Improving Cache Performance by
Selective Cache Bypass [1]

Chi-Hung Chi

Philips Laboratories
345 Scarborough Road
Briarcliff Manor, NY 10510

Henry Dietz

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

In traditional cache-based computers, all memory references are made through cache. However, a significant number of items which are referenced in a program are referenced so infrequently that other cache traffic is certain to "bump" these items from cache before they are referenced again. In such cases, not only is there no benefit in placing the item in cache, but there is the additional overhead of "bumping" some other item out of cache to make room for this useless cache entry. Where a cache line is larger than a processor word, there is an additional penalty in loading the entire line from memory into cache, whereas the reference could have been satisfied with a single word fetch. Simulations have shown that these effects typically degrade cache-based system performance (average reference time) by 10% to 30%.

This performance loss is due to cache pollution; by simply forcing "polluting" references to directly reference main memory — bypassing the cache — much of this performance can be regained. The technique proposed in this paper involves the use of new hardware, called a **Bypass-Cache**, which, under program control, will determine whether each reference should be through the cache or bypassing the cache and referencing main memory directly. Several inexpensive heuristics for the compiler to determine how to make each reference are given.

1. Introduction

Advances in supercomputing and semiconductor technologies have made it possible to design and build high performance computer systems with many processors. However, the performance of these systems is often limited by memory reference bandwidth. While the execution of each operation has become very fast, the time to fetch each datum from main memory (or from another processor's local memory) is at least an order of magnitude longer than the processor operation time — also an order of magnitude longer than the reference time from on-chip or local memory. Use of a cache seems a natural way to attack this mismatch.

It is widely accepted that cache memory is a cost effective way to improve system performance by using locality properties to improve apparent average memory access time. Significant reductions in the average data/instruction access time have been achieved using very simple cache

placement/replacement policies implemented in hardware [Bel74]. If anything, the success of cache has been too complete; the desirability of caching items is rarely questioned and basic research on cache design generally has been reduced to the level of benchmarking and fine-tuning a few well-known parameters.

For example, since cache reference time is so much less than main memory reference time, it is commonly held that as many data as possible should be placed in cache. One typically measures the efficacy of a cache design by determining the cache hit ratio — the fraction of memory references which are satisfied by cache entries. The problem is simply that it is not always beneficial to fetch a line into the cache on a cache-miss even if the cache is infinitely large — *increasing cache hit ratio sometimes reduces system performance!* Other criteria like memory traffic have occasionally been used instead of cache hit ratio, but these measures are also somewhat imprecise and indirect. If one wants to *minimize total memory reference time*, then that is the obvious measure by which cache performance should be judged. Throughout this paper, cache performance is measured in terms of the effect on total memory reference time.

Why are the more commonly used cache performance criteria inaccurate measures of system performance? There is always an overhead associated with fetching a line from memory into cache. If the benefit gained from having that line in cache is not greater than the overhead that loading the cache line implies, then it is faster to reference the data of that line directly from main memory. This is true even if the cache is infinitely large — but even more dramatically true with smaller caches. If some mechanism can be used to *selectively disable or bypass the cache for those references which cache cannot improve*:

[1] the cost of loading the cache with these lines is saved and

[2] for finite-size caches, more cache space becomes available to other references and the probability of accidentally replacing useful lines (those lines that can help improve system performance) is reduced — there will be less cache pollution.

Simulation results, reported in Section 4, strongly support this view. An *average of 10% to 30% reduction in total*

---

reference time can be achieved simply by using the proposed cache bypass mechanism.

Section 2 of this paper presents a survey of current cache designs and bypass concepts. Section 3 discusses the cache bypass mechanism and how the cache bypass control information can be implemented in practical hardware. Section 4 presents simulation results. Continuing research on the cache bypass mechanism is described in Section 5.

2. Current Cache Designs and Bypass Concepts

Before investigating the mechanism for, and benefits of, selective cache bypass, it is useful to briefly survey existing cache management policies; in part, this highlights where the extra performance comes from, but it also clarifies the constraints these traditional policies impose on the cache bypass mechanism. Examples illustrate why some constraints imposed by previous cache replacement policies often cause a large decrease in system performance, as well as how eliminating some of these constraints can regain much of the lost performance.

