6-1-1987

Compiler-Driven Cache Policy (Known Reference String)

Chi Hung Chi
Purdue University

Henry G. Dietz
Purdue University

FILE COPY

Compiler-Driven Cache Policy

(Known Reference String)

Chi-Hung Chi
Henry G. Dietz

TR-EE 87-21
June 1987

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Compiler-Driven Cache Policy

(Known Reference String)

Chi-Hung Chi and Henry G. Dietz

Department of Electrical Engineering
Purdue University
June 12, 1987

ABSTRACT

Increasing cache hit-ratios has proved to be instrumental in improving performance of cache-based computers. This is particularly true for computers which have a high cache-miss/cache-hit memory reference delay ratio. Although software policies are often used for main vs. secondary memory "caching", the speed required for an implementation of a CPU vs. main memory cache policy has prompted only investigation of policies which can be implemented directly in hardware. Based on compile-time analysis, it is possible to predict program behavior, thereby increasing the hit-ratio beyond the capability of pure run-time (hardware) techniques. In this report, compiler-driven techniques for this kind of cache policy are described. The SCP Model (software cache policy model) provides an optimal cache prefetch and placement/replacement policy when given an arbitrary memory reference string. In addition to suggesting a simplified cache hardware model, the SCP Model can be applied to various cache organizations such as direct mapping, set associative, and full associative. Analytic results demonstrate significant improvements in cache performance.

The current work discusses an optimal cache policy which applies where the string of references is known at compile time. However, this constraint can be relaxed to encompass reference strings which are known only statistically, i.e., reference strings in which data aliases make the target of some references ambiguous. Companion reports, currently in preparation, detail the extension of the SCP Model to incorporate aliases, code incorporating loops, and conditional branches.

This page is intentionally blank.

Introduction

Cache memory is a high speed buffer memory between the CPU (central processing unit) or PE (multiprocessor processing element) and the main memory. Its purpose is to obtain high speed data/instruction access without the associated cost of building primary memory entirely using expensive high speed technology. It is widely accepted that cache memory is a cost effective way to improve system performance. Significant reductions in the average data/instruction access time have been achieved using very simple cache placement/replacement policies implemented in hardware [Bel74].

However, because of the limited area for the on-chip cache in VLSI, high ratio of on-chip/off-chip reference delay, and the increasing demand for faster and larger memory, simple hardware cache placement/replacement policies are no longer sufficient to bridge the increasing "memory reference delay gap" between the processor and the main memory.

1. Conventional Cache Policy

Common cache replacement policies[1] like LRU, FIFO, LIFO, random replacement etc., make use of either the history of memory references or probabilistic models to determine what should be kept in cache. In the former cases, heavy time and/or space localities of data/instruction reference are assumed within a program, whereas in the last case the assumption is that a policy of randomly chosing cache entries to replace will achieve good average performance. Each policy achieves good performance under certain program behavior [CoD73]. However, in real programs, the referencing behavior tends to change from one region of code to the next. Different localities of time and space result, and it is very difficult for traditional cache replacement policies to adjust to these changes.

Dynamic switching of these (hardware) policies at run-time is physically and practically impossible [Bab82]. Due to cold start costs (incurred with each policy change) and the increased hardware complexity implicit in implementing all these policies, "hybrid hardware-implemented policies" are not feasible. An even more severe difficulty is that even if "hybrid hardware-implemented policies" were feasible, it is extremely difficult to determine which policy the system should employ at which time. Using only the history of memory references or a probabilistic model, there is not enough information available about the program's future behavior.

1. Conventional replacement policies are discussed in detail in chapter 2.

2. Prefetching

Typically, one considers a cache as a buffer which is filled with useful data upon demand — an entry is made when a cell of memory is referenced and its value is not currently in the cache. Alternatively, prefetching is a technique in which cache entries are made before the program has demanded their values. This tends to beneficially decrease the cache-miss ratio, yet it increases the amount of cache space occupied, and can actually "bump" useful entries from the cache if the cache is of fixed size, thereby increasing the cache-miss ratio.

Most prefetch policies were initially designed for managing page tables in virtual memory systems. However, the similarities between paging and caching are very strong, hence many ideas from paging can be applied to caching.

In 1970, Joseph presented an analysis of program simulations running in a paged memory system [Jos70]. He explored two methods of prefetching: one page lookahead (OPLA) and simple prediction (SP) in a working set environment. In the OPLA, page R and $R+1$ are fetched from the secondary memory to the main memory when there is a demand of page R and it is not in main memory. Page R is fetched into the main memory while the prefetched page $R+1$ is loaded into a buffer (which is temporarily locked against references until the transfer is complete). In the SP, failed predictions are left in the buffer but are not overwritten. Under this situation, there might be more than one prefetched page in the buffer. His results showed that the number of page faults could be reduced by 50% to 70%.

Baer and Sager [BaS76] continued the work of Joseph and explored three methods of prefetching under a least recently used (LRU) page replacement policy. They classified locality into two types: temporal locality (locality of time) and spatial locality (locality of space). Their objective was to prefetch pages which share some type of locality. The first algorithm they proposed was a variant of OPLA. The other two, temporal lookahead, and spatial lookahead, were based on the non-sequential access of memory. What they tried to do was to have the prefetching algorithm adapt to the changes in either type of locality. Of these three prefetching algorithms, spatial lookahead performed best and about one half of the prefetched pages were referenced.

Smith studied prefetching in the context of data base systems [Smi78a] and for general-purpose paged virtual memory systems [Smi78b]. For data base systems with sequential accessing characteristics, he reported that "run length" (the number of consecutive block references with intermediate re-references deleted) was likely to be the most useful predictor for optimizing a data prefetching strategy. For general purpose paged virtual memory systems, he reported that for small page sizes (from 32 to 256), prefetching using OPLA was effective and for large page sizes (from 1024 to 4096 bytes), prefetching under the same conditions (LRU using OPLA) was likely to decrease performance. This decrease is due to the fact that only a small fraction of the prefetched pages are referenced. Thus, if the page size is large, the overhead associated with prefetching useless pages may be larger than the gain in cache-hit ratio.

These pioneering works on page prefetching in virtual memory systems provide many insights into cache prefetching. However, these page-oriented algorithms cannot be directly applied to cache management — mainly due to circuit complexity constraints. In addition, since these algorithms do not embody knowledge of future references, cache pollution is a serious problem: information may be prefetched into the cache even though it never will be referenced. As a result, the cache-miss ratio might be increased to the point where prefetching actually decreases the cache performance. Unless cache pollution is *effectively* solved, prefetching is not of great use. This is why most current systems with cache memory use a strictly demand-driven policy (implemented in hardware) and require all data/instructions to be accessed through the cache.

3. Conclusions

Recent advances in compiler flow-analysis techniques [AlB86] [BuC86] [Die87] make global control/data flow analysis of programs practical. Hence, it is now possible to improve cache performance using predictions of program behavior based on global control/data flow of programs. This technology provides the ability to obtain high-probability reference strings at compile time by simply looking ahead in the program's flow structure. Given a reference string at compile time, both demand-fetch and prefetch cache policies can be "fine-tuned" to the actual references which the program will make.

Compiler-driven prefetch policies can be implemented either by inserting explicit cache prefetch instructions or by tagging references within each instruction. In the case where the reference string is completely known, cache pollution will be reduced to minimum — no pollution whatsoever. Hence, prefetching can only improve cache performance. The implementation of compiler-driven prefetching is discussed in section 2.2.

In the other sections of chapter 2, a few common cache placement/replacement policies are surveyed. Performance of each of these policies is analyzed; some performance limitations are discussed. Chapter 3 outlines the proposed SCP Model, which is based on global control/data flow analysis of programs. Analysis of the SCP Model shows that it provides outstanding cache performance, perhaps halving the cost of cache-misses that would occur in processing a typical trace using a traditional hardware-implemented cache policy. In fact, in the case where the reference string is precisely known at compile time, the SCP Model always achieves the minimum cost cache operation. Applications of the SCP Model as a supplement to different cache hardware organizations is discussed in Chapter 4. In Chapter 5, conclusions and future directions for software cache management are discussed.

Throughout the cache performance analysis in the next few chapters, the memory reference string is assumed to be known precisely. Architectural cache parameters (e.g. cache size) are also assumed to be fixed. The goal is to get the best performance from a fixed-hardware cache.

Hardware Cache Policy

In this section, a survey of some common hardware-implemented cache policies is given. Placement/replacement policies such as least recently used (LRU) and random replacement, prefetching or demand-fetching, and write policies are discussed in detail. Limitations and upper bound performance of these policies are also analyzed — this analysis is extremely important because it allows us to know the limitations of these policies and provides hints as to where and how to improve the cache policies. Actually, as discussed in section 3, the circumstances which cause the poorest upper bound performance are exactly the situations where software cache policies are most effective in improving cache performance.

1. Hardware-Implemented Replacement Policies

Replacement policy is defined as the set of rules by which the choice of cache line to be replaced is made when the cache is full and a new line is to be fetched from the main memory into the cache. Hardware-implemented replacement policies such as LRU (least recently used), random replacement, FIFO (first-in first-out) etc. are commonly used in current cache designs. These polices can be classified as implementing one of two general models: a history-based replacement model or a probabilistic replacement model. For the history-based model, LRU will be used as an example; random replacement will be used as an example of the probabilistic replacement model.

1.1. Least Recently Used (LRU) Policy

The least recently-used (LRU) policy for cache replacement chooses for replacement that line in cache which has not been referenced for the longest period of time [Spi76]. The LRU stack, N_t, is a list of all cache lines referenced by a program in order of recency of usage. Let

$$N_t = [x_1, x_2, ..., x_n]$$

where $x_1, ..., x_n$ are all the cache line references of a program. Under LRU caching with k lines of cache, the cache content at time t will be

$$S_t = \{x_1, x_2, ..., x_k\}$$

which is the first k lines of the LRU stack. This stack therefore implies the contents under LRU of any cache size; it is a summary of the behavior of the program under any LRU-based policy.

As execution progresses, the LRU stack is updated with each cache line reference. Since the stack is simply a list of all cache lines, the update procedure is simple[2]. Suppose reference $r_{t+1} = x_i$, then the complete LRU stack is:

$$N_{t+1} = [x_i,\ x_1,\ \ldots,\ x_{i-1},\ x_{i+1},\ \ldots,\ x_n]$$

Using the notation d_t to represent the stack distance (how deeply buried the reference is on the LRU stack) of the reference at time t, we have

$$\text{if } r_{t+1} = x_i \text{ then } d_{t+1} = i$$

In the simplest distance string model for LRU [Spi76], each distance is assigned a probability called the distance probability:

$$\Pr[d_t = i] = a_i, \qquad \text{for } 1 \le i \le n$$

Locality of reference suggests that the distance probabilities should be generally *decreasing*. The cache lines referenced most recently are those with small distances (near the top of the stack) — the hope is that these cache lines will have the highest probability of reference. To guarantee that the cache lines in the locality sets of every size will be favored over non-locality cache lines, the distance probabilities must be nonincreasing:

$$a_1 \ge a_2 \ge \ldots \ge a_n$$

and it is found empirically [Spi77] that the following relationship is an adequate approximation:

$$A_i = a_1 + a_2 + \ldots + a_i = 1 - ck^{-k}, \qquad \text{for } 1 \le i \le n$$

for parameters c and k with $1 \le k \le 3$ where c is some constant.

Although it is practical for many purposes, this model does not accurately predict all aspects of realistic program behavior. For example, suppose an LRU stack model program executes under LRU caching in a cache of k cache lines — i.e., the first k cache lines in the stack will always be in cache. At each reference, a cache line miss will occur with probability:

$$f_{k,t} = \Pr[r_t \notin S_{t-1}] = a_{k+1} + a_{k+2} + \ldots + a_n = 1 - A_k$$

2. In this paper, we discuss pure LRU. In fact, it is far more common that an approximation to LRU is implemented using a one-bit time stamp [PeS85]. It is unlikely that such an approximation to LRU would perform as well as LRU, and very unlikely that it would perform better.

Under this analytic model, the cache miss probability is constant, and the time τ between any two successive misses is geometrically distributed:

$$\Pr[\tau = q] = A_k^{q-1}(1\text{-}A_k), \qquad q = 1,2,\ldots$$

Note in particular that no matter how many (or how few) cache misses have recently occurred, the statistical properties of the time for the next cache miss are unaffected — the distribution of cache misses is not history sensitive.

Even worse performance is found when the memory reference patterns of real programs are analyzed, since the reference patterns tend to change from one region of code to the next. Locality sets change in time. Frequently, this change is gradual, cache line by cache line; occasionally, the locality set of a real program is completely disrupted as the program begins a new phase of execution. For example, suppose a program makes a pass through successive elements of a large, multi-cache line array. When the first array element in a given cache line is referenced, the cache line enters the locality set and remains there until the last element is referenced. This process continues for each cache line of the array. In this way, the locality set changes slowly in time. However, after the pass is completed, the program may begin an entirely new function, using a new locality set which may overlap little, if at all, with the old. Such phase transactions naturally induce clusters of cache misses, since an entirely new locality set must be acquired on demand but the locality sets in LRU stack model change by only a single page at a time. LRU thus models behavior only within a single phase of execution.

As an example, suppose there is a cache of size two and the memory reference string is `123123`. 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 the LRU policy is shown in Figure 1.

