Pre-print version. Final version is copyrighted by Elsevier:
Journal of Systems Architecture, 2005, vol. 51, no. 4, pp. 223-250
DOI: doi:10.1016/j.sysarc.2004.09.004

Generating Cache Hints for Improved Program Efficiency

Kristof Beyls Erik H. D'Hollander

Department of Electronics and Information Systems
Ghent University
Sint-Pietersnieuwstraat 41
B-9000 Ghent, Belgium

Abstract

One of the new extensions in EPIC architectures are cache hints. On each memory instruction, two kinds of hints can be attached: a *source cache hint* and a *target cache hint*. The source hint indicates the true latency of the instruction, which is used by the compiler to improve the instruction schedule. The target hint indicates at which cache levels it is profitable to retain data, allowing to improve cache replacement decisions at run time. A compile-time method is presented which calculates appropriate cache hints. Both kind of hints are based on the locality of the instruction, measured by the *reuse distance metric*.

Two alternative methods are discussed. The first one profiles the reuse distance distribution, and selects a *static hint* for each instruction. The second method calculates the reuse distance analytically, which allows to generate *dynamic hints*, i.e. the best hint for each memory access is calculated at run-time.

The implementation of the static hints scheme in the Open64-compiler for the Itanium processor shows a speedup of 10% on average on a set of pointer-intensive and regular loop-based programs. The analytical approach with dynamic hints was implemented in the FPT-compiler and shows up to 34% reduction in cache misses.

Key words: compiler optimization, reuse distance, replacement policy, source cache hint, target cache hint, EPIC

Email addresses: kristof.beyls@elis.UGent.be (Kristof Beyls), erik.dhollander@elis.UGent.be (Erik H. D'Hollander).

1 Introduction

Many optimizations have been proposed to improve the cache behavior of programs. The existing software techniques either *improve data layout* (e.g. array padding[38], structure layout[12]), *reorganize the order of computation to increase the locality* (e.g. loop transformations such as loop tiling[51], fusion[30], and others[32]), or *hide the latency* of cache misses through prefetching[33]. In a different vein, the growing speed gap between processor and memory has spurred the development of new hardware remedies to improve cache performance. This resulted in cache hint extensions for memory instructions, e.g. in EPIC (Explicitly Parallel Instruction Computing) architectures[41]. A *source cache hint* indicates the fastest cache level where the requested data is likely to be found. A *target cache hint* specifies at which cache levels data should be kept and from which cache levels data should be removed. Cache hints offer a new dimension for compiler optimizations, since, for the first time, the cache replacement policy can be directly steered from software. Without cache hints, software optimizations can only indirectly influence which data will remain in the cache by changing the order of data access or altering the data layout. This paper is a study on how to use target cache hints, in order to reduce the number of cache misses and how to use source cache hints for hiding the memory latency by scheduling parallel instructions.

The effect of target cache hints is to alter the cache line replacement policy. In current general purpose processors, the cache replacement policy is implemented completely in hardware. The well-known optimal cache replacement policy[2] specifies that the cache line which is used furthest in the future should be replaced. Since it is generally impossible to know which cache line will be used furthest in the future, most processors follow the least recently used (LRU) policy, or a related policy, such as not recently used (NRU) or pseudo-LRU. These policies are based on the assumption that the cache line which has not been used for the longest time has the lowest temporal locality. I.e., it is assumed that the cache line with the lowest "backward" temporal locality, also has the lowest "forward" temporal locality, and thus will be used furthest away in the future. Since the cache replacement policy is implemented in the hardware, the prediction of the line that will be accessed furthest in the future can only be based on the locality exhibited by the previous accesses in the current run of the program. Cache hints change this policy by accepting hints from the compiler. If there are no cache hints, the default cache line replacement policy applies. If cache hints are used, it is expected that the compiler has a more informed estimate of the forward temporal locality, e.g. based on a previous run of the program or on an analysis of the source code.

In contrast to target hints, source cache hints are used only inside the compiler.

During instruction scheduling, the compiler needs to know the latency of the instructions. The latency of an instruction i indicates how many independent instructions the compiler should try to schedule before instructions dependent on i. For load instructions the latency is variable, since it depends on the cache level where the data is found. Traditionally, compilers assume a latency l of a level 1 cache access. Therefore, it schedules instructions depending on a load instruction at least l clock cycles after the load. However, if the load misses the first-level cache, the data is not available yet, and the processor stalls. Source hints indicate the true latency of load instructions to the instruction scheduler. In that way, more independent instructions will be scheduled during a cache miss than during a cache hit.

1.1 Overview

The operation of cache hints in EPIC architectures are discussed in more detail in section 2.

The decision whether data should be placed at a particular cache level depends on both cache hardware parameters, such as cache size and associativity, and the data locality exhibited by the program. The locality of the memory accesses is measured by the *reuse distance*, a metric independent of the cache hardware parameters. Therefore, a single reuse distance measurement can guide the cache replacement decisions for any cache, no matter the associativity or size. Furthermore, the reuse distance is also used to predict whether a memory access is a cache hit or miss. So, this single metric can be used to select both source and target hints. The reuse distance and its relevant properties are elaborated in section 3.