This discussion serves the purpose of illustrating the importance of cache bypass and of giving motivation to research this topic. In the last part of this section, we briefly describe the cache bypass mechanism used in the C1 minisupercomputer manufactured by Convex Computer Corporation [Con86]. Although the strategy used for cache bypass in the C1 is very limited, it does demonstrate the importance of incorporating a bypass mechanism.

2.1. Traditional Replacement Policies

Replacement policy is defined as the set of rules by which the choice of which cache line to replace is made when the cache is full and a new line is to be fetched from the main memory into the cache [HwB84]. Replacement policies such as LRU (least recently used), random replacement, FIFO (first-in first-out), etc., are commonly used in current cache designs.

Although each of these traditional cache replacement policies has its own unique technique for placing and/or replacing cache lines, the option of deciding not to put the requested line in cache was not considered. In all conventional cache replacement policies, immediately after each reference, the line referenced is in cache. This implies that whenever there is a cache miss occurred, the missed line needs to be fetched into the cache and this line fetch is independent of whether the fetched line would bring improvement to system performance.

The main argument for this constraint is that since reference time of data in cache is much smaller than that from main memory and with spatial and temporal behavior of program references [Spi77], having the current referenced line in cache has a high probability to bring improvement in system performance. While this argument is generally true, it is possible to predict with good certainty exactly which lines will not contribute to improving performance; without such prediction, it is easy to envision scenarios where the cache would replace lines it should have kept with lines that will never again be referenced. This leads to a worst-case scenario in which a machine runs slower with cache than without it. Bypassing the cache, hence avoiding this pollution, this worst-case scenario is averted.

An example of this problem is easily constructed. Suppose there is a fully-associative cache of size two, line size one, and the memory reference string is *123123*. (It is interesting to note that this example is exactly the kind of reference sequence one would get in executing a typical loop which references more data than there are cache cells — which is well-known to the worst-case for LRU.) With the cost of different types of memory references shown in Table 1 (and the line-style used to represent each), the cache content after each reference with random replacement, LRU, and modified LRU with cache bypass mechanism are shown in Figures 1, 2, and 3.
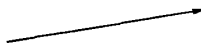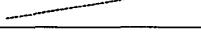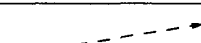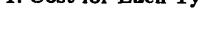
Line Pattern	Cost (Time)	Type of Reference
	none	—
	T_c	Reference from Cache
	T_r	Reference from Main Memory
	$T_c + T_p$	Reference through Cache (with Fetch to Empty Cache Line)
	$T_c + 2(T_p)$	Reference through Cache (with Replacement of a Cache Line)

Table 1: Cost for Each Type of Memory Reference

Cache Policy	Cost	Cost with $T_s = T_r = 10T_c$	$Cost_{Cache-Policy}/Cost_{Optimal}$
Optimal	$2T_p + 2T_r + 4T_c$	$44T_c$	1.000
Random	$7.75T_p + 6T_c$	$83.5T_c$	1.898
LRU	$10T_p + 6T_c$	$106T_c$	2.409

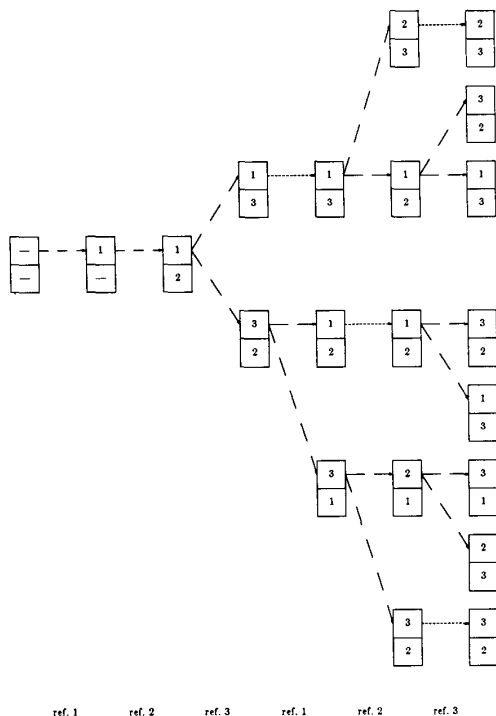**Table 2: Comparsion of Execution Times for *123123***

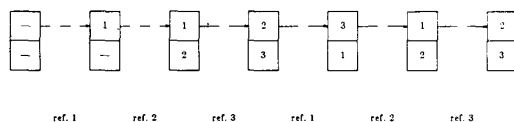**Figure 1: Random Replacement Transactions for** *123123*