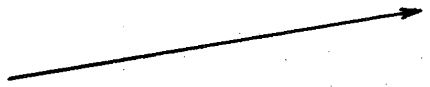
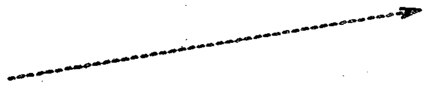
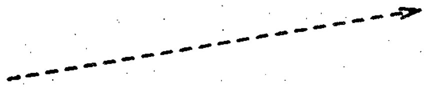
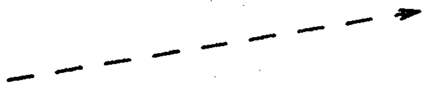
Line Pattern	Cost (Time)	Type of Reference
	*none*	—
	T_c	Reference from Cache
	T_r	Reference from Main Memory
	$T_c + T_p$	Reference through Cache (with Fetch to Empty Cache Line)
	$T_c + 2(T_p)$	Reference through Cache (with Replacement of a Cache Line)

Table 1: Cost for Each Type of Memory Reference

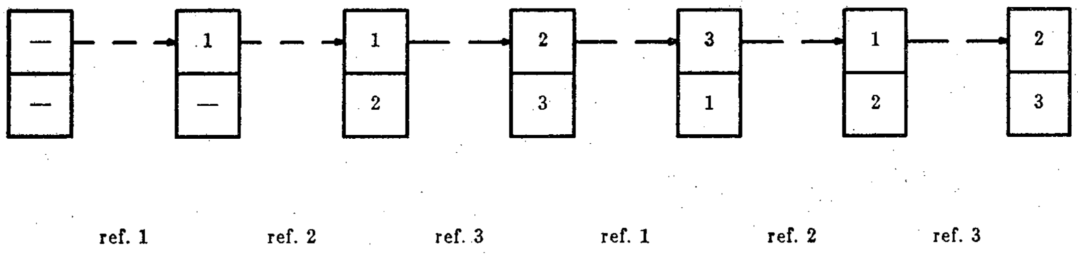


Figure 1: LRU Transactions for 1 2 3 1 2 3

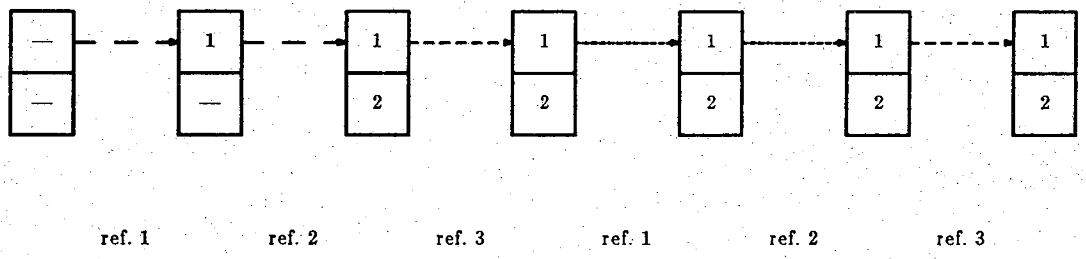


Figure 2: Optimal Transactions for 1 2 3 1 2 3

Cost for referencing 1 2 3 1 2 3 with LRU policy:

$$\begin{aligned} Cost_{LRU} &= 10T_p + 6T_c \\ &= 106T_c \qquad \text{if } T_p = T_r = 10T_c \end{aligned}$$

Compared with the "optimal cache policy" shown in Figure 2:

$$\begin{aligned} Cost_{Optimal} &= 2T_p + 2T_r + 4T_c \\ &= 44T_c \qquad \text{if } T_p = T_r = 10T_c \end{aligned}$$

We found that the ratio of $Cost_{LRU} / Cost_{Optimal}$ is 2.409. This is quite a large ratio and this is within a single locality set.

In the above discussion, we have shown that the main reason for the LRU's poorest upper bound performance is the lack of knowledge of what is going to happen next and lack of the ability to rapidly adjust to a change of memory reference pattern. This is because the history of execution cannot predict a sudden change of locality sets — even increasing cache size does not help. No matter how we improve the LRU, optimal cache performance cannot be obtained under all situations.

1.2. Random Replacement

In the random replacement policy, the fundamental assumption is that references occur at random, i.e., evenly distributed over the range of all program lines [Bel66]. Under this assumption, historical information is irrelevant, and the use of any specific replacement rule does not ensure any relative advantage. Therefore, we might as well choose a simple, random replacement scheme in building the probabilistic model. This scheme chooses the cache line to be replaced at random over the range of all lines in cache.

To determine the performance of this policy, it suffices to compute the probability of a wrong decision being made under this policy. Let n be the number of cache lines in the program. Then the probability of hitting a particular cache line at any memory reference time is $1/n$. Let k be the number of lines in cache. Then the probability of referencing a line in cache is k/n, and the probability of a replacement is $(n - k)/n$. A reference to a line already in cache can be considered a repetition because at least one previous reference must have occurred (when the line was stored to main memory). From the above expressions, we can deduce that the ratio of repetitions to replacements is $k/(n - k)$.

Reusing the previous example (referencing 1 2 3 1 2 3), the cache content after each reference using the random replacement policy is shown in Figure 3.

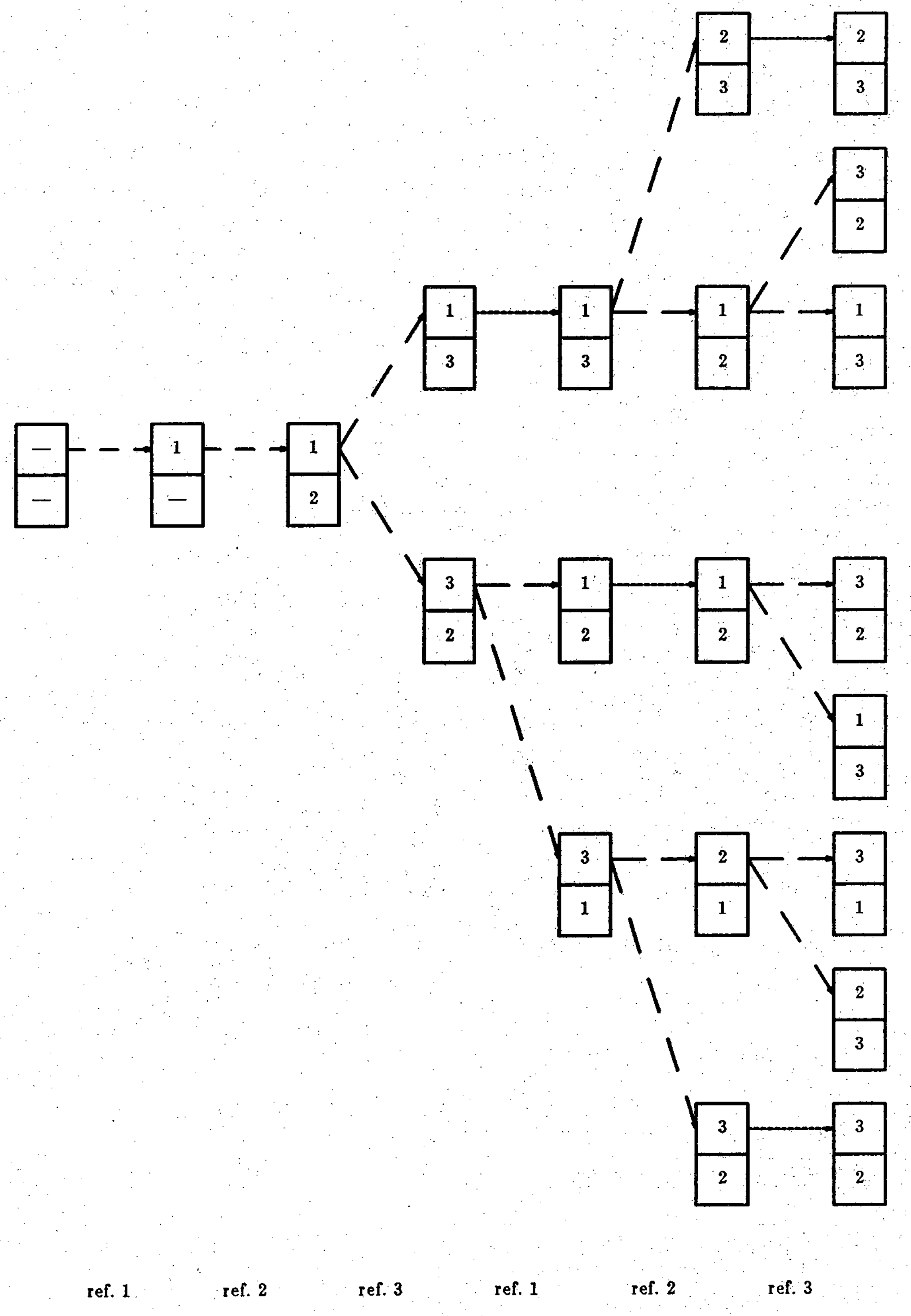


Figure 3: Random Replacement Transactions for 1 2 3 1 2 3

Cost associated with the memory reference string 1 2 3 1 2 3 using random replacement policy:

$$\begin{aligned} Cost_{Random} &= 7.75T_p + 6T_c \\ &= 83.5T_c \qquad \text{if } T_p = T_r = 10T_c \end{aligned}$$

Compared with the "optimal cache policy" shown in Figure 2:

$$\begin{aligned} Cost_{Optimal} &= 2T_p + 2T_r + 4T_c \\ &= 44T_c \qquad \text{if } T_p = T_r = 10T_c \end{aligned}$$

We found that the ratio of $Cost_{Random} / Cost_{Optimal}$ is 1.898. Although this performance is better than the LRU performance for this particular reference string, the performance is still very poor compared to our optimal scheme.

Cache Policy	Cost	Cost with $T_p = T_r = 10T_c$	$Cost_{Cache-Policy} / Cost_{Optimal}$
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 1 2 3 1 2 3

Although the random replacement policy is sometimes optimal, as demonstrated above, it is very unlikely that it will be optimal for a given reference string. This is due to the fact that no locality, nor prediction, of any kind is taken into consideration. Just as random replacement is unlikely to provide very good performance, it is unlikely to provide very bad performance (relative to LRU [SmG85]). Given this, it is surprizing that random replacement is so much less commonly used than LRU-based schemes; perhaps this is because "randomness" carries bad connotations or because it is more difficult to implement an approximation to random replacement than it is to implement an approximation to LRU?

Table 2 shows the cost of referencing 1 2 3 1 2 3 for each cache replacement policy relative to the optimal replacement policy. Although the random replacement policy performs better than LRU, the ratio of $Cost_{Random} / Cost_{Optimal}$ is still very large. Something better is needed. This is our motivation for a software cache policy based on using complex compiler technology and, coincidentally, a greatly simplified hardware design.

2. Demand-Fetching vs. Prefetching

Fetching policy is the mechanism which decides which data to move from main memory to the cache. Fetching policies can be classified as either demand-fetching or prefetching.

Demand-fetching is the policy in which cache lines are brought into the cache only as they are demanded by the processor and found to be absent from the cache. Therefore, the processor has to wait until requested data/instructions arrive from main memory into the cache and from the cache into the processor. In high speed computer systems, this may be a great performance bottleneck.

In prefetching, references may be brought into the cache before they are actually needed. Memory cycles that would otherwise be idle are used to copy data into the cache. There are two appoaches to prefetching: *Static Prefetching* (which is done at compiled time), and *Dynamic Prefetching* (which is done at run time).

Prefetching has great potential to improve cache performance. The key difficulty is deciding what to prefetch and when. For dynamic prefetching, the usual prefetching policy is to prefetch cache line $i+1$ when cache line i is referenced and not in cache, i.e. one line lookahead. However, this causes a serious problem (especically where cache size is small) in that severe cache pollution often results. Information may be prefetched into the cache, replacing some cache line(s) that are referenced later with prefetched information which is never referenced. As a result, cache-misses might actually increase using prefetching.

For static prefetching, prefetching can be made "smart" — so that all information prefetched eventually will be used. Cache pollution is minimized (but not eliminated) with this type of prefetching. This improvement may be achieved without a significant increase in complexity of the cache hardware. Further, since the prefetching operations are scheduled into times when no memory-to-cache traffic is anticipated, there is not likely to be any interference with normal fetching. However, a new compiler technology is needed to implement this "smart" prefetching.

If optimum cache performance is desired, the best approach is to use static prefetching driven by the compiler. Of course, the penalty is that the compiler will need to perform more complex, hence more time consuming, analysis.

3. Write Policies

A write policy is the set of rules whereby it is determined whether a datum being stored should be placed in cache or directly into memory and, if placed in cache, when and how the main memory cell should be updated. Since conventional wisdom marks instructions as read-only (typically, self modifying code is not written), the write policy applies only to stores of data.

If stores do pass through the cache, there are at least two basic strategies for managing them: write through the cache to main memory or copy back data from the cache to main memory only when the cache slot must be re-used. Write through transmits modified data immediately to main memory; thus, all write instructions result in data being transmitted to main memory. Copy back transmits the entire modified line to main memory when a miss occurs and that line is selected for replacement. Since it is only necessary to copy back a cache entry whose value has been changed (so that it no longer matches the value in the location backing it in main memory), a "dirty bit" is often used to mark such cache lines.

For write through, the cache and backing main memory are always **consistent** — corresponding locations always hold the same values. In a multiprocessing environment, where the cache is used in shared memory systems, write through is a simple way of insuring that the numerous caches are consistent with main memory and hence with each other. Also, its implementation is simple, merely forcing a write to main memory for every store instruction. However, it also has several disadvantages.

For example, in a multiprocessor (multi-cache) system, if the write through is accomplished without blocking the processor pending completion of the write into memory, it is possible that the processor would signal another processor to read the value from memory, causing that read request to reach the memory before the write has completed (since it may take a different path through the interconnection network). The alternative, which is blocking the processor until each write has completed, greatly impedes performance in general. In most implementations using write back, longer delay is experienced when a cache miss occurs, since the value originally in the cache must be written back to main memory before it can be replaced by the value just referenced. Also, extra logic is needed to implement "dirty bits." Write back, however, may give a lower cache miss ratio than is achieved using write through [Sim82].

It has been shown that each of these two memory update policies can have better performance than the other under certain conditions [Sim82]. The preferred approach depends on the application program, as well as on the architectural design of the cache. Ideally, a system would incorporate both memory update policies (without excessive overhead) and would optimally chose the update policy to be used for each write in the program. However, just as the replacement policies, it is not possible to implement hardware which can make these choices. This is due to the lack of global knowledge of what is going to be used next.

On the other hand, software, using global information about data/control flow obtained by compiler flow analysis, can make such choices. The SCP Model permits reasonable write policy choices to be made.

