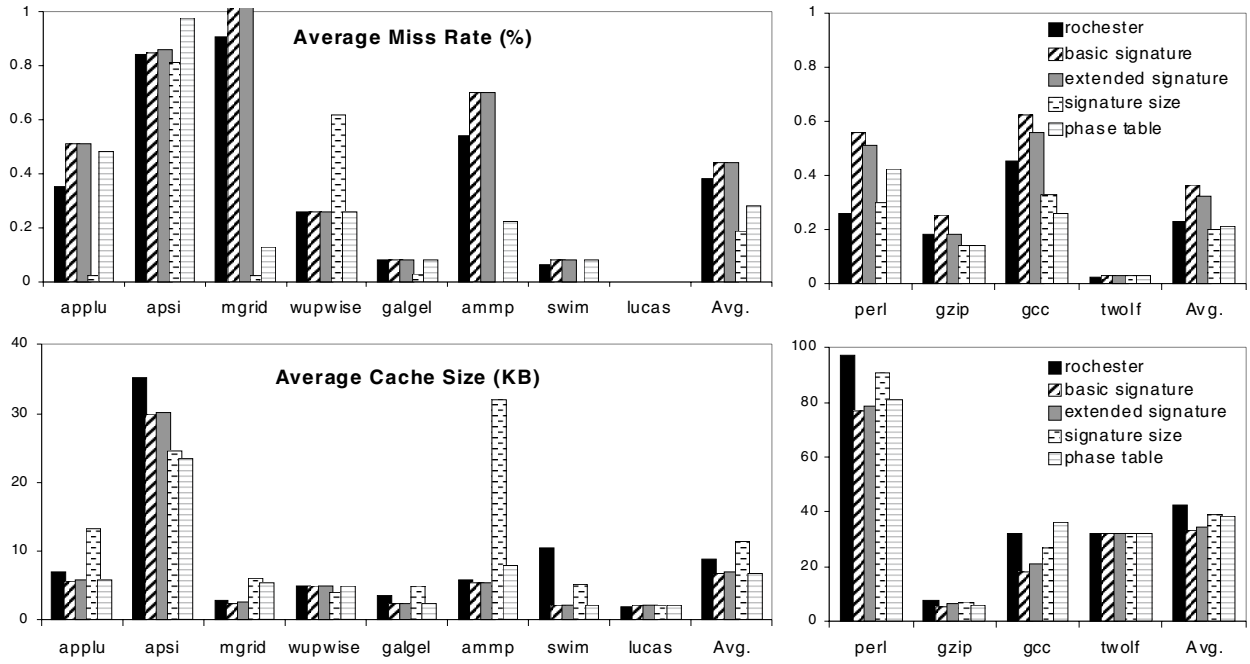

Figure 5. Average miss rates and cache sizes for SPEC2K floating-point (left) and integer (right) benchmarks. The last column in each graph shows the average over all the benchmarks in that graph. Results are shown for the Rochester algorithm, basic signature based and extended signature based algorithms (Sec. 3.3); signature size based algorithm (Sec. 4.2); phase table based algorithm (Sec. 5.1).