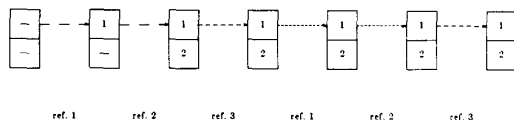**Figure 2: LRU Transactions for** *123123*

Figure 3: Modified LRU with Cache Bypass for *123123*

The total reference costs using these three policies are shown in Table 2. In this table, it can be seen that the ratio of $Cost_{Random}/Cost_{Bypass}$ is 1.898 and the ratio of $Cost_{LRU}/Cost_{Bypass}$ is 2.409.

Notice that while placing data *1* and *2* in cache can improve system performance, placing datum *3* in cache actually decreases the system performance. Unfortunately, if bypass of the cache is not considered, the resulting performance is the worst possible — in fact, it is worse than if no cache were present. With selective cache bypass, one might simply reference datum *3* directly from main memory; yet the cache would speed-up references to data *1* and *2*.

2.2. History of Cache Bypass

Although not commonly accepted as part of traditional cache design, cache bypass is not entirely new.

Nearly all cache-based computers have some provision for disabling the cache so that memory-mapped I/O transactions can take place. However, the idea of enabling/disabling the cache for each memory reference is not well supported by most of these systems (presumably the possibility had not been considered). These systems typically require an entire instruction to be executed to change the cache enable state. Despite this, such systems can be used to implement cache bypass where several consecutive references should be bypassed.

Some machine designers also recognized that the performance of cache could be improved by simultaneously requesting each datum from both main memory and cache. In this scheme, if the item is found in the cache then the cached value is used and the main memory request is cancelled or ignored. If not, the item is returned directly from main memory to the processor, simultaneously initiating a cache update for that datum's line. This technique does improve performance, but may require fairly expensive hardware and does not avert cache pollution — it merely reduces the cost of referencing "through" the cache.

Somewhat closer in spirit to our approach, Convex Computer Corporation has implemented a selective cache bypass mechanism in their C1 minisupercomputer. The strategy employed is [Con86]:

> Upon load or store, the physical control unit either writes the referenced data into its cache or bypasses the cache and accesses main memory directly, leaving the cache unmodified. All aligned 64-bit vector loads and stores result in cache bypass. Loads and stores of aligned, contiguous 32-bit vector elements bypass the cache as well. Since vector accesses dominate supercomputer-class applications software, cache bypass opportunities occur frequently.

Apparently, the cache bypass mechanism is employed only on vector operations because the C1 has a cache with a set size of one, hence, loading a vector register had the effect of totally flushing the cache — obviously negating any benefits of caching. In any case, the Convex scheme is quite reason-

able, and was sufficiently new so as to be patented (patent pending?); the problem is that it equates "vector" with "bypass," and this isn't really correct. Some vectors *should* be cached and some scalars shouldn't be, but on the average the Convex scheme is right often enough to yield a big improvement.

In contrast, the current proposal for cache bypass is to use a compile-time static analysis of the reference behavior of each program to compute a "cache/bypass" tag for each memory reference the compiled code makes. These tags are used at runtime to control a cache enable/disable line.

3. Implementing Cache Bypass

As shown in the example of Section 2 1, LRU referencing of all data through the cache actually performed worse than if no cache were present.

There are two main reasons for this phenomena. First, there is often a large time overhead implied in moving lines of data between cache and main memory. This overhead increases as the cache line size is increased. Consequently, fetching a line into cache can improve system performance *iff* the total number of references to data in that line (before that line is replaced) is such that the savings in referencing cache outweighs the overhead of moving that line between cache and main memory. If not, the total time to make these references will be minimized by ignoring the cache — bypassing to directly reference main memory. Even if the cache is infinitely large, this still holds.

Second, since all real caches are finite, placing one line in cache generally means that some other line cannot be in cache. Hence, placing infrequently referenced lines into cache not only adds a large overhead to total memory access time, but also prevents speed-up that could have been gained if some other (more heavily referenced) line were placed in cache. This effect is what we call "cache pollution."