4. Conclusions

In previous sections, we discussed a number of common hardware-implemented cache policies founded on either the history of execution or a probabilistic model. Each of these policies is tuned to a reference pattern obtained by a "guess" using no knowledge of the program structure; hence, whenever the data/instruction reference pattern of a real program being executed happens to approximate the reference pattern from which the cache policy was derived, good performance is achieved. For example, if a program is in the middle of a region of code and strong localities of time (temporal locality) and/or space (spatial locality) are present, then cache policies based on a historical model may have good performance. However, as the program passes from that region of code to the next, the same cache policy may evidence the worst possible performance. As is shown in Table 2, because the pattern is obtained independent of knowledge of program structure, the performance of traditional hardware-implemented cache policies is typically far from optimal. A similar situation occurs relevant to fetch policy and write policy.

In the fetch policy, it appears that "smarter" prefetching can can increase the cache hit ratio a lot. The main complication is that the impact of cache pollution must be taken into consideration and, even without prefetching, this problem may make it profitable to reference directly from memory as though there were no cache (thereby avoiding pollution of the cache). Since pollution is caused by single-event peturbations in the referencing structure, no history-based model (e.g., OPLA) is effective: when the event becomes history, it has already polluted the cache.

In write policy, the choice of write through or write back cannot generally be decided in favor of one or the other: there are situations in which either is better than the other. To obtain optimal performance, one needs a software-driven technique for chosing the best write policy for each write operation in the program.

The sources of major performance improvement are better handling of the facts that: different regions of code have different locality sets which have little (if any) overlap, branches and subroutine calls also skew localities in a certain way, and different applications have different kinds of locality (spatial versus temporal). These cannot be approached as hardware design problems, since, as discussed above, hardware techniques are inordinately expensive per unit performance improvement: hybrid hardware cache policies (e.g., [Bab82]) are expensive to implement, but their performance in fundamentally limited by the total lack of knowledge about future program behavior. Global information about data/control flow should be incorporated into cache policies.

A relatively minor additional point is that the cost variations for different types of memory references cannot easily be incorporated into the hardware-implemented schemes. For example, in parallel processing systems, referencing from different memory locations may imply different costs, since memory may be partly local and partly global (within a single address space). The difference between referencing a variable stored in global memory and a variable stored in local memory may be a factor 10 or more — any reasonable cache policy must incorporate understanding of these weights.

The flexibility and power of a software-implemented policy, as well as the ability to obtain and use global information about program behavior, make a software policy far more promising. Hence, we propose to migrate hardware-implemented cache policies into software and to use the compiler to improve, and in some cases make optimal, the runtime performance of an architecturally very simple cache.

Software Cache Policy

In Chapter 2, it was suggested that all common hardware cache policies are based on either historical or probabilistic models. Hence, each hardware policy will perform better than the others under certain memory instruction/data reference patterns: no purely hardware "fix" can be made to improve performance because it is not feasible to put all these hardware policies together and to dynamically change policy as the memory referencing pattern changes.

A natural alternative is to improve performance by *modifying the structure of programs*, at compile time, to match the ideal reference patterns for the hardware policy in use. It is, however, impossible to transform arbitrary code into a perfect match for a single hardware policy. For this reason, we propose to allow the compiler to explicitly *control the operation of the cache for each reference.*

Detailed global control/data flow analysis of programs enables us to know more about the order of instruction execution and about the data used or defined by each instruction. In effect, this analysis can determine either the exact reference sequence or a set of possible reference sequences and their associated probabilities of occurrence at runtime. This makes a software cache policy feasible — if this information were not obtainable automatically (using compiler analysis), very few users would be willing or able to explicitly state cache control for each reference.

The optimal control of a cache using compile-time information does not, however, require an increase in the complexity of the cache hardware. Rather, this control simplifies it, since the hardware no longer need make decisions, but merely implement them on command. If a particular reference is "marked" (by the compiler) as being cached in a certain way, it is of no great concern to the hardware that the previous reference was "marked" to be treated differently — as far as the hardware is concerned, the cache policy is consistently just to do what it is explicitly told. In effect, ordinary general-purpose registers within a processor have long been managed in exactly this way: cache "registers" (entries) are not really so different.

As in performing good register allocation, the overhead imposed is that a complex compiler technology must be designed and implemented. But, aside from improving performance in much the same way registers do, this overhead is justified by the simplification of the hardware relative to achieving a given cache hit ratio — the VLSI area saved, particularly in an on-chip cache, is priceless.

In the following sections, a software cache policy model, called the SCP Model, is described as an alternative cache management policy. The basic idea of this model is to analyze global control/data flow of the program and to have the compiler *explicitly* manage cache activity based on these information. Toward this, the global control/data flow graph obtained from the analysis of a program is expanded to include all possible cache contents at each **cache stage** (defined below) in the graph. Cost for each *transaction* of cache content from one **cache state** (defined below) to the next is then placed in the graph as the weight of the edge linking the two cache states. An algorithm (based on shortest path) is executed to obtain an *optimal* cache policy for each cache transaction. This information is used by the compiler to generate code which explicitly controls the cache, either through "cache instructions" (treated much like coprocessor instructions to be executed by the cache engine) or as tags on each instruction.

Throughout this analysis, it is assumed that the reference pattern is *precisely known* at compile time[3].

Section 3.1 states the asumptions made in the SCP Model. In section 3.2, notations and definitions of terms used in the analysis are introduced. Section 3.3 describes the graph formulation of programs. In section 3.4, the algorithm implementing the software cache policy is described. Implementation methodologies for this cache policy are described in section 3.5.

1. Assumptions of the SCP Model

There are several assumptions made in the Generalized SCP Model and few more in the analysis of this SCP Model in this paper.

1.1. Generalized SCP Model Assumptions

The key assumptions of the Generalized SCP Model (there are additional assumptions made in the particular version used in this paper) are:

(1) Full associative cache organization is assumed. As will be discussed in Chapter 4, other organizations, such as direct mapping and set associative, they can be transformed into sets of subproblems with full associative cache organization and smaller cache sizes.

(2) Architectural cache parameters such as set size, line size, and cache size are assumed to be fixed. This research attempts to obtain the best cache performance from a given cache hardware design, rather than to determine the hardware design to achieve a fixed performance goal. (However, it is possible to derive this information using the same basic techniques.)

3. In this report, we ignore the fact that some references will be ambiguous: for example, a pointer might be known to refer to one of two different memory cells, but the compiler may not know which. These complications, as well as code structures including branches (as opposed to the branchless reference strings of this report), will be covered in separate documents.

(3) There will not be any restructuring of program control flow nor any rearrangement of the data/instruction storage patterns. Of course, some kinds of program restructuring and rearrangement of data/instructions storage patterns can improve the localities of data/instructions. However, this will be discussed in a later document.

(4) The reference string is known at compile time (i.e. branchless code, with completely unambiguous data references, is assumed).

1.2. Additional Assumptions

Additional assumptions made in the analysis of the SCP Model within this report are discussed in the next few sections. These assumptions are not crucial to the model, but rather serve to make analysis, and comparison with other alternatives, more managable for this presentation.

(1) The central processing unit has the capability of directly accessing the main memory without going through the cache (with access time T_r) as is shown in Figure 4.

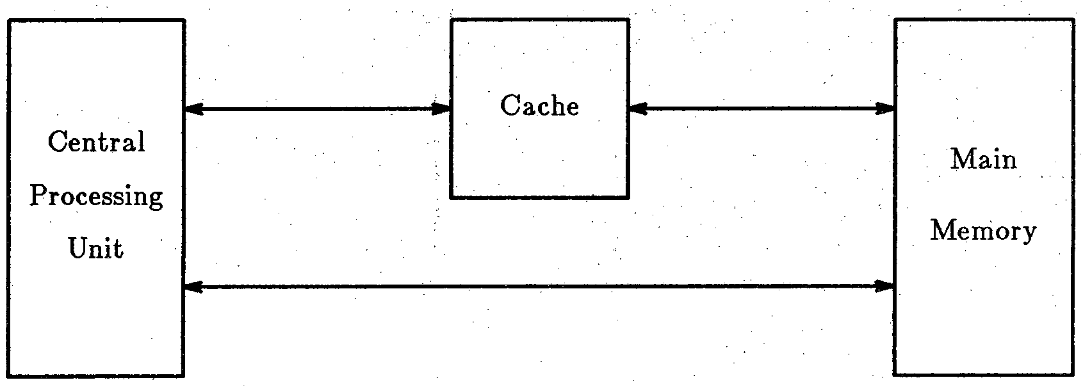


Figure 4: Model of Processor/Cache/Memory Interface

Often, it is more economical to reference an entry directly from the main memory than transferring the whole line into the cache before it is referenced. The large overhead imposed in cache line transfer may not be justified by the infrequent use of a very few entries in that cache line. In this case, direct reference from the main memory is preferred.

(2) In case of direct access to the main memory, the central processing unit need not place a copy of the accessed datum into the cache. This also provides a convenient treatment of different write policies within the SCP Model: write through is modeled by using the direct-to-memory path, whereas write back requests travel through the cache.

(3) Transferring n lines from main memory at one time will take much less time than transferring 1 line from main memory n times. Of course, there is some architectural limit of the number of lines which may be transferred in one request. However, the existence of a limit is ignored in the current work.

(4) The main memory is large enough to hold the whole program and no secondary memory is visible. This is clearly false on virtual-memory machines running relatively large programs, but the assumption simplifies the analysis considerably. Considering the secondary memory (or multi-level cache operation in general) requires a model using more complex cache states.

(5) Let T_c units be the time to reference an entry in a line in cache, T_r units be the time to directly reference an entry in a line in main memory, and T_p units be the time to transfer a cache line between cache and main memory. To simplify the analysis, we assume that all these times are constant — although in some systems the costs will be a function of physical distances to the memory modules containing the addresses referenced, or of other factors involving probabilistic contention for interconnection links. Typically, $T_c << T_r \leq T_p$.

2. Notations and Definitions

Before describing the graph formulation of program, some definitions of notations and terms are needed:

(1) Let $N = \{1, 2, ..., n\}$ be the set of cache lines that may be referenced within a program.

(2) Let M be the set of *distinct* cache lines in the referenced program. The size of M will be represented as m, where $1 \leq n \leq m$. It is found that the ratio of (dynamic cache lines used)/(static cache lines used) (i.e. n/m) is very large; an factor of 100 or 1000 is not unusual.

(3) Let k represent the number of cache lines in cache. It is assumed that $k \leq m$. Here, the general case of program size larger than the cache size is used. The case where program size is smaller than the cache size is not considered because the whole program can then be placed in the cache and any reasonable cache policy will perform very well.

(4) The reference string will be denoted as $\omega = r_1, r_2, ..., r_n$, where $r_i \in M$.

(5) Let S_i represent the subset of M in cache after the reference r_i has been completed. For all i, $0 \leq |S_i| \leq k$ and $S_0 = \emptyset$.

3. Graph Formulation of Software Cache Policy

In this section, the SCP Model for controlling all cache prefetching and placement/replacement activities is described. In this model, the global control/data flow graph obtained is expanded to include all possible cache contents at each cache stage. The graph is constructed in such a way that all possible complete sequences of cache state transactions from the initial cache state to the final cache state are included in the graph. Each of the paths from the initial cache state (defined below) to the final cache state represents one possible complete sequence of cache state transactions for executing the given memory reference string ω.

A simple algorithm is then used to find the path with lowest cache transaction cost (i.e., the shortest path). After the optimal set of cache transactions has been found, the cache control is embeded in the code generated by the compiler (either as explicit cache prefetching instruction or as tagging reference to the end of each instruction).

When the memory reference string known at compile time, this technique results in provably optimal cache performance. This optimality of use of the cache hardware is insured for any cache hardware design (within the bounds given above) and for any transaction cost function. This ability to use arbitrary cost functions makes the SCP Model particularly attractive in control of multiprocessor caches, where hardware-implemented cache policies are typically unable to use the fact that different costs are associated with different memory locations (local or global). The optimality of the SCP Model depends only on the reference string, and the cost function, being known at compile time.

In the graph formulation of the SCP Model, there are four phases:

(1) cache state construction (i.e., vertex construction at a particular time instant in the graph),

(2) cache stage construction (i.e., vertex construction for the whole graph),

(3) cache arc construction (i.e., arc construction in the graph), and

(4) cache arc cost association (i.e., weight assignment to each edge in the graph).

In the SCP Model, all reference string symbol addresses are converted to their corresponding line numbers in the main memory before any analysis is made. This simplifies the analysis in the SCP Model, with no ill effects.

3.1. Cache State Construction

A **cache state** is defined as a possible configuration of cache lines in the cache. In the SCP Model, cache state $v_{i,j}$ in the graph represents the j^{th} possible cache configuration immediately after making the reference r_i. For example, if there are three distinct cache lines {1, 2, 3} in the refernce string and the cache size is two, then the cache may have one of the following possible cache states:

State Number	Cache State	Cache Entries Used
1	$\emptyset$	0/2 (empty)
2	{1}	1/2
3	{2}	1/2
4	{3}	1/2
5	{1,2}	2/2 (full)
6	{1,3}	2/2 (full)
7	{2,3}	2/2 (full)

Table 3: Cache Configurations for k = 2 and m = 3

Given a cache of size k and a program of size M, the maximum possible the number of possible cache states is *(M+1)!/(M+1-k)!*. Obviously, this number is very large for programs of reasonable size and caches of common sizes — but this bound is assuming *fully associative* cache, which means that only the set size, and the fraction of the program lines which fall into each set, are actually valid. For example, if a direct-mapped cache of 4096 lines (set size 2) is used for a program containing 65536 lines, the relevant numbers are simply $k = 2$ and $M = 65536/4096$, hence: there are (16+1)!/(16+1-2)!, or 272, possible cache states (for each of 4096 subproblems). Further, as we have indicated, although the SCP Model can be used for such large caches, the performance increase is most valuable in on-chip and other small caches — typically of size 32 or less — where the analysis becomes very simple (essentially because the set of program lines which need be considered simultaneously decreases as the cache size decreases).

3.2. Cache Stage Construction

Cache stage i is the collection of all possible cache states after memory reference r_i is made. Let $v_{i,j}$ be the cache state j at stage i with the subset of cache lines, S_j contained in cache at the time immediately after the reference r_i. The cache states that make up stage i, $1 \leq i \leq n$ represent all possible subsets of program lines contained in the cache after memory reference r_i. These vertices can be found in the following manner.

The graph representing the reference string is partitioned into n+2 stages, corresponding to the initial stage, r_1, ..., r_n reference stages and the final stage. (The initial and final stages are defined to simplify the graph analysis.)