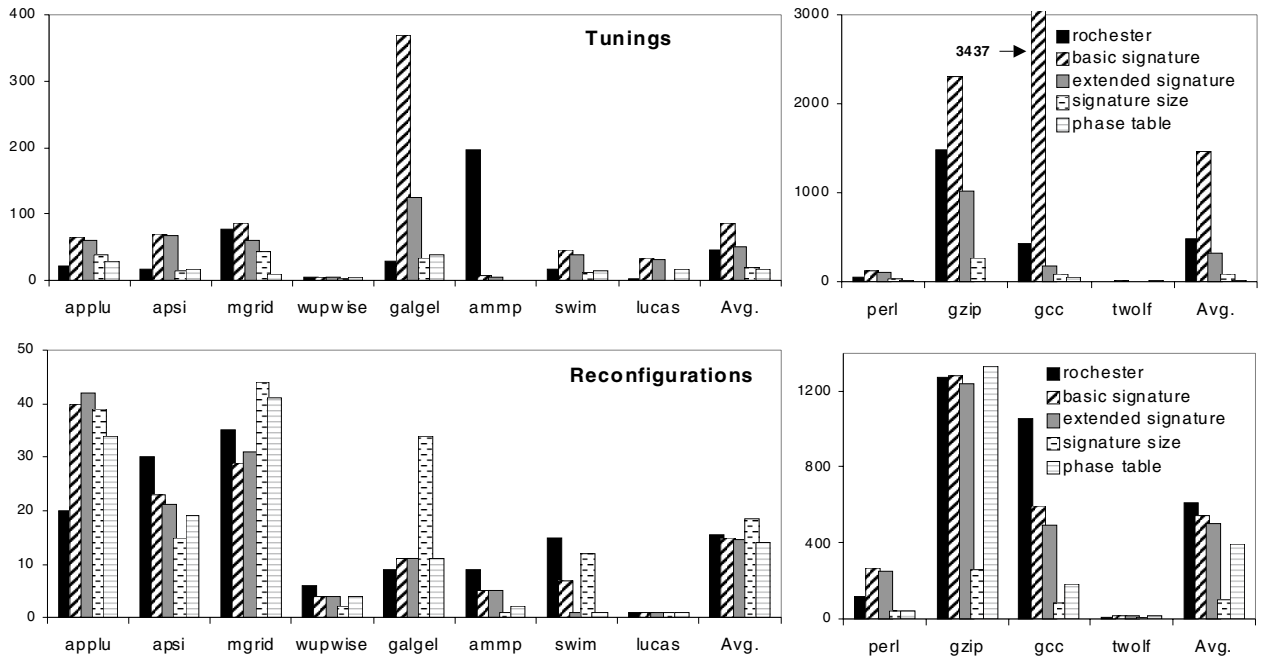

Figure 6. Number of tunings and reconfigurations for SPEC2K floating-point (left) and integer (right) benchmarks. The last column in each graph shows the average over all the benchmarks in that graph. Results are shown for the Rochester algorithm, basic signature based and extended signature based algorithms (Sec. 3.3); signature size based algorithm (Sec. 4.2); phase table based algorithm (Sec. 5.1).

Fig. 5 shows the average cache miss rate and average cache size for the Rochester and basic signature-based algorithms (first two bars; other bars will be described later). On average, all the algorithms perform very similarly in terms of miss rates and average cache sizes. A point to emphasize here is that any algorithm with a sufficient number of tunings will achieve near-optimal instruction cache sizes and miss rates, and in the remainder of the paper, we do not draw any real distinctions among algorithms on that basis. These results do show the advantage of the dynamic configuration approach, however. For example, compared to a configuration having 128KB instruction cache (0% miss rate on average, not shown in the figure), the signature-based algorithm reduces average cache size by 82% for an increased miss-rate of just 0.4%.

The number of tunings and reconfigurations (Fig. 6) are the key distinguishing features directly related to the algorithm's performance overhead, and we focus on these measures in comparing algorithms. Recall that a *tuning* occurs when the algorithm initiates a search for the optimal configuration; a *reconfiguration* occurs whenever the configuration changes.

The signature-based algorithm is comparable to the Rochester algorithm in number of reconfigurations; however, the Rochester algorithm has the advantage of performing far fewer tunings. This is mainly because the Rochester algorithm detects when unnecessary tunings occur and "backs off" by increasing the *noise* levels. This feature is especially useful when there are frequent phase changes that do not require reconfiguration. On the other hand, the basic signature-based algorithm performs tunings every time a phase change is detected; there is no "back off".

To reduce unnecessary tunings, we extend the signature-based algorithm to wait for 4 stable intervals before tuning. Also, if the state is UNSTABLE for more than 10 intervals and performance is below threshold, the cache size is increased to the maximum. This acts as a backup strategy in cases where the working set does not stabilize, so tuning is never performed. With the extended algorithm, the number of tunings is reduced by 74% on average, compared to the basic algorithm (Fig. 6).

4. Measuring working set sizes

As mentioned earlier, the signature size (one's count of the signature) is closely related to the actual working set size. Thus, in those cases where performance is directly related to the working set size, for example instruction and data caches, the signature size can be used to determine the optimal configuration; there is no need for tuning.