Since minimizing the total memory access time is our goal in selective cache bypass and the total access time depends on both the architectural design and the implementation technology of the cache and main memory, some details must be supplied. In the remainder of this paper, we have chosen to discuss cache bypass assuming that the supplied information is that of a typical system; this greatly simplifies the following discussion and reduces the number of graphs needed to support the rest of the paper. For example, the simulations and examples presented in this paper are based on the assumption that LRU is the basic cache management technique and that "typical" CMOS or NMOS ICs implement the relevant system components. This implies, for example, that a main memory reference takes about 10 times as long as a cache reference — in reality, this ratio varies from about 2:1 to greater than 50:1. Of course, the use of specific numbers in the examples and discussion is not indicative of the technique requiring those exact numbers: the technique works for most reasonable cache organizations, only the percentage benefit gained varies.

In Section 3.1, a brief discussion of current IC technologies and their impact on memory access time is given. Criteria or rules to determine whether a reference request is going to bypass the cache and to reference directly from main memory are presented in Section 3.2. Section 3.3 gives a very simple and cheap, yet efficient, way to incorporate a cache bypass mechanism with an LRU policy. Practical implementation schemes for cache bypass control signals to be added to existing systems are presented in Section 3.4.

3.1. Integrated Circuit Technologies

Integrated circuit (IC) technology is one of the major parameters in the criteria for cache bypass mechanism (discussed in the next section). Hence, a brief survey of current different (IC) technologies and its impact on off-chip and on-chip memory reference time is necessary. Table 3 gives the on-chip and off-chip memory access times for some of the current integrated circuit technologies [MiF86]. From this table, we see that the ratio of off-chip to on-chip memory access times is at least 10. Using this ratio, an estimate of the minimum reference frequency that a line needs to justify its placement in cache can be obtained.

Type of Access	Silicon CMOS/SOS	Silicon NMOS	GaAS
On-chip memory access	10-20ns	10-20ns	0.5-2.0ns
Off-chip on-package memory access	40-80ns	20-40ns	4-10ns
Off-chip off-package memory access	100-200ns	100-200ns	20-80ns
Ratio of off-chip on-package to on-chip memory access	4	2	5-8
Ratio of off-chip off-package to on-chip memory access	10	10	40

Table 3. Memory Access Time of Different IC Technologies

3.2. Criteria for Cache Bypass Mechanism

Thoughout the current work, the main focus is the reduction of total memory reference time for a program. Hence, criteria proposed here are based on the comparsion between the time overhead involved in having a line in cache and the total reference time saved by referencing data in a line in cache.

The time overhead of placing a line in cache is the transfer time for all data of that line from main memory to cache. If any dirty[2] line is bumped out of cache using a write-back cache, a similar transfer time to uptime the main memory is also included in this overhead. Since the amount of data transfer between main memory and cache is constant for a cache design, this overhead is only architecture design and implementation technology dependent, and is independent of program behavior.

On the other hand, the time savings for placing a line in cache accumulates every time data in that line is referenced. Hence, the savings are, in addition, program dependent.

There are additional factors which can influence the costs and the savings of placing/replacing a line in cache, resulting in slightly different cache bypass decisions of references in a program. For example, if a reference is going to bypass the cache and directly reference main memory, the average probability of bumping a line from cache decreases, and cache space could also be viewed as available to other lines.

These effects are easily recognized and advantageously used in the cache bypass mechanism. In fact, a complete analytical model of the cache bypass mechanism for common cache replacement policies to take all these factors into consideration can easily be derived from the compiler-driven cache (SCP) model [ChD87] [ChD88]. While the SCP model can fully account for cache bypass, and can promise *optimal performance*, the complete SCP model does entail relatively complex analysis and compiler technology; hence, the technique presented here is a sub-optimal, but quite effective and simple, approximation to the SCP model[3].

To define an algorithm for determining when to bypass the cache for a particular reference, some definitions and notations are useful.

overhead(i) = time overhead of placing/replacing line *i* in cache

saving(i) = time saving of having line *i* in cache before it is replaced

n(i) = total number of referencing line *i* in cache before it is replaced

With the cost notations defined in Table 1, the *overhead(i)* and *saving(i)* are as follows:

If no dirty line is bumped out of cache, the overhead is:
$$overhead(i) = T_p$$
If a dirty line is replaced (bumped) from the cache, then the overhead is:
$$overhead(i) = 2 * T_p$$
The savings for having line *i* in cache (before it is replaced) is:
$$saving(i) = n(i) * (T_r - T_c)$$

In order for a reference line *i* to bypass the cache, the overhead *overhead(i)* must be greater or equal to the total time savings *saving(i)*. Only in this case can the placement of line *i* contribute to improve system performance.