At stage 0, there is only one cache state, since the initial cache contents are presumably either known or the cache holds no pertinent entries ($S_0 = \emptyset$; the cache contains no valid entries).

For stage i, $1 \leq i \leq n$, the stage consists of all possible (reachable) cache states defined by the cache size and the distinct lines of the memory reference string. Using the example in the *cache state construction*, there are 7 cache states in each cache stage i, $1 \leq i \leq n$, for a cache with size two. The main reason for constructing these cache stages is that by including all possible cache states in each cache stage, we can guarantee that there must be one cache state in each cache stage i which shows the cache content after the memory reference r_i. Since all possible sequences of cache state transaction are included, the optimal cache policy is one which traverses the shortest (cheapest) path from the initial state to the final state. For each memory reference r_i, exactly one cache state j in cache stage i along this optimal path is used.

At stage $n+1$, there is a cache state indicating the final contents of the cache (when the memory reference string has completed). The cache content S_{n+1} at this final stage is unimportant because after a given memory reference string has completed, there is no reason to prefer one cache state to another — all cache states have the same effect, hence they can be collapsed into a single cache state.

An example cache stage construction helps clarify this. Suppose the given memory reference string, ω, is `1231`, the size of the cache, k, is two. The number of distinct lines in the reference string, M, is `123`. All cache stages in the graph are shown in Figure 5. In Figure 5, each circle represents one possible cache state and inside the circle is the cache content S_j. There are 8 cache stages ($n = 6$ in this case). Stage 0 has one cache state, corresponding to the initial state of the cache (which we will assume to be $\emptyset$). For stage i, $1 \leq i \leq 6$, there are 7 cache states within each stage (as given in Table 3). Stage $n+1$ also has one stage, indicating the final state of the cache (whose content is unimportant).

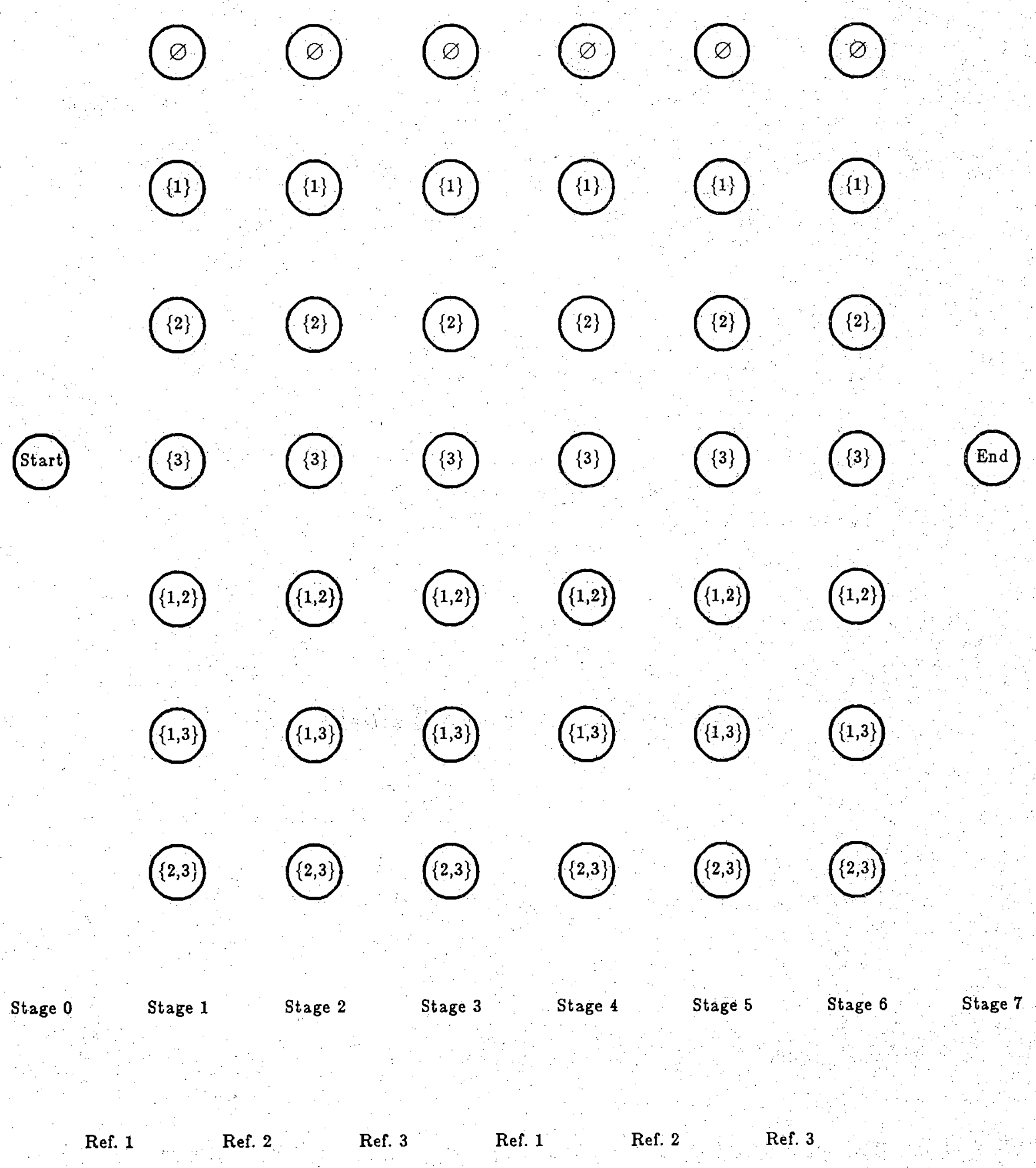


Figure 5: Stages for SCP Model

3.3. Cache Arc Construction

In the SCP Model, construction of an arc from cache state $v_{i,j}$ to cache state $v_{i+1,j'}$ represents a possible cache-control operation in performing reference r_{i+1}. The arc leaves a cache state in stage i and points into the cache state in stage $i+1$ which differs only in that reference r_{i+1} may have altered the cache contents. In this way, arcs are only created between states in successive stages, and it is possible that some states in each stage may not be reached; this is because most cache architectures severly constrain the action which can be taken in a single cache transaction. For example, if only one cache line can be replaced at one time, cache states with more than one line different are not connected by an arc. (Indeed, it is this fact more than any other which makes use of an optimal software cache management scheme feasible — without this constraint, the compile times would explode.)

In this phase, arcs for all possible cache state transactions must be constructed. From each cache state $v_{i,j}$, there are three classes of arcs leaving $v_{i,j}$:

No Placement/Replacement:
From each cache state $v_{i,j}$, an arc is constructed to $v_{i+1,j'}$, where $S_j = S_{j'}$. This kind of cache state transaction can occur under two situations. First, memory reference r_{i+1} is in the cache. In this case, the memory reference is directly from the cache. Second, memory reference r_{i+1} is not in the cache. In this case, the memory reference is directly from the main memory. In both cases, there is *no* placement/replacement in the cache.

Placement (without replacement):
If $|S_j| < k$, an arc is created to $v_{i+1,j'}$, where $S_{j'} = S_j + r_{i+1}$. This represents a reference which is not available from the cache, but which may be placed in the cache in any entry which was previously not valid (empty). Under typical cache hardware constraints, the cache content will be changed by just one entry and there is usually no reason to differentiate between multiple empty entries; hence, only one arc of this type will be drawn from that cache state.

Replacement:
If $|S_j| = k$ then for each x in S_j, let $S_{j'} = S_j$ - x + r_{i+1} and arcs are constructed to $v_{i+1,j'}$. This represents the situation where the cache is full and the next reference is placed into the cache, thereby replacing an existing entry. Since each line in the cache can be replaced, arcs corresponding to the replacement of each line in the cache are drawn. Any line could be replaced, hence there will be one arc of this type drawn to each cache state where the referenced line is in the cache and the cache is full.

Continuing the example in the previous phase, the graph resulting from cache arc construction is shown in Figure 5. Note that there are cache states which do not have arcs coming in. These are *unreachable*, or "dead," cache states and they can be removed from the graph.

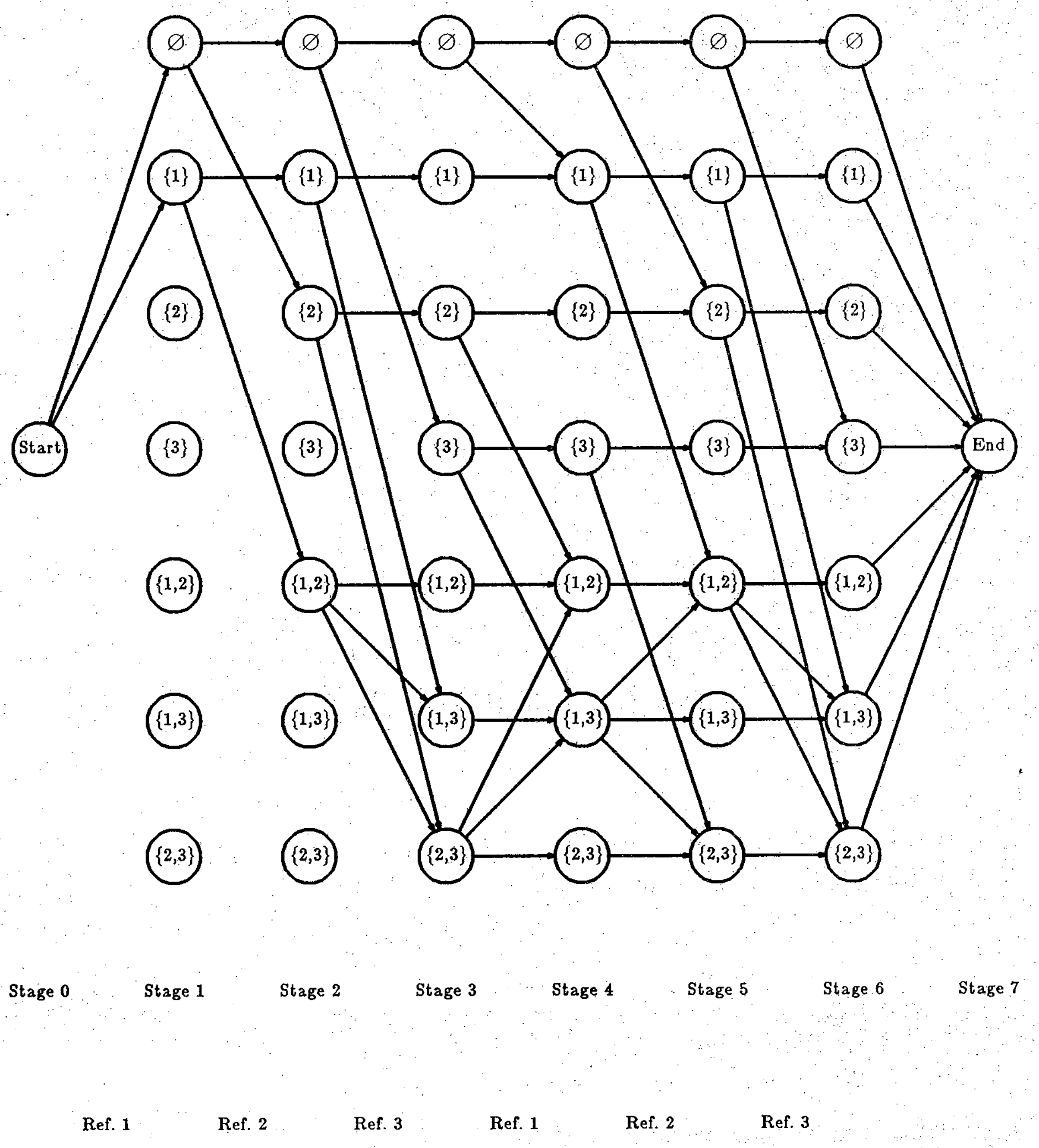


Figure 6: Arcs for SCP Model

3.4. Cache Arc Cost Association

In the SCP Model, the cost associated with an arc represents the expected "relative" cost of going from one cache state to the next in the graph. This cost may be a constant or a variable (as in the case of multiprocessing). Generally speaking, its value depends on the change of the cache content and the delay in accessing the source memory module for the reference r_i. This second aspect is most important for read and instruction-fetch references (under placement, replacement, and direct reference) and for write references (under replacement and direct reference).

The cost of each arc (cache state transaction) in the graph is computed and assigned by the following rules. For each arc, the cost of the arc connecting cache state $v_{i,j}$ and $v_{i+1,j'}$ is:

$S_j = S_{j'}$
: If $r_{i+1} \in S_{j'}$, the cost is T_c. This represents a memory reference which is satisfied by an existing entry in the cache. If not, the cost is T_r, representing a reference directly from main memory (bypassing the cache entirely).

$S_j \subset S_{j'}$
: The cost is $(T_p + T_c) * (|S_{j'}| - |S_j|)$. The cost is the product of the cost of placing one line in cache and the number of cache lines that need to be placed in cache. This represents cache placement (without replacement). Note that the cost T_p for each cache line placement may be different.

$(S_j \neq S_{j'})$ and $(|S_j| = |S_{j'}|)$
: The cost is $(2 * T_p + T_c) *$ (number of different lines between S_j and $S_{j'}$). The cost is the product of the number of replacement and the number of replacement lines. This is the case where replacement occurs. Here again, note that the cost T_p for each placement/replacement may be different.

Each cache state in stage n has one exit arc which enters the final cache state at stage $n+1$. The costs associated with these arcs are 0 (since they do not represent a physical action).

Continuing the example from the previous phases, and given the graph in Figure 6, the cost associated with each type of arc, and the line-style used to represent each, is given in Table 1. Figure 7 gives the graph of Figure 6 after each arc has been given a line-style indicating its cost.