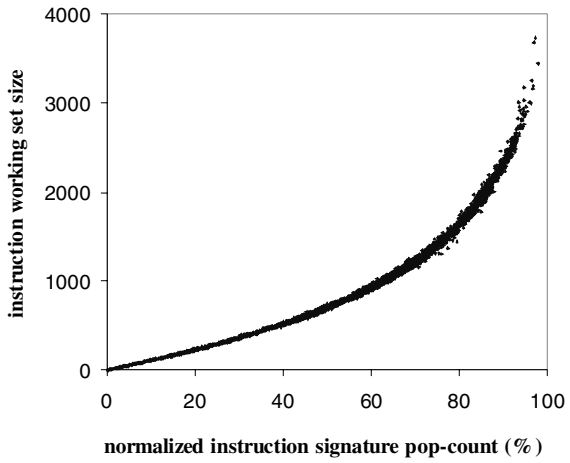

Figure 7. Working set size vs. normalized one's count of the signature for instruction working set of SPEC2K benchmark *gcc*. Signature size used is 128 bytes.

4.1 Working set size experiments

We collected the true working set and the working set signature for each window of 100K instructions. Then, the true size of the working set versus the signature size was plotted. Since a randomizing hash is used, the graphs for all the benchmarks are essentially identical (and fit Eq. 2b). A representative plot for the instruction working set is shown in Fig. 7.

As expected, for small working sets, the graph is close to linear with a slope of 1 and as the working set gets bigger, the graph becomes non-linear. Even in the non-linear section, the signature can give reasonably accurate working set size estimates 3-4x the maximum signature size. This means that a typical signature size we have been considering (32-128 bytes) with line-size granularities (32-128 bytes) can be used to estimate working set sizes of many tens to hundreds of Kbytes – adequate for reconfiguring L1 caches. By increasing the granularity (future research), we expect the reach to be extended to L2 cache sizes.

4.2 Evaluation: reconfiguration using signature size

The extended signature-based algorithm can be modified to use the signature size for selecting an optimal cache configuration – the smallest that holds the current working set (plus 10% to allow for some noise). To determine the appropriate size, equation 2 (Section 2) is used. This eliminates the need to tune, and it typically reduces the number of reconfigurations as well. The main advantage lies in the significantly smaller number of reconfigurations (Fig. 6: *signature size*) – on average, 75-80% fewer than the Rochester and extended signature algorithms. The effect is much more prominent in a benchmark like *gzip*. *Gzip* has lots of dynamic phases with a cache requirement of 8KB, separated by phases with a requirement of 2KB. When tuning, the Rochester and signature-based algorithms try the 2KB configuration before trying the 8KB. On the other hand, the signature-size based algorithm sets the size to 8KB directly, avoiding half of the reconfigurations.

5. Identifying recurring phases

As a program executes, it goes through many phase changes. However, the same phases often recur multiple times during program execution. As far as we know, no previous work has proposed the saving of recurring phase information to avoid re-tuning. In this section, we study such an algorithm. Briefly, this will be done by maintaining a *phase table* in memory. After tuning has determined the optimal configuration for a particular phase, it will be stored in the table. Later, if the phase recurs, the optimal configuration can be reinstated without going through the tuning process.

5.1 Phase statistics

Table 1 shows some general characteristics of phases as identified in the simulations. The program execution consists of a sequence of stable 100K instruction intervals separated by unstable intervals. Each "run" of stable intervals is defined as one *dynamic phase*. If the relative signature distance between two different dynamic phases is within the 0.5 threshold, we say that they are the same *static phase*. The average phase lengths are computed by averaging the lengths of all the dynamic phases.

In general, the floating-point benchmarks have longer phases, typically 10's of millions of instructions – primarily due to the long loops of numerical code. The integer benchmarks on the other hand have much shorter phases; less than one million instructions for *gzip* and *gcc*. For many of the floating point benchmarks 99+% of the time is spent in stable phases. As the average phase length decreases there are more transitions, and hence the fraction of time in a stable region decreases (60-80% for *gzip* and *gcc*).