3.3. Algorithm for LRU Bypass-Cache

In this section, LRU (least recently used) cache replacement is chosen as the basic scheme and the cache bypass control is added on top of this policy. We have choosen to discuss an LRU Bypass-Cache because the basic LRU policy is probably the most commonly used and most commonly trusted to yield good performance. Hence, the comparsions of simulated performance with/without cache bypass (in Section 4) are very good estimates of the expected improvement derived by converting commonly available computers to use Bypass-Cache instead of traditional cache.

In this section, a fast, simple, efficient (yet sub-optimal) algorithm to determine when a reference should bypass the cache is proposed. The algorithm is based on the concept of a **trace**, as discussed in **trace scheduling** techniques used for automatic parallelizing compilers [Ell85]. The procedure to determine, for each reference in the program, whether to bypass or to reference through the cache is:

1. Perform traditional flow analysis and build the program flow graph. (This step should be considered "free" because any *good* compiler will use this same analysis to aid in generating efficient code.)

---

[2] A line in cache is considered dirty *iff* some protion of the value it contains does not match the value stored in the corresponding main memory line.

[3] In fact, if the SCP model is used with more radically redesigned cache, performance is much better than using a Bypass-Cache and the analysis is essentially the same. Hence, we feel that if one wants to achieve optimal performance, one should be willing to make the more drastic hardware and software changes to support it — here, we have simply given a technique whereby only trivial hardware and software changes result in large, but sub-optimal, performance gains.

2. For each trace (a possible control flow path which has not yet been processed), do the following:

    a. Mark all references in this trace as "cachable" (put in cache).

    b. Scan this trace, keeping track of which items would be resident in cache assuming that all items marked as cachable are always referenced through the cache and that LRU is used to determine which item is bumped from cache when line replacement occurs. As the references are scanned, the time overhead and savings realized for each cachable line are accumulated. As a simple heuristic, the savings for referencing an item within a loop is multiplied by a factor[4] of 10.

    c. At the end of the trace, mark all references which have a larger overhead than savings as "non-cachable".

    d. The above set of markings can be somewhat improved, although not made optimal, by repeating steps 2b and 2c. Such repetition is, however, completely optional. All the simulation results given in this paper used only a single pass.

This algorithm, although very crude and simple, reaps speedups ranging from a few percent to a factor of nearly 100, depending on the cache configuration and the benchmark used. Speedups greater than 2 are not unusual for commonly used cache configurations.

3.4. Implementation of Bypass Control

With the results of compiler analysis of a program (or with statistical results gleaned from previous runs), the bypass/cache question is easily answered with good enough accuracy so as to permit huge performance increases. However, this information must be transmitted to the Bypass-Cache control logic for each reference. The information for each reference requires only a single bit — a 1 means "bypass" and 0 means "go through the cache." The natural question is how does the compiler get this one bit of information for each reference into the Bypass-Cache control at runtime?

There are a number of alternative solutions to this problem and each of these solutions trades off some resources or capabilities.

The conceptually easiest and most efficient way to transmit this cache bypass information is to embed a bit in each instruction for each memory reference the instruction may cause. For new machine design, this is fairly convenient; reserving a control bit to obtain speedups of total

memory access time by factors of 2 or more is virtually always worthwhile. Also, existing machines with at least one currently unused bit in each instruction should probably use this implementation.

Alternatively, the instruction set of the machine can be expanded to include explicit Bypass-Cache control instructions. In fact, these instructions exist for virtually all computers which have cache. An extreme example of this explicit cache control is the IBM 801, where individual cache lines can be explicitly allocated and deallocated; most systems simply permit the cache to be enabled/disabled as a whole. Since bypasses may come in "clumps", even this crude bypass control can gain some improvement; however, bypasses do not always come in clumps. By defining a new instruction specifically to implement Bypass-Cache control, one could permit each cache control instruction to set the pattern of bypass/cache decisions for the next n references, where n is somewhat less than the machine word length. Again, some performance would be gained, but the high frequency of Bypass-Cache control instructions would limit performance.

