Optimal Worst Case Formulas
Comparing Cache Memory Associativity

Håkan Lennerstad*    Lars Lundberg*

1 Abstract

Consider an arbitrary program \( P \) which is to be executed on a computer with two alternative cache memories. The first cache has \( k \) sets and \( u \) blocks in each set, this is denoted a \((k, u)\)-cache. The other is a fully associative cache with \( q \) blocks; a \((1, q)\)-cache.

We present optimal formulas comparing the performance of a \((k, u)\)-cache compared to a \((1, q)\)-cache for worst case scenarios. Optimal mappings of the program variables to the cache blocks are assumed.

Let \( h(P, k, u) \) denote the number of cache hits for the program \( P \) using a \((k, u)\)-cache and an optimal mapping. We establish an explicit formula for the function \( r(n, k, u, q) \) and prove that

\[
\inf_P \frac{h(P, k, u)}{h(P, 1, q)} = 1 - r(n, k, u, q),
\]

where the infimum is taken over all programs \( P \) which contain \( n \) variables.

We also deduce a formula for the infimum taken over all programs of any number of variables:

\[
\inf_P \frac{h(P, k, u)}{h(P, 1, q)} = 1 - R(k, u, q).
\]

Further we prove that programs which are extremal for this minimum may have any hit ratio, i.e. any ratio \( h(P, 1, q)/m(P) \). Here \( m(P) \) is the total number of memory references for the program \( P \). An optimal bound on the hit ratio using a \((k, u)\)-cache, given the hit ratio with a \((1, q)\)-cache, also follows.

*Both authors were supported in part by the Blekinge Research Foundation.
We assume the LRU replacement policy, that each variable can be stored in one memory block and is free to be stored in any block.

Since the problems of finding optimal mappings are NP-hard, the results provide optimal bounds for NP-hard quantities.

The results on cache hits can easily be transformed to results on access times. The results make it possible to compare the efficiency of a heuristic \((k, u)\)-cache mapping algorithm to an optimal mapping algorithm. Not only for small \(k, u\) and \(q\), the results further support performance trade-offs when choosing size and degree of associativity for a cache, which are optimal for worst-case scenarios.

2 Introduction

Cache memories reduce the memory access time in computer systems; an excellent survey can be found in [19]. There is, however, a limited hardware budget in terms of gates, routing etc. when designing cache memories. This hardware budget can be used in different ways, e.g. one can include different amounts of associative memories thus yielding a fully associative, set-associative or direct mapped cache. Due to the limited hardware budget, using associative memories reduces the number of storage elements in the cache. Moreover, the access time usually increases with increasing associativity. Consequently, trade-offs between the access time, the degree of associativity and the number of storage elements are interesting. Such design trade-offs are far from trivial. One aspect here is the implementation cost of associative memories compared to the storage elements. This ratio depends on the implementation technique [16].

Another aspect is the maximum difference in hit ratio between small fully associative caches and larger set-associative and direct mapped caches. Calculating the maximum hit ratio decrease when decreasing the number of storage elements is trivial, i.e. in the worst-case scenario the hit ratio may drop to zero. However, estimating the maximum hit ratio increase when increasing the degree of associativity and decreasing the total number of storage elements is non-trivial.

In this paper we establish optimal upper bounds on the maximum relative increase of cache hits when using a fully associative cache compared to a set-associative or direct mapped cache. We will assume that the compiler is free to map variables to arbitrary memory addresses. Figure 1 illustrates...
The present report is a part of a series of reports on optimal worst case bounds in computer science contexts. We have also derived optimal bounds for static versus dynamic allocation of parallel programs ([7] and [10]), cluster versus dynamic allocation ([8] and [11]), for all programs with a specified parallelism [12], for the worst case performance drop at memory reorganization [13], and for the cluster execution time for a program $P$ only knowing one execution time for $P$, executed with dynamic allocation and any schedule [14]. All these reports provide optimal bounds for NP-hard quantities. In the present report we consider the NP-hard problem of finding an optimal mapping for a certain program $P$ with a $(k,u)$-cache, i.e. a mapping maximizing the number of cache hits. We give a formula for this number for worst case programs.

The method in all the reports consists of two steps:

1. Successive elimination of unnecessary programs, leading to a subset of all programs, which contains at least one extremal (worst-case) program and further can be represented by matrices.

2. In the remaining set of programs, sometimes regarded as matrices, those where all possible rows occur equally frequently are proven to be extremal. Here we have obtained a formulation with simplicity enough to allow explicit formulas to be derived.

The first step is naturally application dependent to a higher degree than the second. In the present report the first step is taken care of in sections 3 - 7, while sections 8 and 9 deal with the second step. Section 10 discusses the
applicability of the results, and section 11 displays some graphics concerning the results.

3 Problem definition and notation

A data cache is defined by five parameters: the number of sets, the number of blocks in each set, the block size, the replacement policy within a set and the update policy to main memory. We focus here on two parameters: the number of cache sets and the number of blocks in each set. A cache with \( k \) sets containing \( u \) blocks each is denoted a \((k, u)\)-cache. In some reports this cache configuration is referred to as a \( u \)-way set associative cache with \( k \) sets. A mapping maps each variable in a program onto a specific memory address and thus to a specific cache set, during the complete execution each variable will be stored on this address. Thus a mapping can be thought of as a partition of the variables in \( k \) sets.

If \( u \) equals one, the cache is direct mapped. If \( k \) equals one, the cache is fully associative. It is well known [5] that with a fixed number of blocks, the hit ratio is higher, or at least equal, with a fully associative cache than with a direct mapped cache. The left part of Figure 1 shows a \((4, 1)\)-cache and the right part shows a \((2, 2)\)-cache, for a data segment of size 8.

In this study, the block size plays no important role, i.e., the results are valid for any block size. However all blocks have the same size. We are only interested in the hit ratio, i.e., the percentage of all memory references that can be handled by the cache. The hit ratio is unaffected by the update policy, e.g., write-through or copy-back. Therefore, the update policy plays no important role.

In cache memory systems code references are much easier to handle than data references. In this report we only consider data references. When ignoring the code references, the execution of the program \( P \) can be regarded as a sequence of \( m(P) \) references to \( n \) different program variables. All variables are stored in the data segment. We assume that the data segment is infinitely large, and that the compiler or programmer is free to map any variable onto any block. In section 10 this assumption is discussed. We also assume that all memory words in a variable are stored in the same block. We only consider variables which are referenced at least once, hence \( n \leq m \).

The only limit on the size of the program variables is that no variable exceeds one cache memory block. Programs with some small variables may have several variables mapped to the same cache block. If a variable in one
block is transferred to or from the cache, all other variables mapped to the same block are simultaneously transferred. In Theorem 3.2, we show that we may in the remainder of the report consider only the case of having exactly one variable mapped to each block.

No initial storing is assumed, hence the first referenced block will be stored in the cache without replacing any other variable. We assume that the LRU [15] replacement policy is used within each set. This means that if a cache set is full and a variable mapped to this set is referenced which is not in the cache, then the variable, together with other variables mapped to the same block, replaces the least recently referenced block in the cache set. This is a reasonable replacement policy which has been used in some machines [6] as well as in other studies (e.g., see [5] and [18]).

If the number of sets exceeds one, different mappings of variables to memory addresses may result in different hit ratios, e.g., consider a direct mapped cache and a program which repetitively accesses two variables in a loop. If these two variables are mapped to the same set, each access will result in a cache miss (this phenomenon is often referred to as bumps). However, if the two variables are mapped to different sets, there would be no cache misses, except the initial references. A mapping which, for a certain program $P$, results in a maximal number of hits is referred to as an optimal mapping for $P$. The number of hits for a program $P$, using an optimal mapping and a $(k,u)$-cache, is denoted $h(P,k,u)$. This number with any specific mapping $A$ is denoted by $h(A,P,k,u)$.

In this study we are interested in the minimal number of hits using a $(k,u)$-cache compared to the minimal number of hits with a $(1,q)$-cache, for programs $P$ of $n$ variables. Certainly some of the memory references that can be handled by the $(1,q)$-cache may lead to a miss using the $(k,u)$-cache, at least for some values on $n$, $k$, $q$, and $u$. References which, using optimal mappings, are hits using the $(1,q)$-cache but misses using the $(k,u)$-cache, are referred to as extra misses. We now consider a mapping which, using a $(k,u)$-cache, minimizes the number of extra misses. Using this mapping, the number of extra misses for a certain program $P$ is denoted $em(P,k,u,q)$. The number of extra misses with a specific mapping $A$ is denoted by $em(A,P,k,u,q)$. In mathematical language all possible mappings are all possible partitions of the $n$ variables in $k$ sets. We summarize:

**DEFINITION 3.1** Given a program $P$ we denote the maximal number of
hits and the minimal number of extra misses as follows:

\[ h(P, k, u) = \max_A h(A, P, k, u), \]

and

\[ \em(P, k, u, q) = \min_A \em(A, P, k, u, q). \]

Note that \( h(P, k, u) \) and \( \em(P, k, u, q) \) may be realized with different mappings. The problem of finding an optimal mapping using a \((k, u)\)-cache is NP-hard, e.g. see problems P01 and P02 in [2].

We next prove that in the interest of worst case estimates, we only need to consider programs where each variable have the size of one block. Note that in this case \( h(A_1, P, 1, q) = h(A_2, P, 1, q) \) for any mappings \( A_1 \) and \( A_2 \). Because of this we need no notation of mappings in conjunction to the quantity \( h(P, 1, q) \). In the theorem, \( n \)-programs denote programs of \( n \) variables.

**Theorem 3.2**

\[
\inf_{n \text{-programs } P} \frac{h(P, k, u)}{h(P, 1, q)} = \inf_{n \text{-programs } P \text{ with block sized variables only}} \frac{h(P, k, u)}{h(P, 1, q)}.
\]

**Proof:** Consider any program \( P \) with \( n \) variables, for which some variables are mapped to the same block when using an optimal mapping for a \((1, q)\)-cache. Suppose that the variables \( i_1, i_2, ..., i_j \) are mapped to the same block. Then there is another program \( P' \), which has none of the variables \( i_1, i_2, ..., i_j \), but a new variable \( i_0 \). The size of \( i_0 \) equals the sum of the sizes of the variables \( i_1, i_2, ..., i_j \), and each reference to \( i_1, i_2, ..., i_j \) in \( P \) is replaced by a reference to \( i_0 \) in \( P' \). Since all variables mapped to the same block are transferred to and from the cache simultaneously, we have \( h(P, 1, q) = h(P', 1, q) \).

Next consider a \((k, u)\)-cache. By the construction the mappings of the variables of \( P' \) can be regarded as those mappings of the variables of \( P \) where the variables \( i_1, i_2, ..., i_j \) are mapped to the same block. Further, for each mapping \( A' \) of the variables of \( P' \) there is a mapping \( A \) of the variables of \( P \) so that \( h(A', P', k, u) = h(A, P, k, u) \). Hence:

\[ h(P', k, u) = \max_{A'} h(A', P', k, u) \leq \max_A h(A, P, k, u) = h(P, k, u). \]
Thus:
\[
\frac{h(P', k, u)}{h(P', 1, q)} \leq \frac{h(P, k, u)}{h(P, 1, q)}.
\]

By repeating this argument we obtain a program \(P''\) where no variables are mapped to the same block, and where
\[
\frac{h(P'', k, u)}{h(P'', 1, q)} \leq \frac{h(P, k, u)}{h(P, 1, q)}.
\]

We also increase the size of each variable in \(P''\) to equal the size of one block. Clearly this also preserves the inequality.

Denote the number of variables of the program \(P''\) by \(y\); obviously \(y < n\).

