Selecting Long Atomic Traces for High Coverage

Roni Rosner[*], Micha Moffie[*], Yiannakis Sazeides[+], Ronny Ronen[*]

[*]Microprocessor Research
Intel Labs (formerly MRL)
Haifa, Israel
{roni.rosner, micha.moffie, ronny.ronen}@intel.com

[+]Department of Computer Science
University of Cyprus
Nicosia, Cyprus
yanos@ucy.ac.cy

ABSTRACT

This paper performs a comprehensive investigation of dynamic selection for long atomic traces. It introduces a classification of trace selection methods and discusses existing and novel dynamic selection approaches – including loop unrolling, procedure inlining and incremental merging of traces based on dynamic bias. The paper empirically analyzes a number of selection schemes in an idealized framework.

Observations based on the SPEC-CPU2000 benchmarks show that: (a) selection based on dynamic bias is necessary to achieve the best performance across all benchmarks, (b) the best selection scheme is benchmark and maximum trace-length specific, (c) simple selection, based on program structure information only, is sufficient to achieve the best performance for several benchmarks.

Consequently, two alternatives for the trace selection mechanism are established: (a) a "best performance" approach relying on complex dynamic criteria; (b) a "value" approach that provides the best performance (and potentially the best power consumption) based on simpler static criteria. Another emerging alternative advocates adaptive based mechanisms to adjust selection criteria.

Categories and Subject Descriptors

C.1.3 [**PROCESSOR ARCHITECTURES**]: Other Architecture Styles – *pipeline processors.*

General Terms

Algorithms, Performance, Design.

Keywords

Trace processors, trace cache, trace selection, trace atomicity.

1. INTRODUCTION

Trace caching ([27][31][23][32][12]) is a promising method for providing high bandwidth instruction supply at low latency. A trace cache enables a processor to fetch in parallel long sequence of instructions across basic blocks thus overcoming the sequential resolution of control transfer instructions.

ICS'03, June 23-26, 2003, San Francisco, California, USA.

Although alternative techniques may be used to achieve the high instruction bandwidth of trace caches (e.g. the collapsing buffer [5], the basic block trace-cache [3] or extended basic-block cache [16]), trace caches possess two unique properties that make them more attractive: (a) the construction of traces can be done effectively off the critical path, and (b) the storage of complete traces in a cache for reuse.

These trace cache properties can facilitate additional important optimizations. In particular, trace construction may be enhanced with optimization functionality that can increase performance [10][14][25][7][36] and reduce power consumption [35][30]. This optimization potential can be significantly facilitated by (a) selecting ***longer*** traces, and (b) by treating each trace as a single ***atomic*** block although it may contain control transfer instructions [22][26].

Trace selection defines a set of criteria for when a consecutive dynamic sequence of instructions constitutes a trace. Different selection criteria result in the construction of a different set of traces and lead to the execution of a different sequence of traces, thus obtaining different trace length (and content) and different dynamic coverage. Therefore, successful trace selection is crucial for maximizing performance.

In this work, we perform a comprehensive study of trace selection targeted at producing ***long atomic traces*** with ***high coverage***. These two objectives are typically in conflict, because selecting long traces may result in low coverage whereas high coverage may require short traces. This trade-off is central to this work. The paper introduces a classification of trace selection methods and discusses how existing and novel selection schemes address this trade-off. An empirical analysis of a number of selection schemes is performed for a scenario with no trace cache misses and a large predictor to establish a performance limit for long atomic traces.

One of the key results is that selection schemes that rely on dynamic information provide the overall best performance: With maximum trace length of 256 instructions, the average hot trace length is 156 instructions for floating point programs and 52 for integer, and coverage is 98% for floating point and 89% for integer programs. The results also revealed that for several benchmarks, simpler selection schemes relying on program structure may be sufficient to achieve the best performance.

At the microarchitecture design level, we identify two alternative approaches for the trace selection mechanism. An aggressive approach based on complex dynamic criteria with overall the best performance. The other approach is based on simpler static criteria. Such a "value" approach can provide the best performance for a large set of benchmarks, and due to its design simplicity, may

also offer the best power consumption. A third, adaptive-based alternative emerges from our results. Adaptive trace-selection may introduce novel trade offs for microarchitecture design and represents an important direction for future work.

The rest of the paper is organized as follows: the front-end model assumed in this work is described in Section 2. The basics of trace selection are described in Section 3. Section 4 introduces a classification of trace selection methods. This section also discusses a number of existing and novel selection schemes. The experimental framework is described in Section 5. Results are presented in Section 6. Related work is discussed in Section 7. Finally, Section 8 concludes the paper and suggests directions for future work.

2. FRONT-END MODEL

This section describes the model used in this study. The focus of this work is on the front-end and therefore back-end microarchitecture parameters are not discussed. This section also includes a discussion on performance metrics for a trace-cache driven front-end and introduces the objective function used in this study.

2.1. Traces and Related Notions

A ***basic block*** (BB) is a sequence of non-CTI (control transfer instructions) where only the last instruction can be a CTI (this work does not account for potential labels inside such a block).

A ***trace*** is a finite sequence of dynamically consecutive instructions [31]. Traces in this paper are single-exit and atomic. A trace is atomic if it is treated as a single instruction, or single-entry/single-exit [19][20][26]. Single-exit atomic traces may be necessary when applying optimizations that do not preserve sufficient information for reuse of the head of a partially correct trace, e.g. when instructions are reordered within the trace.

A ***trace identifier*** (TID*)* is a finite sequence of basic block starting ***IP***s (instruction pointers).

A ***path*** or ***BB-history*** is a finite sequence of BB-starting IPs.

2.2. Front-End Model

A pictorial representation of the front-end model is shown in Figure 2-1. Note that this front-end model is similar to the one described in [32].

Instructions are supplied to the processor only in the form of traces. A trace predictor provides an identifier—a TID—that is used to access the trace cache. In the case of a trace cache hit the trace is send for execution. In the case of a miss, the TID is used to construct a trace from the instruction cache.

When a trace misprediction occurs, the correct trace is built starting from the IP of the **first** instruction in the mispredicted trace (this reflects the atomicity of traces). The construction is done using a fetch engine that can sequence at the granularity of instructions (regular instruction cache hierarchy and conventional branch prediction). The new trace is build based on trace selection criteria. The different selection criteria are discussed in Section 4.