A single memory instruction can be executed multiple times, resulting in multiple memory *accesses*. The memory accesses originating from the same instruction could exhibit different amounts of locality, requiring different cache hints. However, cache hints are traditionally defined to be specified for a memory instruction, so that all accesses generated by a single instruction must have the same cache hint. In this paper, we refer to a cache hint specified for an instruction as a *static hint*, whereas a cache hint specified for an individual access is called a *dynamic hint*. Both methods for generating static hints and generating dynamic hints are specified. In section 4, a general method applicable to all programs is proposed which measures the reuse distance by instrumenting and profiling a program. Based on the profiled reuse distance, static cache hints are generated. In contrast, in section 5, a program analysis for loop-oriented programs is defined, which calculates reuse distances analytically. Based on the analysis, in section 6, code is generated which produces dynamic hints, i.e. the hint is generated for each individual memory access.

In section 7, the results of applying these optimizations on both pointer-intensive programs as well as numerical programs are presented. With static target hints, the improved cache replacement policy leads to a maximum speedup of 20%, with an average of 3%. When static source hints are also enabled, the better informed instruction scheduler produces code which is 10% faster, on average, with a maximum speedup of 56%. Furthermore, on a set of numerical programs, the analytical calculation of the reuse distance allows to generate dynamic target hints. The static hints remove 5% of the cache misses for these programs, while dynamic hints remove 10% of the cache misses, on average. In section 8 concluding remarks are formulated.

1.2 Related Work

The speed gap between processor and memory has been doubling every two years since 1980. This has attracted a lot of researchers to propose improved caching schemes. Most of these proposals can be categorized either as hardware modifications[9,26,43], loop transformations[32,31,51], data layout optimizations[10,28,38] or hardware or software prefetching schemes[27,32,43,46]. The prefetching schemes aim at hiding the latency of cache misses with parallel instructions, while most other methods try to reduce the number of cache misses either by improving the programs locality or by improving the cache placement and replacement policy. The proposed cache hint selection scheme performs both kinds of optimizations: the source cache hints are used to hide the latency of the cache misses, while the target cache hints improve the replacement policy of the data cache.

Below, we discuss the most closely related work in three areas: work related to source cache hint selection, work which measures or calculates reuse distances and work related to target cache hint selection.

Work related to source cache hint selection The source cache hints allow the compiler to try to hide the latency of cache misses with parallel instructions, while fetching the data from a lower cache level. Most related research focuses on generating prefetch instructions to do this. However, prefetching requires extra prefetch instructions to be inserted in the program. In the source cache hint approach, the latency is hidden without inserting prefetch instructions. Similar techniques are proposed in [21] and [35]. In [21], the cache behavior of numerical programs is examined using miss traffic analysis. The detected cache miss latencies are hidden by techniques such as loop unrolling and shifting. In comparison, our technique also applies to non-numerical programs and the latencies are compensated by scheduling low level instructions. In [35] load instructions are classified into normal, list and stride access. List and stride

accesses are maximally hidden by the compiler because they cause most cache misses. However the classification of memory accesses in two groups is rather coarse. The reuse distance provides a more accurate way to measure the data locality, and as such permits the compiler to generate a more balanced schedule.

Work related to reuse distance measurement In [16], the reuse distance is proposed as a metric for whole-program locality, and techniques are proposed for fast profile-based measurement of the reuse distance. Next to the profile-based measurement of reuse distance distributions, in this paper, we also propose the mathematical representation of reuse distances for *every single access*. The calculation of reuse distances is related to the field of analytical calculation of cache behavior. Most of the previous work on analytical cache behavior calculation is based on reuse vectors[19,20,39,47]. These reuse vectors only capture the reuses between uniform references, i.e. array references for which the index expressions differ at most by a constant. As such, not all reuse is captured, and the method results in less exact estimates. Recently, Chatterjee et al.[11] proposed a technique to exactly calculate the cache behavior. While their technique leads to the exact calculation for low-associative caches, the proposed cache equations do not allow to efficiently obtain cache behavior for high associativity ($\geq$ 4 way set associative). Furthermore, the cache behavior cannot be calculated when the loop bounds contain symbolic parameters. In contrast, the calculation of reuse distances allows to compute the cache behavior of fully associative caches. When assuming a cache line size equal to the array element size, it is even possible to compute cache behavior in the presence of symbolic loop bounds. Furthermore the proposed cache equations can easily be extended to also handle set-associative caches. In [8], Caşcaval also calculates the reuse distance analytically, but only for loop nests for which the data dependences are uniformly generated.