We next construct a program \(P'''\) of \(n\) variables by adding one reference each to \(n - y\) new variables as the \(n - y\) last references in the program \(P''\). Each new variable has the same size as one block. The \(n - y\) last references in the program \(P''\) are all misses in any cache and with any mapping since the first reference to a block always is a miss. Hence we can also obtain an optimal mapping from an optimal mapping of the program \(P''\) by adding the new variables to any block. Hence: \(h(P'', 1, q) = h(P''', 1, q)\) and \(h(P'', k, u) = h(P''', k, u)\).

Given any program \(P\) of \(n\) variables, we can thus construct a program \(P'''\) of \(n\) variables where each variable have the size of one block, and where
\[
\frac{h(P''', k, u)}{h(P''', 1, q)} \leq \frac{h(P, k, u)}{h(P, 1, q)}.
\]

The theorem follows.

We also need the quantity \(eh(A, P, k, u, q)\), the extra hits. This is the number of memory references for \(P\) which using the mapping \(A\) which are misses with a \((1, q)\)-cache but hits using a \((k, u)\)-cache. We clearly have the relation

**Lemma 3.3** \(h(A, P, k, u) - h(P, 1, q) = eh(A, P, k, u, q) - em(A, P, k, u, q)\).

The significance of the results rely on the calculation of an explicit formula for the function

**Definition 3.4**
\[
r(n, k, u, q) = \max_P \frac{em(P, k, u, q)}{h(P, 1, q)},
\]

where the maximum is taken over all programs \(P\) with \(n\) variables.
The formula is presented in Theorem 4.2.
In Theorems 4.3 and 4.4 we also present results corresponding to the following function:

**DEFINITION 3.5**

\[ R(k, u, q) = \sup_P \frac{em(P, k, u, q)}{h(P, 1, q)}, \]

where the supremum is taken over all programs.

### 4 Main results

Our first result states that the formula for max\(_P \, \text{em}(P, k, u, q)/h(P, 1, q)\) can be used to obtain optimal worst case estimates for the hit ratio comparing the two caches. Here we thus obtain an optimal bound on the NP-hard quantity \(h(P, k, u)\).

**THEOREM 4.1 (i)** The worst case ratio of cache hits comparing two caches is given by

\[ \inf_P \frac{h(P, k, u)}{h(P, 1, q)} = 1 - r(n, k, u, q), \]

where the infimum is taken over all programs \(P\) with \(n\) variables.

(ii) Consider a program \(P\) with \(n\) variables, and suppose that the hit ratio \(h(P, 1, q)/m(P)\) for \(P\) with a \((1, q)\)-cache is known. Then an optimal estimate of the hit ratio for the program \(P\) using a \((k, u)\)-cache is given by

\[ \inf_P \frac{h(P, k, u)}{m(P)} = \left(1 - r(n, k, u, q)\right) \frac{h(P, 1, q)}{m(P)}. \]

The proof of Theorem 4.1 is given immediately after Theorem 4.10. Note that we do not claim the existence of extremal programs. The function \(1 - r(n, k, 1, k)\), which compares a direct mapped cache to a fully associative cache, is plotted in section 11.2 together with the limit \(1 - R(k, 1, k)\).

The formula for \(r(n, k, u, q)\) is given in the next theorem. The formulas in the report are defined in terms of certain decreasing sequences of nonnegative integers. For such a sequence \(I = \{i_1, \ldots, i_k\}\) we need the following notation.
\[ b(I) = \text{the number of distinct integers in } I. \]

\[ a(I, j) = \text{the number of occurrences of the } j\text{th distinct integer in } I, \text{ enumerated in size order, } 1 \leq j \leq b(I). \]

\[ d(I, u) = \sum_{i_j > u} i_j, \text{i.e. } d(I, u) \text{ is the sum of all entries in } I \text{ which are larger than } u. \]

**Theorem 4.2** If \( n \leq ku \) or \( u \geq q \), then \( r(n, k, u, q) = 0 \). If \( ku < q \) and \( ku < n \), then \( r(n, k, u, q) = 1 \). For the remaining part of the theorem, these cases are disregarded.

If \( w = n/k \) is an integer, we have

\[
r(n, k, u, q) = \frac{k!}{q^n} \sum_I \binom{w}{i_1} \cdots \binom{w}{i_k} \frac{d(I, u)}{\prod_{j=1}^{b(I)} a(I, j)!}.
\]

The sum is taken over all sequences of non-negative integers \( I = \{i_1, \ldots, i_k\} \) which are decreasing: \( i_j \geq i_{j+1} \) for all \( j = 1, \ldots, k - 1 \), is bounded by \( w \) but have some entry greater than \( u \): \( w \geq i_1 > u \), and have sum \( q : \sum_{j=1}^k i_j = q \).

If \( w = n/k \) is not an integer, we let \( w = \lfloor n/k \rfloor \) and denote the remainder of \( n \) divided by \( k \) by \( n_k \). Then we have:

\[
r(n, k, u, q) = \frac{(k-n_k)!n_k!}{q^n} \sum_{i=\max(0, q-w(k-n_k))}^{\min(q, (w+1)n_k)} \binom{w}{i_1, \ldots, i_k} \prod_{j=1}^{b(I)} \frac{d(I_1, u) + d(I_2, u)}{\prod_{j=1}^{b(I_1)} a(I_1, j)! \prod_{j=1}^{b(I_2)} a(I_2, j)!}.
\]

The sum over \( I_1 \) is taken over all decreasing sequences \( I_1 \) of bound \( w + 1 \), length \( n_k \) and sum \( l \). The sum over \( I_2 \) is taken over all decreasing sequences \( I_2 \) of bound \( w \), length \( k - n_k \) and sum \( q - l \).

This theorem, and Theorem 4.4, are proved in section 9. Here also a fast algorithm which generates all decreasing sequences is given. We remark that the formula can alternatively be formulated with multinomial coefficients and taking the sum over certain partitions of the integer \( k \).

For programs of any number of variables we have the following results:

**Theorem 4.3**

\[
\inf_P h(P, k, u) = 1 - R(k, u, q),
\]

9
where the infimum is taken over all programs $P$ of any number of variables.

Furthermore, an extremal program for this ratio have no reference which is a miss with a $(1,q)$-cache but a hit with a $(k,u)$-cache, when using optimal mappings.

The function $1 - R(k/u,u,k)$ for $u = 1, 2, 4, 8$ is plotted in section 11.1, comparing caches with increasing associativity.

**THEOREM 4.4** If $u \geq q$, then $R(k,u,q) = 0$. If $ku < q$ then $R(k,u,q) = 1$. Otherwise, the formula for $R(k,u,q)$ is given by:

$$R(k,u,q) = \frac{k!(q-1)!}{k^q} \sum_{I} \frac{1}{i_1! \cdots i_k!} \frac{d(I,u)}{\prod_{j=1}^{k} a(I,j)!}.$$  

The sum is taken over all sequences of non-negative integers $I = \{i_1, \ldots, i_k\}$ which are decreasing: $i_j \geq i_{j+1}$ for all $j = 1, \ldots, k-1$, have some entry greater than $u$: $i_1 > u$, and have sum $q : \sum_{j=1}^{k} i_j = q$.

Note that the sequences in this formula only differ from those appearing in Theorem 4.2 in that here the bound of $w$ or $w + 1$ is removed.

We will next describe the results for specific programs from which the generals results of theorems 4.1 to 4.4 follow. These also have some interest in their own right. To this end we next describe the programs which are extremal for the ratio $\max_P \frac{em(P, k, u, q)}{h(P, 1, q)}$: the complete programs.

As discussed previously, a program can be represented by a sequence of $m$ integers, each between 1 and $n$, and each integer occurring at least once since each variable is referenced at least once. In this sequence each entry represents one memory reference, and its value denotes the variable which is referenced.

When executed using a $(1,q)$-cache the program induces a sequence of states for the cache. Our results are established by analysing this sequence of states, we will next describe its representation.

For any given program $P$ we initially add $n$ references, one reference to each of the $n$ variables in the same order as they are initially referenced in the program $P$. This gives the program $P'$ with $m(P') = m(P) + n$. Note that if $n > q$ we add $n$ $(1,q)$-cache misses.

For any reference in the program $P'$ after the initial $n$ references, the state of the cache is described by a so called $q,n$-hit vector. This vector contains one entry for each variable, the content describes whether the variable
is in the cache or not. There are \( n - q \) 0s, for variables not in the cache. Variables in the cache are denoted by "1", except the least recently referenced variable in the cache, which is represented with "q" at the corresponding position.

The program \( P \) is represented by a matrix \( M(P) \) having as rows all induced \( q,n \)-state vectors of \( P' \) which represent hits. If \( n > q \), \( P' \) has at least \( n + 1 \) initial misses, i.e. the number of rows of the matrix \( M(P) \) is less than \( m(P) \). Obviously there exist \( q \binom{n}{q} \) different \( q,n \)-hit vectors.

The raison d'être for the \( q,n \)-hit vectors is that we here represent enough information for our purpose, and that any collection of \( q,n \)-hit vectors represent a program (Theorem 7.12). Therefore results about programs can be derived by studying collections of \( q,n \)-hit vectors.

**DEFINITION 4.5** A program \( P \) is complete if all \( q \binom{n}{q} \) possible \( q,n \)-hit vectors are equally frequent in the matrix \( M(P) \).

**THEOREM 4.6** All complete programs are extremal for the ratio \( \text{em}(P,k,u,q)/h(P,1,q) \), i.e. for each complete program \( P \) we have

\[
\text{em}(P,k,u,q)/h(P,1,q) = \max_{P'} \text{em}(P',k,u,q)/h(P',1,q).
\]

Here the maximum is taken over all programs \( P' \) with the same number of variables as \( P \).

The main part of the report is devoted to the proof of this theorem. In parallel with this we prove the usefulness of the \( q,n \)-hit vector description.

Concerning optimal mappings we have the following theorem, which is essential in order to compute the formulas of theorems 4.2 and 4.4.

**THEOREM 4.7** For any complete program \( P \) of \( n \) variables there is an optimal mapping for a \((k,u)\)-cache where each set in the mapping contains \( \lceil n/k \rceil \) or \( \lfloor n/k \rfloor \) variables.

Clearly, if \( n/k = w \) is an integer there is an optimal mapping where each set has exactly \( w \) variables.

In section 8 we prove that complete programs may have any hit ratio:

**LEMMA 4.8** For any number \( H, 0 \leq H \leq 1 \), and for any \( \epsilon > 0 \), there is a complete program \( P \) such that

\[
\left| \frac{h(P,1,q)}{m(P)} - H \right| < \epsilon.
\]
We conclude this section by showing how Theorem 4.1 and 4.3 follow once Theorem 4.6 and Lemma 4.8 have been established.

**THEOREM 4.9 (i)** For any program \( P \) with \( n \) variables we have

\[
h(P,1,q)(1-r(n,k,u,q)) \leq h(P,k,u).
\]

(ii) For any complete program \( P \) of \( n \) variables we have

\[
h(P,1,q)(1-r(n,k,u,q)) \leq h(P,k,u) < m(P) - h(P,1,q)r(n,k,u,q).
\]

*Proof:* Given a program \( P \), denote by \( A_0 \) and \( A_1 \) optimal mappings for the hit ratio and for the number of extra misses, respectively. From Lemma 3.3 we then get

\[
h(A_0,P,k,u) \geq h(A_1,P,k,u) \geq h(P,1,q) - em(A_1,P,k,u,q).
\]

Hence

\[
h(P,k,u) \geq h(P,1,q) - em(P,k,u,q).
\]

By invoking \( r(n,k,u,q) \geq em(P,k,u,q)/h(P,1,q) \) the inequality in (i) follows.