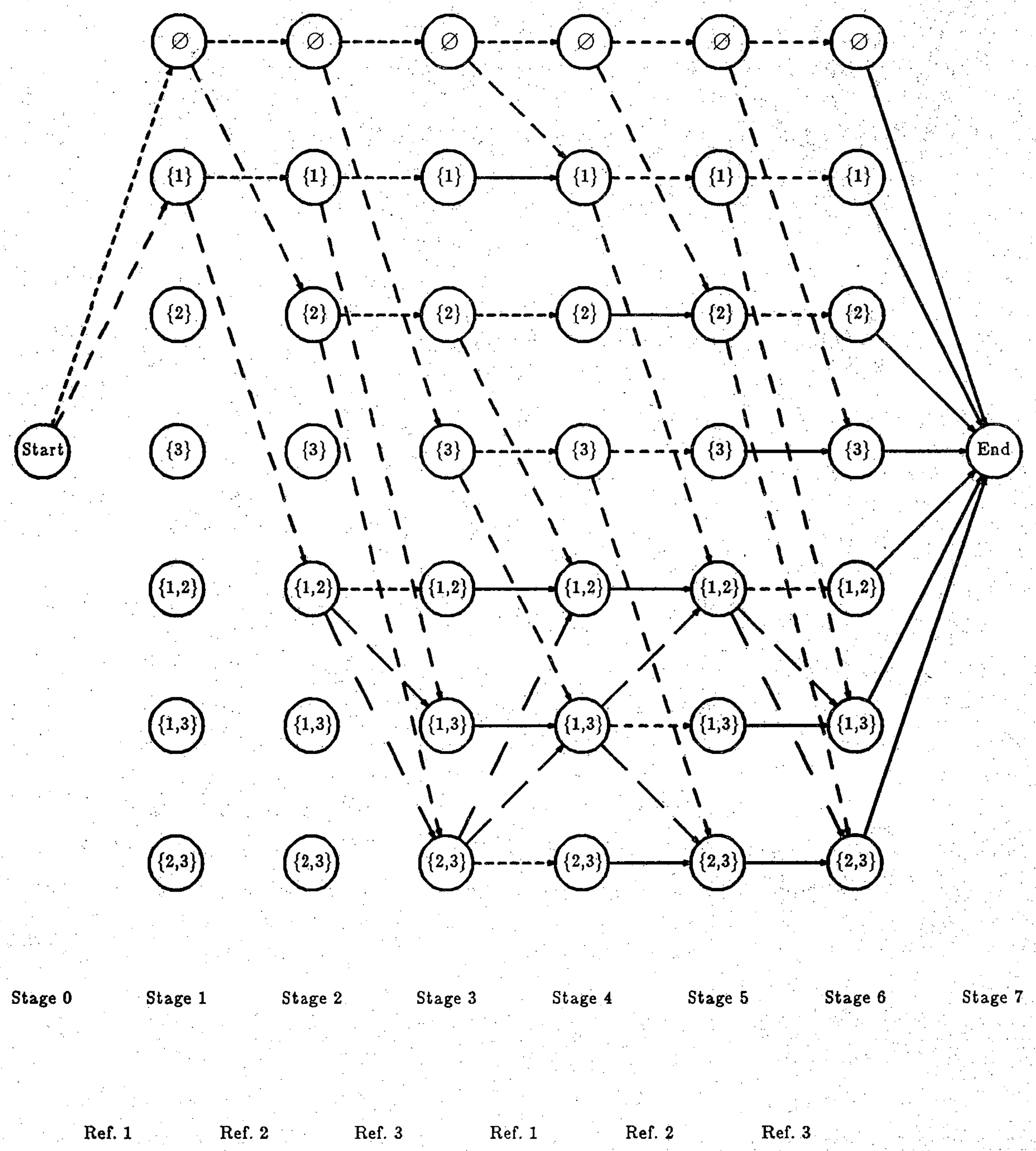


Figure 7: Arc Costs for SCP Model

4. Algorithms for Cache Placement/Replacement Policy

The directed graph obtained in section 3.4 and in Figure 7 includes all possible paths corresponding to all possible cache placement/replacement policies for the memory reference string 1 2 3 1 2 3. Each of these paths represents a complete sequence of cache state transactions as the given memory string is referenced.

In Figure 8, the graph actually generated by the SCP Model (with all dead cache states removed) is shown.

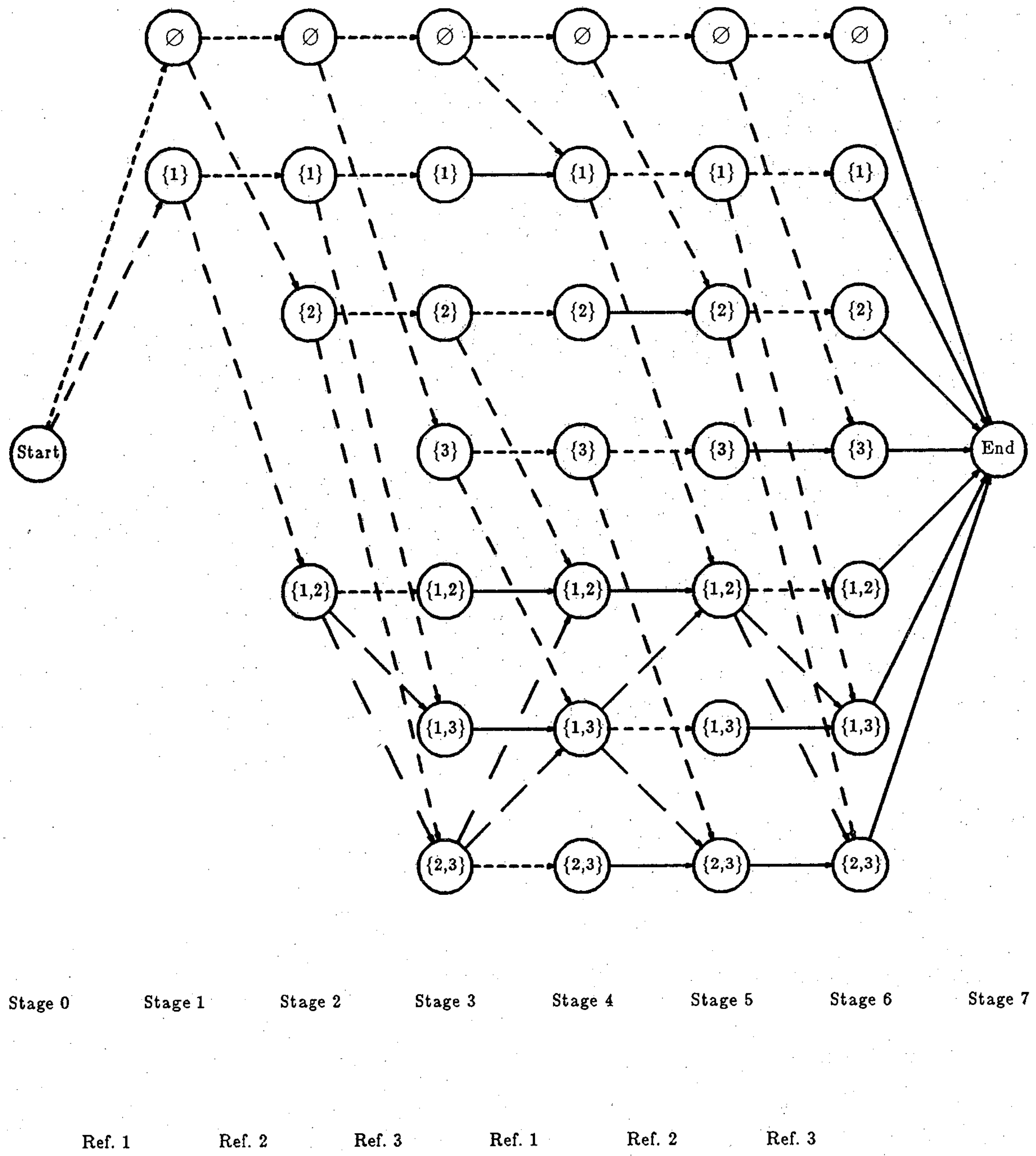


Figure 8: Complete SCP Model (w/o Dead States)

The problem now is to select the shortest path from the initial cache state at stage 0 to the final cache state at stage $n+1$ in the graph. Standard algorithms for the shortest path problem [Wag76] [Joh77] can provide a solution to this problem[4]. Figure 9 shows the optimal path obtained by performing the shortest path algorithm on the graph of Figure 8. This optimal path represents the exact cache state transactions given the memory reference string 1 2 3 1 2 3.

4. A more computationally-desirable approach would be to use a pruned search which truncated the search in such a way as to avoid generating most states within the later stages.

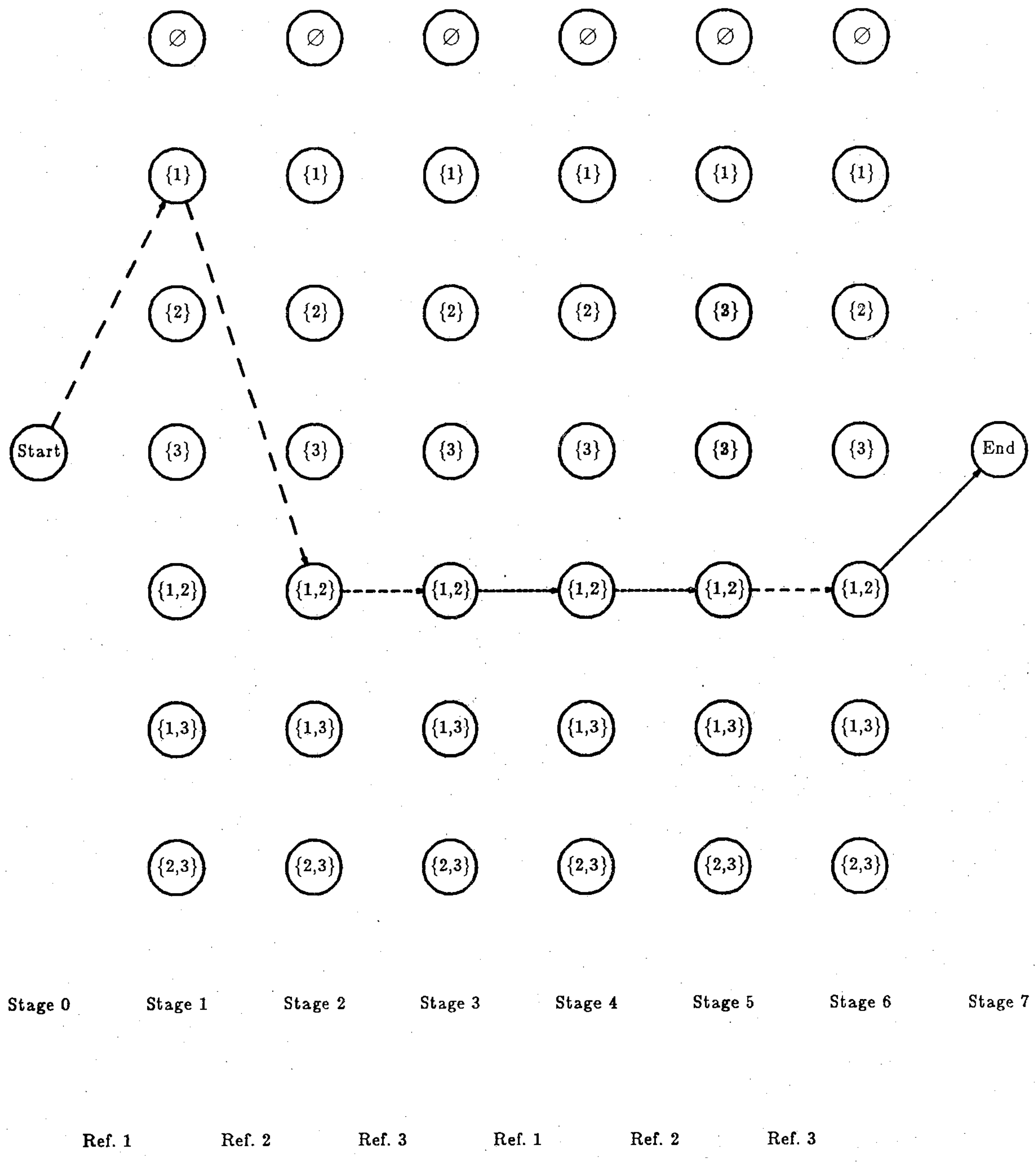


Figure 9: Optimal Cache Control for 123123

The shortest path solution for each cache state transaction for referencing string 123123 is given in Figure 9. Detailed information about the cache content, type of placement/replacement, place of reference, line number to be feteched, and line number to be replaced are collected and are shown in Table 4. This information, especially the cache operation for each reference, and the cache line to be operated on, can be embeded into the code generated by the compiler. In this way, optimal cache performance is assured, since the shortest path is, by definition, the optimal set of cache operations.

Cache State	Reference String	Cache Content after Reference	Place of Reference	Line No. Fetched	Line No. Replaced
Start	—	$\emptyset$	—	—	—
$v_{1,2}$	1	{1}	Cache	1	—
$v_{2,5}$	2	{1,2}	Cache	2	—
$v_{3,5}$	3	{1,2}	Main Memory	—	—
$v_{4.5}$	1	{1,2}	Cache	—	—
$v_{5.5}$	2	{1,2}	Cache	—	—
$v_{6.5}$	3	{1,2}	Main Memory	—	—

Table 4: Optimal Cache Control Sequence for 123123

The graph so constructed contains O(nm^k) cache states and O(knm^k) edges. Wagner's shortest path algorithm for a directed acyclic graph from a single source has an execution time of O(max(v,e,d)), where v is the number of vertices, e is the number of edges, and d is the maximum cost of any edge. For our graph, the computational complexity of the SCP Model is O(knm^k) (i.e. O(e)) using Wagner's algorithm.

Although this computational complexity is not promising, it is not actually a severe limitation. Most computer systems employ very simple cache using direct mapping with a small set size and a small line size. For these simple cache organizations, the compile times are expected to be quite acceptable. The key factor in the complexity of the algorithm is the set size — which, for very practical hardware implementation reasons, cannot be very large and is usually one or two.

As is shown in Chapter 4, the complexity is O(nm) for direct mapping and O(nm^2) for set associative with line size of two. The value of m is typically much smaller than the value of n because the static code size of a program is always less than its dynamic code size. Further, since caching is based on lines, each of which may contain a number of memory locations, the number of distiguishable address references is reduced by this factor.

Moreover, heuristic algorithms can always help us to reduce the graph analysis time, thereby improving compile time. Alpha-beta pruning, and generating the graph only as the search requires portions of it, can dramatically improve the graph analysis time without sacrificing optimality.