Work related to target cache hints Work related to target cache hints is found in [25], [40], [50] and [53]. In [25], Jain et al. propose keep and kill instructions. The keep instruction locks data into the cache, while the kill instruction indicates it as the first candidate to be replaced. They also prove under which conditions the keep and kill instructions improve the cache hit rate. In [50], Wang et al. propose to extend each cache line with an EM(Evict Me)-bit. The bit is set by software, based on a locality analysis. If the bit is set, that cache line is the first candidate to be evicted from the cache. In contrast to our approach, the locality analysis is based on reuse vectors, which is less accurate because only reuses between uniform references are taken into account. In [40], a cache with 3 modules is presented. The modules are optimized respectively for spatial, temporal and spatial-temporal locality. The compiler indicates in which module the data should be cached, based upon

compiler analysis or a profiling step. These approaches all suggest interesting modifications to the cache hardware, which allow the compiler to improve the cache replacement policy. However, the proposed modifications are not available in todays architectures. An advantage of our approach is that it uses cache hints available in existing processors. The results show that the presented cache hint selection scheme is able to increase the performance on real hardware.

In addition, previous work generates the equivalence of static hints, i.e. a fixed hint per instruction. In contrast, the presented analytical reuse distance calculation generates dynamic hints, optimized for each individual memory access.

2 Cache Hints

Recently, steering the cache replacement policy from software was enabled by cache hints, which emerged in EPIC[41] architectures. This is in line with the central idea of the EPIC architectures: the compiler is responsible for deciding when to issue instructions and which resources to use. In contrast with superscalar processors, the compiler should specify which instructions to execute in parallel, how to predict branches and where to place data in the cache hierarchy. In EPIC instruction sets, cache hints provide a means to the compiler to communicate its decisions about cache placement and replacement to the processor. The major EPIC architectures (HPL-PD[29] and IA-64[23]) both provide cache hints with similar semantics.

In the HPL-PD research architecture, cache hints annotate regular memory instructions, and occur in two kinds: the source and target hints. The first kind, the *source cache specifier*, indicates at which cache level the accessed data is likely to be found. The second kind, the *target cache specifier*, indicates at which cache level the data is kept after the instruction is executed. An example is given in figure 1, where the effect of the load instruction `LD_C2_C3` is shown.

The *source cache specifiers* are used by the compiler to know the estimated data access latency. Without these specifiers, the compiler assumes that all memory instructions hit in the L1 cache. Using the source cache specifier, the compiler is able to determine the true memory latency of instructions. When the memory access has a longer latency, the compiler tries to hide that latency by scheduling more instructions explicitly in parallel with the memory operation. The *target cache specifiers* are used by the processor, where they indicate the highest cache level at which the data should be kept. A carefully selected target specifier will maintain the data at a fast cache level, while minimizing the cache pollution.

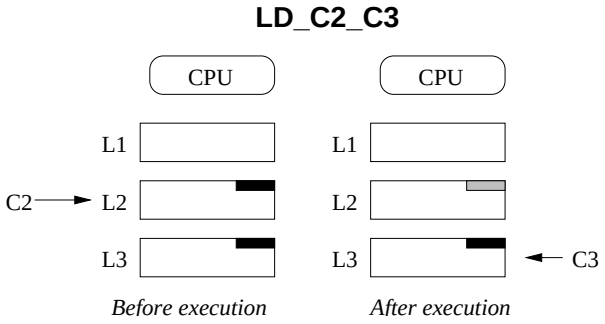


Fig. 1. Example of the effect of the cache hints in the load instruction `LD_C2_C3`. The source cache specifier `C2` in the instruction suggests that the data resides in the L2-cache. The target cache specifier `C3` indicates that the data should be stored no closer than the L3-cache. As a consequence, the data becomes the first candidate for replacement in the L2-cache.

Table 1
The effect of the target cache hints on the replacement policy in the Itanium processor[24]. alloc=$\surd$ means that the data is allocated in that cache level. update LRU=$\surd$ means that the LRU bits are updated, i.e. that cache line is considered the most recently accessed line.

hint	**L1**		**L2**		**L3**	
	alloc	update LRU	alloc	update LRU	alloc	update LRU
t1	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
nt1			$\surd$	$\surd$	$\surd$	$\surd$
nt2			$\surd$		$\surd$	$\surd$
nta			$\surd$			

Since source cache hints are used inside the compiler, there's no need to communicate them to the processor. The source cache hints can be used in any compiler, there is no need for support in the instruction set. They merely are of help to the instruction scheduler. In contrast, target cache hints require architectural support. The IA-64 defines the target cache hints `.t1`, `.nt1`, `.nt2` and `.nta`. These cache hints specify whether there is temporal locality at a given cache level. The IA-64 model assumes that there are two parallel cache hierarchies: one for accesses which exhibit temporal locality, and one for accesses that exhibit only spatial locality, or even no locality at all. The semantics of the different cache hints in this model are depicted in figure 2. `.t1` specifies that the memory instruction has temporal locality at all cache levels; `.nt1` indicates no temporal locality at cache level 1, `.nt2` means no temporal locality at cache level 1 and 2, and `.nta` means no temporal locality at all.