The right inequality for complete programs follows from the relation

\[
h(A,P,k,u) + em(A,P,k,u,q) < m(P).
\]

To the left we here have all hits and a subset of the misses when using a \((k,u)\)-cache, to the right we have all memory references. Only the misses which are also misses with a \((1,q)\)-cache are not represented. Since the initial reference is a miss for both caches, a strict inequality is valid. By taking optimal mappings \( A_0 \) and \( A_1 \) as before we get

\[
h(A_0,P,k,u) < m(P) - em(A_0,P,k,u,q) \leq m(P) - em(A_1,P,k,u,q).
\]

Now by invoking Theorem 4.6, for complete programs \( P \) we have \( r(n,k,u,q) = em(P,k,u,q)/h(P,1,q) \), the right inequality follows. \( \square \)

The same proof gives the following \( r \)-independent version:

**THEOREM 4.10 (i)** For any program \( P \) we have

\[
h(P,1,q)(1 - R(k,u,q)) \leq h(P,k,u).
\]
(ii) For any complete program $P$ we have
\[ h(P,1,q)(1 - R(k,u,q)) \leq h(P,k,u) < m(P) - h(P,1,q)R(k,u,q). \]

Proof of Theorem 4.1(i) and 4.3: By taking $h(P,1,q)/m(P)$ arbitrarily close to 1, Theorem 4.1(i) and Theorem 4.3 follow immediately from Lemma 4.8 and from Theorem 4.9 and 4.10, respectively.

Proof of Theorem 4.1(ii): Since, by Lemma 4.8, there are complete programs with hit ratio arbitrarily close to $h(P,1,q)/m(P)$, we can use Theorem 4.9(ii). We thus assume that the hit ratio is not measured with infinite accuracy.

For the sake of the optimality we need to prove that for any given program $P$ there is a program $P'$ with hit ratio arbitrarily close to the hit ratio of $P$, and so that the difference
\[ \frac{h(P',k,u)}{m(P')} - \frac{h(P,1,q)}{m(P)}(1 - r(n,k,u,q)) \]
is arbitrarily small.

By Lemma 4.8, for any $\epsilon > 0$ there is a complete program $P''$ so that $h(P'',1,q)/m(P'') > 1 - \epsilon$. Theorem 4.9(ii) then gives $h(P'',k,u)/m(P'') > 1 - r(n,k,u,q)(1 - \epsilon)$. Now let $P'''$ be a program with no cache hits; let for example $P'''$ continuously reference the $n$ variables cyclically. Now we choose the relative size of $P'$ and $P'''$ so that the concatenated program $P' = P'' + P'''$ has hit ratio arbitrarily close to the hit ratio of $P$: $0 < m(P''')/m(P') - h(P,1,q)/m(P) < \epsilon$.

$P'''$ have no hits so $h(P',1,q) = h(P'',1,q) = h(P',1,q)$ and $h(P',k,u) = h(P'',k,u) = h(P',k,u)$. We then get
\[ \frac{h(P',k,u)}{m(P')} = \frac{h(P'',k,u)}{m(P'')} < \frac{m(P''')}{m(P)}(1 - r(n,k,u,q)(1 - \epsilon)) < \]
\[ (\frac{h(P,1,q)}{m(P)} + \epsilon)(1 - r(n,k,u,q))(1 - \epsilon)) < \]
\[ \frac{h(P,1,q)}{m(P)}(1 - r(n,k,u,q)) + \epsilon(1 - r(n,k,u,q)). \]

By taking $\epsilon$ small, the optimality of (ii) follows. \hfill $\square$
5 Method overview

The results in section 4 are established when we have proved the following facts for complete programs:

1. Complete programs are extremal: Theorem 4.6.

2. The formulas in Theorem 4.2 and Theorem 4.4 are valid for complete programs. Part of this is to establish that a complete program has an optimal mapping where all mapping sets have the same number of variables, or as close to this as possible (Theorem 4.7).

3. There are complete programs with any hit ratio: Lemma 4.8.

We start by considering $r(n, k, u, q)$. Having obtained a mathematical formula for this function, $R(k, u, q)$ is obtained in Section 9 by observing that $r(n, k, u, q)$ is increasing as a function of $n$, and then by letting $n$ approach infinity in the formula for $r(n, k, u, q)$.

In order to derive a mathematical formula for $r(n, k, u, q)$ for all positive integers $n, k, q$ and $u$, we start by taking care of trivial cases of integers $n, k, q$ and $u$ (section 6). For the non-trivial cases, we successively take away uninteresting sets of programs with the argument that if a program which is removed is extremal, then there is another program which also is extremal and which is not removed. Thus we always keep at least one program which is extremal for the problem.

We next represent the sequence of states for a $(1, q)$-cache which is induced by the program $P$ with a sequence of so called $q, n$-state vectors. This vector describes the state of the cache and thus contains enough information about the history of $P$ to decide whether the next reference is a hit or a miss.

We have seen that the minimum of the ratio $h(P, k, u)/h(P, 1, q)$ can be obtained by calculating the maximum of $em(P, k, u, q)/h(P, 1, q)$. For this ratio the $(1, q)$-cache misses are uninteresting and not represented in the sequence of $q, n$-state vectors. By frequent use of this flexibility, i.e. by adding $(1, q)$-cache misses in certain ways, it is possible to prove that an arbitrary sequence of $q, n$-state vectors represent a program. Theorem 7.9 thus states that, starting with any $q, n$-state vector, one can add $(1, q - 1)$-cache misses so that the next hit occurs at any other state for the $(1, q - 1)$-cache.

By Theorem 7.11 we can always add cache misses so that the next hit always references the variable which in this time cycle is the least recently
referenced one. This allows the representation to be further simplified into so called $q,n$-hit vectors. The aim of section 7 is Theorem 7.12. By this theorem it is valid to consider arbitrary sequences of $q,n$-hit vectors.

In section 8 complete programs are defined and studied. In this section Theorems 4.6 and 4.7, and Lemma 4.8 are proved. The proof of Theorem 4.6 consists of the so called duplication argument which is a cornerstone in all the reports [7] to [14].

The calculation of the explicit formulas, Theorem 4.2 and 4.4, is done in section 9. Here also an algorithm which generates all sequences appearing in Theorems 4.2 and 4.4 is given (Lemma 9.1). In section 10 the applicability of the results is discussed. Section 11 displays some graphics concerning the results.

Figures 2 and 3 present the logical structure of the report.
It is enough to consider programs as in Theorem 7.1 and where further each variable is initialized during the first \( n \) references.

**Theorem 7.2**

There are extremal programs for the ratio \( \text{em}(P, k, u, q)/\text{h}(P, 1, q) \) where there is room for exactly one variable in each block.

**Theorem 7.1**

No variable is larger than one block.

**Theorem 7.4**

Sequences of \( q,n \)-state vectors contains enough information for our purpose.

**Theorem 7.9**

If \( n > q+1 \) then \( \exists \) an extremal program where each hit is a reference to a variable \( i \) with \( V(i) = q \).

**Theorem 7.11**

For any reference to \( i_0 \) which is a miss with a \((k,u)-cache\) and \( V(i_0) < q \), there is another \( q,n \)-state vector \( V' \) which also is a miss and \( V(i_0) = q \).

**Theorem 7.10**

Figure 3. Logical structure of section 7.

**Lemma 7.8**

\( V_1 \rightarrow V_2 \) with a \((1,q-1)\)-cache if \( V_1 \) and \( V_2 \) are such that \( V_1(i)=0 \) if and only if \( V_2(i)=0 \).

**Lemma 7.7**

\( V_1 \rightarrow V_2 \) with a \((1,q-1)\)-cache if \( V_1(i) = V_2(i) \) for all \( i \) except two indices \( i_1 \) and \( i_2 \) where \( V_1(i_1) = V_2(i_2) = 0 \) and \( V_1(i_2) = V_2(i_1) > 0 \).

**Lemma 7.6**

\( V_1 \rightarrow V_2 \) if \( V_1 \) and \( V_2 \) are such that \( V_1(i)=0 \) if and only if \( V_2(i)=0 \) and for all other \( i \): \( V_1(i) = ((V_2(i)+x) \mod q) + 1 \) for some integer \( x \).

**Definition**

\( q,n \)-hit vector \((V(1),...,V(n))\):
- \( V(i) = 0 \): variable \( i \) not in the cache
- \( V(i) = 1 \): variable \( i \) in the cache, not least recently referenced
- \( V(i) = q \): variable \( i \) in the cache, least recently referenced

**Definition**

\( q,n \)-hit vector \((V(1),...,V(n))\):
- \( V(i) = 0 \): variable \( i \) not in the cache
- \( V(i) = 1 \): variable \( i \) in the cache, not least recently referenced
- \( V(i) = q \): variable \( i \) in the cache, least recently referenced

**Assumptions**

LRU replacement policy

No variable is larger than one block.

**Lemma 7.3**

A reference to the variable \( i_0 \) is a miss with a \((k,u)-cache\) if and only if there are \( v \geq u \) variables \( i_j \) with \( 0 < V(i_j) < V(i_0) \) which are mapped to the same set as \( i_0 \).

**Theorem 7.2**

It is enough to consider programs as in Theorem 7.1 and where further each variable is initialized during the first \( n \) references.

**Theorem 7.11**

If \( n > q+1 \) then \( \exists \) an extremal program where each hit is a reference to a variable \( i \) with \( V(i) = q \).

**Theorem 7.9**

\( V_1 \rightarrow V_2 \) with a \((1,q-1)\)-cache.

**Lemma 7.5**

Take arbitrary \( V_1 \) and \( V_2 \).
Then \( V_1 \rightarrow V_2' \), where \( V_1(i)=0 \) if and only if \( V_2(i)=0 \).

**Lemma 7.7**

\( V_1 \rightarrow V_2 \) with a \((1,q-1)\)-cache if \( V_1 \) and \( V_2 \) are such that \( V_1(i)=0 \) if and only if \( V_2(i)=0 \).

**Lemma 7.8**

\( V_1 \rightarrow V_2 \) with a \((1,q-1)\)-cache if \( V_1(i) = V_2(i) \) for all \( i \) except two indices \( i_1 \) and \( i_2 \) where \( V_1(i_1) = V_2(i_2) = 0 \) and \( V_1(i_2) = V_2(i_1) > 0 \).

**Theorem 7.12**

If \( n > q+1 \), any sequence of \( q,n \)-hit vectors represent a program within the remaining set of programs. This set contains an extremal program.

**Theorem 7.10**

For any reference to \( i_0 \) which is a miss with a \((k,u)-cache\) and \( V(i_0)<q \), there is another \( q,n \)-state vector \( V' \) which also is a miss and \( V(i_0)=q \).
Transformation group: Set of \( q,n \)-hit vectors which have the 0:s in the same positions and where "\( q \)" occurs equally frequently in all remaining positions. This frequency is the repetition factor.

\[ \exists \text{an extremal complete program for a (1,}q\text{-cache).} \]

Lemma 8.4
All complete programs \( P \) have the same ratio \( \text{em}(P,k,u)/h(P,1,q) \).

Theorem 4.7
For any complete program there is an optimal mapping for a \((k,u)\)-cache where each set have size \( \lceil n/k \rceil \) or \( \lfloor n/k \rfloor \).

Lemma 8.3
Let \( P \) be complete and \( A \) any mapping. If \( A' \) has one variable remapped to a smaller set, then \( \text{em}(A',P,k,u,q) \leq \text{em}(A,P,k,u,q) \).

Lemma 8.2
Let \( P \) be complete and \( A \) any mapping. If \( A' \) have two variables switched between sets within \( A' \), then \( \text{em}(A',P,k,u,q) = \text{em}(A,P,k,u,q) \).