While all the above schemes have some merit, there is another scheme which both permits a cache control bit to be associated with each instruction and does not require changes in the instruction set design or encoding. In current machine designs, the addressable space is typically very large and programs rarely use the entire addressable space of the machine. Thus, it is possible to trade one address bit (e.g., the most significant bit of an address) for use as the control bit for the Bypass-Cache. In fact, this solution is suggested by Intel in their 80386 programmer's reference manual [Int86] as a way to provide a cache control bit for use in multiprocessor cache coherency control. Worst case, this effectively reduces the addressable space by 50%[5]. Of course, it also causes the compiler writer a bit of grief in that not only must all addresses be correctly tagged, but the compiler must also be careful about operations such as pointer arithmetic or comparisons.

Other methods, such as using a separate cache controller to explicitly control the cache (similar to the remote PC idea [Rad83]) are also possible. However, the overhead and the synchronization cost involved may be too large to be practical.

---

[4] This is a rough approximation to weighting each reference in the trace by its expected number of executions — it assumes each loop executes an average of 10 times. If the compiler has a better estimate, this can be used instead. Techniques for the compiler to make more intelligent estimates of expected execution frequencies are discussed in [Die87].

[5] The actual address space may not be affected because address mapping mechanisms may be able to circumvent the loss.

4. Simulation Results

To measure the effect of cache bypass in reducing total reference time, detailed simulation of the LRU Bypass-Cache was performed using the single-pass compiler algorithm described above. For comparison, the same simulations were performed using a conventional LRU cache with the same configuration as the Bypass-Cache.

The benchmark programs were taken from the DARPA MIPS package, and are widely used as benchmarks of cache and/or system performance. Data are given for four of these programs:

Bubble
A typical bubble sort program, executed on a set of 500 random data.
Puzzle
This is a compute-bound program from Forest Basket, run with a size of 511.
Realmm
A program which performs a matrix multiplication of two real matricies, each of which is 40 by 40.
Tower
The standard recursive tower-of-Hanoi solution, given the problem of moving 18 disks.

Each of the programs was simulated for about 500,000 references of execution, hence "cold start" cache effects are negligible.

Since our primary concern is minimizing the total reference time, rather than maximizing hit ratio, it was also necessary to assume specific ratios of reference times for each of the different types of reference. The cost functions used for the data in this paper were based on cost estimates for a typical CMOS-based system:

- Cost of referencing data from cache is 1 time unit.
- Cost of referencing data from main memory is 10 time units.
- Cost of placing a line in an empty or non-dirty cache entry is $10 + (line\_size - 1) * 7$ time units.

The fact that fetching/storing n consecutive data into/from cache in one request takes less time than fetching/storing n data in n requests is reflected in the above costs. We were actually quite generous in this assumption, using a formula giving a 30% benefit for multi-word fetch/store; however, this simply has the effect of making the benefit due to Bypass-Cache appear smaller.

To make the simulations as complete as possible, all possible power-of-2 cache organizations (e.g. different line sizes, set sizes) for a fixed cache size of 128 words[6] were simulated and are presented in this paper. The absolute reference times for the different benchmarks naturally differ, however, the speedups and curve shapes are fairly consistent across all the simulations.

Figures 4 through 7 graph speedup of total memory reference times with Bypass-Cache as compared to the same configuration conventional cache. Each curve in the graphs is marked with the power-of-2 which was used as the associative set size. These graphs clearly demonstrate that the speedup in total memory reference time using Bypass-Cache is very large — in fact, it is plotted on a log scale, and averages about 2.

The speedup with Bypass-Cache is usually smallest for a line size of one or two. With an increase in line size (leaving cache size and set size fixed), the speedup with Bypass-Cache increases greatly. This agress with confirms the argument given in Section 3. This is because a larger line size implies a larger overhead in cache line placement and replacement. Although the total number of references of a line with increasing line size increases, this increase is much less than the increase in overhead. Consequently, cache more easily becomes polluted, and the Bypass-Cache becomes more critical in improving system performance.

These curves also show that the speedup with Bypass-Cache is usually smaller for cache with small set size (fixed cache size and line size). Although the cause of this is not yet known, we suspect that this is related to the increase in traffic seen by each cache set (becuase there are fewer sets). Even though the speedup is much smaller in these cases, it is still typically about 1.2 (i.e., 20 percent).

Figure 8 shows the total reference time for the Tower benchmark. The dotted lines indicate the times taken using conventional cache, whereas the solid lines show the times taken with Bypass-Cache.