After a trace is built, the trace cache is updated and the trace is sent down the pipe for execution. When the build is a result of a misprediction the trace is also used to correct the history information used to access the trace predictor.

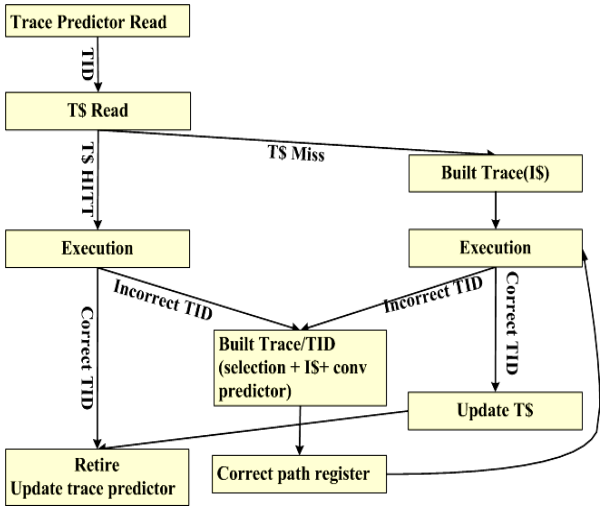

Figure 2-1 Front-End Model Flow

The history information is used to index the ***trace prediction table*** (TPT) to read the next TID. The TPT itself is updated at retirement.

This base model is extended with advanced selection schemes as explained in Section 4.

2.3. Front-End Performance Metrics

Trace selection together with the trace predictor and trace-cache are the three main independent variables that influence various front-end performance metrics. Table 2.1 presents a number of metrics and shows the variables influencing each.

The different metrics are explained below:

Length: Average trace length for all traces.

Unique traces: Number of unique traces created

Unique patterns: Number of unique history patterns that were used to index the trace predictor.

METRIC	DEPENDS ON
Hot/Cold Length	Selection, cache, predictor
Unique Traces	Selection
Unique Patterns	Selection, information vector
Coverage	Selection, cache, predictor
Predictability	Selection, predictor
Builds	Selection, cache, predictor

Table 2.1 What influences front-end performance metrics

Coverage: Percent of executed instructions in traces that were correctly predicted and that were found in the trace cache.

Predictability: Number of mispredictions from the trace predictor. The ***information vector*** is a trace predictor related variable and refers to the information used to form predictor indices (number of history items, type of items, hashing function etc).

Builds: Number of trace builds, performed as a result of trace mispredictions or trace cache misses.

The ***length*** metric is divided into hot and cold because 'not all traces are equal'. As shown in Figure 2-2, a retired trace can be the result of four execution scenarios. Each scenario may have different cost/benefit and thus the length metric needs to reflect this. In this work, retired traces are divided into two categories: hot and cold. Hot are those traces that hit in the trace cache and predicted correctly, whereas the other three types of traces are grouped as cold. We argue that trace-cache based front-ends should aim to maximize hot-trace length and minimize cold-trace length. As shown in the results section, this is particularly important for configurations that support very long traces.

Henceforth unless indicated otherwise, the term trace length will be used to refer to the overall average trace length.

The number of unique traces and number of unique patterns are meant to measure the "pressure" on the trace cache and trace predictor table resources. The higher these numbers are - the more likely these resources would suffer from capacity, conflict and cold misses.

Coverage captures directly the effects of builds and mispredictions, and, indirectly, it may capture the effects of unique traces and unique patterns.

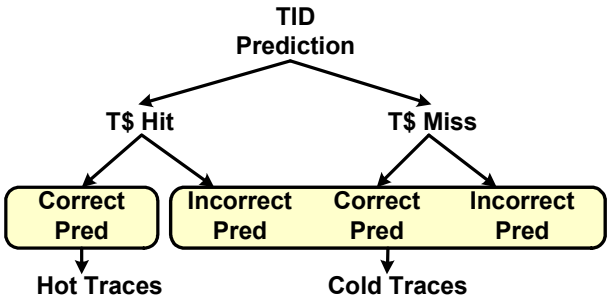

Figure 2-2 Types of Retired Traces

2.4. Objective Function

Two important metrics that a front-end should maximize are (a) length of hot traces, and (b) execution coverage from traces. However, the two objectives are typically in conflict, because to achieve *long traces may require low coverage* whereas *high coverage may require short traces*. Therefore, selection should aim for a trade-off between the two. The following *objective function* attempts to capture this trade-off:

$$\frac{1}{\left(\frac{C}{L}+1-C\right)}$$

In this formula ***L*** denotes the average hot trace length (it can take values from 1 to maximum trace length). ***C*** denotes coverage (it can take a value between 0 and 1).

The objective function value is intended to capture the speedup potential of a microarchitecture with trace-cache based optimizations over a single-issue microarchitecture. The fraction of execution covered by the trace-cache, ***C***, is assumed to be optimized to have parallelism equal to ***L***.

It is noteworthy that the above function is an adaptation of Amdahl's Law [1]. This indicates that the objective function assigns a lot of significance on coverage. Therefore, unless ***C*** is high enough the performance potential will be small.

This objective function, together with some of the other metrics, is used in subsequent sections to assess the performance of different selection heuristics. We recognize that other objective functions could have been used and that the ultimate metric is actual performance, however, the above enables for comparison of different selection schemes without full microarchitectural simulation and independent of specific implementation.

3. PHILOSOPHY OF TRACE SELECTION

Trace selection defines the set of criteria for when a consecutive dynamic sequence of instructions constitutes a trace. It is evident from Table 2.1 that trace selection influences all front-end metrics, considered in this work, and thus may be crucial to trace-cache performance.

It can be deduced from previous work ([23][32]) that two general principles should guide the choice of selection criteria: capturing as much of the determinism in the control flow of the program and avoid thrashing table resources for the trace cache and trace predictor.

The first principle – capture determinism – usually translates to criteria that do not terminate a trace when there is high determinism for what the next instruction will be. Therefore it is crucial to have criteria whose coverage is rather insensitive to increasing trace length.

The second principle – avoid trashing – is partially covered by the first. By terminating traces when there is low determinism thrashing can be prevented. However, it is possible even when there is high determinism in the control flow to create a lot of unique traces and patterns that may thrash resources. Thus, a throttling mechanism may be needed to terminate traces even in the presence of biased behavior.