Despite the model of two separate cache hierarchies for temporal and non-temporal data (see fig. 2), the first implementations of the IA-64 architecture,

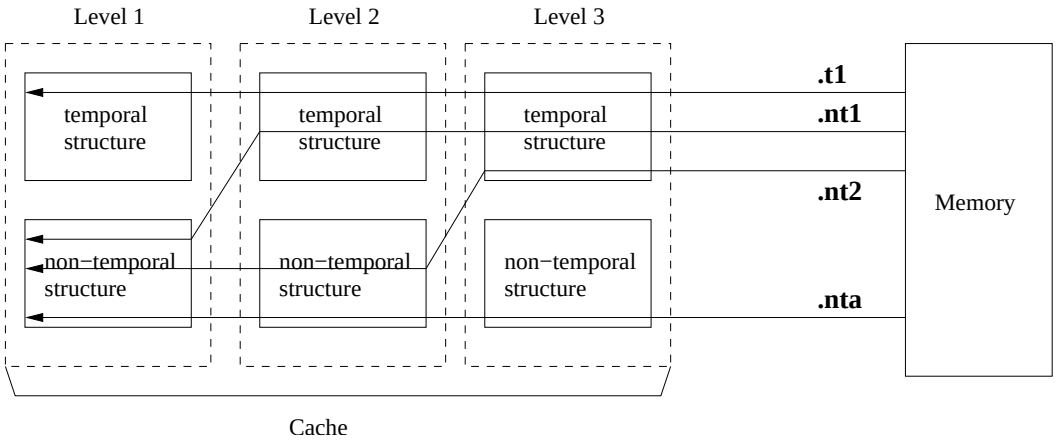


Fig. 2. The semantics of the target cache hints in the IA-64 model[23]. If the cache hint indicates temporal locality for a given cache level, the data is allocated in the temporal buffer; otherwise it is placed in the non-temporal buffer. `.nt{1,2,a}` means "no temporal locality at level 1, 2, and at all levels". Detailed effect of these hints are shown in table 1.

the Itanium1 and Itanium2 processors, implement only a single cache hierarchy. Nonetheless, the cache hints influence the placement and replacement policy. The single cache hierarchy on the Itanium processors is viewed as the temporal cache hierarchy from the IA-64 model. In order to exploit spatial locality for memory accesses where the cache hint indicates no temporal locality, data is always put in a single way in the second cache level. When the cache hint indicates that there is no temporal locality, the corresponding LRU bits are not updated. The detailed effect of the different hints are shown in table 1.

3 Reuse Distance

In order to select an appropriate source cache specifier, the compiler should estimate at which cache level requested data is found. Likewise, for a target cache specifier, the compiler should estimate the profitability of keeping data at a certain cache level. These estimates are based on a single locality metric: the *reuse distance*, which is defined within the framework of the following definitions.

Definition 1. A **memory reference** corresponds to a read or write instruction, while a particular execution of that read or write at runtime is a **memory access**[20].

Definition 2. A **reuse pair** $\langle a_1, a_2 \rangle$ is a pair of memory accesses in a memory access stream, which touch the same memory location, without intermediate accesses to that location. The **accessed data set (ADS)** of a reuse pair

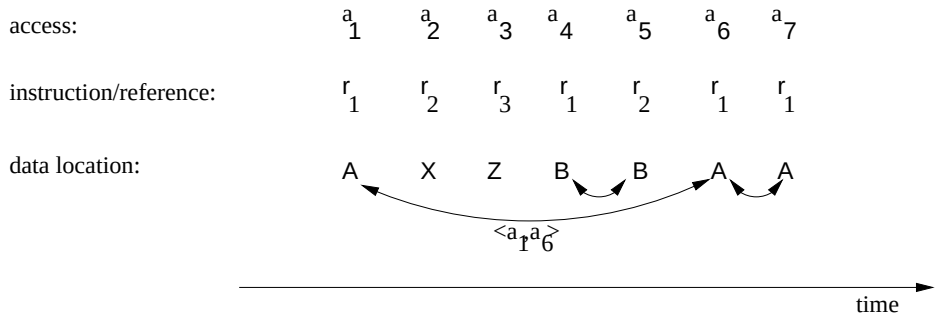


Fig. 3. The top row indicates 7 sequential memory accesses, which are generated by the references in the second row, where the reference index reflects the instruction number. The bottom row shows the corresponding memory locations A, B, X or Z. The accesses to X and Z are not part of a reuse pair, since they are accessed only once in the stream. $\mathsf{ADS}\langle a_1, a_6\rangle = \{B, X, Z\}$, and $\mathsf{RD}(\langle a_1, a_6\rangle) = |\mathsf{ADS}\langle a_1, a_6\rangle| = 3$. $\mathsf{RD}(\langle a_6, a_7\rangle) = 0$. $\mathsf{FRD}(a_1) = 3$, $\mathsf{BRD}(a_1) = \infty$. $\mathsf{FRD}(a_6) = 0$, $\mathsf{BRD}(a_6) = 3$.

$\langle a_1, a_2\rangle$ is the set of unique memory locations accessed between a_1 and a_2, and is denoted by $\mathsf{ADS}\langle a_1, a_2\rangle$. The **reuse distance** of a reuse pair $\langle a_1, a_2\rangle$ is the number of unique memory locations accessed between accesses a_1 and a_2. It is denoted by $\mathsf{RD}(\langle a_1, a_2\rangle)$, and equals $|\mathsf{ADS}\langle a_1, a_2\rangle|$.