The presence of recurring phases is evident in all the benchmarks by comparing the number of dynamic phases with the number of static phases. However, the degree to which phases recur is reduced by the relatively short benchmark runtimes (2 billion instructions). Several benchmarks were run for 10 billion instructions and the number of dynamic phases was almost three orders of magnitude greater than the number of static phases. This

Table 1. Benchmark characteristics. Columns are benchmark name, number of dynamic and static phases, number of static phases that lead to 95% of stable time, average phase length in units of 100K instructions and the percentage of time spent in stable phases.

Bench-mark	#Dynamic Phases	# Static Phases	95% Stable Time	Avg. Phase Length	Stable Time (%)
applu	66	26	12	301	99.61
apsi	72	15	6	276	99.36
mgrid	86	9	4	230	99.01
wupwise	6	5	3	3331	99.95
galgel	371	39	12	51	95.14
ammp	7	1	1	2854	99.92
swim	45	14	10	442	99.66
lucas	33	18	13	604	99.69
perl	119	21	5	166	99.13
gzip	2301	10	4	7	81.79
gcc	3437	57	19	3	61.77
twolf	17	13	2	1174	99.84

indicates that the gains of reusing configuration information for recurring phases increase with time.

The "95% stable time" column of the table is the number of static phases that account for 95% of the dynamic phases. These numbers are quite low, fewer than 20 in every case. This indicates that a relatively small signature table will be sufficient for covering most recurring phases.

5.2 Evaluation: recurring working sets

The algorithm for exploiting recurring working sets is similar to the one given in Fig. 4. However, on detecting a phase change, the algorithm first performs a table lookup to see if configuration information for the phase exists in the table. If so, the optimal configuration is reinstated. If not, the algorithm goes into the TUNING state. At the end of tuning, the optimal configuration is committed to the signature table.

In addition to the configuration information, the table also keeps track of phase lengths. If, during its last execution, the length was fewer than four intervals (400,000 instructions), then tuning is not performed. This avoids tuning for insignificant phases. Four intervals are chosen because the tuning process takes a maximum of four intervals.

The results for the algorithm are shown in Figs. 5 and 6, labeled *phase table*. The important difference lies in the number of tunings performed by the phase table algorithm. The algorithm performs 67% fewer tunings for floating point benchmarks and 92% fewer tunings for the integer benchmarks compared with the extended signature based algorithm. In situations where the tuning process is complex, this can lead to significant improvements in performance.

6. Implementation

To implement configuration algorithms, we propose a combination of hardware and software. Software performs higher-level configuration decisions, and hardware collects working set signatures, and, possibly, performs some of the lower level analysis.

6.1 VMM based configuration management

To perform working set analysis and manage configurable hardware of wide variety and complexity, we are developing a co-designed virtual machine monitor (VMM) [16] – a layer of software designed concurrently with the hardware implementation. This software is hidden from all conventional software and would typically be developed as part of the hardware design effort. The base technology is used in the Transmeta Crusoe [20] and the IBM Daisy/BOA projects [21] primarily to support whole-system binary translation. In this work, we are not interested in the binary translation aspect. In fact, for managing configurable hardware, there needs to be no changes made to existing binaries.

Of course, VMM software is not the only option for managing the optimization process. Low-level operating system software could also be used. This, however, requires the addition of implementation dependent code to the OS. One could also consider microcode in place of VMM software. The microcode can reside in ROM, but there must still be some hidden memory for maintaining data structures such as the phase table. A special purpose co-processor [22] is another good candidate for managing the hardware configuration. It has the advantage of saving optimization time overhead at the expense of additional hardware.

In the most straightforward implementation, working set signatures are collected by hardware, and then the raw signature data is read and analyzed by VMM software. The working set size/difference algorithms we propose can easily be performed in software. With the assumed window size, VMM software is invoked very 100K instructions. Because in most cases the relative signature distance will be very small, the VMM overhead will also be small – probably a few tens of instructions. If this overhead is still too high, a longer sampling interval can be used, or hardware can be used to perform some of the low level analysis. This is described in the next subsection.

A phase table lookup ostensibly requires a linear search of signatures, but it can be made more efficient by using techniques such as hashing based on the signature size,

early exits when the phase is same as the previous one, etc. This will be a topic for future study as the VMM is implemented.