Definition 4.5
A program is complete if the corresponding sequence of \( q,n \)-hit vectors representing \( (1,q)\)-cache hits contain all possible \( q,n \)-hit vectors equally frequently.

Theorem 4.8, 4.9
For any program \( P \) of \( n \) var.: 
\[ h(P,1,q)/(1 - R(n,k,u,q)) \leq h(P,k,u) \].
For any complete program \( P \) of \( n \) var.: 
\[ h(P,k,u) \leq m(P) - h(P,1,q)R(n,k,u,q) \].
Version for any number of variables.

Theorem 4.10
Complete programs may have any hit ratio \( h(P,k,u)/m(P) \).

Lemma 8.1
\[ \exists \text{an extremal complete program for a (1,q)-cache.} \]

Theorem 4.6
Complete programs are extremal for the ratio \( \text{em}(P,k,u,q)/h(P,1,q) \).
(duplication argument)

Lemma 8.5
All complete programs \( P \) have the same ratio \( \text{em}(P,k,u,q)/h(P,1,q) \).

Theorem 4.2
Formula for \( r(n,k,u,q) \).

Theorem 4.4
Formula for \( R(k,u,q) \).

Lemma 9.1
Algorithm which generates all decreasing sequences.

Lemma 9.2
\[ r(n,k,u,q) \text{ is increasing in } n. \]

Definition 3.1
\[ h(A,P,k,u) - h(P,1,q) = \text{eh}(A,P,k,u,q) - \text{em}(A,P,k,u,q). \]

Theorem 3.2

Lemma 3.3
\[ \frac{h(A,P,k,u)}{h(P,1,q)} = \text{eh}(A,P,k,u,q) - \text{em}(A,P,k,u,q). \]

Theorem 4.1, 4.3
\[ \inf \text{all } n \text{-programs } P \frac{h(P,k,u)}{h(P,1,q)} = 1 - r(n,k,u,q). \]
\[ \inf \text{all programs } P \frac{h(P,k,u)}{h(P,1,q)} = 1 - R(k,u,q). \]

Optimal hit ratio estimate:
\[ h(P,k,u)/m(P) \geq (1-r(n,k,u,q))h(P,1,q)/m(P). \]

Theorem 4.7
Complete programs are extremal for the ratio \( h(P,k,u)/h(P,1,q) \).

Lemma 9.3
For any program \( P \) of \( n \) var.: 
\[ h(P,1,q)(1 - R(n,k,u,q)) \leq h(P,k,u). \]
For any complete program \( P \) of \( n \) var.: 
\[ h(P,k,u) \leq m(P) - h(P,1,q)R(n,k,u,q). \]
Version for any number of variables.

Theorem 4.8
If \( n > q+1 \), any sequence of \( q,n \)-hit vectors represent a program within the remaining set of programs. This set contains an extremal program.

Theorems 4.1, 4.3
\[ \inf \text{all } n \text{-programs } P \frac{h(P,k,u)}{h(P,1,q)} = 1 - r(n,k,u,q). \]
\[ \inf \text{all programs } P \frac{h(P,k,u)}{h(P,1,q)} = 1 - R(k,u,q). \]

Optimal hit ratio estimate:
\[ h(P,k,u)/m(P) \geq (1-r(n,k,u,q))h(P,1,q)/m(P). \]

Theorem 4.7
Complete programs are extremal for the ratio \( h(P,k,u)/h(P,1,q) \).

Lemma 8.4
All complete programs \( P \) have the same ratio \( \text{em}(P,k,u,q)/h(P,1,q) \).

Theorem 4.6
Complete programs are extremal for the ratio \( \text{em}(P,k,u,q)/h(P,1,q) \).
(duplication argument)

Lemma 8.5
All complete programs \( P \) have the same ratio \( \text{em}(P,k,u,q)/h(P,1,q) \).

Theorem 4.2
Formula for \( r(n,k,u,q) \).

Theorem 4.4
Formula for \( R(k,u,q) \).

Lemma 9.1
Algorithm which generates all decreasing sequences.

Lemma 9.2
\[ r(n,k,u,q) \text{ is increasing in } n. \]

Definition 3.1
\[ h(A,P,k,u) - h(P,1,q) = \text{eh}(A,P,k,u,q) - \text{em}(A,P,k,u,q). \]

Theorem 3.2

Lemma 3.3
\[ \frac{h(A,P,k,u)}{h(P,1,q)} = \text{eh}(A,P,k,u,q) - \text{em}(A,P,k,u,q). \]

Theorem 4.6
Complete programs are extremal for the ratio \( h(P,k,u)/h(P,1,q) \).

Lemma 9.1
Algorithm which generates all decreasing sequences.

Lemma 9.2
\[ r(n,k,u,q) \text{ is increasing in } n. \]

Definition 3.1
\[ h(A,P,k,u) - h(P,1,q) = \text{eh}(A,P,k,u,q) - \text{em}(A,P,k,u,q). \]

Theorem 3.2

Lemma 3.3
\[ \frac{h(A,P,k,u)}{h(P,1,q)} = \text{eh}(A,P,k,u,q) - \text{em}(A,P,k,u,q). \]

Theorem 4.10
Complete programs may have any hit ratio \( h(P,k,u)/m(P) \).

Lemma 8.1
\[ \exists \text{an extremal complete program for a (1,q)-cache.} \]

Definition
Transformation group:
Set of \( q,n \)-hit vectors which have the 0:s in the same positions and where "\( q \)" occurs equally frequently in all remaining positions. This frequency is the repetition factor.

Theorem 7.12

Lemma 7.1
If \( n > q+1 \), any sequence of \( q,n \)-hit vectors represent a program within the remaining set of programs. This set contains an extremal program.
6 Trivial cases

THEOREM 6.1 If $n \leq ku$, then $r(n, k, u, q) = 0$.

Proof: If $n \leq ku$, then all variables can be contained within a cache with $k$ sets containing $u$ blocks each. Consequently, there will be no extra misses for the $(k, u)$-cache.

THEOREM 6.2 If $ku < q$ and $ku < n$, then $r(n, k, u, q) = 1$.

Proof: From the definition of $r(n, k, u, q)$ it is obvious that $r(n, k, u, q) \leq 1$, for any values of $n, k, q$ and $u$.

If $ku < q$, it is possible to write a program $P$ which repeitively accesses a set of variables which requires $q$ blocks of memory space. Since we are using the LRU replacement policy, we will get a cache-miss for every reference using the $(k, u)$-cache. However, for the $(1, q)$-cache we will get no misses after the first iteration. By increasing the number of such iterations, it is possible to make the ratio $em(P, k, u, q)/h(P, 1, q) > 1 - \epsilon$, for any $\epsilon > 0$. Thus $\sup em(P, k, u, q)/h(P, 1, q) = 1$.

THEOREM 6.3 If $u \geq q$, then $r(n, k, u, q) = 0$.

Proof: If $u \geq q$, then each set in the $(k, u)$-cache is larger than the set in the $(1, q)$-cache. Consequently, using the LRU replacement policy, no extra misses can occur.

Having isolated the trivial cases above, unless stated otherwise we assume that $n > ku \geq q > u$, in the rest of this paper. Further, "extremal programs" will unless stated otherwise denote programs which maximize the quantity $em(P, k, u, q)/h(P, 1, q)$.

7 From programs to $q,n$-hit vectors

THEOREM 7.1 For any values of $n, k, u$ and $q$, there is at least one extremal program such that each variable occupies exactly one block.

Proof: The only difference to the statement here compared to the one in Lemma 3.2 is that we here want to maximize $em(P, k, u, q)/h(P, 1, q)$, while we in that proof want to minimize $h(P, k, u)/h(P, 1, q)$. Since maximizing misses are equivalent to minimizing hits, it is easily seen that the same proof
works also for this theorem.

Hence there are extremal programs which has room for exactly one variable in each block, from now on we consider only those.

Since there are \( n \) variables in the program, we know that there will be at least \( n \) cache misses, because each variable will be referenced at least once. The first reference to a variable, which always results in a miss, is referred to as the initial reference.

**THEOREM 7.2** Consider an arbitrary extremal program \( P \), and denote the first referenced variable by \( i_1 \), the secondly referenced variable by \( i_2 \), and so on. Then consider a program \( P' \) which is identical to \( P \) with the exception that a sequence of \( n \) memory references to variables \( i_1, i_2, \ldots, i_n \) in this order has been added at the start of the program. Then \( P' \) is also an extremal program.

Proof: Since we consider the case when \( n > q \), the number of hits using a \((1, q)\)-cache is the same for \( P \) and \( P' \). I.e. memory reference number \( r \) in \( P \) results in a miss using a \((1, q)\)-cache if and only if memory reference number \( r + n \) in \( P' \) results in a miss using a \((1, q)\)-cache. Recall that we use the LRU replacement policy. Hence \( h(P, 1, q) = h(P', 1, q) \).

Consider memory reference number \( r_2 \) in program \( P \), and denote the referenced variable by \( i \). If \( r_2 \) results in a hit, using a \((1, q)\)-cache, then there is another reference \( r_1 \) which also accesses variable \( i \), such that \( r_1 < r_2 \) and there are no references to variable \( i \) between \( r_1 \) and \( r_2 \). The sequence of memory references between \( r_1 \) and \( r_2 \) completely determines if \( r_2 \) will result in a miss using a \((k, u)\)-cache, i.e. this sequence determines if an extra miss will occur. Obviously this sequence corresponds to the sequence \( r_1 + n \) to \( r_2 + n \) in \( P' \). Consequently, the number of extra misses is the same for \( P \) and \( P' \), i.e. \( em(P, k, u, q)/h(P, 1, q) = em(P', k, u, q)/h(P', 1, q) \), where \( m \) is the number of memory references in \( P \). Consequently, if \( P \) is an extremal program, \( P' \) is also an extremal program.

Hence there are extremal programs which has no hits during the \( n \) first memory references using a \((1, q)\)-cache, from now on we consider only those.

A \( q, n \)-state vector \( V \) defines the variables currently stored in a \((1, q)\)-cache, and the order in which they have been referenced. \( V \) has one entry for each variable. If variable \( i \) is not in the cache then \( V(i) = 0 \). Otherwise \( V(i) \) denotes the number of variables referenced since variable \( i \) was last referenced, including variable \( i \). Hence if variable \( i \) is the most recently
accessed variable, \( V(i) = 1 \). If it is the second most recently accessed variable, \( V(i) = 2 \), and so on (see Figure 4). Hence after the first \( n \) memory references the \( q, n \)-state vector consists of the \( q \) first positive integers and \( n - q \) zeros. As discussed before, the LRU replacement policy is used.

Clearly, a \( q, n \)-state vector has \( n \) entries and contains each of the first \( q \) positive integers exactly once, and \( n - q \) zeros. If reference to variable \( j \) is a cache miss, and \( V(i) = q \), then variable \( i \) is replaced in the cache. We then put \( V(i) = 0 \), increment all positive integers and put \( V(j) = 1 \). If the reference to variable \( j \) is a hit we increment all entries which are smaller than \( V(j) \), and then put \( V(j) = 1 \). Figure 4 shows a \( q, n \)-state vector for the case when \( q = 4 \) and \( n = 7 \).

We are interested in the ratio \( \varepsilon(P, k, u, q) / h(P, 1, q) \) for extremal programs. Using a \( (1, q) \)-cache there are no hits during the first \( n \) memory references. Therefore, we do not consider the \( n \) first memory references, i.e. we assume that there are always \( q \) non-zero entries in a \( q, n \)-state vector. The numbering of the \( q, n \)-state vectors is chosen so that the \( q, n \)-state vector number \( r \) describes the state of the cache immediately before reference \( r \) in the program.

