科學敎育
Journal of Science Education 16, 191-201, 1999

The Analysis of Prefetch Accuracy Using The Ideal Instruction Prefetch Model

Seong Baeg Kim* Jung hoon Lee**

Abstract

Prefetching aims at reducing memory latency by fetching, in advance, instructions and data that are likely to be requested by the processor in a near future. The effectiveness of prefetching depends on how accurate the prediction on the needed instructions and data is. Most previous studies on prefetching were limited to proposing a particular prefetch scheme and evaluating its performance, giving little attention to its theoretical aspects. This paper focuses on the limit of instruction prefetching. For this purpose, we propose an ideal prefetch model that makes use of a post-mortem analysis of a program execution to derive an upper bound on prefetch accuracy. Based on this model, we analyzed the upper bounds on prefetch accuracy (i.e., theoretically achievable prefetch accuracy) of programs from the SPEC benchmarks. The results show that when there is no instruction cache the upper bounds are very high. However, as the size of the instruction cache increases, the prefetch accuracy bound drops drastically. For example, in the case of the *spice* benchmark, the achievable prefetch accuracy drops from 53 % to 39 % when the cache size increases from 2 Kbytes to 16 Kbytes (assuming 16 byte block size). One implication of this result is that when the cache size is over some threshold and the prefetch miss penalty is high, instruction prefetching may degrade the overall performance rather than improve it.

1. Introduction

Recent advances in electronic technology have drastically reduced processor cycle time. In the DRAM technology, however, the cycle time improvement has been slow although density growth has been tremendous. The resultant processor-memory cycle time disparity necessitated the use of high speed buffers such as instruction and data caches. The use of caches has been very successful in bridging the gap between

* 제주대학교 컴퓨터교육과
** 제주대학교 전산통계학과

the processor and memory cycle times during the last decade. Today, with large caches and careful mapping of memory blocks to cache blocks [5, 9, 11], misses due to limited cache sizes or set-associativities are not very likely [2]. On the other hand, cold start misses and misses due to context switches are becoming the major source of cache misses [10].

There have been a number of attempts to reduce the adverse effects of cold start and context-switch misses. One such attempt is to use prefetch techniques such as sequential prefetching [13, 14] or threaded prefetching [7]. One of the important aspects of prefetching is prefetch accuracy. Although many studies have proposed efficient prefetch schemes, none of them studied on the intrinsic limitation of prefetching such as an upper bound on prefetch accuracy. Furthermore little has been studied on the effects of prefetch time and the presence of cache memory on the theoretically achievable prefetch accuracy.

Our work focuses on the theoretical aspects of instruction prefetching. In particular, we derive an upper bound on prefetch accuracy using an ideal prefetch model. We also investigate the effects of prefetch time and the presence of cache memory on the prefetch accuracy bound.

One of the advantages of these analyses is that they reveal the intrinsic limitation of prefetching. Another advantage is that the derived prefetch accuracy bound provides a ground on which the effectiveness of other prefetch schemes is evaluated.

The remainder of this paper is organized as follows. In the next section, we survey the related work. Section 3 describes an ideal prefetch model from which the upper bound on prefetch accuracy is derived. In Section 4, we give upper bounds on prefetch accuracy of SPEC benchmarks. This section also investigates how the prefetch time and the presence of cache memory affect the theoretically achievable prefetch accuracy. Finally, we conclude this paper in Section 5.

2 Related Work

Caching and prefetching have been the subjects of intensive investigations since they are critical to the performance of high performance processors and their role is becoming increasingly important due to the growing gap between the processor cycle time and the DRAM access time. Caching is efficient since it makes effective use of localities (both temporal and spatial) of programs. On the other hand, prefetching draws most of its advantages by bringing, in advance, the memory blocks that are likely to be requested by the processor in a near future.

Compared with cache memories, prefetching has received far less attention.

In most previous studies on instruction prefetching, prefetching was limited to the next sequential instruction block. For example, Smith, in [14], investigated three sequential prefetch schemes with increasing sophistication: *always prefetch*, *prefetch on misses*, and *tagged prefetch*. In the *always prefetch* scheme, the next sequential instruction block is prefetched after each instruction block reference. On the other hand, *prefetch on misses* prefetches the next block only on a cache miss. *Tagged prefetch*, an enhancement of *prefetch on misses*, associates with each memory block a tag bit that guides the prefetching.