In the next section, a classification of trace selection methods is introduced and a discussion on how different selection schemes address the above principles is presented.

4. CLASSIFICATION OF TRACE SELECTION

Trace selection methods can be classified into one or combination of the following four categories, according to the information used during the trace construction to decide when to terminate a trace:

- ***Capacity*** – based on hardware constraints
- ***Local*** – based on single instruction
- ***Global*** – based on all instructions in a trace
- ***Dynamic*** – based on dynamic behavior

The following sections describe for each category specific selection algorithms and discuss in what way they address the issues of control flow determinism and thrashing (Section 3). We emphasize that in this work we only considered run time trace selection schemes.

4.1. Capacity

Trace capacity limitations can be viewed as the most basic type of selection. Capacity refers to termination that occurs when no more information can be stored per trace. Examples of such information are maximum number of instructions and maximum number of

CTIs. The former limitation reflects the storage limitations for the trace content, while the later reflects the need to limit the size of trace identifiers used for trace indexing in various structures such as the trace cache and trace predictor. It is apparent that capacity selection has the "highest priority" among all schemes and need to be considered in combination with any other selection scheme.

4.2. Local

For this class of heuristics the decision to terminate a trace is taken by using mainly information about the "current" instruction (provided capacity selection is not forcing a trace to be terminated). Typically, this means to examine if the instruction type satisfies a trace-termination condition. This examination does not require any global state about the trace. Table 4.1 shows different types of control transfer instructions (CTI) that can terminate a trace.

CF	**conditional forward branch**
CB	**conditional backward branch**
DJ	**unconditional direct jump**
IJ	**unconditional indirect jump**
DC	**unconditional direct call**
IC	**unconditional indirect call**
RET	**Return**

Table 4.1 Code Names for CTI

Local selection criteria attempt to capture control flow determinism derived from either instruction semantics or typical program behavior. Example criteria from previous work [31] are (a) not terminating on direct unconditional branches because their target is always deterministic, (b) not terminating on backward branches because they tend to be more taken, and (c) terminating on returns because functions can be called from several points in a program and thus their continuation may not always be the same.

In this work, several local schemes were considered. Each row in Table 4.2 defines a unique selection scheme. The types of instructions that terminate a trace are denoted by shaded boxes while those that do not terminate a trace by white boxes.

B: Select traces that are only basic blocks – terminates on all CTI types.

F: Traces may contain forward branches.

FJ: Traces may contain forward branches and jumps.

UB: Allow loops without any CTI in their bodies to be unrolled into a trace.

UF: Allow for loops with only forward branches in their bodies to be unrolled into a trace

P: Traces do not cross function boundaries

ND: Terminates traces on CTI with high determinism, while including non-deterministic CTIs.

NU: Terminates traces only on backward branches.

C: Does not terminate on any CTI.

Note that all schemes terminate traces on exceptions, traps and hardware interrupts.

	CF	CB	DJ	IJ	DC	IC	RET
B	■	■	■	■	■	■	■
F		■	■	■	■	■	■
FJ		■			■	■	■
UB	■		■	■	■	■	■
UF			■	■	■	■	■
P					■	■	■
ND			■		■		
NU		■					
C							

Table 4.2 Criteria for Local Selection

The **B** and **C** schemes represent the end-points in the spectrum of selection schemes as far as trace length. **B** is expected to produce the shortest traces and the highest coverage. **C** is expected to produce the longest traces but suffer for most non-determinism and therefore low coverage.

Why Local Selection may not be Sufficient?

Local selection may work well for small trace lengths. However, the uniform treatment of instructions based on their type and not their behavior may be insufficient for longer traces.

For example, the likelihood of a trace to be correctly predicted usually decreases as more CTIs are included. Thus, correct prediction and coverage may decrease with longer traces. This is not always the case, e.g., a return from a function called only from one caller should not terminate a trace because its continuation is always the same.

These limitations motivate selection schemes that rely on global and dynamic information which are discussed in the next sections.

4.3. Global

For global selection schemes, the decision to terminate a trace may be based on information about other instructions in the trace. Such information can include the relation between two distant instructions, recurring patterns, or other structural characteristics that cannot be observed with the perspective of a single instruction. Below we discuss two such heuristics.

Loop Unrolling (LU): a trace selection heuristic aiming to exploit the regularity in the iterative control flow of a program. Specifically, during the construction of a trace, if a taken backward branch (or jump) is detected and its target is already in the trace then the loop gets unrolled. Effectively, loop unrolling aims to improve local schemes that do not terminate on backward branches by reducing their non-determinism.

The following **LU** selection schemes were considered:

- **LUNB:** Disallow intermediate branches between target and backward branch,
- **LUB:** Allow intermediate branches,
- **LUH**: Require the target to be the first instruction in the trace. This was examined in conjunction with intermediate branches (**LUHB**) and without them (**LUHNB)**

Loop unrolling needs to be considered in combination with a local scheme that terminates traces on conditional backward branches. The local schemes **F** and **FJ** (see Section 4.2) terminate traces on backward branches. These local schemes were evaluated in combination with the above **LU** configurations, resulting in eight unique configurations.

The trace length of a selection scheme that combine **LU** with **F**(**FJ**) cannot exceed that of the local schemes **UF**(**P**), because (a) the schemes **UF** and **P** never terminate on backward branches, and (b) **UF** is inclusive of **F** and **P** of **FJ**. The expectation is that **LU** unrolling will have higher trace length than the local schemes that terminate on backward branches and higher coverage than schemes that always include conditional backward branches in traces.

The amount of unrolling is another important design parameter. This design dimension is not discussed due to space limitation. The **LU** results in this paper were obtained with maximum unrolling of ten.

Continue Beyond Returns (IN): in most previous work, trace selection schemes terminate a trace on a return instruction to prevent the creation of large number of unique traces. Not terminating a trace on a return instruction can lead to building several traces because functions are usually called from several locations in a program. However, when the return and its corresponding call instruction appear in the same trace then not terminating on a return may be desirable because the next address is unique (and deterministic). This type of selection tries to keep together the caller and callee code, resembling the effect of procedure inlining. Consequently, this selection is denoted by **IN**.