Now consider a mapping \( A \) of the \( n \) variables to \( k \) sets. Each mapping can be regarded as a partition of the first \( n \) positive integers in \( k \) sets \( A_j \), a mapping \( A \) is thus the sets \( A_1, ..., A_k \).

**Lemma 7.3** Consider a \( (k, u) \)-cache, a mapping \( A \) and a \( q, n \)-state vector \( V \) corresponding to reference \( r \) in a program \( P \). Assume that \( i_0 \in A_0 \).

Then the variable \( i_0 \) is not in the \( (k, u) \)-cache at reference \( r \) in program \( P \) if and only if there are \( j \) variables \( i_j \), which are also mapped to \( A_0 \), such that \( 0 < V(i_j) < V(i_0) \) and \( j \geq u \).
Proof: This is a direct consequence of the definition of the $q,n$-state vector and of the fact that the LRU replacement policy is used. □

The next theorem states that a $q,n$-state vector contain enough information about the history of the program for our purpose.

**THEOREM 7.4** If the $q,n$-state vector after $r$ references and the mapping of variables to sets are known, then this is possible to determine from this information whether reference $r$ is an extra miss or not.

Proof: Consider a $q,n$-state vector $V$, a mapping $A$ and a reference to a variable $i_0$. If $V(i_0) > 0$, no miss will occur using a $(1,q)$-cache. This is a direct consequence of the definition of the $q,n$-state vector.

Assume that $i_0 \in A_n$ using the $(k,u)$-cache and there are $j$ variables $i_j$, which are also mapped onto $A_n$. In that case the previous lemma guarantees that if $0 < V(i_j) < V(i_n)$ and $j \geq u$, a cache miss will occur using the $(k,u)$-cache and mapping $A$.

Consequently, if the $q,n$-state vector and the mapping of variables to memory blocks are known, it is possible to determine if an extra miss will occur.

□

Our next aim is to prove that any sequence of $(q,n)$-state vectors represents the sequence of $(1,q)$-hits for some program (Theorem 7.9). This allows us to consider arbitrary sequences of $(q,n)$-state vectors.

Two $q,n$-state vectors $V$ and $V'$ are said to be congruent if $V(i) = V'(i)$, for all $i$ such that $V(i) = 0$.

**LEMMA 7.5** For two arbitrary $q,n$-states vectors $V_1$ and $V_2$, it is possible to generate a sequence of memory references, such that, starting from $V_1$, we reach a state vector which is congruent with $V_2$ without having any hits using a $(1,q)$-cache.

Proof: Generate a reference to the first variable $i$ (the variables are enumerated in some fixed arbitrary order) for which $V_1(i) = 0$ and $V_2(i) \neq 0$, thus obtaining a new $q,n$-state vector $V'$. If $V'$ and $V_2$ are not congruent we again issue a reference to a variable $i$ where $V'(i) = 0$ and $V_2(i) \neq 0$. This procedure can be repeated until we reach a vector $V'$ which is congruent with $V_2$.

For the sequence of memory references so generated there are no hits using a $(1,q)$-cache. □
Consider two congruent \(q, n\)-state vectors \(V\) and \(V'\). The \(q\) indexes \(i_j\) for which \(V(i_j) \neq 0\) are enumerated in such order that \(V(i_1) = 1, V(i_2) = 2, \ldots, V(i_q) = q\). \(V\) and \(V'\) are modular congruent if there is an integer \(x:\)
\[
V(i_j) = ((V'(i_j) + x) \mod q) + 1\]
for all \(j : 1 \leq j \leq q\).

**Lemma 7.6** For two modular congruent \(q, n\)-state vectors \(V_1\) and \(V_2\), it is possible to generate a sequence of memory references such that, starting from \(V_1\), we reach \(V_2\) without having any hits using a \((1, q - 1)\)-cache.

**Proof**: If we issue a reference to the variable \(i_q : V_1(i_q) = q\), we will obtain a new state vector \(V_1'\) which is modular congruent to \(V_1\). Also this reference will clearly be a miss when using a \((1, q - 1)\)-cache. By repeating this kind of reference we can generate any vector \(V\) which is modular congruent to \(V_1\) with no hits when using a \((1, q - 1)\)-cache. \(\square\)

Two \(q, n\)-state vectors \(V_1\) and \(V_2\) are quasi-identical if \(V_1(i) = V_2(i)\) for all \(i\) except two indexes \(i_a\) and \(i_b\) such that \(V_1(i_a) = 0, V_2(i_b) > 0, V_1(i_b) = V_2(i_b)\) and \(V_1(i_a) = V_2(i_a)\).

**Lemma 7.7** For two quasi-identical \(q, n\)-state vectors \(V_1\) and \(V_2\), it is possible to generate a sequence of memory references, such that, starting from \(V_1\), we reach \(V_2\) without having any hits using a \((1, q - 1)\)-cache.

**Proof**: Consider two states vectors \(V_1'\) and \(V_2'\) which are modular congruent with \(V_1\) and \(V_2\) respectively. \(V_1'\) and \(V_2'\) have been chosen in such a way that \(V_1'(i_a) = q\) and so that \(V_1'\) and \(V_2'\) are also quasi-identical. By issuing one reference to the variable \(i_a\) with \(V_2'(i_a) = q\) we obtain \(V_2''\) which is modular congruent to \(V_2\).

From Lemma 7.6 we know that it is possible to generate a sequence of memory references, such that, starting from \(V_1\) it is possible to reach \(V_1'\) without having any hits using a \((1, q - 1)\)-cache. From \(V_1'\) we get \(V_2''\) by issuing a reference to variable \(i_a\). Since \(V_1'(i_a) = 0\), this reference leads to a miss using a \((1, q - 1)\)-cache. Again, Lemma 7.6 guarantees that, starting from \(V_2''\), it is possible to generate a sequence of memory references, such that we reach \(V_2\) without having any hits using a \((1, q - 1)\)-cache. \(\square\)

**Lemma 7.8** For two congruent \(q, n\)-state vectors \(V_1\) and \(V_2\), it is possible to generate a sequence of memory references, such that, starting from \(V_1\), we reach \(V_2\) without having any hits using a \((1, q - 1)\)-cache.
Proof: Assume that $V_1(z) = V_2(z) = 0$, and that $V_2(ij) = j$ for all $j : 1 \leq j \leq q$. In that case, the algorithm below reaches $V_1' = V_2$, starting from $V_1$, without generating any hit using a $(1,q-1)$-cache.

1. Let $j = q$ and let $V_j'$ be modular congruent to $V_1$ such that $V_j'(ij_q) = q$.

2. If $j = 1$ the algorithm terminates, else let $j = j - 1$.

3. If $V_j'(ij) = j$, then $V_j' = V_{j+1}'$ and go to 2.

4. Go to state $X_j$, which is quasi-identical with $V_j'+1$, i.e. $X_j(i) = V_j'(i)$; for all $i : 1 \leq i \leq n$ except that $X_j(i_j) = V_j'(i_j) = 0$ and $X_j(z) = V_j'(i_j) = t$ ($t < j$).

5. Go to state $X_j'$, which is quasi-identical with $X_j$, i.e. $X_j'(i) = X_j(i)$; for all $i : 1 \leq i \leq n$ except that $X_j'(i_y) = X_j(i_j) = 0$ and $X_j'(i_j) = X_j(i_y) = j$, i.e. $i_y$ is the index for which $X_j(i_y) = j$.

6. Go to state $V_j'$, which is quasi-identical with $X_j'$, i.e. $V_j'(i) = X_j'(i)$ for all $i : 1 \leq i \leq n$ except that $V_j'(z) = X_j'(i_y) = 0$ and $V_j'(i_y) = X_j'(z) = t$, n.b. $V_j'$ is congruent with $V_1$ and $V_2$.

7. Go to 2.

The algorithm above guarantees that $V_1'(i) = V_2(i)$ for all $i : j \leq i \leq n$. Moreover, Lemma 7.7 guarantees that there will be no hits using a $(1,q-1)$-cache during steps 3, 4 and 5 in the algorithm above.

**THEOREM 7.9** If $n > q+1$, it is possible to go from any $q,n$-state vector $V_1$ to any other $q,n$-state vector $V_2$ without having any hits in a $(1,q)$-cache.

Proof: This follows immediately from Lemma 7.5 and 7.8. □

For the moment we only consider cases, in which $n > q + 1$. The case when $n = q + 1$, will be handled later. As discussed previously, a program can be represented by the sequence of memory references with associated $q,n$-state vectors corresponding to hits using a $(1,q)$-cache. From Theorem 7.9 we know that if $n > q + 1$, each $q,n$-state vector in this sequence is completely independent from any other vector in the sequence.
**THEOREM 7.10** Consider a mapping $A$ and two memory references $r_1$ and $r_2$, both referring to the same variable $x$. Denote the $q,n$-state vectors corresponding to $r_1$ and $r_2$ respectively by $V_1$ and $V_2$. Assume that these vectors are such that $0 < V_1(x) = p < q$, $V_2(x) = q$, $V_2(i) = V_1(i)$, when $V_1(i) < p$, otherwise $V_2(i) = V_1(i) - 1$ for all $i \neq x$. Assume also that reference $r_1$ is a miss using a $(k,u)$-cache and a mapping $A$. Then reference $r_2$ is also a miss using a $(k,u)$-cache and the mapping $A$.

*Proof:* From Lemma 7.3 we know that if reference $r_1$ leads to a miss using a $(k,u)$-cache and mapping $A$, then there are $i$ variables $z_i$, which are mapped to the same set as $x$, such that $0 < V(z_i) < V(x) = p$ and $i \geq u$. We also know that $V_2(i) = V_1(i)$, when $V_1(i) < p$. Consequently, at reference $r_1$ the same $z_i$ variables are mapped to the same set as $x$. Under these conditions, Lemma 7.3 tells us that reference $r_2$ will result in a cache miss.

**THEOREM 7.11** If $n > q + 1$ there exists an extremal program such that each hit using a $(1,q)$-cache is a reference to the variable for which the corresponding entry in the $q,n$-state vector equals $q$.

*Proof:* Consider a program $P$ and its induced sequence of $q,n$-state vectors. If a reference $r$ occurs to a variable $i$ with $V(i) < q$, it is possible by Theorem 7.9 to add memory references immediately before reference $r$ so that $V'(i) = q$, where $V'$ is the new $q,n$-state vector preceding reference $r$. If this is done for all $i$ such that $V(i) < q$ we obtain a new program $P'$ which has the same number of hits using a $(1,q)$-cache: $h(P,1,q) = h(P',1,q)$. It remains to check that the number of extra misses does not decrease. By Theorem 7.9 it is possible to add memory references in such a way that $V'$ fulfills all conditions in Theorem 7.10, without new hits. It then follows from Theorem 7.10 that all extra misses in $P$ are extra misses also in $P'$. If sequences of memory references are added in this way at each spot where a reference $r$ occurs to a variable $i$ with $V(i) < q$, then from the program $P$ we obtain a program $P'$ where each hit is a reference to a variable $i$ with $V(i) = q$, and $em(P,k,u,q)/h(P,1,q) = em(P',k,u,q)/h(P',1,q)$. The theorem is proved.

From now on we only consider programs for which each hit using a $(1,q)$-cache is a reference to the variable for which the corresponding entry in the current $q,n$-state vector equals $q$. There is at least one extremal program within this set. However, the ratio $em(P,k,u,q)/h(P,1,q)$ is unchanged if
all \( q, n \)-state vectors corresponding to misses using a \((1, q)\)-cache are deleted - both quantities in the ratio deals with \((1, q)\)-cache hits. It is clear that each arbitrary sequence of \( q, n \)-state vectors represent a program of this kind, having all misses removed. Hence we may look for extremal programs among the set of arbitrary sequences of \( q, n \)-state vectors.