Definition 3. Consider the reuse pairs $\langle a_1, a_2\rangle$ and $\langle a_2, a_3\rangle$. The **forward reuse distance** of a memory access a_2 is the reuse distance of the pair $\langle a_2, a_3\rangle$. If there is no such reuse pair, its forward reuse distance is ∞. The **backward reuse distance** of a_2 is the reuse distance of $\langle a_1, a_2\rangle$. If there is no such pair, the backward reuse distance is ∞. The forward reuse distance of a_2 is denoted by $\mathsf{FRD}(a_2)$, its backward reuse distance is denoted by $\mathsf{BRD}(a_2)$.

Figure 3 shows three reuse pairs in a short memory access stream.

Theorem 1. Reuse Distance Theorem In a fully associative LRU cache with n lines, an access a hits the cache if and only if $\mathsf{BRD}(a) < n$. The memory line accessed by a will stay in the cache until the next access of that memory line if and only if $\mathsf{FRD}(a) < n$.

Proof. In a fully associative LRU cache with n cache lines, the n most recently referenced memory lines are retained. For an access a, exactly $\mathsf{BRD}(a)$ different memory lines were referenced since the previous access to the same location. If $\mathsf{BRD}(a) \geq n$, the referenced memory line is not one of the n most recently referenced lines, and consequently will not be found in the cache.

If the forward reuse distance is infinite, the data will not be used in the future, so there is no next access. If the forward reuse distance is not infinite, consider the forward reuse distance of access a_1 and assume that the next access to the data occurs at access a_2, resulting in a reuse pair $\langle a_1, a_2\rangle$. By definition, $\mathsf{FRD}(a_1)$=$\mathsf{BRD}(a_2)$. Therefore, the data will be found in the cache at access

a_2, if and only if $\mathsf{FRD}(a_1) < n$. □

The theorem above can be used to predict the cache misses of fully associative caches. Traditionally, cache misses have been classified as either cold, conflict or capacity misses[22]. A cold miss occurs when the data is accessed for the first time, i.e. when the backward reuse distance is ∞. A capacity miss occurs when that memory access results in a cache miss in a fully associative cache, i.e. when the backward reuse distance exceeds the cache size. The theorem allows to predict cold and capacity misses. Even though conflict misses are not predicted by the reuse distance, previous research[5,6,22] indicates that also for lower-associative, and even for direct mapped caches, the reuse distance can be used to obtain a good estimate of the cache behavior. The authors of [5,6,22] independently measure the error made by the theorem above, when predicting cache behavior for low-associative caches. Both statistical analysis and measurements based on a large number of program traces indicate that the relative error is low, typically less than 5%[22]. This is further confirmed by the cache miss measurements performed on the SPEC2000 benchmark in [7]. The measurements show that for typical caches in general purpose processors (associativity 2 or greater and cache size larger than 8KB), the capacity misses are responsible for more than 85% of all misses.

The reuse distance as a metric for cache locality has the benefit that a single measurement can be used to base optimizations on for all cache levels, since it is independent of cache size or associativity. Therefore, it is a suitable locality metric for multi-level cache optimizations such as cache hint selection.

4 Static Cache Hint Selection

Traditional software cache optimizations, such as loop tiling[4] reduce the number of cache misses by reducing the reuse distance of the accesses. However, due to data and control dependences, for many programs, it is not possible to legally perform these program transformations. Here, we exploit the possibility to adapt the instruction scheduling and the cache replacement policy through cache hints when the reuse distance is larger than the cache size.

4.1 Cache Hint for a Single Access

The source cache hint should indicate the highest cache level where the data can be found at the time the memory access occurs. If a faster cache level is indicated, the compiler would assume a latency that is smaller than the true latency and it wouldn't try to hide the full latency of the load. If a slower

cache level is indicated, the assumed latency would be too large. As a consequence, the compiler would generate a sub-optimal schedule, because the target register of the memory instruction will be kept live longer than necessary, which increases register pressure. Theorem 1 specifies that the backward reuse distance indicates the minimal cache size which is needed for the access to be a cache hit in a fully associative cache. As described at the end of section 3, it also indicates the minimal cache size for the access to be a hit for lower-associative caches with high probability. Therefore, the source cache hints are selected using the following rule.

Source Cache Hint Selection Strategy. *The hint indicating the smallest cache level larger than the backward reuse distance is selected as the source cache hint.*

The target cache hints should indicate the smallest cache level where the data will be retained until its next use, i.e. the fastest cache level C where the locality of the data will be exploited. If a larger cache level is selected, the locality would not be exploited in the fast level C, leading to an extra miss for that level. If a smaller level is selected, the data is fetched into a level where the locality cannot be exploited. Even worse, the data pollutes the smaller cache, and can throw out other data with higher locality. According to theorem 1, the forward reuse distance indicates the cache size that is needed for the data to be retained until the next reuse in a fully associative cache. As described at the end of section 3, it also indicates the cache size needed to keep the data for lower-associative caches with high probability. Therefore, the target hints are selected by the following rule.