5. Implementation of Software Cache Policy

Finally, the SCP Model requires that information about each cache state transaction (such as that given in Table 4) be inserted into the code generated by the compiler. These inserted "cache directives" can be embedded in the generated code in two different forms: new cache control instructions or tagging references at the end of each instruction.

The first method is to define new cache control instructions (such as load a line to cache, or store a line in cache to main memory) for fetching and storing memory lines and to explicitly insert these new instructions into the code generated by the compiler. The cost of this implementation is the extra execution times needed to execute these explicit new cache control instructions. The main advantage is, however, that the cache management can then be implemented as a coprocessor — *permitting use of existing, conventional, processors.* In fact, even without any specialized hardware, some benefit can be gained in multiprocessor environments by using a software-simulated cache and block transfers between global and local memory spaces (feasible only because global memory references may have extremely long access times).

The other alternative, which is more appropriate in custom-designed processors, is to place the cache control directives (from the compiler) within each instruction, by use of a cache-directive tag field. This trades the time to execute coprocessor-instruction cache directives for the need to "borrow" instruction bits (in every instruction which could cause a reference) for cache directives. Although the need for extra instruction bits may increase the instruction length (and hence the cycle time), the fact that only a couple of bits are typically needed leads us to predict that such extension of instruction lengths will not be necessary.

A combination of these two methods is also possible. Cache directives may be implemented by tagging references within instructions. Although this document has not been much concerned with prefetching, it is more difficult to embed prefetch cache directives within each instruction than to make separate prefetch instructions, since the prefetch offset may require a large number of bits for its representation. Hence, cache control instructions would be used for prefetch. This minimizes the number of cache control instructions in the execution stream and, at the same time, solves the problem of large offsets in prefetch references.

This SCP Model can be implemented using demand fetch, prefetch, or a combination of the two; in many cases, the distinction between the two is quite vague. For example, a delayed load instruction (as found in most RISC processors) can be considered to be either demand fetch — because it requests data on demand relative to the memory system outside the processor) — or prefetch — because the request is separated from the use within the processor. In either demand fetch or prefetch, the same delays are encountered and the compiler must schedule the fetch/cache activities to minimize idle time: only the positioning of the control "directives" within the execution stream is different. For example, the use of NOP instructions to fill-in the gap between issuing and completing a delayed load in RISC processors is not fundamentally different from the (much older) techniques which do not advertise a delayed load but none-the-less allow loads to take several cycles and each instruction waits for a hardware "valid" tag before it uses the content of a register.

SCP Model in Different Cache Organizations

In cache design, one of the main factors in obtaining high efficiency is the cache organization. Basically, there are three common cache organizations: direct mapping, set assocative and full associative. (Direct mapping and full associative are actually special cases of set associative where set size is 1 and k, respectively.) In this chapter, applications of the SCP Model to these different cache organizations are discussed. As will be seen in the next few sections, the SCP Model discussed in the last chapter can be applied to these three cache organizations with only minor modifications.

In section 1 of this chapter, application of the SCP Model to direct mapping cache organization is discussed. Section 2 investigates the application of SCP Model to set associative cache organization. Finally, application of the SCP Model to full associative cache organization is discussed in section 3.

1. SCP Model in Direct Mapping Cache Organization

This is the simplest, and therefore most commonly implemented, of all possible cache organizations. In this direct mapping, line i in the memory maps into the line i modulo k of the cache, where k is the size of the cache [HwB84]. Every M/k (where M is the size of the main memory) will be mapped to the same cache line. The direct mapping cache organization is shown in Figure 10.

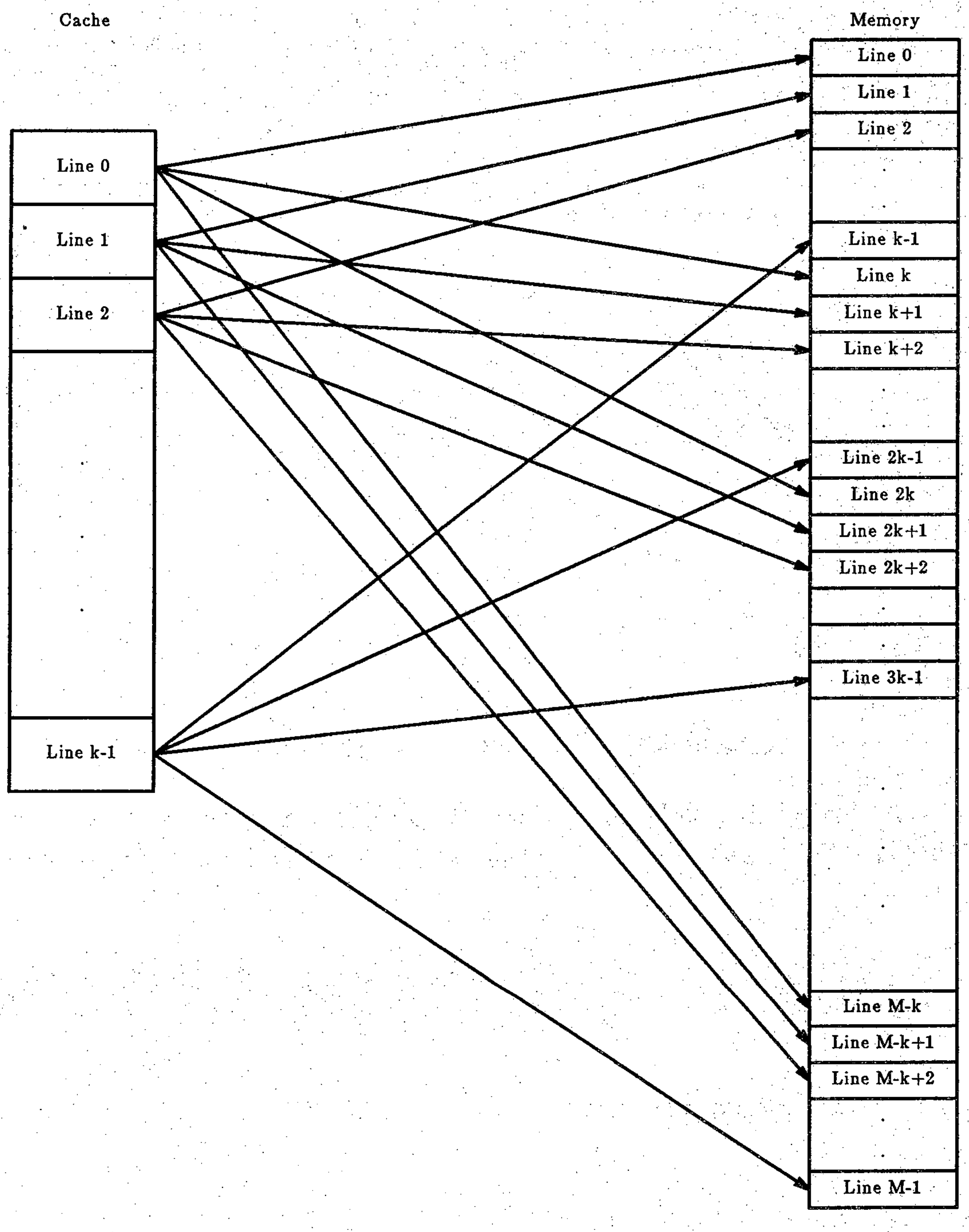


Figure 10: Direct Mapping Cache Organization

In the SCP Model, direct mapping cache organization can be visualized *as k independent sub-organizations* as shown in Figure 11. Each of these *k* sub-organizations consists of a cache of size one and a main memory of size M/k. For the given memory reference string, it also subdivides into k sub-strings. In each sub-string, all the line numbers are mapped to the same line in cache — thus reducing the alternatives for performing a reference to just two: either reference from main memory or reference using the line in cache. Since each sub-organization and its corresponding sub-string is independent of the others, it can be analyzed separately using the technique described in Chapter 3.

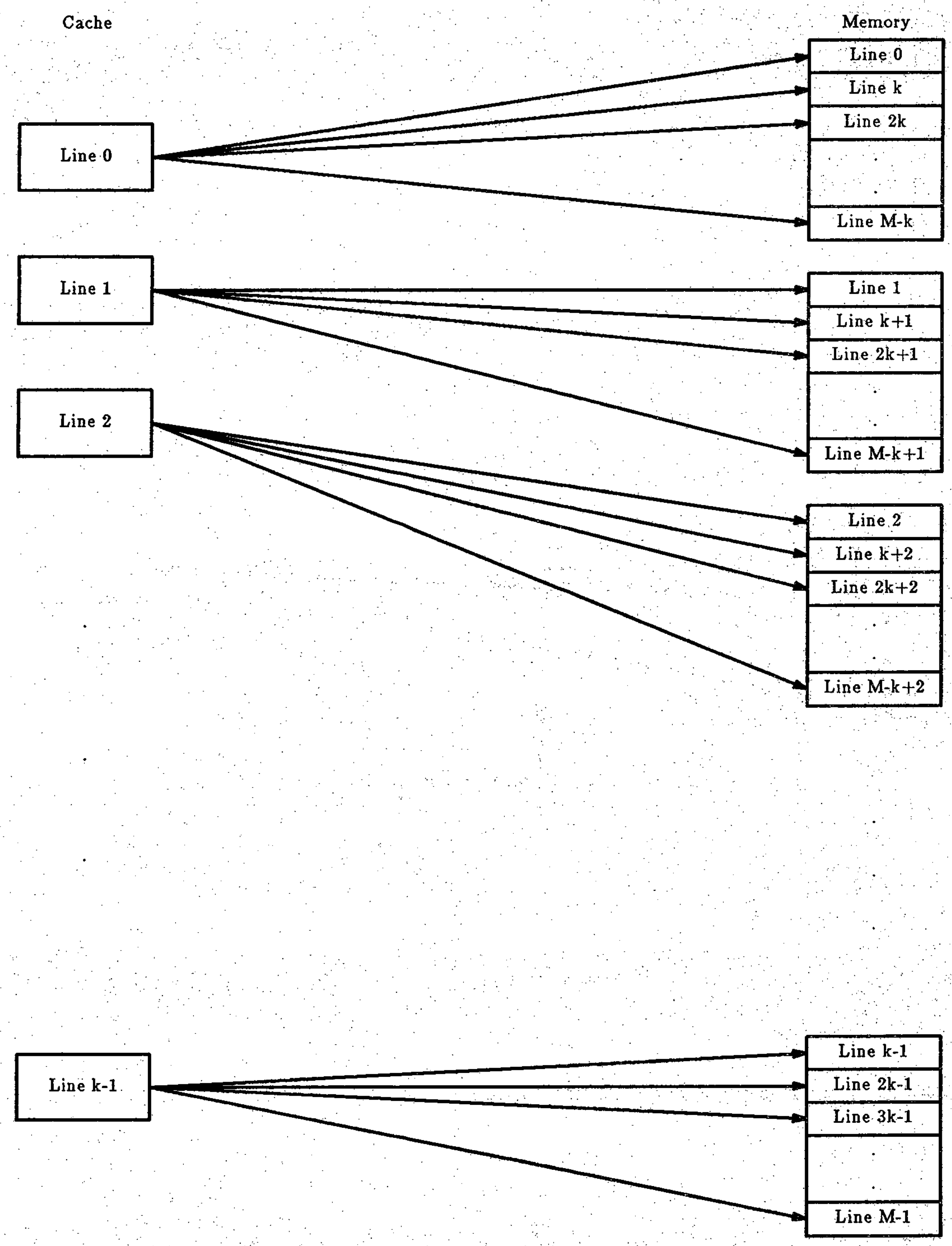


Figure 11: SCP Model of Direct Mapping

Using the example in the last chapter with line 1 and 3 in main memory mapped to the same line in cache and line 2 in main memory to another line in cache, sub-string 1313 and 22 are formed. The graph generated by the SCP Model in direct mapping cache organization is shown in Figure 12. In this case, there are 2 subgraphs of Figure 12 (each of which is analyzed separately).

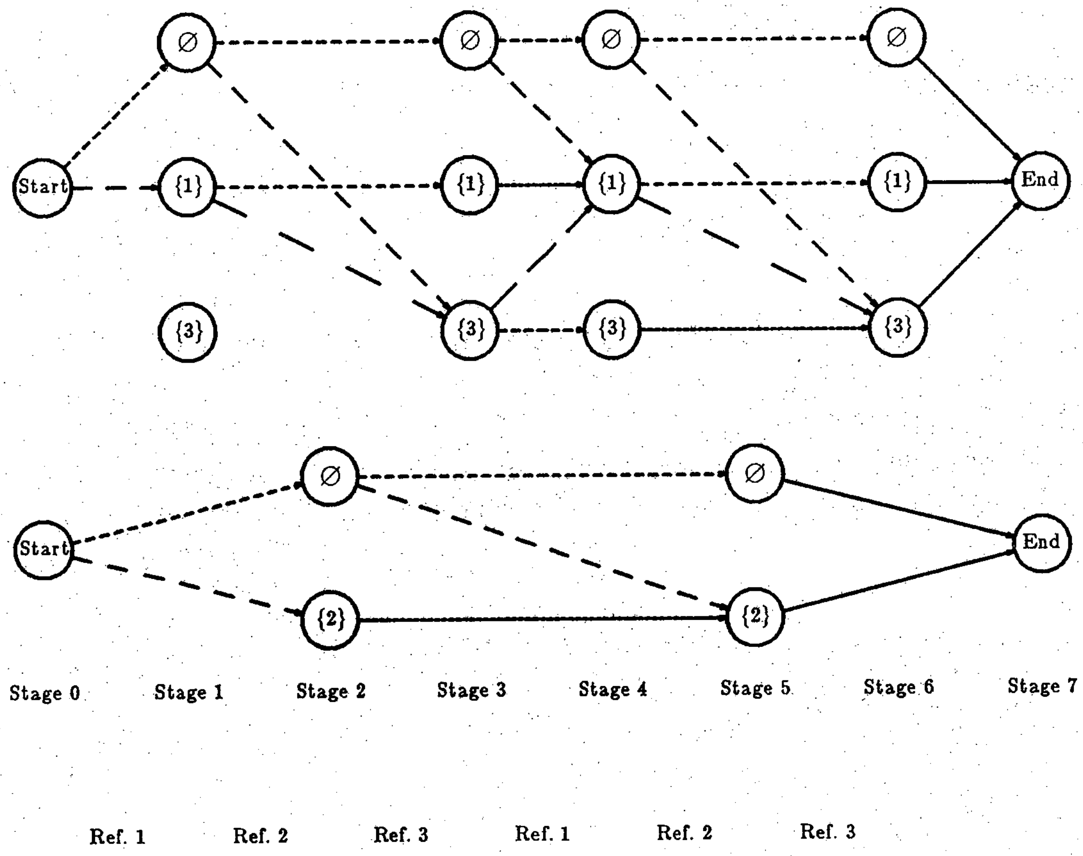


Figure 12: SCP Model for Direct Mapping 123123

The resulting optimal cache use for referencing string 123123 using direct mapping cache organization in the SCP Model is shown in Figure 13. Note that the two independent optimal paths found by the SCP Model together provide all the information needed to control the cache.