The **IN** global selection needs to be combined with a local selection scheme that does not terminate on function calls (**IC** and **DC**) and returns (**RET**). The **C** local selection scheme (see Section 4.2) does not terminate on any CTI, thus it is expected to suffer from low coverage due to non-determinism. The combination of **C** with the **IN** global selection (**IN+C**) will terminate a trace on a return instruction when its corresponding call instruction is not in the current trace. The expectation is that global selection (**IN+C**) as compared to **C**, will increase the coverage at the expense of shorter traces. In the worst case, the trace length may be as short as that of the local selection scheme that terminates traces on function boundaries (scheme **P**).

4.4. Partial Ordering of Local and Global Selection Methods

It should be evident by the discussion so far that local and global selection heuristics can be ordered according to *inclusion* of termination criteria. The ordering for the various schemes considered in this work is shown in Figure 4-1. For example, scheme **B** includes scheme **F**, because **B** contains all termination criteria of **F**. Virtually always, a scheme that is included in another scheme, say **X**, will produce longer traces than **X**, since it has fewer conditions for terminating traces. Thus, scheme **F** is likely to produce longer traces than **B**. However, the relative performance of schemes that do not have inclusion relation (e.g. **P** and **ND**) will be determined by the distribution of instruction types.

Inclusion is trivial to determine across local selection schemes (based on Table 4.2). The inclusion relation of a global scheme can be defined with respect to local schemes that have less or more constraints. For instance, scheme **LUNB+F** is included in **F,** since **F** always terminates on backward branches. Scheme **LUNB+F** includes **UB** because **UB** never terminates on backward branches. Recall that **LUNB+F** always terminates when the target is not in the trace.

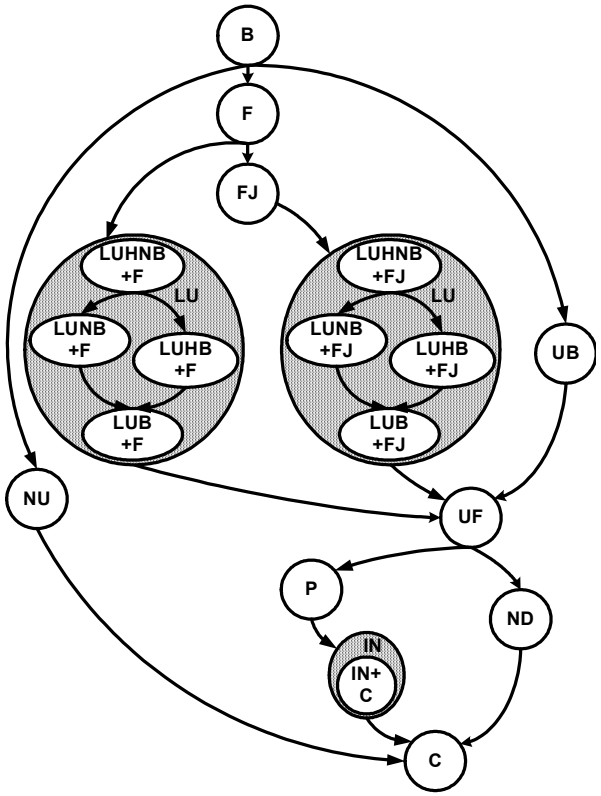

Figure 4-1 Partial Ordering of Local and Global Selection Methods

It is possible that interactions between trace selection and maximum trace length to cause violation of the inclusions with respect to the trace length. Nevertheless, the inclusion graph was found to be usually correct and useful in understanding the relations and potential of various selection schemes we considered.

4.5. Dynamic Selection and Trace Merging

Local and global schemes rely mainly on information in the static program structure. Dynamic selection is intended to capture *run-time program behavior*. In this respect, dynamic selection is closely related to well-known microarchitectural techniques such as caching and history-based prediction.

The dynamic information considered in this study is the ***bias*** of instructions [24][26]. Qualitatively, bias measures the likelihood of a transition from a CTI instruction to one of its possible targets. A biased instruction will tend to have the same successor. Consequently, biased information can be used to prolong a trace that would otherwise terminate.

The notion of bias can be naturally extended to the bias between larger segments of code such as basic-blocks and traces. This is "natural" when combining local and global selection schemes with dynamic. The non-dynamic criteria can be viewed as base schemes upon which dynamic selection can improve. Local and global selection criteria are used in the construction of traces from

instructions while dynamic selection is used to merge previously created traces into newer, longer traces. This improvement process is incremental, and may be repeated several times. In this work bias is considered at the granularity of traces and is used for merging traces. In previous work bias was considered only between basic blocks [26].

There are two types of bias information that can be considered: the ***in-bias*** and ***out-bias***. In-bias refers to the likelihood of a trace to be preceded by another one, whereas the out-bias refers to the tendency of a trace to be followed by a second trace. If a trace does not exhibit in-bias, it may be a good point to start a new trace because otherwise it may lead to the construction of multiple traces. Similarly, a trace not exhibiting out-bias may be a good point to terminate a trace.

The decision to merge traces can be taken upon one of the following bias combinations, represented as a Boolean formula:

- **In**: In-bias only
- **Out**: Out-bias only
- **In&Out**: In-bias AND out-bias
- **In|Out**: In-bias OR out-bias

Bias Predictor

Bias based schemes require a predictor. This is a table that can contain in each entry a TID and a count that indicate whether a trace is biased and towards what trace. In [24][25][26] two options are presented for indexing the bias predictor: IP or path based.

This work considers tables for in and out bias indexed with a TID. Alternatively, a BB path or a sequence of TIDs could have been used. Exploring different information vectors to index the bias table may represent important direction for future work.

Bias counters are incremented when a trace is biased to its successor (predecessor) and reset on at most two consecutive unbiased events. The bias predictor is updated at retirement. It is consulted at retirement to decide according to a threshold whether to merge traces. When a merge takes place both the TPT and trace cache are updated.

In the experimental section, dynamic selection is considered in combination with the local selection schemes presented in Section 4.2. This produces 36 combinations (4 dynamic x 9 local). The dynamic selection schemes are evaluated for several thresholds.

5. SIMULATION FRAMEWORK

Trace driven simulation is used to empirically study the performance of various selection schemes. The simulator models only the front end; it does not model pipeline latency effects. Front-end simulation is considerably faster and enables larger design space exploration. Future work should consider full microarchitectural simulation that includes the timing effects of various trace related functions.