Target Cache Hint Selection Strategy. *The hint indicating the smallest cache level larger than the forward reuse distance is selected as the target cache hint.*

Further evidence of the appropriateness of this target hint selection scheme is given in [25], where Jain et al. prove that this cache hint choice is guaranteed to perform equal or better than the LRU replacement policy for a fully associative cache.

4.2 Static Selection for all Accesses from a Single Instruction

A single memory instruction generates multiple memory accesses when that instruction is executed multiple times (e.g. in loops). The different accesses originating from the same memory instruction may exhibit different locality, requiring different cache hints. For example, in figure 3, the instruction r_1 generates accesses a_1, a_4, a_6 and a_7. These accesses respectively have forward reuse distances 3, 0, 0 and ∞. Therefore, for the second and third access

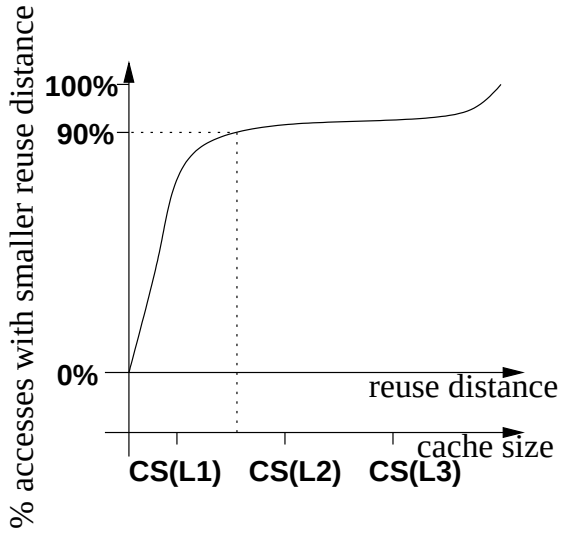


Fig. 4. A cumulative reuse distance distribution for an instruction is shown and how a threshold value of 90% maps it to cache hint C2. CS(Lx) = cache size for cache level x.

generated by instruction r_1, the data will be retained for any cache size; while for the fourth access, the data will never be reused.

However, it is not possible to specify different cache hints, since the hint is specified on the instruction. As a consequence, all accesses originating from the same instruction share the same cache hint. The following approach is used to obtain a single cache hint per instruction which is applicable for most accesses generated by the same instruction.

First, for every instruction, the cumulative reuse distance distribution of all generated memory accesses is collected. An example of such a distribution is shown in figure 4. Based on the distribution, a source cache hint is selected so that for at least x% of the accesses, the data will be found in that cache level. Figure 4 shows how to find the cache hint so that for at least 90% of the accesses, the data will be found in the indicated cache level. Similarly, the distribution can be used to select the target cache hint so that for at least y% of the accesses, the data will be retained at that cache level. Clearly, for some distributions, it is impossible to find a cache hint which indicates the correct cache level for the majority of all accesses. Therefore, the hints should be selected so that if they are wrong, they don't incur a high cost, i.e. err on the safe side.

In table 2, the expected cost of too small and too high source and target hints are indicated. It is less costly to have a source hint which is too large instead of one which is too small. It is costlier to indicate a too small cache level, since the compiler won't try to hide the full latency of the access. When the hinted cache level is too large, the compiler merely tries to schedule more parallel

Table 2
Expected cost of a wrong cache hint.

	indicated level too small	indicated level too large
source cache hint	high cost	low cost
target cache hint	low cost	high cost

instructions between the access and the next dependent instructions. For a target hint, it is costlier to indicate a cache level too large instead of a level too small. When the hinted level is too small, the data is brought in a level where it won't be reused, potentially leading to cache pollution. This would also happen with an LRU replacement policy. On the other hand, when the hinted level is too large, the data won't be fetched in all levels where it can be reused, and unnecessary cache misses will result.

The optimal values for the parameters x and y may vary with cache and memory access time, the execution of parallel instructions on the specific processor, the scheduling algorithm in the compiler, the program which is compiled and the input set[16]. Taking table 2 into account, in the experiments, x is chosen to be 90%, while y is chosen to be 10%. In this way, at most 10% of the accesses could have a wrong cache hint which incurs high cost.

The cumulative reuse distance distributions can be measured during a training run of an instrumented version of the program. The implementation of such a profile-based scheme in the Open64-compiler and its application to a number of benchmarks is presented in section 7.

The two main limitations of cache hint selection based on measured reuse distance distributions are the following. First, the cache hint selection is based on a reuse distance distribution measured for one particular execution of the program. Another execution, processing a different amount of data, might produce a different reuse distance distribution, and could therefore require different hints. Second, it is often impossible, based on the distribution, to select a single static hint which is suited for all individual memory accesses. Even though the first limitation might be alleviated by predicting the change of the reuse distance distribution in function of some characteristic of the program input, e.g. as proposed in [16,17], the use of reuse distance *distributions* still prohibits to select different hints for different memory accesses generated by a single instruction.