6.2 Hardware working set analysis

Besides collecting working set signatures, hardware can also be used for estimating working set size and/or to detect working set changes, thereby reducing software overhead. In particular, detecting working set changes in hardware avoids invoking the VMM between each interval; the VMM has to be invoked only when the working set actually changes. Furthermore, for very simple reconfigurations that are directly related to working set size (e.g. cache configurations), it may not be necessary to enter the VMM at all; hardware can determine the proper configuration based only on the size of the working set signature. It is important to emphasize that this hardware is not on the critical path and hence can be implemented with slow, low power transistors.

To measure size, there must be a hardware counter which increments whenever a bit in the signature changes from 0 to 1. This requires reading the signature entry before writing to it.

To measure the relative signature distance, a second signature register is required to hold the signature for the previous window. As defined in section 2.2, the relative signature distance is the ratio of the exclusive-OR to the inclusive-OR of the signatures – say X/N. X and N can be evaluated dynamically as follows.

Initially, X=N=count of ones in the previous signature. For each signature access, both the previous and current signature values are read. If previous=0 and current=0, both X and N are incremented. If previous=0 and current=1, nothing is done; if previous=1 and current=0, then the bit in the previous signature is cleared and X is decremented; the case previous=1 and current=1 should never happen. Then at the end of the interval, hardware can find the relative signature distance X/N (or approximate it by shifting and comparing, when the threshold is a power of two). The VMM can set up the hardware to trap to VMM software on values above the threshold.

6.3 Implementation cost

The primary cost is the working set signature. This consists of 128 bytes and can be placed off the critical path. Using smaller signatures can further reduce the hardware cost. Fig. 8 shows that a signature as small as 32 bytes can resolve most of the dynamic phases resolved by a 512-byte signature. Small signatures are unable to resolve certain phase changes for benchmarks with large working sets (*perl* and *gcc*) due to collisions in the signature table, which lead to smaller relative distances. Preliminary experiments have shown that using smaller thresholds is a solution. Dynamically varying thresholds and/or signature sizes, to accommodate larger working sets is a topic of future research.

In the simple implementation (where the VMM performs the relative distance computation) the memory only has to be written in normal operation. Furthermore, it is not critical that every instance of an element of the working set be recorded. Only one occurrence of the element has to be recorded and most elements appear multiple times. Thus, if occasionally dropping an element simplifies hardware (for example, retiring instructions from different cache blocks in one cycle) little accuracy is lost. For the determining the relative signature distance in hardware, two copies of the signature memory are needed, and they are both read and written during the collection phase.

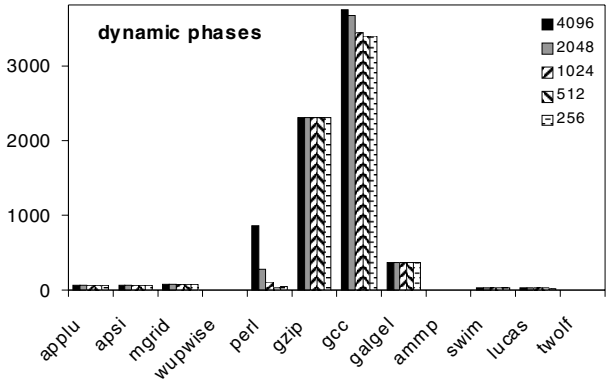

Figure 8. Number of dynamic phases resolved by signatures of sizes 256–4096 bits (32-512 bytes). The elements sampled are 32-byte blocks except for the 256-bit signature, where they are 128 bytes. This is done to increase the "reach" of the signature.

7. Related work

Previous work related to hardware reconfiguration was discussed in Sec. 1.1. In this section, we briefly discuss work related to working set analysis.

Sherwood et al. [24] proposed the use of program phase information to speed up simulation. They use basic block execution frequency information as a *fingerprint* for an interval of execution. The goal then, is to find a small set of intervals whose fingerprint matches that of the whole program. Detailed simulation over these intervals can give a fairly accurate estimate of the performance of the whole program.

Adaptive mode control (AMC) caches, proposed by Zhou et al. [25], keep track of the working set in order to enable/disable cache lines. The AMC cache keeps a counter for each of the tags to measure activity. If the cache line is not accessed for a particular interval, then it

is put to sleep. However, the corresponding tag entry is not put to sleep, thereby allowing continuous monitoring of the working set and avoiding "just-in-case" periodic upsizing.