Now consider a fixed mapping \( A \), i.e. a partition of the variables in \( k \) sets consisting of \( u \) variables in each set. We want to find how many of the \( q, n \)-state vectors which result in a miss for a \((k, u)\)-cache, this is the number of extra misses. Each reference to a variable \( i \) corresponds to an entry equalling \( q \) in the current \( q, n \)-state vector, \( V(i) = q \). We know that the variable \( i \) is discarded using a \((k, u)\)-cache if the number of variables \( j : V(j) > 0 \) belonging to this partition set is larger than \( u \). This is a necessary and sufficient condition for an extra miss. Since we thus are only interested in whether a certain entry in a \( q, n \)-state vector is \( 0, q \) or some value between \( 0 \) and \( q \), we can further simplify our notation. In a \( q, n \)-state vector “0”, “1”, and “q” are left unchanged, but all other integers are replaced by “1”. Hence each vector has one “q”, \( q - 1 \) “1” and \( n - q \) “0”. Such a vector is referred to as a \( q, n \)-hit vector. Also this vector contains enough information to be able to decide whether an extra miss occurs or not.

We will in the following often represent a program as a matrix which has \( q, n \)-hit vectors as rows. Again only the hits are represented, all \( q, n \)-hit vectors representing misses with a \((1, q)\)-cache are deleted.

Since a program by definition has \( h(P, 1, q) \) hits using a \((1, q)\)-cache, we obtain an \( h(P, 1, q) \times n \) matrix. Figure 5 shows a program for which there are five cache hits using a \((1, 3)\)-cache, i.e. \( h(P, 1, q) = 5 \). The program has six variables. The first cache hit that occurs is a reference to variable number three. When this hit occurs variables number one and two are also
The example in Figure 5 illustrates that if the mapping of variables to sets is known, it is possible to determine if the memory reference corresponding to a certain \( q,n \)-hit vector will result in a miss using a \((k,u)\)-cache or not. Remember, we are interested in the mapping of variables to sets which results in a minimum number of misses using a \((k,u)\)-cache. The figure also shows that the number of extra misses depends on the mapping.

The problem of calculating \( r(n,k,u,q) = \max_P em(P,k,u,q)/h(P,1,q) \) for all programs \( P \) of \( n \) variables can now be formulated as to find a sequence of \( q,n \)-hit vectors for which the ratio obtained when dividing the minimum number of misses for the sequence, using a \((k,u)\)-cache, with the length of the sequence, is maximal.

From Theorem 7.11 and 7.4 the following theorem follows, which summarizes the aim of the present section to reformulate the problem into arbitrary sequences of \( q,n \)-hit vectors:

**THEOREM 7.12** If \( n > q + 1 \), any sequence of \( q,n \)-hit vectors represents a program.

It is obvious that any program with \( n \) variables corresponds to a sequence of \( q,n \)-hit vectors, as described before Definition 4.5.

8 Complete programs

There are \( q \binom{n}{q} \) distinct \( q,n \)-hit vectors. A program for which the corresponding sequence of \( q,n \)-hit vectors contains exactly \( x \) copies of each distinct vector is referred to as a complete program; \( x \) is referred to as the
repetition factor. Such a sequence of \( q,n \)-hit vectors is referred to as a complete sequence.

A set of \( q,q,n \)-hit vectors form a transformation group if all zeros are positioned at the same entries in all vectors and if the “\( q \)” is positioned differently in all vectors. Consequently, there are \( x \left( \begin{array}{c} n \\ q \end{array} \right) \) transformation groups in a complete program.

Figure 6 shows a \( q,n \)-state vector \( V \), and the corresponding \( q,n \)-hit vector \( V \). The figure also shows the transformation group \( T \), in which \( V \) is a member. The arrows in the right part of the figure correspond to a sequence of four memory references, where the first reference is to a variable mapped to block 5, the second reference is to a variable mapped to block 6, the third to block 7 and the fourth to block 2. These four memory references go through all \( q,n \)-hit vectors in \( T \) exactly once, reaching \( q,n \)-hit vector \( V \) again after the fourth memory reference. Using a \( (1,4) \)-cache, none of the four memory references would result in a cache miss.

So far, we have only considered the case when \( n > q + 1 \). The reason for this is that if \( n = q + 1 \), we cannot guarantee that there is a program corresponding to any arbitrary sequence of \( q,n \)-hit vectors (see theorem 7.12). However, in the next theorem we will show that there are complete programs even when \( n = q + 1 \).

**THEOREM 8.1** If \( n = q + 1 \), then there is a program \( P \) such that the sequence of \( q,n \)-hit vectors corresponding to hits using a \( (1,q) \)-cache forms a complete sequence.
Proof: We are going to describe a complete program $P$ with $n = q + 1$ variables.

The program $P$ starts by generating the $n - 1$ memory references $2, 3, ..., q + 1$. This gives cache misses only and the $q, n$-hit vector $V = (0, q, 1, ..., 1, 1)$. The following $q$ references are references to the variables $2, ..., q, q + 1$ in this order. By the definition of transformation groups we then clearly issue cache hits such that we go through all $q, n$-hit vectors in the transformation group $T$ of $V$ exactly once, finally returning to $V$.

Having done this we generate a memory reference to variable 1. This issue gives a miss and a new $q, n$-hit vector $V'$ belonging to a different transformation group $T'$, where $V'(1) = 0$. Again we generate a sequence of $q$ memory references, such that we go through all $q, n$-hit vectors in $T'$ exactly once, finally returning to $V'$ again with no misses using a $(1, q)$-cache. This procedure is repeated until we have got all the $q + 1$ different transformation groups.

Clearly the cache hits in the program $P$ correspond to all possible $q, n$-hit vectors, and each $q, n$-hit vector occurs once. Consequently, $P$ is a complete program for the case when $x$ is one. Note that by repeating the $(q + 1)^2$ memory references it is possible to obtain a complete program with any arbitrary value on $x$.

Hence there are complete programs when $n = q + 1$.

Proof of Theorem 4.8: Consider a complete program $P$, for which the hits occur in such order that $q, n$-hit vectors belonging to the same transformation group are listed consecutively. If $h(P, 1, q)/m(P) > H$ we add $y$ cache misses at the end of the program, thus obtaining a new program $P'$ which also is complete since only cache hits are involved in the definition of complete programs. Let $h(P', 1, q) = h(P, 1, q) + y$. If $h(P, 1, q)/m(P) < H$, we consider another complete program $P'$ where we have increased the repetition factor from $x$ to $x'$, still with $q, n$-hit vectors belonging to the same transformation group listed consecutively. Since only hits occur within a transformation group, we thus introduce no new misses. Let $\alpha = \frac{x'}{x}$. Denoting $b(P) = m(P) - h(P, 1, q)$, the misses using a $(1, q)$-cache, and $\alpha = \frac{x'}{x}$, it is clear that

\[
\frac{h(P', 1, q)}{m(P')} = \frac{\alpha h(P, 1, q)}{\alpha h(P, 1, q) + b(P')}.
\]
Hence we can increase the ratio arbitrarily close to 1. By repeating the two techniques it is possible to find a complete program which has hit ratio arbitrarily close to $H$. \hfill $\Box$

**Lemma 8.2** Consider a complete program $P$. The variables in $P$ include $i_1$ and $i_2$. If $A_1(i_1) = S_1$ and $A_1(i_2) = S_2$ and there is another mapping $A_2$ which is identical to $A_1$ except $A_2(i_1) = S_2$ and $A_2(i_2) = S_1$, then the number of extra misses is the same for mappings $A_1$ and $A_2$.

**Proof:** Obviously, the number of misses using a $(k, u)$-cache for vectors $V$, such that $V(i_1) = V(i_2)$ is the same for mappings $A_1$ and $A_2$. The symmetry of complete sequences guarantees that all $q, n$-hit vectors $V$ in $P$ for which $V(i_2) \neq V(i_1)$ can be grouped into pairs $(V, V')$, such that $V(i) = V'(i)$, except that $V(i_1) = V'(i_2)$ and $V(i_2) = V'(i_1)$. If $V$ results in a miss for mapping $A_1$ using a $(k, u)$-cache, then $V'$ results in a miss for mapping $A_2$ using a $(k, u)$-cache. Similarly, if $V$ results in a miss for mapping $A_2$ using a $(k, u)$-cache, then $V'$ results in a miss for mapping $A_1$ using a $(k, u)$-cache. Consequently, the number of extra misses is the same for $A_1$ and $A_2$. \hfill $\Box$

**Lemma 8.3** Consider a complete program $P$. If there are $n_1$ variables mapped onto set $S_1$ and $n_2$ variables mapped onto set $S_2$ and $n_1 + 1 < n_2$, the number of extra misses cannot decrease if one variable is remapped from $S_2$ to $S_1$.

**Proof:** Consider a mapping $A_1$, such that there are $n_1$ variables mapped onto set $S_1$ and $n_2$ variables mapped onto set $S_2$, where $n_1 + 1 < n_2$. From Lemma 8.2 we know that only the number of variables mapped to a each set is interesting. In this case the variables are mapped in such a way that $A_1(i_1) = S_1, A_1(i_2) = S_1, \ldots, A_1(i_{n_1}) = S_1, A_1(i_{n_1+1}) = S_2, \ldots, A_1(i_{n_1+n_2}) = S_2, A_1(i_{n_1+n_2+1}) = S_3$ and so on. Consider also another mapping $A_2$ which is identical to $A_1$ except that $A_2(i_{n_1}) = S_2$. We are going to show that the number of extra misses cannot be larger using $A_2$ than $A_1$.

Obviously, the number of misses for vectors $V$, such that $V(i_{n_1}) = 0$ is the same for mappings $A_1$ and $A_2$, i.e. we only have to consider vectors $V$ for which $V(i_{n_1}) \neq 0$. We now introduce the concept of $(n_1, n_2)$-permutations. $V'$ is by definition an $(n_1, n_2)$-permutation of $V$ if $V'(i_j) = V(i_{n_1+n_2+1-j})$ and $V'(i_{n_1+n_2+1-j}) = V(i_j)$ for all $j : 1 \leq j \leq n_2$, and all other entries are equal: $V'(i_j) = V(i_j)$ for all $j : n_1 + 1 \leq j \leq n_2$ and $n_1 + n_2 + 1 \leq j \leq n$.

If $V(i_{n_1}) = "q"$ and $V(i_j) = "1"$ for all $j : n_1 + 1 \leq j \leq n_2$, then $V$ may, using a $(k, u)$-cache, lead to a miss using mapping $A_2$ even if $V$ did
not cause a miss using mapping $A_1$. The symmetry of complete programs
guarantee that each such vector $V$ can be grouped together with another
vector $V'$, forming a pair $(V, V')$ such that $V'$ is an $(n_1, n_2)$-permutation of
$V$. From the definition of $(n_1, n_2)$-permutations and we know that if, using
a $(k, u)$-cache, vector $V$ leads to a miss using mapping $A_2$ and a hit using
mapping $A_1$, then $V'$ leads to a miss using mapping $A_1$ and a hit using
mapping $A_2$.

We now introduce the function #a$(V, [x, y])$, i.e. there are two param-
eters: a $q, n$-hit vector $V$ and a range $[x, y]$. The value of this function
is a natural number indicating the number of "1"s in $V$ in the interval
$V(i_x) \ldots V(i_y)$.