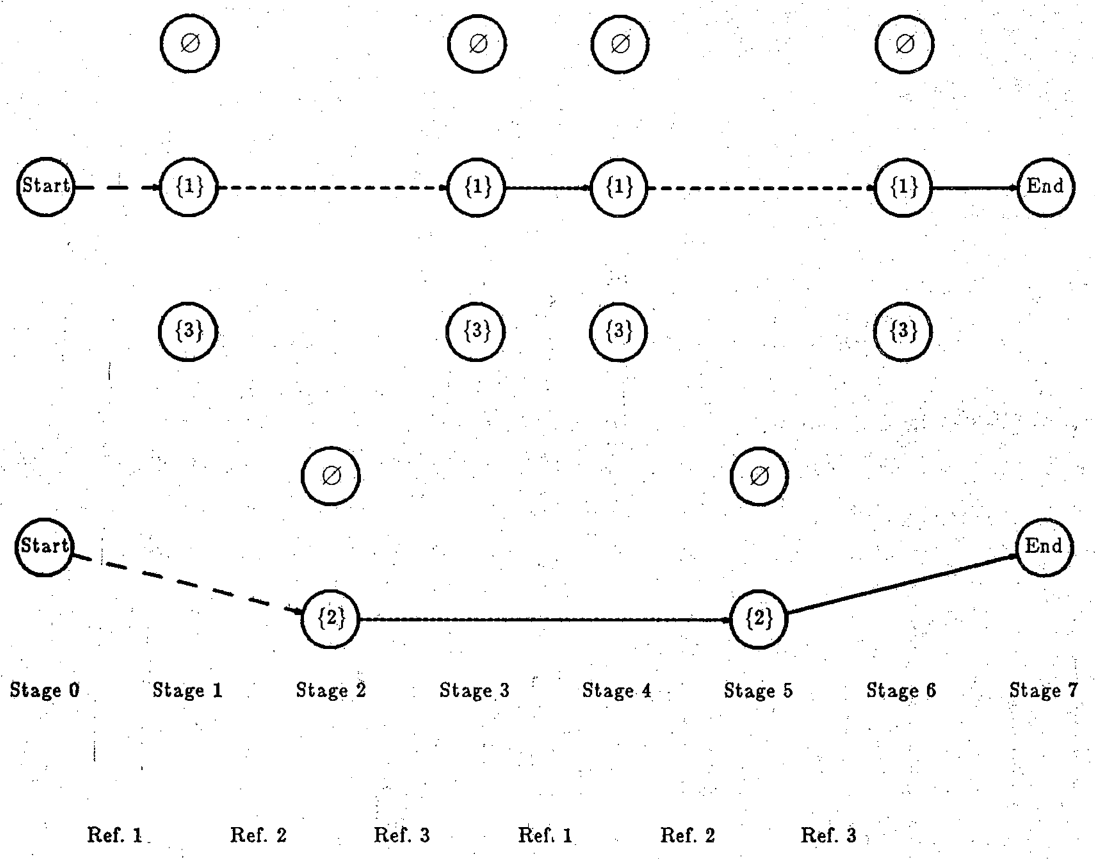


Figure 13: Optimal Cache Control for Direct Mapping 1 2 3 1 2 3

If this optimal cache management (in Figure 13) is used instead of simply always using the cache (as in Figure 14), the performance, using the cost functions given earlier, improves by a factor of 2. *By slectively disabling the cache, cache performance was dramatically improved!* This surprising result is the direct effect of avoiding cache pollution.

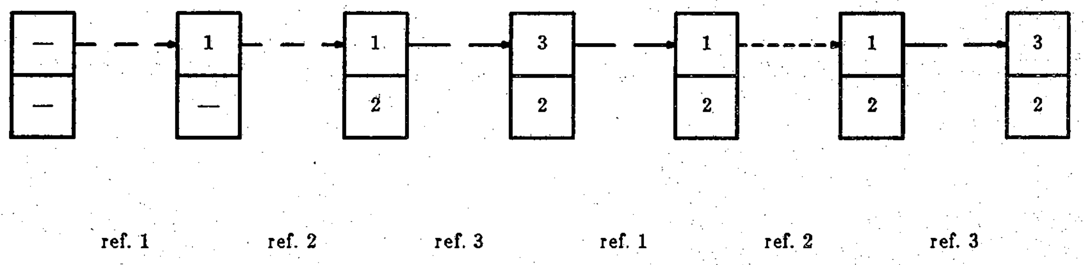


Figure 14: Direct Mapping Transactions for 1 2 3 1 2 3

The worst-case complexity of the SCP Model in direct mapping cache organization is O(nm), which makes the technique practical even for very large caches. On the average, however, there will be k sub-problems each of which concerns m/k distinct cache lines and n/k references. This results in a complexity of O($k^{-1}mn$), which reflects the fact that *the analysis becomes easier as the cache becomes larger.*

2. SCP Model in Set Associative Cache Organization

The set-associative organization divides the cache into S sets with $E = k/S$ lines per set, where k is the total number of lines in the cache. For reasons of hardware complexity, E is rarely greater than four [Smi78c], and is most often two. A line i in main memory can be cached in any line belonging to the set i modulo S [HwB84]. Set Associative cache organization with set size of two is shown in Figure 15.

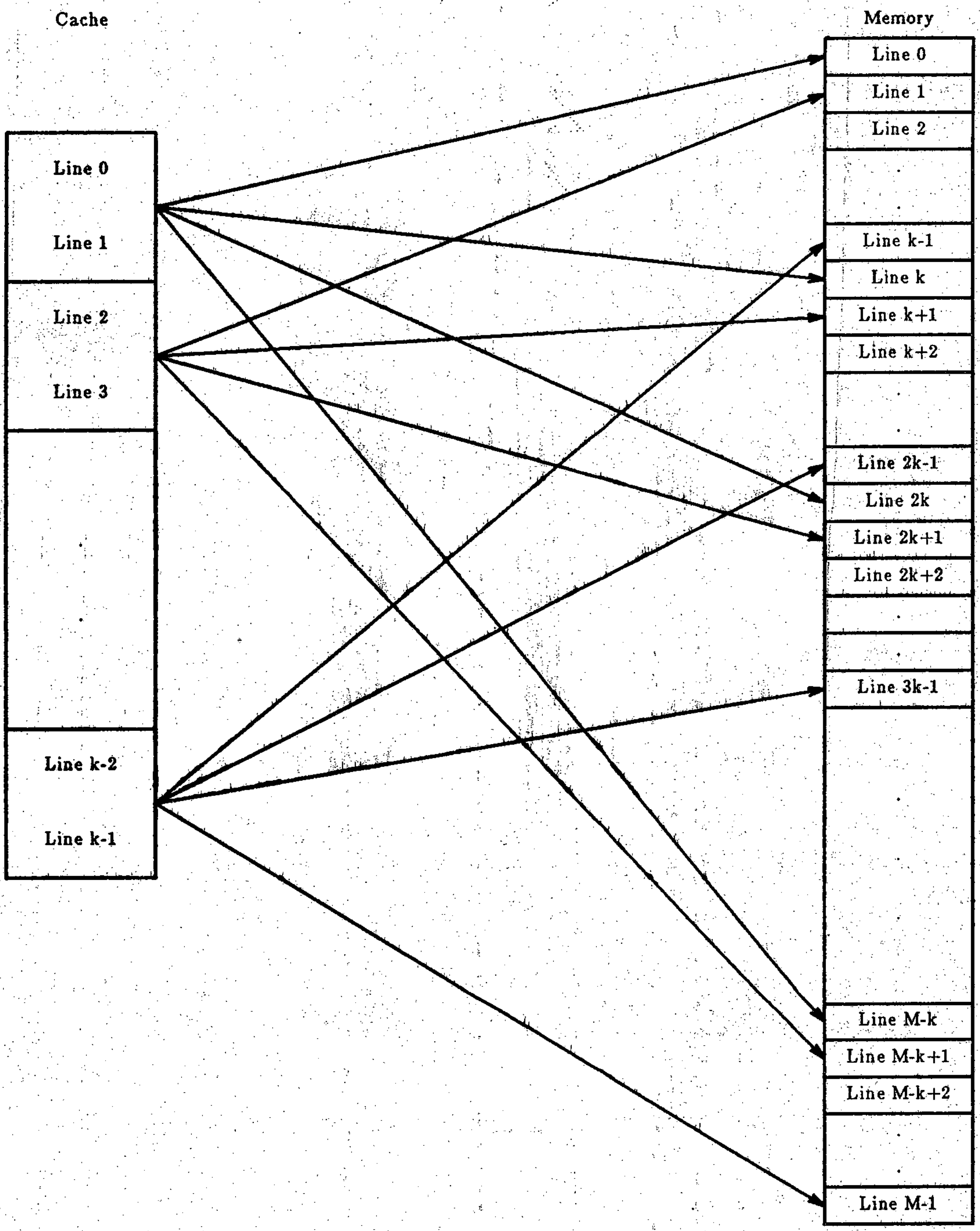


Figure 15: Set Associative Cache Organization

In the SCP Model, set associative cache organization can be visualized as S independent sub-organizations. Each of these S sub-organizations consists of a cache of size k/S and a main memory of size M/k. The memory reference string is also subdivided into S sub-strings, each of which has all its symbols mapped to the same cache set. Since each sub-organization and its corresponding sub-string is independent of the others, it can be analyzed separately using the technique described in Chapter 3. An example of set size of 2 is shown in Figure 16.

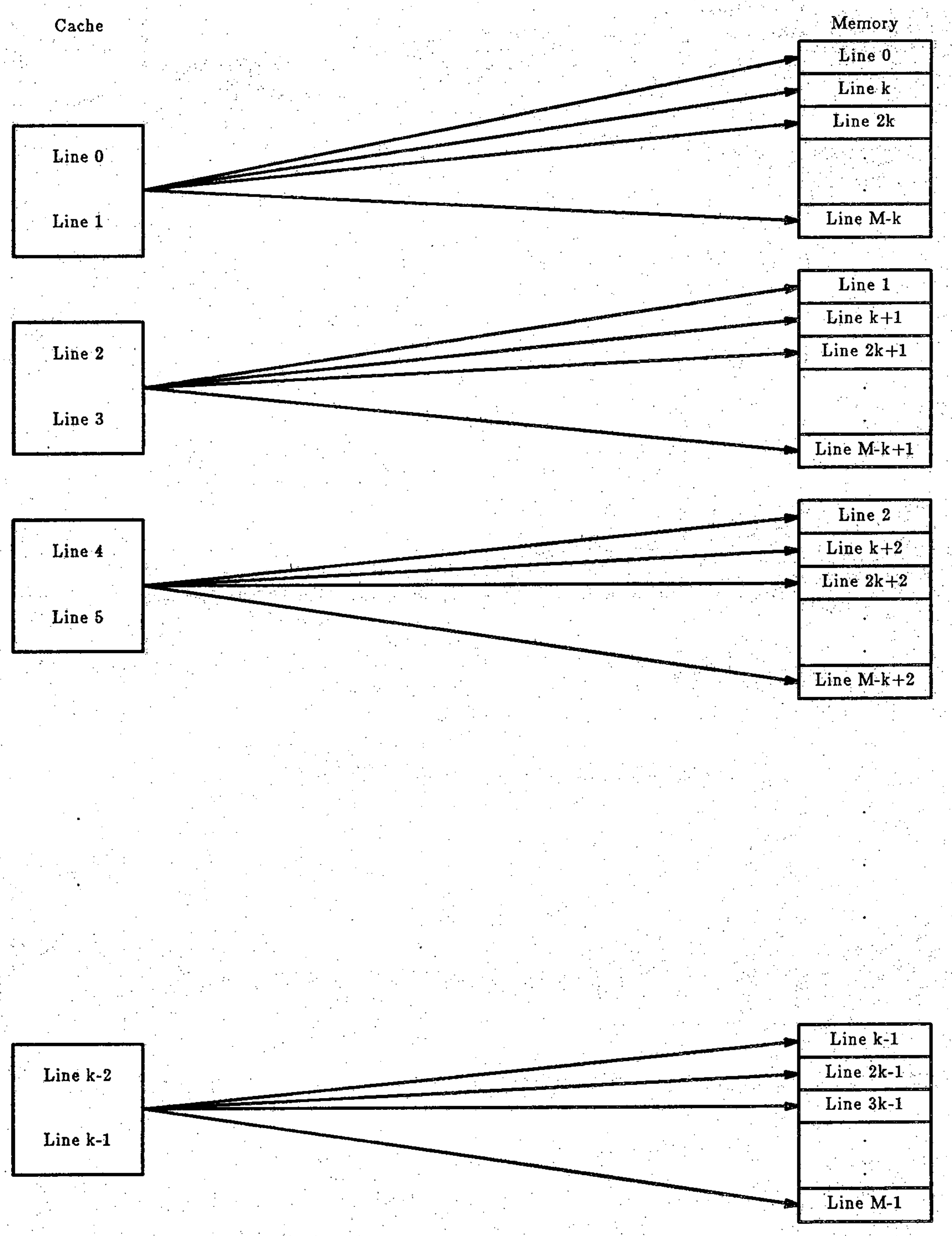


Figure 16: SCP Model of Set Associative (Size Two)

The worst-case complexity of the SCP Model in the direct mapping cache organization is O(Snm^E). For small set sizes, which hardware complexity issues generally force, the SCP Model is still practical. For example, if E is 2, then the worst-case complexity is just O(Snm^2). Again, it is useful to note that, on the average, increasing the cache size but not increasing the set size will simplify the problem; the complexity is O($nm(m/S)^{E-1}$), where S increases as k.