This bit allows a prefetch not only on a cache miss but also on a cache hit on a prefetched block. Other studies on sequential instruction prefetching include the *stream-buffer* approach in which several sequential blocks are prefetched on a cache miss to hide the ever increasing memory latency [6]. Examples of processors that make use of sequential instruction prefetching are the IBM System/370 Models 145, 158, 168, 195 [12]; the CDC 6600 [12]; the Manchester University MU5 [12]; the VAX-11/780 [3]; the M68020 /68030 [4]; the Intel 8086/80286/80386 /80486 [1].

The above prefetch schemes are limited to the prefetching of sequential blocks and, therefore, share the same problem caused by taken branches. To rectify this problem,

Kim et al. proposed a scheme called *threaded prefetching* [7]. In this scheme, each instruction block has an instruction block pointer called *thread*. The thread indicates the instruction block to be prefetched when the block containing it is accessed by the processor. This scheme operates in two different modes: real-time and non real-time modes. In the non real-time mode, the thread is dynamically updated during program execution so that it indicates the instruction block that is most likely to be accessed next. By the principles of locality, the block that is most likely to be accessed next is the block that was previously accessed after the present block. Therefore, the thread is made to point to such a block in the non real-time mode. In the real-time mode where the worst case performance is of utmost importance, the prefetching of instruction blocks is made in the direction that improves the worst case execution time. For this purpose, the thread is generated by the compiler through an analysis on the worst case execution path [8].

In a prefetch scheme, if the time to execute an instruction block is shorter than the time needed to prefetch a block, the processor has to stall even on a prefetch hit. This stall time can be reduced by prefetching a block that is two blocks ahead of the current block rather than just one block [7]. When this lookahead technique is

used in sequential prefetching, prefetch is made for the block physically situated next to the next block of the current block. On the other hand, if this technique is applied to the threaded prefetching, the thread will point to the block that is two blocks ahead of the current block in the most likely execution path. This lookahead threaded prefetching can easily be done by providing a buffer that contains the address of the previous instruction block and updating the thread of the this instruction block based upon the instruction block accessed after the current block. A more aggressive approach is to use a queue, rather than a buffer, that contains the addresses of the previously accessed instruction blocks and to update the thread of the instruction block at the front of the queue based upon the instruction block accessed after the current block [7]. In this case, the capacity of the queue determines the degree of lookahead.

3 Ideal prefetch model

In this section, we describe an ideal prefetch model and give methods for deriving an upper bound on prefetch accuracy using the ideal model. The ideal prefetch model is based on a post-mortem analysis of program execution. To see the motivation behind our ideal prefetch model, consider the example given in Figure 1.

block b_1 : b_2 -- b_2 -- b_2 -- b_2
block b_2 : b_3 -- b_{10} -- b_3 -- b_{10}
block b_3 : b_4 -- b_{20} -- b_{20} -- b_4
block b_4 : b_5 -- b_{30} -- b_{30} -- b_5
block b_5 : b_6 -- b_{15} -- b_{25} -- b_6

Figure 1: Block behavior table

Figure 1 shows, for each instruction block, the instruction blocks that were accessed immediately after the block during program execution. We call this table a *block behavior table*. In the example block behavior table, the program always accessed the next sequential block (i.e., b_2) after b_1. On the other hand, both the sequential and non-sequential blocks were accessed after b_2 -- b_5. In the case of b_5, more than one non-sequential block were accessed. This could be because b_5 has more than one branch instruction or it has an indirect branch instruction. The above instruction reference behavior is typical of program execution.