HP Dynamo [26], a run time dynamic optimization system, uses a measure of working set change to flush stale data translations from a cache. Dynamo optimizes traces of the program to generate fragments, which are stored in a fragment cache. At steady state, most of the instructions are fetched from the fragment cache. When the working set changes, the rate of fragment formation increases. This is used as a trigger to flush stale fragments from the cache, making room for the new ones.

Merten et al. [27] describe a framework for dynamic optimization, which profiles branches to detect working set *hot-spots*. This is mainly done using a branch behavior buffer, which collects frequently executed branches. The hot-spot information can be fed into a run-time optimizer such as Dynamo to achieve performance improvements.

8. Conclusions and future research

We introduced the concept of a working set signature, a lossy-compressed representation of the program working set. The signatures provide a robust mechanism for detecting working set changes. Also, unlike previously reported methods, the signatures can be used to identify specific working sets. This provides an opportunity for storing configuration information associated with recurring working sets. Algorithms using complex tuning mechanisms can benefit significantly from reuse of configuration information.

When applied to an instruction cache reconfiguration algorithm, the signatures detect most of the major working set changes. This algorithm achieves 27% fewer tunings and 18% fewer reconfigurations than the Rochester algorithm – probably the best published to date.

Working set size information can be derived from the signature and can be used to configure the instruction caches directly. An algorithm based on this achieves performance similar to the signature-based algorithm using 74% fewer reconfigurations.

Finally, an algorithm based on reuse of configuration information leads to 80% fewer reconfigurations compared to the Rochester algorithm. These results suggest that an algorithm based on reuse of configuration information can potentially perform much better than other algorithms when the tuning overhead is high.

We plan to continue the development of a VMM that implements these algorithms. This development will include

- Algorithms for tuning multiple interacting units in a way that optimizes performance and/or power efficiency. The work in [12] is an important first step in this direction.
- Study of the relationship between the signature size, the PC bits, sample interval and thresholds. It is likely that the VMM can adjust the PC bits and sample interval dynamically to adapt to working set size.
- Study of sampling schemes such as periodic sampling, to reduce sampling overhead.
- Study of algorithms for building and managing the signature table. In particular, it will be necessary to develop fast algorithms for searching the table to find recurring phases.

The ultimate goal is to define the overall VMM structure and to apply it to a highly configurable microarchitecture.

9. Acknowledgements

We would like to gratefully acknowledge Todd Bezenek and Timothy Heil for several valuable discussions. This work is being supported by SRC grants 2000-HJ-782 and 2001-HJ-902, NSF grants EIA-0071924 and CCR-9900610, Intel and IBM.

10. References

[1] D. Albonesi, "Dynamic IPC/Clock Rate Optimization," *Proc. of the 25th Intl. Sym. on Computer Architecture*, July 1998, pp. 282-292.

[2] S.-H. Yang, M. Powell, B. Falsafi, K. Roy and T. N. Vijaykumar, "An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep Submicron High-Performance I-Caches," *Proc. of the 7th Intl. Sym. on High Performance Computer Architecture*, Jan. 2001.

[3] A. Veidenbaum, W. Tang, R. Gupta, A. Nicolau and X. Ji, "Adapting Cache Line Size to Application Behavior," *Intl. Conf. on Supercomputing*, July 1999, pp. 145-154.

[4] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu and S. Dwarkadas, "Memory Hierarchy Reconfiguration for Energy and Performance in General Purpose Architectures," *Proc. of* 33rd *Intl. Sym. on Microarchitecture*, Dec. 2000, pp. 245-257.

[5] D. H. Albonesi, "Selective Cache Ways: On-demand Cache Resource Allocation," *Proc. of* 32nd *Intl. Sym. on Microarchitecture*, Dec. 1999, pp. 248-259.

[6] P. Ranganathan, S. Adve and N. Jouppi, "Reconfigurable Caches and Their Application to Media Processing," *Proc. of the 27th Intl. Sym. on Computer Architecture*, June 2000, pp. 214-224.

[7] T. Juan, S. Sanjeevan and J. Navarro, "Dynamic History-Length Fitting: A Third Level of Adaptivity for Branch Prediction," *Proc. of the 25th Intl. Sym. on Computer Architecture*, July 1998, pp. 155-166.