The trace cache has no misses. The TPT is path based [13], contains 64K-tagged entries and is fully associative. The predictor is indexed using a path register 35 IPs long. The in-bias and out-bias tables have both 64K entries, are fully associative and tagged. LRU replacement is used with all tables and individual entries are updated using one bit hysteresis. Fully associative tables were chosen to eliminate interference from collisions.

The following thresholds were considered in conjunction with dynamic selection: 4, 8, 16, 32, 64, 128, 256, 512, 1024, and 4096. The same threshold was used for in and out bias for configurations that consider both.

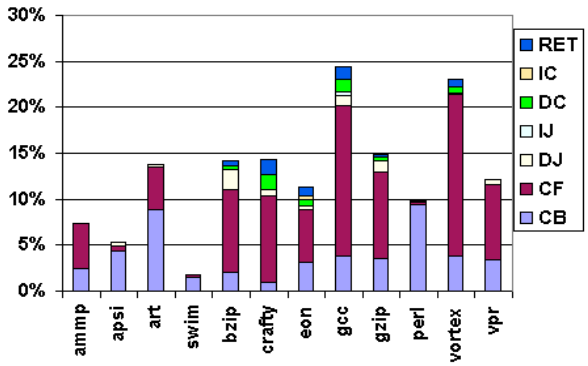

Figure 5-1 Dynamic distribution of CTI

The evaluation workload consists of traces from the SPEC-CPU2000 suite: (integer benchmarks): *bzip*, *crafty*, *eon*, *gcc*, *gzip*, *perl*, *vortex* and *vpr*, (floating point benchmarks): *ammp*, *art*, *apsi* and *swim*. These benchmarks were selected after a cursory analysis indicated they were representative of SPEC-CPU2000 behavior. The benchmarks were compiled and run on the IA32 architecture. Each trace is approximately 30 million instructions. The dynamic contribution of CTI for each benchmark is depicted in Figure 5-1**.**

6. RESULTS

6.1. Overall Best Performance

Table 6.1 summarizes the best selection performance, according to our objective function, for each benchmark. The results are divided into three categories according to the proposed selection classification: Local, Global and Dynamic. For each category, results are further divided according to maximum trace length (MTL): 64 and 256 (results for MTL 16 are only shown for the local category). Dynamic schemes were not considered with MTL 16, because non-dynamic schemes leave small room for improvement. For each subcategory and benchmark – represented as a cell in the table – the table contains the configuration that achieved the highest objective function value. Therefore, each row contains the best performing selection schemes for each selection category and maximum trace length. For each benchmark (row), the configuration that achieved the best performance for MTL 64 across all categories is marked with light shadowing. Similarly, dark shadowing marks the configuration that achieved the best performance for MTL 256. When there is no clear advantage of one configuration over another, both were shadowed. Finally, each benchmark has a table cell with bold border marking the selection scheme that achieved the highest objective function value across all categories (irrespective of MTL).

	Local			Global		Dynamic	
	16	**64**	**256**	**64**	**256**	**64**	**256**
Eon	ND	P	P	LUB+FJ	LUB+FJ	Out_4+UB	Out_32+B
Vortex	ND	UF	UF	LUB+F	LUB+F	Out_4+UB	Out_32+B
Vpr	UF	UB	UB	LUB+F	LUNB+F	In&Out_64+B	Out_1024+B
Gcc	ND	F	F	LUHB+F	LUHNB+F	Out_8+B	Out_16+B
Gzip	UB	UB	UB	LUNB+F	LUNB+F	Out_64+UB	Out_64+UB
Ammp	ND	F	F	LUB+F	LUHNB+F	Out_32+B	Out_128+B
Art	P	UF	P	IN+C	LUB+FJ	In|Out_16+UB	In|Out_4+B
Perl	ND	ND	NU	IN+C	LUNB+NU	Out_4+ND	Out_16+B
Apsi	P	P	P	IN+C	IN+C	Out_256+P	Out_4+P
Swim	UF	UF	UF	IN+C	LUHNB+FJ	Out_8+UB	Out_16+F
Crafty	ND	F	F	LUNB+F	LUNB+F	Out_16+UB	Out_128+UB
Bzip	ND	F	F	LUHNB+F	LUHNB+F	Out_256+B	Out_256+B

Table 6.1 Summary of best selection schemes according to category and maximum trace length.

We explain the table contents with benchmark *swim*. The selection schemes with the highest objective function value for MTL 64 are **local-UF** and **dynamic-Out_8+UB**. These two configurations have roughly equal performance. For MTL 256 the best performance comes from **global-LUHNB+FJ**. Across all categories, irrespective of MTL, the highest objective function value is provided by the configuration **global-LUHNB+FJ.**

Several observations can be made from the table. Overall, the best performing category is dynamic. This is true for both MTL 64 and MTL 256. Nevertheless, the data also suggest that simple schemes from local and global configurations may suffice with MTL 64 for six benchmarks (*bzip*, *crafty*, *swim*, *apsi*, *perl, and art*) and for four benchmarks (*bzip*, *crafty*, *swim*, and *apsi*) with MTL 256.

Another observation from the data (by examining columns) is that the best performing selection scheme is benchmark specific. This is particularly prevalent for local and global selection. For example, six different selection schemes provide the best performance for local MTL 64. For the dynamic category there is less variance: selection based on out bias seems to be the best choice for most benchmarks. However, there is variation with respect to the local scheme to use in combination with bias information and the threshold value. The above suggests that trace selection may need to be enhanced with adaptive mechanisms to achieve the best performance. Analysis, not shown here, suggests that a mechanism that can effectively adapt a bias threshold is most promising.

The benchmark behavior comes in several flavors. For *crafty* and *bzip* there is no benefit going to longer traces, MTL 16 is sufficient to achieve the best performance. As shown in the next subsection, for these two benchmarks coverage decreases with increasing MTL while their hot-trace length is not increasing. The low coverage of bzip is the result of low prediction rate due to data driven control in this benchmark.

For one benchmark, ammp, performance degrades when going to longer traces. Two benchmarks, *gcc* and *gzip*, do not benefit from increasing the MTL form 64 to 256. As shown in the next section, this is caused by low coverage and/or insignificant increase in hot trace length. Recall that the only events in the experimental framework that influence coverage are trace mispredictions. For benchmarks *swim*, *apsi*, *perl*, *art*, *vpr*, *vortex* and *eon* there is potential with long traces (the best overall performance is achieved with MTL 256). These benchmarks should exhibit reasonable coverage and hot trace length and potentially could benefit from even MTL longer than 256.