Both limitations are addressed in the next two sections. In section 5, the reuse distance is *calculated parametrically* for each individual memory access, and for all possible program executions. In section 6, the parametric reuse distances are used to generate dynamic target hints.

5 Reuse Distance Calculation

A program which runs for more than a couple of seconds typically performs billions of memory accesses. Therefore it is not feasible to store the reuse distance for every memory access separately. Furthermore, at run-time, cache hints should require only a minimum of extra bytes and computations to encode them, in order not to increase the pressure on instruction and data caches. Therefore, both at compile-time, a concise way of representing the reuse distances is needed, and at run-time, an efficient way of computing the cache hints is needed.

The profiling method described in section 4, used to determine the static cache hints, generates a small compile-time representation by only storing the distribution of the reuse distance per instruction. At run-time, the static hints do not require extra computations. However, only one cache hint is specified per instruction, even when the different accesses require different cache hints. In contrast, dynamic cache hints are computed at run time, and are tailored to the locality of each individual memory access. In order to make dynamic cache hints feasible, without much run-time overhead, in this section a method for calculating a concise representation of exact reuse distances for each memory access is described. The reuse distance equations are defined for the class of programs that fit the *polyhedral model*[18]. In section 6, the results of the calculation are used to generate code which dynamically selects the best cache hint for each access.

5.1 Polyhedral Program Model

Central to the polyhedral model[18,1,14] are integer polytopes.

Definition 4. An *integer polyhedron* is a set of integer points

$$\{x \in \mathbb{Z}^n \mid Ax \leq b\}, \tag{1}$$

where A is an integer matrix and b is an integer vector[42]. The set of integer points described by eq (1) is an *integer polytope*[42] if the number of points in the set is finite.

A polyhedral model of a program describes the access patterns of indexed arrays by integer polytopes. Basically, the programs which fit the polyhedral model satisfy the following conditions:

- Constant scalar variables are considered symbolic program parameters. An example of such a parameter is `N` in figure 5.

```
do k = 1, N
  A(k,k)=k
enddo
do i = 1, N
  A(2i,1)=1
  do j = i+2, N-i
    if (i<>j) A(i,j-i) = A(3+j,i)
  enddo
enddo
```

Fig. 5. Example program.

- The program consists of assignment, if and loop statements.
- The set of loop iterations for which a statement is executed must be representable as a union of integer polyhedra. The union of integer polyhedra is called the iteration space[52] of that statement, which contains a point for every iteration in which the statement is executed. This is possible if the loop boundaries are affine functions of outer loop induction variables and program parameters.
- The variables in the program are either scalars or arrays. Scalars can be treated as one-dimensional arrays with size 1. The index expressions of the array variables are affine functions of the loop induction variables and program parameters.

An example program which fits the polyhedral model is shown in figure 5.

To be consistent with the terminology used in the analytical calculation of cache behavior, "reference" is used as a synonym for "memory instruction" in this section (see definition 1). In order to calculate the reuse pairs $\mathsf{reuse}\,(\rightarrow)$, their accessed data sets ADS and the corresponding reuse distance; the set of references $\mathcal{R}$, the iteration space of a reference $\mathsf{IS}(r)$, the set of array variables $\mathcal{V}$, and the execution order of two accesses should be formally described. They are defined as follows:

Definition 5. The **set of all the references** in a program is denoted by $\boldsymbol{\mathcal{R}}$. The **set of variables** in a program is denoted by $\boldsymbol{\mathcal{V}}$. The **iteration space** of the statement in which a reference r occurs is denoted by $\boldsymbol{\mathsf{IS}(r)}$. The **memory location** which is accessed by r at iteration i is denoted by $\boldsymbol{r@i}$. The fact that iteration i of reference r is **executed before** iteration j of reference s is expressed as $\boldsymbol{i_r \lessdot j_s}$. The set of the program parameters is denoted by $\boldsymbol{\mathcal{P}}$.

Example 1. In figure 5, a small program is shown which fits the polyhedral model. To clarify the notations introduced in definition 5, some examples applied to the program in figure 5 are given here:

- The variable set $\mathcal{V} = \{A\}$.
- The reference set $\mathcal{R} = \{\mathtt{A(k,k)}, \mathtt{A(2i,1)}, \mathtt{A(i,j-i)}, \mathtt{A(3+j,i)}\}$.
- The iteration space $\mathsf{IS}(\mathtt{A(i,j-i)}) =$

$\{(i,j) : (1 \leq i \leq N) \wedge (i+2 \leq j \leq N-i) \wedge (i < j \vee i > j)\}$.
- The mapping of iteration to data space $\mathtt{A(3+j,i)}@(i=3,j=6) = A(9,3)$.
- The execution order constraint $(i)_{\mathtt{A(2i,1)}} \lessdot (i',j')_{\mathtt{A(i,j-i)}} \;=\; i \leq i'$.
- The parameter set $\mathcal{P} = \{\mathtt{N}\}$.