Aside from the obvious benefit in using Bypass-Cache, this graph suggests an interesting general cache design rule. **If the total memory reference time is to be minimized, rather than the hit-ratio maximized, it is usually better to choose small line size and small set size.** This makes perfect sense in that although large line sizes increase hit-ratio, they imply overhead increases which are greater than the hit-ratio increases — in fact, expotentially greater. That increasing set size is not beneficial is less intuitive, but probably is related to the increased traffic per set and use of a poor replacement algorithm (i.e., one can do a whole lot better than LRU [ChD87]).

For Bypass-Cache, the difference in total memory access time for different line sizes (with same cache size and size) is not as great as those for cache without bypass. This is true because a lot of cache pollution can be avoided with Bypass-Cache.

---

[6] About 500 simulations were performed, encompassing a wide variety of cache sizes and configurations. However, all the simulation results obtained were very consistent, hence we have chosen to present only the data for the largest cache size we examined — 128 words. Other simulation data are avalable upon request.

5. Conclusion

In this paper, we present a new cache design — Bypass-Cache — which is able to avert polluting the cache by bypassing the cache for entries for which caching would not result in faster total execution time. From our simulation results, we see that the speedup is tremendous, with an average of about 2. Various methods for implementing the Bypass-Cache architecture are presented as well as an outline of the compiler technology required for its effective use.

Perhaps the most significant result, however, is that **cache hit ratio is not necessary related to the total reference time**. This will be discussed more deeply in a later paper.

References

[AlB86]   Allen, R., Baumgartner, D., Kennedy, K., Porterfield, A., "PTOOL: A Semi-Automatic Parallel Programming Assistant," *1986 International Conference on Parallel Processing*, August 1986, pp. 164-170.

[Bel74]   Belady, L.A., Palermo, F.P., "On-line Measurement of Paging Behavior by the Multi-valued MIN Algorithm," *IBM Research and Development*, 18, 1, January, 1974, pp. 2-19.

[BuC86]   Burke, M., Cytron, R., "Interprocedural Dependence Analysis and Parallelization," *SIGPLAN Symposium on Compiler Construction*, 1986, pp. 613-641.

[Con86]   "C1 Processor Series: Architecture," Convex Computer Corporation, 1986.

[ChD87]   Chi, C.H., Dietz, H., "Compiler-Driven Cache Policy," Technical Report EE-87-21, Purdue University, May, 1987.

[ChD88]   Chi, C.H., Dietz, H., "Register Allocation for GaAs Computer Systems," *Proceedings of the 1988 Hawaii International Conference on Systems Sciences*, January 1988, pp. 266-274.

[Die87]   Dietz, H. G., *The Refined-Language Approach To Compiling For Parallel Supercomputers*, Ph.D. Dissertation, Polytechnic University, June 1987.

[Ell85]   Ellis, J. R., *Bulldog: A Compiler for VLIW Architectures*, 1985 ACM Doctoral Dissertation Award, MIT Press, 1986.

[HwB84]   Hwang, K., Briggs, F.A., *Computer Architecture and Parallel Processing*, McGraw Hill Book Company, 1984.

[Int86]   Intel Corporation, *80386 programmer's reference manual*, 1986, pp. 11-6.

[Rad83]   Radin, G., "The 801 Minicomputer," IBM Journal of Research and Development, May 1983, pp. 237-246.

[Smi82]   Smith, A.J., "Cache Memories," *Computing Surveys*, Vol. 14, No. 3, September, 1982, pp. 473-530.