Another general trend, across benchmarks, is that for local selection with increasing MTL the local scheme that performs best has the same or more criteria (more restrictive) as compared to the selection scheme for a smaller MTL. For example, for *perl* the best local selection with MTL 16 and 64 is **ND**, however, for 256 it is **NU.** As displayed in Figure 4-1, **NU** is inclusive of the **ND** criteria. This trend should be expected since as MTL increases local selection would prefer to keep traces shorter since the non-determinism increases with trace length. The same phenomenon for the same reasons occurs for global schemes. For example, for *vpr* with MTL 64 the best performing scheme is **LUB+F** whereas for MTL 256 is **LUNB+F**. The **LUB** allows intermediate branches whereas **LUNB** does not.

The next section presents different performance metric values for the configurations in Table 6.1.

6.2. Objective Function and Performance Metrics for Best Configurations

Figure 6-1 to Figure 6-4 show the objective function and various performance metrics for the benchmarks and configurations in Table 6.1. The ratio of hot to cold trace-length is presented in Figure 6-5. The results in these figures are useful for establishing the performance potential with different trace selection schemes, identifying correlations and understanding the causes of different benchmark behavior. Due to limited space, we only elaborate on some of these issues.

Figure 6-1 shows the value of the objective function and Figure 6-2 shows the normalized objective function values. The normalization is computed separately for each benchmark. The data in these figures show that benchmarks can be divided into two categories: those with high performance potential, defined to have at least one configuration with objective function value greater than 20 (*art*, *perl*, *vortex*, *apsi*, *swim*, *eon* and *ammp*), and those with low potential, (*bzip*, *crafty*, *gcc*, *gzip* and *vpr*). *bzip* exhibits the worst performance due to low predictability caused by data driven control flow.

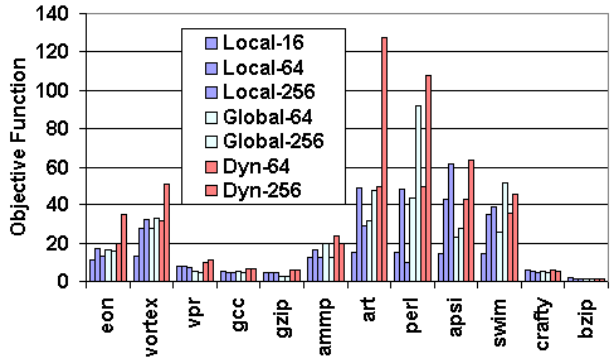

Figure 6-1 Best objective-function per category/length

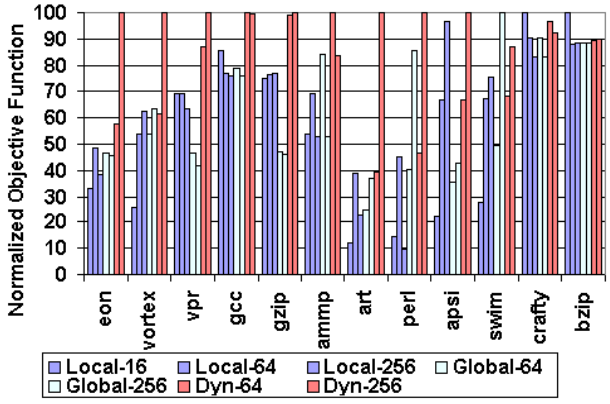

Figure 6-2 Normalized objective-function

Figure 6-3 and Figure 6-4 can be used to explain the dichotomy observed in the benchmarks objective function behavior. It can be seen in Figure 6-3 that the coverage for benchmarks with high potential is virtually insensitive to the trace length and the selection scheme used, and that coverage is typically higher than 95%.

Figure 6-4 shows that the hot trace length for benchmarks with high potential is typically 40 instructions or above for several configurations. Furthermore, for benchmarks with high potential we observe that (a) within a given selection category, hot trace length increases with increasing MTL, and (b) hot traces tend to be longer than the cold ones (see Figure 6-5).

On the other hand, benchmarks with low potential have low coverage and display more sensitivity to the information used for selection. Typically, their coverage is lower than 90%. These benchmarks have short trace length, typically less than 30, and the ratio of hot/cold length trace is less than one.

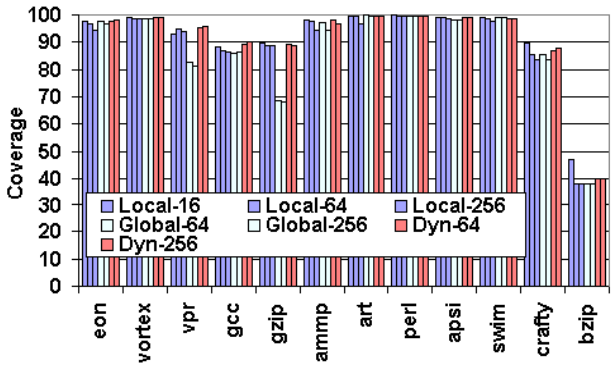

Figure 6-3 Coverage

The latter underlines the importance of selecting traces so that mispredictions and misses are isolated in short traces.

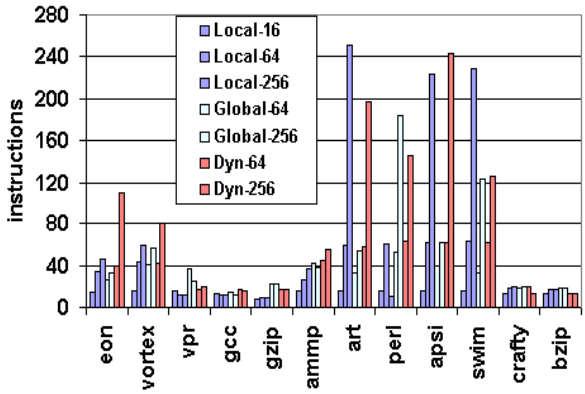

Figure 6-4 Hot-trace length

From Figure 6-3 and Figure 6-4 it can also be observed that overall the average coverage is 98% for floating point programs and 88% for integer programs, these numbers are the same with maximum trace length 64 and 256. The average hot trace length for MTL 64 is 57 IA32 instructions for FP programs and 28 instructions for integer programs, whereas for MTL of 256 it is 156 instructions for floating point and 52 for integer programs.