If $V(i_{n_1}) = "1"$ and #a$(V_i, [1, n_1]) < #a(V_i, [n_1 + 1, n_1 + n_2])$, then $V$
may, using a $(k, u)$-cache, lead to a miss using mapping $A_2$ even if $V$
did not cause a miss using mapping $A_2$. The symmetry of complete programs
guarantee that each such vector $V$ can be grouped together with another
vector $V'$, forming a pair $(V, V')$ such that $V'$ is an $(n_1, n_2)$-permutation
of $V$. From the definition of $(n_1, n_2)$-permutations we know that if, using
a $(k, u)$-cache, vector $V$ leads to a miss using mapping $A_2$ and a hit using
mapping $A_1$, then, using a $(k, u)$-cache, $V'$ leads to a miss using mapping
$A_1$ and a hit using mapping $A_2$. \hfill \Box

Proof of Theorem 4.7: By repeating the argument of the preceeding
theorem we finally obtain an optimal mapping of the type described in
the theorem. \hfill \Box

**Lemma 8.4** All complete programs $P$ with $n$ variables have the same ratio
$em(P, k, u, q) / h(P, 1, q)$.

**Proof:** Obviously, the ratio $em(P, k, u, q) / h(P, 1, q)$ is not affected by the
order in which the $q, n$-hit vectors are listed in the sequence corresponding
to the complete program. Therefore, the only potentially significant differ-
ence between two complete programs is the repetition factor $x$. Creating $x$
copies of each row, increases both the number of extra misses $em(P, k, u, q)$
and the number of hits using a $(1, q)$-cache $h(P, 1, q)$ with a repetition fac-
tor $x$. Consequently, the ratio $em(P, k, u, q) / h(P, 1, q)$ is not affected by
$x$, i.e. all complete programs $P$ with $n$ variables have the same ratio
$em(P, k, u, q) / h(P, 1, q)$, where $m$ is the number of memory references in
$P$. \hfill \Box

28
Proof of Theorem 4.6: Consider a sequence of $q,n$-hit vectors corresponding to an arbitrary program $P_0$ (see Figure 7), and take a mapping $A_0$ which is optimal for the program $P_0$. Let further $A_1$ be any mapping where a set with most variables have at most one more variable than a set with fewest variables. We then clearly have

\[
\frac{em(P_0, k, u, q)}{h(P_0, 1, q)} = \frac{em(A_0, P_0, k, u, q)}{h(P_0, 1, q)} \leq \frac{em(A_1, P_0, k, u, q)}{h(P_0, 1, q)}.
\]

As discussed previously, the $q,n$-hit vectors form a $h(P_0, 1, q) \times n$ matrix; in the example in Figure 7 we have $h(P_0, 1, q) = 4$ and $n = 3$.

We next duplicate the program $P_0$ in $n!$ copies, where each copy corresponds to one of the $n!$ permutations of the $n$ columns in the $h(P_0, 1, q) \times n$ matrix. The copies are concatenated, resulting in a sequence of $q,n$-hit vectors of length $n! \times h(P_0, 1, q)$. This sequence corresponds to a program $P_1$. It is clear that $P_1$ is a complete program (see the rightmost version of $P_1$ in Figure 7).
Now the mapping $A_1$ on each of the $n!$ permutations of the columns of $P_0$ can alternatively be regarded as mappings $A_{1,i}, i = 1,\ldots,n!$ on the program $P_0$. Here all mappings $A_{1,i}$, some of which may be equal, have all the property that the sizes of the sets as equal as possible. Now fix the mapping $A_1$ to be such that $em(A_1, P, k, u, q) \leq em(A_{1,i}, P, k, u, q)$ for all $i = 1,\ldots,n!$.

Then $em(A_1, P, k, u, q)$ is a lower bound also of the mean value of the quantities $em(A_{1,i}, P_0, k, u, q)$. Since the sum of these is $em(A_1, P_1, k, u, q)$,
we get
\[
\frac{em(A_1, P_0, k, u, q)}{h(P_0, 1, q)} \leq \frac{1}{n!} \sum_{i=1}^{n!} \frac{em(A_1, i, P_0, k, u, q)}{h(P_0, 1, q)} = \frac{em(A_1, P_1, k, u, q)}{n! h(P_0, 1, q)} = \frac{em(A_1, P_1, k, u, q)}{h(P_1, 1, q)}.
\]

Further, from Lemma 8.4 we know that all complete programs \(P\) with \(n\) variables have the same ratio \(em(P, k, u, q)/h(P, 1, q)\). By this and Theorem 4.7, mappings as \(A_1\) are optimal mappings. Thus, for an arbitrary program \(P_0\);
\[
\frac{em(P_0, k, u, q)}{h(P_0, 1, q)} \leq \frac{em(A_1, P_1, k, u, q)}{h(P_1, 1, q)} \leq \frac{em(P_1, k, u, q)}{h(P_1, 1, q)}
\]
where \(P_1\) is complete. Hence complete programs are extremal. \(\square\)

9 Calculating \(r(n, k, u, q)\) and \(R(k, u, q)\)

Proof of Theorem 4.2: By Theorem 6.1, 6.2 and 6.3 we know that if \(n \leq ku\) or \(u \geq q\), then \(r(n, k, u, q) = 0\), and if \(ku < q\) and \(ku < n\), then \(r(n, k, u, q) = 1\). In the remaining part of the proof we disregard from these cases.

By Theorem 4.6 and 4.7 and Lemma 8.4 we may consider a matrix where each permutation appears exactly once, and a partition where the partition sets have equal sizes, or as close to this as possible. We first consider the case when \(w = n/k\) is an integer. Consider a complete matrix with repetition factor \(x = 1\). Each permutation group will be represented by a vector with all 1s and 0s, also the entry \(q\) is here replaced by “1”. There are obviously \(\binom{n}{q}\) such vectors. Now the columns are partitioned into a fixed partition of \(k\) sets of size \(w\) each. For a specific vector of 1s and 0s we consider the entry \(q\) replacing each of the 1s. For each “1” appearing in a set with \(u\) 1s thus an extra miss will occur. Hence the number of 1s appearing in sets which contain at least \(u\) 1s is the number of extra misses represented by that vector of 1s and 0s.

The set of all permutations are partitioned in subsets according to the number of 1s in each of the \(k\) sets. Each subset of permutations is represented by a decreasing sequence of integers. Each sequence \(I\) consists of \(k\) integers, \(I = \{i_1, \ldots, i_k\}\) where \(0 \leq i_j \leq w\), and with sum \(q : i_1 + \ldots + i_k = q\). Clearly \(i_j\) represents the number of 1s in set \(j\). All permutations in a subset generate the same number of extra misses: \(d(I, u) = i_1 + \ldots + i_c\), where \(i_c\) is
the last entry where \( i_c > u \). Hence, for each permutation in the permutation set corresponding to the decreasing sequence \( I \), the ratio of extra misses is \( d(I, u) / q \).

The number of permutations corresponding to a specific decreasing sequence is computed in two steps. First the 1s in each set are permuted, which gives rise to a factor \( \binom{w}{i_1} \cdots \binom{w}{i_k} \). Next the sets are permuted, which gives the factor \( k! \). But sets with equal number of 1s need not be permuted, in order to count each permutation exactly once. We thus need to divide by the number of permutations of the sets which has the same number of 1s.

This gives the factor \( \frac{d(I, u)}{u} \).

If \( n/k \) is not an integer we assume that the first \( n_k \) subsets contain \( w + 1 \) columns, and the other \( k - n_k \) contains \( w \) columns each. We next consider permutations with a maximum number of 1s in the sets of size \( w + 1 \), and the remaining 1s in the sets of size \( w \). The quantity \( d(I, u) \) is computed and multiplied with the number of such permutations, which is obtained as above for the sets of size \( w + 1 \) separately times this number for the sets of size \( w \). Next the same procedure is repeated after having moved one of the 1s from the sets of size \( w + 1 \) to the smaller sets. The procedure is repeated until either the sets of size \( w + 1 \) run out of 1s or the sets of size \( w \) run out of 0s.

The limits of the sums exclude cases when there are no permutations, i.e., we are avoiding summing zeros. For example, if \( \max(0, q - w(k - n_k)) > 0 \), then we have in total \( w(k - n_k) \) slots in the sets of size \( w \). Thus if \( q > w(k - n_k) \), packing this side full of 1s will still leave \( q - w(k - n_k) \) 1s for the sets of size \( w + 1 \). Since \( l \) is the number of 1s here, \( l \) should thus start with \( q - w(k - n_k) \). Also, the terms corresponding to sequences when both \( I_1 \) and \( I_2 \) have all entries smaller than \( u \) can be omitted, since then \( d(I_1, u) + d(I_2, u) = 0 \).

The decreasing sequences can easily be generated by the algorithm described in the following lemma. We say that the least decreasing sequence of length \( k \) and sum \( q \) is the sequence \( \{ \lfloor \frac{q}{k} \rfloor, \ldots, \lfloor \frac{q}{k} \rfloor, \lfloor \frac{q}{k} \rfloor, \ldots, \lfloor \frac{q}{k} \rfloor \} \). If \( q_k \) is the remainder when \( q \) is divided by \( k \), the number of \( \lfloor \frac{q}{k} \rfloor \)s is \( q_k \), and the number of \( \lfloor \frac{q}{k} \rfloor \)s is \( k - q_k \), making the sum of the sequence \( q \).

**Lemma 9.1** Let \( q, u \) and \( k \) be positive integers with \( u < q \).

Every sequence \( I = \{ i_1, \ldots, i_k \} \) of \( k \) non-negative integers which is decreasing, has \( i_1 > u \) and has sum \( q \) is generated exactly once by the following algorithm:
1. As starting sequence take $i_1 = u + 1$ and take $\{i_2, ..., i_k\}$ as the least decreasing sequence of length $k - 1$ and sum $q - u - 1$.

2. Find the rightmost position in $I$, say $j_1$, which fulfills:
   
   (a) $i_{j_1} < i_{j_1-1}$ or $j_1 = 1$
   
   (b) $i_{j_1+1} > 0$

   The algorithm terminates if no such $j_1$ can be found.

3. The next sequence is obtained from $I$ by increasing the entry in position $j_1$ by one, and replacing the subsequence $\{i_{j_1+1}, ..., i_k\}$ with the least decreasing subsequence of length $k - j_1$ and sum $\sum_{j=j_1+1}^{k} i_j - 1$.

4. Go to step 2.

The lemma is proved in [7].