In the ideal prefetch model, the instruction reference behavior of a block is first transformed into what we call a *reference pattern*. In the reference pattern, each instruction block is represented by a letter. For example, the first distinct instruction block in the pattern is represented by a and the second distinct instruction block is represented by b and so on. The reference pattern for b_1 constructed in this way is a--a--a--a. Likewise,

b_2's reference pattern is a−−b−−a−−b. Figure 2 shows the reference patterns for other instruction blocks.

block b_1 : $a -- a -- a -- a$
block b_2 : $a -- b -- a -- b$
block b_3 : $a -- b -- b -- a$
block b_4 : $a -- b -- b -- a$
block b_5 : $a -- b -- c -- a$

Figure 2: Symbolic block behavior table

prefix	reference letter					
	a	b	c	d	e	···
a	1	4	−	−	−	....
a−a	1	−	−	−	−	....
a−b	1	2	1	−	−	....
a−a−a	1	−	−	−	−	....
a−a−b	−	−	−	−	−	....
a−b−a	−	1	−	−	−	....
a−b−b	1	−	−	−	−	....
a−b−c	1	−	−	−	−	....

Table 1 : Frequencies of reference patterns

From this symbolic block behavior table, another table is constructed that lists, for each possible prefix of reference patterns, the frequencies of letters that appear after the prefix. For example, there are three possible reference patterns of length 3 that have a−−b as their prefix: a−−b−−a, a−−b−−b. and a−−b−−c. Since a−−b−−a, a−−b−−b, and a−−b−−c appeared once (in block b_2), twice (in blocks b_3 and b_4) and once (in block b_5), respectively, the table has an entry for prefix a−−b that has frequencies of 1, 2, and 1 for a, b, and c respectively. Table 1 shows the entries for other prefixes for the previous example.

The ideal prefetch model uses this table to make the prefetch decision. For example, if the processor is about to initiate a new prefetch and the reference pattern associated with the current instruction block is a−−b, then the ideal prefetch algorithm will prefetch the instruction block corresponding to letter b. This decision is dictated by the fact that among the three possible reference patterns that have a−−b as their prefix (i.e., a−−b−−a, a−−b−−b and a−−b−−c), a−−b−−b has the highest frequency. Likewise, the ideal prefetch model can determine which instruction block to prefetch for all possible reference pattern prefixes. Table 2 shows the instruction block that will be prefetched by the ideal model for each possible prefix of reference patterns for the program execution described in Figure 1.

To prefetch the instruction block corresponding to a letter in a reference pattern, the ideal prefetch model should have address information associated with each letter. For this purpose, the model maintains, for each instruction block, address information for each letter in its

reference pattern. For example, the address information associated with block b_1 has an entry for a $->$ b_1. This indicates that the letter a in its reference pattern corresponds to block b_2. Likewise b_2's address information has entries for a $->$ b_3 and b $->$ b_{10}. Table 3 shows the address information maintained for each instruction block for the example given in Figure 1.

prefix	Prefetch letter
a	a
a−a	a
a−b	b
a−a−a	a
a−a−b	−
a−b−a	b
a−b−b	a
a−b−a	a

Table 2 : Prefetch decision for each prefix

block	reference letter					
	a	b	c	d	e	...
block b_1	b_2	-	−	−	−	....
block b_2	b_3	b_{10}	−	−	−	....
block b_3	b_4	b_{20}	−	−	−	....
block b_4	b_5	b_{30}	−	−	−	....
block b_5	b_6	b_{15}	b_{25}	−	−	....

Table3 : Block address information for each letter in reference patterns

block b_1 : b_2(no prefetch)→b_2(b_2 prefetched:hit)→b_2 (b_2 prefetched:hit)→b_2(b_2 prefetched:hit)

block b_2 : b_3(no prefetch)→b_{10}(b_3 prefetched:miss)→ b_3(b_{10} prefetched:miss)→b_{10}(b_{10} prefetched:hit)

block b_3 : b_4(no prefetch)→b_{20}(b_{20} prefetched:miss) →b_{20}(b_{20} prefetched:hit)→b_4(b_4 prefetched:hit)

block b_4 : b_5(no prefetch)→b_{30}(b_5 prefetched:miss) →b_{30}(b_{30} prefetched:hit)→b_5(b_5 prefetched:hit)

block b_5 : b_6(no prefetch)→b_{15}(b_6 prefetched:miss) →b_{25}(b_{15} prefetched:miss)→b_6(b_6 prefetched:hit)

Figure 3: Prefetch hits and misses as determined by the ideal prefetch algorithm

Figure 3 shows the prefetch hits and misses as determined by the ideal prefetch algorithm for our example. One case to note is when an instruction block is executed for the first time. In this case, the current reference pattern and block address information needed to make a prefetch decision are not available and, therefore, no prefetch request is made.