[Spi77]   Spirn, J., *Program Behavior: Models and Measurements*, Elsevier-North Holland, N.Y., 1977.
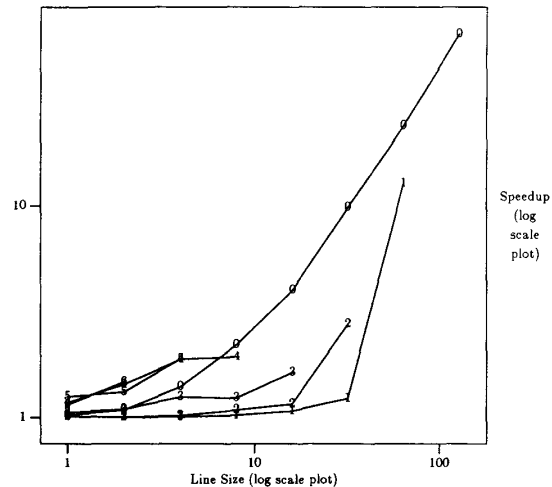
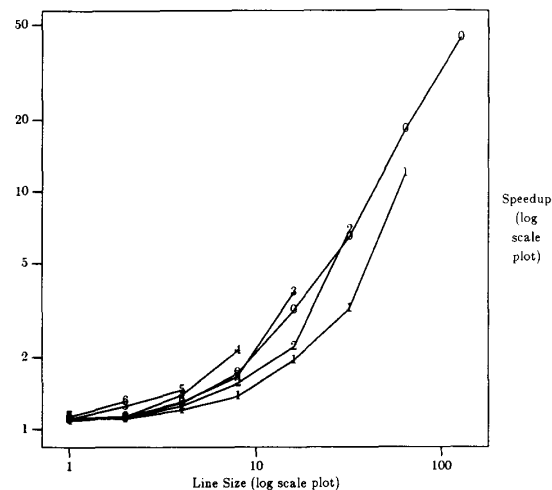**Figure 4:** Speedup in Total Reference Time for Bubble

Figure 5: Speedup in Total Reference Time for Puzzle
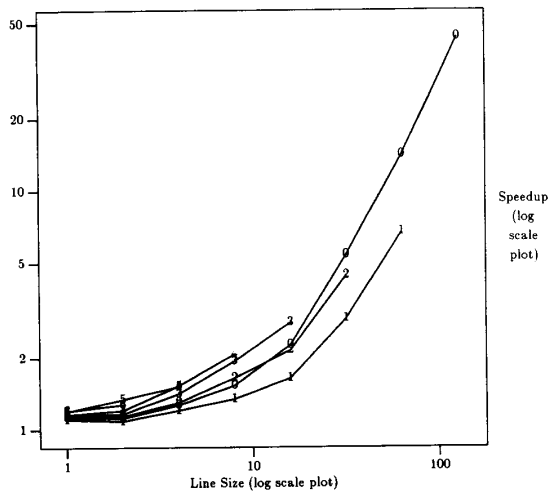
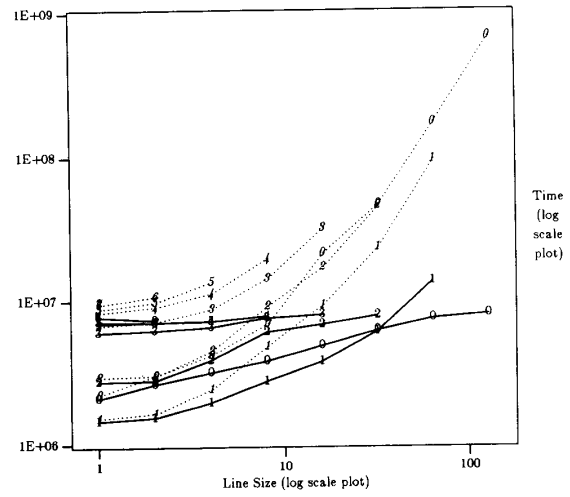**Figure 6:** Speedup in Total Reference Time for Realmm

Figure 8:
Total Reference Time WITH/*WITHOUT* Bypass for Tower
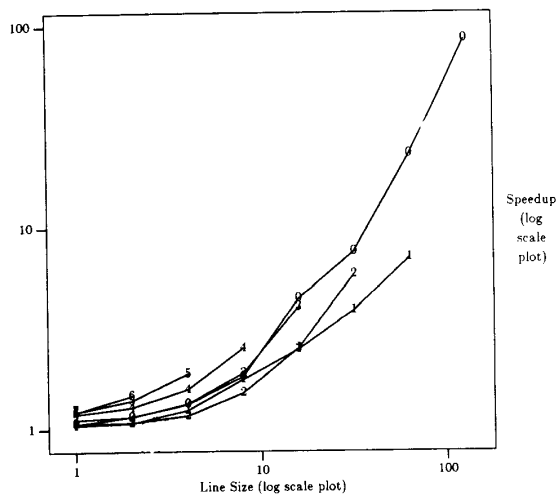(WITH is solid lines, *WITHOUT* is dotted lines)

Figure 7: Speedup in Total Reference Time for Tower