**Lemma 9.2** For any $n > 0$, $r(n, k, u, q) \leq r(n + 1, k, u, q)$.

*Proof:* Suppose that $P_0$ is a complete matrix and $A_0$ is an optimal mapping for $P_0$. Hence

$$r(n, k, u, q) = \max_{\text{programs } P \text{ of } n \text{ variables}} \frac{em(P, k, u, q)}{h(P, 1, q)} = r(n, k, u, q),$$

$$\frac{em(P_0, k, u, q)}{h(P_0, 1, q)} = \frac{em(A_0, P_0, k, u, q)}{h(P_0, 1, q)}.$$  

We next construct a program $P_1$ by adding to the program $P_0$ a new variable of size exactly one block, as the very last reference. Since this variable is not referenced before it is a miss with any cache. Hence we can obtain an optimal mapping $A_1$ to the program $P_1$ by adding the new variable to any set. The program $P_1$ has $n + 1$ variables but may not be optimal. Thus:

$$\frac{em(P_0, k, u, q)}{h(P_0, 1, q)} = \frac{em(P_1, k, u, q)}{h(P_1, 1, q)} \leq$$

$$\max_{\text{programs } P \text{ of } n + 1 \text{ variables}} \frac{em(P, k, u, q)}{h(P, 1, q)} = r(n + 1, k, u, q).$$

$\square$
**Proof of Theorem 4.4:** By the previous lemma it is clear that
\[
\sup_n r(n, k, u, q) = \lim_{n \to \infty} r(n, k, u, q).
\]

Further it follows that
\[
\lim_{n \to \infty} r(n, k, u, q) = \lim_{w \to \infty} r(wk, k, u, q).
\]

Note also that exactly the same decreasing sequences are generated for all \( w \geq k \), if \( k \) is fixed. This fact makes it possible to explicitly compute the function \( R(k, u, q) \). When letting \( w \to \infty \), it is enough to study the behaviour of the \( w \)-dependent part of each term, which is the quotient
\[
\frac{\left( \begin{array}{cccc}
  w \\
  i_1 \\
  \vdots \\
  i_k \\
\end{array} \right)}{\left( \begin{array}{c}
  wk \\
  q \end{array} \right)}.
\]

We will use the fact that \( i_1 + \ldots + i_k = q \). From the Stirling formula \( n! \approx (\frac{n}{e})^n \sqrt{2\pi n} \) we get for large \( n \):
\[
\frac{n}{k} \approx e^{-k} \frac{\sqrt{n}}{k} \frac{n^n}{(n-k)^{(n-k)}}.
\]

Thus the quotient approximately equals
\[
\frac{\frac{e^{-i_1}}{i_1!} \sqrt{\frac{w}{w-i_1}(\frac{w}{w-i_1})^{w-i_1}} \ldots \frac{e^{-i_k}}{i_k!} \sqrt{\frac{w}{w-i_k}(\frac{w}{w-i_k})^{w-i_k}}}{\frac{q!}{e^q} \sqrt{\frac{kw-q}{kw}} \frac{(kw-q)^{kw-q}}{(kw)^{kw}}} =
\]
\[
\frac{q!}{i_1! \ldots i_k!} \sqrt{\frac{w}{w-i_1} \ldots \frac{w}{w-i_k}} \frac{kw-q}{kw} \frac{(kw-q)^{-q}}{(w-i_1)^w \ldots (w-i_k)^w (w-i_1)^{-1} \ldots (w-i_k)^{-1}}.
\]

The first line after the last equality clearly is independent of \( w \). The second line tend to 1. The first factor of the third line tend to \( e^{-q} \). The second factor can be written as
\[
(1 + \frac{i_1}{w-i_1})^w \ldots (1 + \frac{i_k}{w-i_k})^w.
\]
Now,
\[(1 + \frac{i - 1}{w - i})^{w-i} \rightarrow e^{i-1} \text{ as } w \rightarrow \infty,\]
so we get
\[e^{i_1} \cdot \ldots \cdot e^{i_k} = e^q\]
as \(w \rightarrow \infty\), cancelling the factor \(e^{-q}\). From the third factor we get
\[(\frac{kw - q}{w - i_1})^{-i_1} \cdot \ldots \cdot (\frac{kw - q}{w - i_k})^{-i_k} \rightarrow k^{-i_1} \cdot \ldots \cdot k^{-i_k} = k^{-q}.\]
Thus the third line above contributes with a factor \(k^{-q}\) as \(w \rightarrow \infty\). Hence:
\[
\lim_{w \rightarrow \infty} \left(\frac{w}{i_1} \cdot \ldots \cdot \frac{w}{i_k}\right)^{\frac{kw - q}{q}} = \frac{q!k^{-q}}{i_1! \cdot \ldots \cdot i_k!}.
\]

The proof is completed. \(\square\)

10 Discussion and Conclusions

The results reported here make it possible to obtain optimal worst-case bounds on the cache hit ratio for set-associative and direct mapped caches. This can be understood by considering the following example.

Consider a program \(P\), with \(n\) variables, for which we have obtained a certain hit ratio \(X\) using a \((1,q)\)-cache. Now we are interested in the hit ratio \(Y\) for \(P\) using an optimal mapping of variables to memory addresses for another cache organization – a \((k,u)\)-cache – such that \(q < ku\) and \(u < q\). We know that \(X \leq h(P,1,q)/m(P)\), and that \(Y = h(P,k,u)/m(P)\). Consequently, from Theorem 4.1 (ii) we know that \(X(1 - r(n,k,u,q)) \leq Y\). One practical implication of this is that if we should obtain a hit ratio which is smaller than \(X(1 - r(n,k,u,q))\) using a \((k,u)\)-cache, then we know that the mapping of variables to memory addresses is not optimal. Generally we do not know if \(X = h(P,1,q)\) or not, and Theorem 4.1 (ii) gives an optimal lower bound on \(h(P,k,u)/m(P)\). Consequently, based on the information we have the lower bound on \(Y\) is optimal.

In the example above the results were used for evaluating the performance (hit ratio) of one particular program \(P\). However, it is also possible to use our results when evaluating different cache architectures, e.g. Theorems 4.3 and 4.4 directly tell us how much worse the hit ratio we may obtain for a \((k,u)\)-cache compared to a \((1,q)\)-cache. However, the results can also
be used for more complex scenarios, e.g., consider two cache memory systems, in system one we have a (1, 20)-cache and in system two we have a (20, 2)-cache. The main memory access time is the same in both cases, and the access time for the (1, 20)-cache is 8 times faster than a main memory access whereas the access time for the (20, 2)-cache is 10 times faster than a main memory access, i.e. the (20, 2)-cache is slightly faster than the (1, 20)-cache.

Let $M$ be the access time for one memory reference on the main memory, and let $C$ denote the access time on the cache memory. For a program $P$ with length $L$ and hit ratio $H$, on system 1 we get the total access time

$$A_1(P) = L(C_1H + M(1 - H)) = L(M - (M - C_1)H).$$

Let $H'$ be the hit ratio with system 2. From Theorem 4.1 we know that $H(P) (1 - r(n, k, u, q)) \leq H'(P) \leq 1$. From this lower and upper bounds of the ratio $A_2(P)/A_1(P)$, the access time for system 2 when the access time for system 1 is known, follow:

$$\frac{C_2}{M - (M - C_1)H} \leq \frac{A_2(P)}{A_1(P)} \leq \frac{M - (M - C_2)H(1 - r(n, k, u, q))}{M - (M - C_1)H}.$$

In section 11.3 these bounds, $f_1(H)$ and $f_2(H)$ respectively, are plotted as a function of the program parameter $H$ when system 1 is a (1, 20)-cache with $M/C_1 = 8$, and system 2 is a (20, 2)-cache with $M/C_2 = 10$. Here we need $R(20, 2, 20) = 0.2459$. If the number of parameters $n$ is known, one can obtain a slightly sharper upper bound by replacing $R(20, 2, 20)$ by $r(n, 20, 2, 20)$. The functions $f_1(H)$ and $f_2(H)$ are interesting when we want to compare the two cache organizations.

The average behaviour of set-associative caches have been studied extensively. One rule of thumb is that a $(k, 1)$-cache gives about the same hit ratio as a $(k/4, 2)$-cache [3]. On page 421 in [3] there is a table where the hit ratio using different cache sizes and different degrees of associativity is shown for a "typical" work load. A smaller version of the same table is also shown in [5]. These tables show that the hit ratio gain of going from a $(k, 1)$-cache to a $(k/2, 2)$-cache is larger than the gain of going from a $(k/2, 2)$-cache to a $(k/4, 4)$-cache, and going from a $(k/4, 4)$-cache to a $(k/8, 8)$-cache is even smaller. The plot in Section 11.1 supports this. There has also been a great interest for developing models for memory access patterns, which make it possible to evaluate average performance of fully associative, set-associative and direct mapped caches.
It is well known that one of the problems with direct mapped caches compared to fully or set associative caches is that they exhibit a terrible worst-case performance [4]. However, before this result it has not been possible to quantitively compare the worst-case behaviour of direct mapped caches to fully associative caches. Section 11.2 displays a plot of this performance relation.

On pages 408 to 425 in [3] there is a very comprehensive discussion on the performance implications of various cache parameters, e.g. the LRU replacement policy outperforms random replacement particularly for caches with relatively few blocks (up to a couple of thousand blocks). Moreover, the “optimal” block size increases with increasing cache size. Based on these observations it is reasonable to assume that large caches will have large blocks and that the LRU replacement policy is used within each set. Consequently, the LRU assumption made in this report is reasonable. The assumption that each variable can be mapped to one block becomes more reasonable for large cache blocks. However, large arrays will probably have to be mapped to consecutive parts of the address space exceeding one cache block. There are however options open to the compiler or programmer, e.g. one can either store a two dimensional array row by row or column by column.

Another approach is to use non-linear mapping functions. When ordinary linear mapping is used, every k:th memory block is mapped to the same set. However, in some cases non-linear mapping functions have been suggested [1][17]. In those cases, the problem of optimally mapping variables to memory addresses is transferred into the problem of defining an optimal mapping function, and in that case different parts of large arrays can be mapped independently of each other.
11.3 Maximal and minimal access time

As described in section 10, estimates on cache hits can easily be transformed to estimates on access times. Consider a (20,2)-cache with cache access time 1/8, the main memory access time is normalized to 1. Denote the hit ratio for the program $P$ on this system by $H$, and the access time $A_1(H)$. Next the same program is to be executed on a (1,20)-cache with cache access time 1/10. Here optimal upper and lower bounds $f_1(H)$ and $f_2(H)$ for the access time $A_2(H)$ on the second system, relatively $A_1(H)$, are plotted.

\[
 f_1(H) = \frac{0.8}{8-7H} \leq \frac{A_2(H)}{A_1(H)} \leq \frac{8-5.436H}{8-7H} = f_2(H)
\]

Notes:
1. The lower bound is trivial, while the upper bound follows from Theorem 4.1. Both bounds are optimal, i.e. there are programs with any access time in the interval given by $f_1(H)$ and $f_2(H)$.
2. Usually higher associativity results in slightly higher cache access time. If this relationship is known, i.e. if $C(k,u)$ is known, the above argument can be extended to provide optimal associativity in the sense of minimal access time for worst case programs.
11.2 Direct mapping versus full associativity

This plot shows the worst case hit ratio comparing a direct mapped cache with a fully associative cache. The cache has $k$ blocks, and programs of $n$ variables are considered. The function plotted is

$$\inf_P \frac{h(P, k, 1)}{h(P, 1, k)} = 1 - r(n, k, 1, k).$$

The following plot shows sections of the plot above together with the limit $1 - R(k, 1, k)$, which is the infimum taken over programs of any number of variables.
11 Graphics Section

The graphics in this section were produced with Mathematica Version 2.0 and edited with Framemaker Version 4.

11.1 Increasing degree of associativity

This plot shows the worst case number of cache hits for four degrees of associativity, compared to full associativity. The worst case corresponds to the infimum

\[ \inf_p \frac{h(P, k, u)}{h(P, 1, q)} = 1 - R(k, u, q) \]

taken over programs of any number of variables.

Note:
1. Only values for integer arguments have been plotted.
2. The improvement in relative number of cache hits for worst case programs, as a function of the associativity, occurs fairly slowly. This is in accordance with [5]; see discussion in section 10.
References:


Figure 1: A direct mapped and a set-associative cache with two sets for a data segment of size 8.

Variables 1, 3 and 4 are not in the $(1,q)$-cache

If a miss occurs, variable 5 is replaced

Figure 2: A $q,n$-state vector where $q = 4$ and $n = 7.$
Figure 3: Two mappings of the same program with a (3,1)-cache, compared to a (1,3)-cache.

Number of extra misses: 2
Number of extra misses: 4

Sequence of memory block references: 5, 6, 7, 2

Figure 4: An example of lemma 8.1.
Figure 5: Creating a complete program $P'$ from a program $P$. 

The process involves reordering the $q.n$-hit vectors in $P'$.