The prefetch accuracy of the ideal prefetch algorithm is calculated by dividing the number of prefetch hits by the total number of prefetch requests:

$$\text{Prefetch accuracy} = \frac{\text{no. of prefetch hits}}{\text{total no. of prefetch requests}} = \frac{9}{15} = 0.6$$

Let $B_p = \{b_1, b_2, ... b_{|Bp|}\}$ be the set of instruction blocks used in program P.
Let L_b be the list of blocks referenced immediately after b (i.e., block behavior table entry for b).

for each $b \in B_p$
L'_b *symbolize*(L_b)

for each prefix p that appears in the symbolic lists $L'_{b1}, L'_{b2}, \ldots L'_{b|BP|}$

Let $A = \{a_1, a_2, \ldots a_k\}$ be the set of letters that appear in p.

for each a in A

compute the frequency of $p||a$ in the prefixes of $L'_{b1}, L'_{b2}, \ldots L'_{b|BP|}$

where || is the concatenation operation.

Let $a_{p,max}$ be the letter in A that has the highest frequency.

$n_{hits} \leftarrow 0$
$n_{misses} \leftarrow 0$
for each $b \in B_p$
while (L'_b is not empty)
$l \leftarrow$ null
$actual \leftarrow delete\_first(L'_b)$
// $delete\_first(L)$ deletes the first letter in L and
// returns the deleted letter

$prediction = a_{l,max}$
if $actual = prediction$ then $n_{hits} \leftarrow n_{hits} + 1$
else $n_{misses} \leftarrow n_{misses} + 1$
$l \leftarrow l \,||\, actual$

achievable_prefetch_accuracy_bound $\leftarrow n_{hits}/(n_{hits} + n_{misses})$

Figure 4 : Algorithm for calculating the prefetch accuracy bound

The prefetch accuracy calculated in this way represents an upper bound on prefetch accuracy that can be achieved by any prefetch algorithm. The validity of this upper bound depends only on the following two assumptions:

1. The prefetch algorithm is deterministic.

2. The algorithm does not differentiate instruction blocks for prefetching purposes.

These assumptions hold for all the prefetch algorithms described in Section 2 and, therefore, their prefetch accuracies are subject to the upper bound derived from the model. Figure 4 describes, procedurally, the steps taken in the ideal prefetch algorithm. Since the information needed by the ideal prefetch model is available only after the program execution is completed, the ideal prefetch algorithm is not implementable. Nevertheless this ideal prefetch algorithm is important not only in studying the intrinsic limitation of prefetching but also in evaluating the effectiveness of existing and/or newly proposed prefetch schemes (i.e., to see how close their prefetch accuracies are to the limit).

Extensions

● When there is an instruction cache : So far we have not considered the effects that the presence of instruction cache memory has on the achievable prefetch accuracy. When an instruction cache is present and prefetching is performed between the cache and main memory, the decision on which instruction block to prefetch is generally made based on the instruction references that missed in the cache. This is especially true when the

instruction cache is placed on-chip while the prefetcher is located off-chip. The prefetch accuracy bound in this case can be derived by constructing the block behavior table based on the instruction references that missed in the cache.

● When lookahead prefetching is used : As we explained earlier, when lookahead prefetching with degree K is used, the block prefetched during the execution of an instruction block is the one that is K blocks ahead of the current block in the predicted execution path. In this case, the block behavior table entry for an instruction block can be constructed based on the instruction blocks that were accessed K blocks later.

4 Analysis results

In this section, we give prefetch accuracy bounds for six programs from the SPEC benchmark suite [16] using the ideal prefetch model described in the previous section. The six SPEC benchmark programs are *gcc*, *xlisp*, *espresso*, *doduc*, *spice*, and *fpppp*. For each benchmark, 100 million memory references were gathered to extract the information needed by the ideal prefetch model. The gathering of the memory references was performed on a MIPS RS-2030 workstation using the pixie [15] utility.