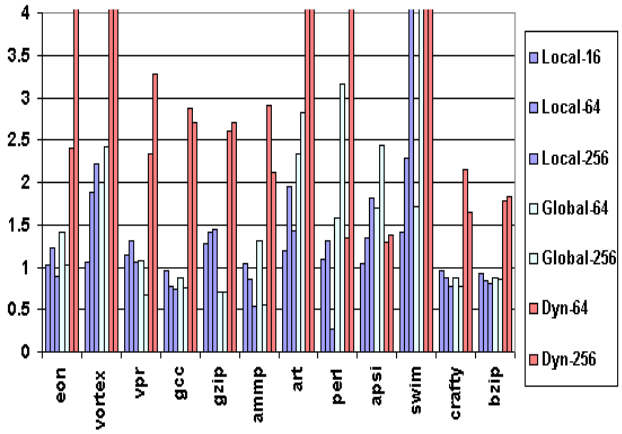

Figure 6-5 Ratio Hot/Cold Trace Length

Figure 6-5, presents the ratio of hot trace length to cold trace length. We focus on the most interesting behavior that is around ratio of one, therefore the graph does not show maximum values for few benchmarks that exceed 4. It can be observed that across all benchmarks only the dynamic selection provides significantly longer hot than cold traces (usually at least twice as long).

6.3. Performance of Different Selection Schemes in Same Category

The analysis of results so far was done macroscopically by considering only the best selection scheme from each category. Figure 6-6 provides data for comparing selection schemes in the same category. Due to limited space, only the normalized average and range of objective function values across all benchmarks is shown for MTL 64. Objective function averages are computed separately for each scheme, and then averages are normalized across all selection schemes. For **LU** and dynamic selection the best combination with a local scheme is shown. The best local selection with MTL 16 is shown for reference.

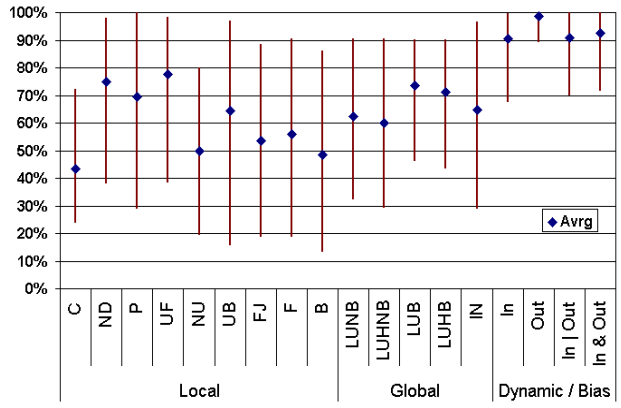

Figure 6-6 Normalized average and range of Objective Function for MTL 64

As observed in previous sections, dynamic selection provides the best performance across all categories and benchmarks. The data in Figure 6-6 shows that the dynamic selection scheme with the best performance is **out-bias**. On the average, out-bias is 10% better than any other scheme for MTL 64. Analysis, not shown here, also revealed that out-bias is always within 95% of the best performing scheme.

For global selection, the best performance is obtained by **LUB** and **LUHB,** whereas for local selection it is obtained by **UF** and **ND**. The average performance of dynamic schemes is the highest. The results for MTL 256 are similar, however, the difference between out-bias and the next best selection scheme is larger.

7. RELATED WORK

The notion of identifying frequently executed paths (traces) across basic blocks for optimization has been prevalent for the software-based schemes reported in [8][18][6][2][11] and more recently for hardware based schemes [21][22][26][25][7][15]. The various proposals mainly differ in one or more of the following: the methodology and resources used for detecting the hot paths, the structure and address space used for storing the hot paths, and the timing and resources used for optimization. The common characteristic in all these schemes is the use of run time (dynamic) information for selecting hot paths.

This paper draws from previous work in trace-caches [27][31][23]. The original proposals of selection for trace caches were local and for short traces. As shown in this work such an approach may provide sufficient coverage for short traces but not with increasing maximum trace length. The first paper to consider dynamic information in conjunction with traces promoted biased branches to assertions [24].

The work closest to this paper is [26]. The goals in that paper are similar to these in this work. [26] considers path-based out-bias information to merge basic blocks. Our paper builds on the idea for dynamic criteria for trace selection and atomic traces [19][20].

This work is distinguished from the study reported in [26] in several aspects. Our goal is to maximize trace length while *preserving* a high degree of coverage. We quantify this tradeoff using an objective function that combines the effect of coverage and hot trace length that emphasizes the importance of coverage. [26] appears to have more emphasis on increasing trace length.

We provide a framework that *combines* different types of dynamic information (in, out, in or out and in-out) in conjunction with different local schemes, and merging at trace granularity. In previous work, only the out bias was considered and merging was done at the granularity of basic blocks.

Furthermore, in this work the trace sequencing is done with a predictor whereas in [26] an ideal next fetch address predictor was assumed. In a follow-up work, [7], a real path-predictor was employed.

The biased scheme used in this work is TID-based whereas in previous work it was path based. The two could possibly be combined but we did not explore that option. As far as we know, the global schemes considered here have not been considered elsewhere for trace selection. Analogous concepts exist for the selection of hot paths that can be stored in memory [22].

8. CONCLUSIONS AND FUTURE WORK

This paper presents a comprehensive study of selection schemes for long atomic traces. The paper introduces a classification of trace selection methods based on the information used to select traces. Existing and novel selection methods are discussed. New selection methods considered include loop unrolling, procedure in-lining and incremental merging of traces based on different types of dynamic bias.

The paper includes an empirical analysis of various selection schemes with increasing trace length in an idealized framework. Several observations are made using benchmarks from the SPEC-CPU2000 suite. Selection based on dynamic bias information is necessary to achieve the best performance across all benchmarks. However, simple selection schemes relying on program structure are sufficient to achieve the best performance for half the benchmarks when maximum trace length is 64 instructions. Another observation is that adaptive trace selection may be needed because the best selection scheme depends on the benchmark and maximum trace length used.

Two alternatives for the trace selection mechanism are established: (a) a “best performance” approach relying on complex dynamic criteria; (b) a “value” approach that provides the best performance (and potentially the best power consumption) based on simpler static criteria. Another emerging alternative advocates adaptive based mechanisms to adjust selection criteria.