3. SCP Model in Full Associative Cache Organization

Full associative cache organization permits any line in main memory to be mapped into any line in cache. In other words, E is k — which hardware can implement only if k is very small. Full associative cache organization is shown in Figure 17.

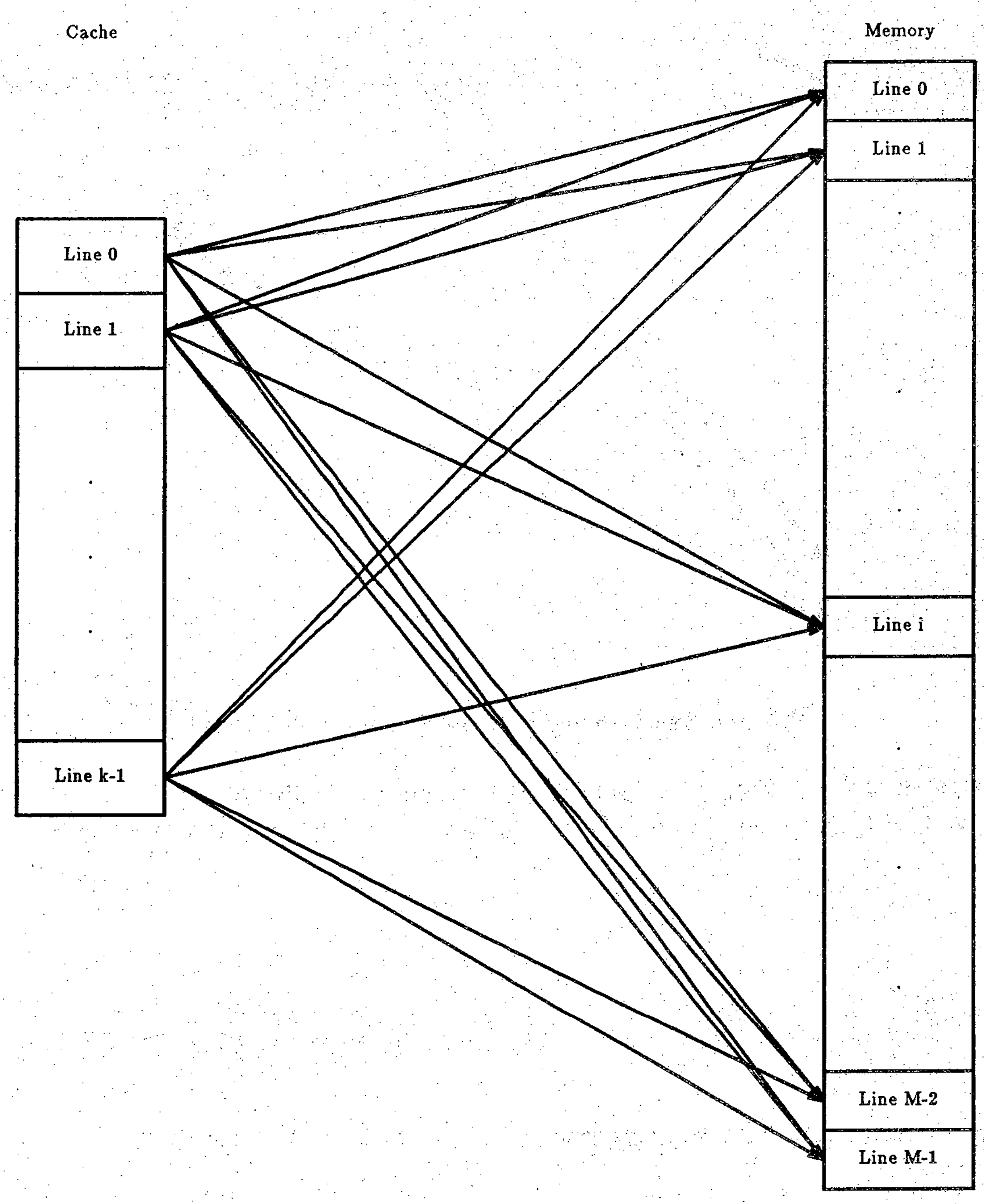


Figure 17: Full Associative Cache Organization

The techniques described in Chapter 3 can be used to analyze the full associative cache organization. Full associative cache organization in the SCP Model is shown in Figure 18.

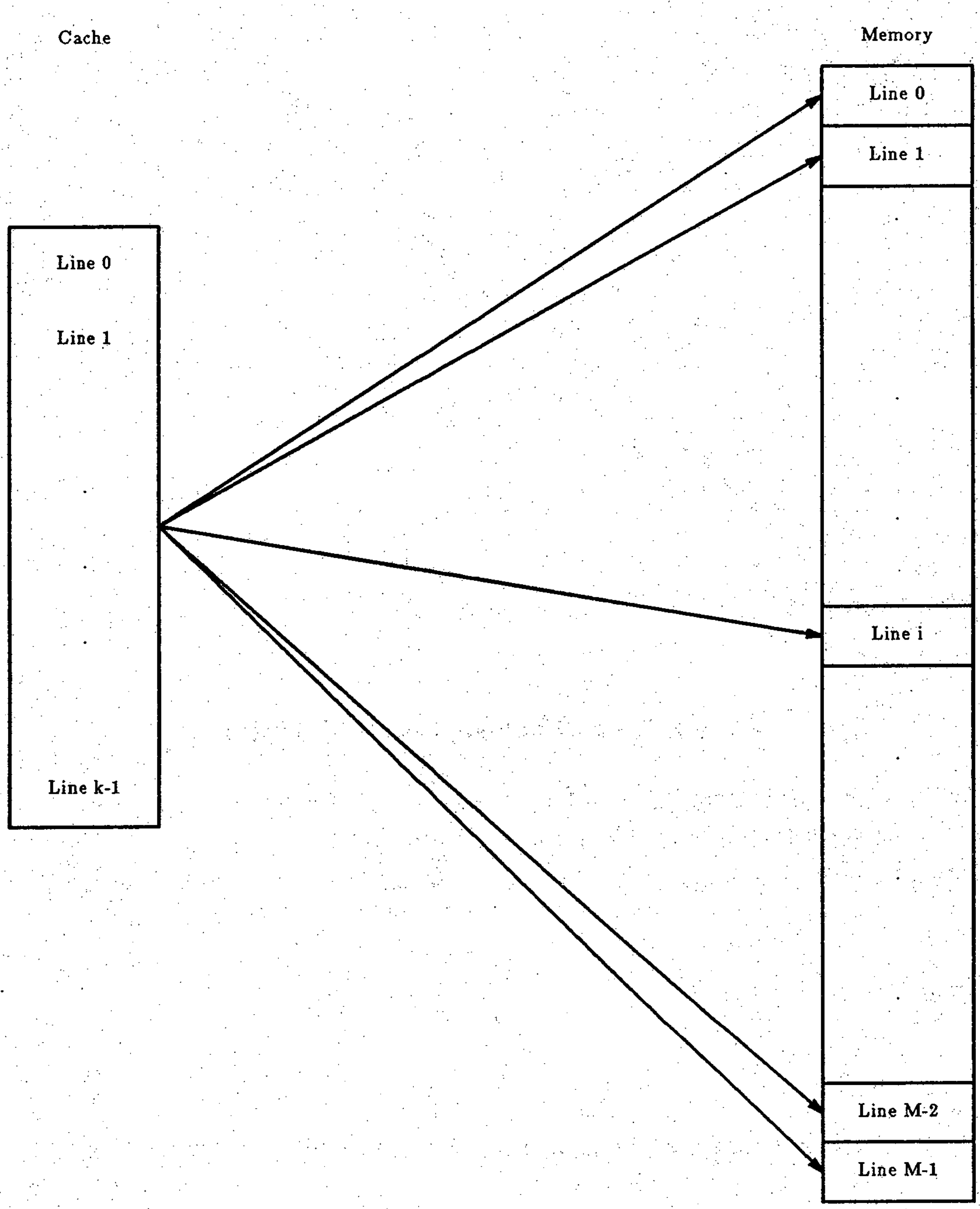


Figure 18: SCP Model of Full Associative

The worst-case complexity of the SCP Model in a full associative cache organization is $O(knm^k)$, which is only practical for very small caches. However, as discussed earlier, the hardware implementation complexity also restricts full associative caches to very small k.

Conclusions

This research offers a new methodology for the management of cache activities. Although still in the initial stages, it has demonstrated the potential for a major "breakthrough" in cache performance.

With the graph formulation of a program and shortest-path based analysis, the algorithm *guarantees optimal cache performance*; both demand-fetching/prefetching and placement/replacement policy are optimally managed given a reference string which is known at compile time. Although this precise knowledge of the reference string is unlikely to be available at compile time, the techniques presented here do not actually require such perfect knowledge — *compile-time-ambiguous alias problems merely block our claim of optimality, not the general applicability of the techniques.* Further, the exact reference string is known in performing register allocation — the technique presented here is also an *optimal register allocation algorithm.*

Another innovation in this work is that arbitrary cost functions for placement/replacement are permitted. The cost functions can be re-defined to fit any computer system, including parallel processing systems with complex memory organizations, yet the SCP Model insures optimal performance. These cost functions can even be statistical (e.g., they may represent estimates of expected traffic over interconnection links, etc.), although optimality cannot be insured is such a case. This flexibility is a key step forward in cache performance optimization relative to all previous research concerning optimal page algorithms like MIN [Bel66], VMIN [PrF76], GOPT [DeS78], and DMIN [BuD81], where cost functions are not considered and optimal performance, even assuming uniform cost, is not guaranteed. (We feel that this is a fair comparison, since the reference string is also asssumed to be known in these algorithms.)

There is, of course, some concern over the admittedly high complexity of software to implement a compiler-driven cache policy. However, as discussed above, *the hardware to implement a cache structure becomes intolerably complex under the same circumstances which make the SCP Model unwieldy*: most cache structures that are easy to build are also practical to control using the SCP Model. It also seems likely that the complexity of the SCP technique can be greatly reduced using simple search heuristics, such as alpha-beta pruning.

In addition to this new methodology, we have proposed a hardware structure which can cheaply and efficiently implement the new cache policy. The SCP Model can be implemented by explicitly inserting "cache control instructions" into the code at compiled time or by adding a prefetching offset to the end of each instruction generated by the compiler. In fact, using this new model, it is possible to implement a processor which will use a small software-managed on-chip cache as/instead of registers: this new architecture is discussed in a paper currently in preparation.

In summary, we have shown that a software cache policy can greatly improve cache performance in ways that traditional hardware-implemented cache policies, as such, cannot. The analytic results given in this paper are currently being confirmed by simulations of the software cache policy, and will be published in a later document.

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.

[Bab82] Babaoglu, O., "Hierarchical Replacement Decisions in Hierarchical Stores," *Proceeding of the 1982 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1982, pp. 11-19.

[BaS76] Baer, J.L., Sager, G.R., "Dynamic Improvement of locality in virtual memory systems," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 1, 1976, pp. 54-62.

[Bel66] Belady, L.A., "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Sys. Journal*, Vol. 5, 1966, pp. 78-101.

[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.

[BuD81] Budzinski, R.L., Davidson, E.S., Mayeda, W., Stone, H.S., "DMIN : An Algorithm for Computing the Optimal Dynamic Allocation in a Virtual Memory Computer," *IEEE Transactions on Software Engineering*, Vol. SE-7, No. 1, January 1981, pp. 113-121.

[CoD73] Coffman, E.G., Denning, P.J., *Operating System Theory*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[DeS78] Denning, P.J., Slutz, D.R., "Generalized Working Sets for Segment Reference Strings," *Communications of the ACM*, Vol. 21, No. 9, September, 1978, pp. 750-759.

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

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

[Joh77] Johnson, D.B., "Efficient Algorithms for Shortest Paths in Sparse Networks," *Journal of the ACM*, Vol, 24, No. 1, January 1977, pp. 1-13.

[Jos70] Joseph, M., "An Analysis of paging and program behavior," *Computer Journal*, 13, 1, 1970, pp. 48-54.

[PeS85] Peterson, J. L., Silberschatz, A., *Operating Systems Concepts*, Addison-Wesley Publishing Company, 1985, pp. 222-226.

[PrF76] Prieve, B.G., Fabry, R.S., "VMIN - An Optimal Variable-Space Page Replacement Algorithm," *Communications of the ACM*, Vol. 19, No. 5, May 1976, pp. 295-297.

[SmG85] Smith, J.E., Goodman, J.R., "Instruction Cache Replacement Policies and Organizations," *IEEE Transactions on Computers*, Vol. c-34, No. 3, March 1985.

[Smi78a]	Smith, A.J., "Sequentiality and Prefetching in data base systems," *ACM Transactions on Data Base Systems*, Vol. 3, No. 3, 1978, pp. 223-247.
[Smi78b]	Smith, A.J., "Sequential Program Prefetching in Memory Hierarchies," *IEEE Computer*, Vol. 11, No. 12, 1978, pp. 7-21.
[Smi78c]	Smith, A.J., "A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory," *IEEE Transactions on Software Engineering*, Vol. 4, No. 2, Mar. 1978, pp. 121-130.
[Smi82]	Smith, A.J., "Cache Memories," *Computing Surveys*, Vol. 14, No. 3, September, 1982, pp. 473-530.
[Spa74]	Spaniol, 0., "Demand prepaging algorithms basing on a model of locality of programs," in Gelenbe, E. and Mahl, R. eds., *Computer Architectures and Networks*, North-Holland, Amsterdam, 1974, pp. 515-527.
[Spi76]	Spirn, J., "Distance String Models for Program Behavior," *IEEE Computer*, November, 1976, pp. 14-20.
[Spi77]	Spirn, J., *Program Behavior: Models and Measurements*, Elsevier-North Holland, N.Y., 1977.
[Wag76]	Wagner, R.A., "A Shortest Path Algorithm for Edge-Sparse Graphs," *Journal of the ACM*, Vol. 23, No. 1, January 1976, pp. 50-57.