Figure 5 shows the effects of cache size, block size, and the degree of lookahead on the prefetch accuracy bound. When cache memory is not used, the upper bounds are over 99 % for all the six programs regardless of the degree of lookahead and the block size. However, when cache memory is used, the achievable prefetch accuracy decreases drastically. For example, the achievable prefetch accuracy is below 50 % for *espresso* and *spice* when an instruction cache of only 2 Kbytes is used. This can be explained by the fact that cache memory filters out most of the memory reference localities exhibited by the program. This randomization by the cache memory severely limits the achievable prefetch accuracy when the prefetch decision has to be made based on the memory references that are missed in the cache. As the size of the instruction cache increases, the prefetch accuracy drops further. For example, in the case of the *spice* benchmark, the achievable prefetch accuracy drops from 53 % to 39 % when the cache size increases from 2 Kbytes to 16 Kbytes (assuming 16 byte block size). The results also show that when cache memory is used, the achievable prefetch accuracy decreases as the block size and the degree of lookahead increase. In one extreme case (*spice*: cache size = 16 Kbytes, block size = 32 bytes, degree of lookahead = 4), the achievable prefetch accuracy is below 3 %. One implication of these results is that the use of prefetching between the cache and main memory may

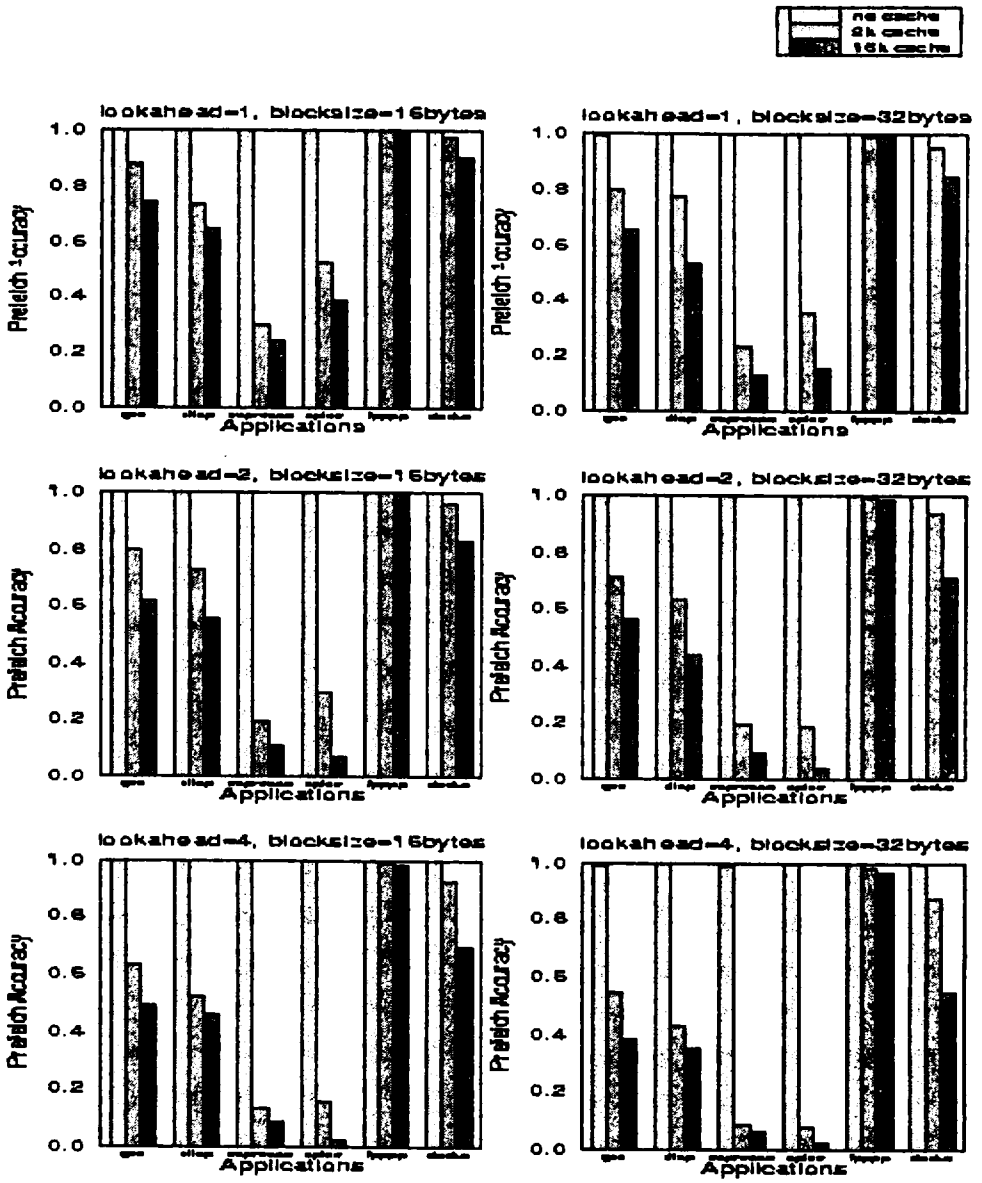

Figure 5: Prefetch accuracy bounds of SPEC benchmarks

actually degrade the overall performance rather than improve it when the size of the instruction cache and the penalty for a prefetch miss are over some thresholds.

Unlike the other four programs, the prefetch accuracy bounds of fpppp and doduc are relatively high even when cache memory is used. This can be explained by their sequential behavior. For fpppp, more than 99 % of instruction blocks that are referenced after an instruction block are sequential blocks. Most of these sequential references result in prefetch hits in the ideal prefetch model and this makes the prefetch accuracy bound of fpppp largely insensitive to the presence and/or the size of the instruction cache.

5 Conclusions

This paper has proposed an ideal prefetch model to derive an upper bound on prefetch accuracy. The proposed model makes use of a post−mortem analysis based on complete instruction reference history to derive the upper bound. Using this model, we analyzed the prefetch accuracy bounds of programs from the SPEC benchmark suite. The results show that the prefetch accuracy bound is very dependent on the presence of instruction cache memory. The results also show that as the size of the instruction cache increases, the theoretically achievable prefetch accuracy decreases drastically. One implication of this result is that when the cache size is over some threshold and the prefetch miss penalty is high, instruction prefetching may degrade the overall performance rather than improve it.