This work points to several directions for future work. Enhancing trace selection with adaptive mechanisms is one that appears very promising. Future work, should investigate the performance with a complete microarchitectural model for specific selection schemes. Such a study should address trace-specific phenomena such as trace cache miss penalty, and provide detailed performance and power analysis. Another direction of research is to consider selection for long but non-atomic traces.

9. REFERENCES

[1] G.M. Amdahl, "Validity of the single-processor approach to achieving large scale computing capabilities", in *AFIPS* vol. 30, pp. 483-485, 1967.

[2] V. Bala, E. Duesterwald and S. Banerjia, "Transparent Dynamic Optimization: The Design and Implementation of Dynamo", TR HPL-1999-78, HP Labs.

[3] B. Black, B. Rychlik and J. Shen, "The Block-based Trace Cache", in *ISCA26*, May 1999.

[4] P.P. Chang and W.W. Hwu, "Trace selection for compiling large C application programs to microcode" in *MICRO21*, pp. 188-198, Nov. 1988.

[5] T. M. Conte, K. N. Menezes, P. M. Mills and B. A. Patel, "Optimization of instruction fetch mechanisms for high issue rates",in *ISCA22*, Jun. 1995.

[6] K. Ebcioglu and E.R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility", in *ISCA24*, pp. 26-37, 1997.

[7] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S.J. Patel and S.S. Lumetta, "Permormance Characterization of a Hardware Mechanism for Dynamic Optimization", *MICRO34*, Dec. 2001.

[8] J.A. Fisher, "Trace Scheduling: A technique for Global Microcode Compaction", in *IEEE Transactions on Computers*, 30(7), pp. 478-490, July 1981.

[9] D. Friendly, S. Patel and Y. Patt, "Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism", in *MICRO30*, Dec. 1997.

[10] D. Friendly, S. Patel and Y. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors", in *MICRO31*, Nov. 1998.

[11] M. Gschwind, E.R. Altman, S. Sathaye, P. Ledak and D. Appenzeller, "Dynamic and Transparent Binary Translation", in *IEEE Computer Magazine* 33(3), pp. 54-59, 2000.

[12] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker and P. Roussel, "The Microarchitecture of the Pentium® 4 Processor", in *Intel Technology Journal*, 2001.

[13] Q. Jacobson, E. Rotenberg and J.E. Smith, "Path-Based Next Trace Prediction", in *MICRO30*, 1997.

[14] Q. Jacobson and J.E. Smith, "Instruction Pre-Processing in Trace Processors", in *HPCA5*, 1999.

[15] Q. Jacobson and J.E. Smith, "Trace Preconstruction", in *ISCA27*, pp. 37-46, May 2000.

[16] S. Jourdan, L. Rappoport, Y. Almog, M. Erez, A. Yoaz, and R. Ronen, "eXtended Block Cache", in *HPCA6*, Jan. 2000.

[17] T. Juan, S. Sanjeevan and J.J. Navarro, "Dynamic History-Length Fitting: A Third Level of Adaptivity for Branch Prediction", in *ISCA25*, pp 155-166, 1998.

[18] S.A. Mahlke and D.C. Lin and W.Y. Chen and R.E. Hank and R.A. Bringmann, "Effective Compiler Support for Predicated Execution using the Hyperblock", in *MICRO25*, 1992.

[19] S.W. Melvin and Y.N. Patt, "Performance Benefits of Large Execution Atomic Units in Dynamically Scheduled Machines", in *Proc. ICS'3*, pp. 427-432, 1989.

[20] S. Melvin and Y Patt, "Enhancing Instruction Scheduling with a Block-Structured ISA", in *Intern. Journal of Parallel Prog.*, 23(3) pp 221-243, Jun. 1995

[21] M.C. Merten, A.R. Trick, C.N. George, J. Gyllenhaal, and W.W. Hwu, "A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization", in *ISCA26*, 1999.

[22] M.C. Merten, A.R. Trick, E. M. Nystrom, R.D. Barnes and W. Mwu, "A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hot Spots", in *ISCA27*, May 2000.

[23] S. Patel, D. Friendly and Y. Patt, "Critical Issues Regarding the Trace Cache Fetch Mechanism", Univ. of Michigan Technical Report CSE-TR-335- 97

[24] S. Patel, M. Evers and Y. Patt, "Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing", in *ISCA25*, June 1998.

[25] S. Patel and S. Lumetta, "rePlay: A Hardware Framework for Dynamic Optimization", in *IEEE Trans. on Computers*, 50(6), pp 590-608, June 2001

[26] S. Patel, T. Tung, S Bose and M. Crum, "Increasing the Size of Atomic Instruction Blocks using Control Flow Assertions", in *MICRO33*, 2000.

[27] A. Peleg and U. Weiser. "Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independent of Virtual Address Line", U.S. Patent 5,381,533, Jan. 1995.

[28] M. Postiff, G. Tyson and T. Mudge, "Performance Limits of Trace Caches", in *Journal of ILP,* vol. 1, Oct. 1999.

[29] A. Ramírez, J. L. Larriba-Pey, C. Navarro, J. Torrellas and M. Valero, "Software trace cache", in *Proc. ICS13*, pp. 119–126, 1999.

[30] R. Rosner, A. Mendelson and R. Ronen, "Filtering Techniques to Improve Trace-Cache Efficiency", in *PACT'01*, Sept. 2001.

[31] E. Rotenberg, S. Bennett and J.E. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching", in *MICRO29*, Dec. 1996.

[32] E. Rotenberg, S. Bennett and J. Smith, "A trace cache microarchitecture and evaluation", in *IEEE Trans. on Computers*, 48(2), pp 111–120, Feb. 1999

[33] E. Rotenberg, Q. Jacobson, Y. Sazeides and J.E. Smith, "Trace Processors", in *MICRO30*, Dec. 1997.

[34] E. Rotenberg and J.E. Smith, "Control Independence in Trace Processors", in *MICRO32,* Nov. 1999.

[35] B. Solomon, R. Ronen, D. Orenstien, Y. Almog and A. Mendelson "Micro-Operation Cache: A Power Aware Frontend for Variable Instruction Length ISA", in *ISLPED'01*, Aug. 2001.

[36] B. Slechta et al, "Dynamic Optimizations of Micro-Operations", in HPCA9, Feb. 2003.