[8] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook and D. Albonesi, "An Adaptive Issue Queue for Reduced Power at High Performance," *Workshop on Power-Aware Computer Systems (PACS2000, held in conjunction with ASPLOS-IX)*, Nov. 2000.

[9] D. Folegnani and A. González, "Reducing Power Consumption of the Issue Logic," *Workshop on Complexity-Effective Design (WCED2000, held in conjunction with ISCA27),* June 2000.

[10] R. Bahar and S. Manne, "Power and Energy Reduction via Pipeline Balancing," *Proc. of the 28th Intl. Sym. on Computer Architecture*, July 2001.

[11] S. Ghiasi, J. Casmira and D. Grunwald, "Using IPC Variations in Workloads with Externally Specified Rates to Reduce Power Consumption", *Workshop on Complexity Effective Design 2000 (WCED2000, held in conjunction with ISCA27)*, June 2000.

[12] M. Huang, J. Reneau, S.-M. Yoo and J. Torrellas, "A Framework for Dynamic Energy Efficiency and Temperature Management," *Proc. of the 33rd Intl. Sym. on Microarchitecture*, Dec. 2000, pp. 202-213.

[13] D. Brooks and M. Martonosi, "Adaptive Thermal Management for High-Performance Microprocessors," *Workshop on Complexity Effective Design 2000 (WCED2000, held in conjunction with ISCA27)*, June 2000.

[14] Timothy Sherwood and Brad Calder, "Time Varying Behavior of Programs," *UC San Diego Technical Report UCSD-CS99-630*, August 1999.

[15] Bingxiong Xu and D. H. Albonesi, "Runtime Reconfiguration Techniques for Efficient General-Purpose Computation," *IEEE Design and Test of Computers*, Vol. 17, Issue 1, Jan. – Mar. 2000, pp. 42-52.

[16] A. S. Dhodapkar and J. E. Smith, "Saving and Restoring Contexts via Co-Designed Virtual Machines," *Workshop on Complexity-Effective Design (held in conjunction with ISCA28),* June 2001.

[17] P. Denning, "Working Sets Past and Present," *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 1, Jan. 1980.

[18] A. Batson and W. Madison, "Measurements of major locality phases in symbolic reference strings," *Proc. of the Intl. Sym. Computer Performance and Modeling, Measurement and Evaluation*, ACM SIDMETRICS and IFIP WG7.3, Mar. 1976, pp. 75-84.

[19] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0," *University of Wisconsin-Madison Computer Sciences Department Technical Report #1342*, June 1997.

[20] A. Klaiber, "The Technology Behind Crusoe Processors," Transmeta Technical Brief, ***http://www.transmeta.com/dev,*** Jan. 2000.

[21] K. Ebcioglu and E. Altman, "DAISY: Dynamic Compilation for 100% Architecture Compatibility, *Proc. of the 24th Intl. Sym. on Computer Architecture*, June 1997, pp. 26-37.

[22] Y. Chou and J. Shen, "Instruction Path Coprocessors," *Proc. of the 27th Intl. Sym. on Computer architecture*, 2000, pp. 270-281.

[23] Rajeev Balasubramonian, Sandhya Dwarkadas and David Albonesi, personal correspondence.

[24] T. Sherwood, E. Perelman, and B. Calder, "Basic Block Distribution Analysis to Find Periodic Behaviour and Simulation," *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sep. 2001, pp. 3-14.

[25] H. Zhou, M. Toburen, E. Rotenberg, and T. Conte, "Adaptive Mode Control: A Static-Power-Efficient Cache Design," *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sep. 2001, pp. 61-72.

[26] V. Bala, E. Duesterwald, S. Banerjia, "Dynamo: A Transparent Dynamic Optimization System," *Proc. of the Conf. on Prog. Language Design and Implementation*, ACM SIGPLAN, 2000, pp. 1-12.

[27] M. Merten, A. Trick, R. Barnes, E. Nystrom, C. George, J. Gyllenhaal, and W.-M. Hwu, "An Architectural Framework for Runtime Optimization," IEEE Transactions on Computers, June 2001, pp. 567-589.