References

[1] Intel Corporation. Microprocessors, 1990.

[2] J. D. Gee and M. D. Hill and A. J. Smith. Cache Performance of the SPEC Benchmark Suite. Technical Report UCB /CSD 91/648, Computer Science Division, University of California, Berkeley, Oct. 1991.

[3] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative approach (second edition)*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[4] W. Hilf and A. Nausch. *The M68000 Family Volume 1.* page 48. Prentice Hall, Englewood Cliffs, NJ, 1989.

[5] W. W. Hwu and P. P. Chang. Achieving High Instruction Cache Performance with an Optimizing Compiler. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 242−251, 1989.

[6] N. P. Jouppi. Improving Directed −Mapped Cache Performance by the Addition of a Small Fully−Associative Cache and Prefetch Buffers. In

Proceedings of the 17th Annual International Symposium on Computer Architecture, pages 364−373, 1990.

[7] S. B. Kim and M. S. Park and S. Park and S. L. Min and H. Shin and C. S. Kim and D. Jeong. Threaded prefetching: An adaptive instruction prefetch mechanism, *Microprocessing and Microprogramming*, 39, 1993.

[8] M. Lee and S. L. Min and C. Y. Park and Y. H. Bae and H. Shin and C. S. Kim. A Dual−mode Instruction Prefetch Scheme for Improved Worst Case and Average Case Program Execution Times. In *Proceedings of the 14th Real−Time Systems Symposium*, pages 98−105, 1993.

[9] S. McFarling. Program Optimization for Instruction Caches. In Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, pages 183−191, 1989.

[10] J. C. Mogul and A. Borg. The Effect of Context Switches on Cache Performance. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75−84, 1991.

[11] K. Pettis and R. C. Hansen. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16−27, 1990.

[12] B. R. Rau and G. E. Rossman. The Effect of Instruction Fetch Strategies upon the Performance of Pipelined Instruction Units. In *Proceedings of the 4th Annual International Symposium on Computer Architecture"*, pages 80−89, 1977.

[13] A. J. Smith. Sequential Program Prefetching in Memory Hierarchies. *IEEE* 1978.

[14] A. J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473−530, Sep. 1982.

[15] M. D. Smith. Tracing with pixie. Technical Report CSL−TR−91−497, Stanford University, Nov. 1991.

[16] SPEC Newsletter, Vol. 1, 1989.