The execution order constraint $i_r \lessdot j_s$ is computed as follows. First, the `do`-loops surrounding both references r and s are looked up. i_r is executed before j_s if i_r is a lexicographically[52] smaller iteration point than j_s for the common surrounding loops, or if $i_r = j_s$ for the surrounding loops and r comes textually before s in the body of the surrounding loops. Note that the execution order constraints can also be computed in loop-structures based on `goto` or `while` statements, as long as they can be converted into `do`-loops[55].

5.2 Reuse Distance Equations

The reuse distances of the individual references in the program are calculated in 3 steps:

(1) The reuse pairs in the memory access stream are calculated. For every couple of *references* (r,s), a set of polytopes $\mathsf{reuse}\,(r \rightarrow s)$ is generated, which represents all reuse pairs for which the first access is generated by an execution of reference r, and the second access is generated by s.
(2) For each set of reuse pairs, a set of polytopes is constructed which describes the accessed data set (ADS) of the reuse pairs in the set.
(3) The number of different memory locations in the ADS is counted, which equals the reuse distance of the reuse pair. The count is expressed by an Ehrhart polynomial[13].

The three steps are explained below.

5.2.1 Reuse pair

Every memory access $a = (r, I_r)$ is uniquely defined by the reference r which generates the access, and the iteration point I_r at which the access occurs. Therefore a reuse pair, consisting of two memory accesses $a_1 = (r, I_r)$ and $a_2 = (s, J_s)$, is uniquely defined by the tuple (r, s, I_r, J_s).

All reuse pairs $\langle a_1, a_2 \rangle$ for which the first access a_1 originates from reference r and the second access a_2 originates from reference s, are combined into the set of reuse pairs denoted by $\mathsf{reuse}\,(r \rightarrow s)$, which contains the iteration points I_r and J_s that generate a reuse. These iteration points are described by the

following simultaneous equations:

$$\forall r, s \in \mathcal{R} : \mathsf{reuse}\,(r \rightarrow s) = \{(I_r, J_s) \in \mathbb{Z}^n \mid \text{subject to conditions (2a)– (2d)}\} \tag{2}$$

$$I_r \in \mathsf{IS}(r) \wedge J_s \in \mathsf{IS}(s) \qquad (\textit{iteration space}) \quad (2a)$$

$$I_r \lessdot J_s \qquad (\textit{execution order}) \quad (2b)$$

$$r@I_r = s@J_s \qquad (\textit{same location}) \quad (2c)$$

$$\forall t \in \mathcal{R} : \neg(\exists K_t \in \mathsf{IS}(t) : I_r \lessdot K_t \lessdot J_s \wedge t@K_t = r@I_r) \qquad (\textit{no intervening access}) \quad (2d)$$

The above formula gives the constraints which must be satisfied before a reuse occurs between $r@I_r$ and $s@J_s$. Equation (2a) expresses that I_r and J_s are part of the iteration space of respectively r and s. (2b) indicates that I_r must be executed before J_s; (2c) encodes that the same memory location must be accessed; and (2d) ensures that no intervening memory access touches the same memory location. Furthermore, the following formulas define the iteration points at which *forward* and *backward* reuse, respectively, occurs:

$$\begin{aligned} \mathsf{reuse_F}\,(r) &= \bigcup_{\forall s \in \mathcal{R}} \{I_r \mid \exists J_s : (I_r, J_s) \in \mathsf{reuse}\,(r \rightarrow s)\} \\ \mathsf{reuse_B}\,(s) &= \bigcup_{\forall r \in \mathcal{R}} \{J_s \mid \exists I_r : (I_r, J_s) \in \mathsf{reuse}\,(r \rightarrow s)\} \end{aligned} \tag{3}$$

An example of the above equations for a simple program is shown in figure 6.

5.2.2 Accessed data set of a reuse pair

The function map_r^A maps an iteration space to the elements of array A accessed by r, while $\mathsf{iters}_t(I_r, J_s)$ is the set of iterations of reference t executed between iteration I_r and iteration J_s:

$$\mathsf{map}_r^A = \{I \rightarrow r@I \mid I \in \mathsf{IS}(r) \wedge r \text{ accesses array } A\} \tag{4}$$

$$\mathsf{iters}_t(I_r, J_s) = \{K_t \in \mathsf{IS}(t) \mid I_r \in \mathsf{IS}(r) \wedge J_s \in \mathsf{IS}(s) \wedge I_r \lessdot K_t \lessdot J_s\} \tag{5}$$

Now $\mathsf{ADS}^A\,(\mathsf{reuse}\,(r \rightarrow s))$, the elements of A that are in the ADS of the reuse pairs in $\mathsf{reuse}\,(r \rightarrow s)$, is expressed as follows:

$$\mathsf{ADS}^A\,(\mathsf{reuse}\,(r \rightarrow s)) \;=\; \bigcup_{t \in \mathcal{R}} \mathsf{map}_t^A(\mathsf{iters}_t(\mathsf{reuse}\,(r \rightarrow s))), \tag{6}$$

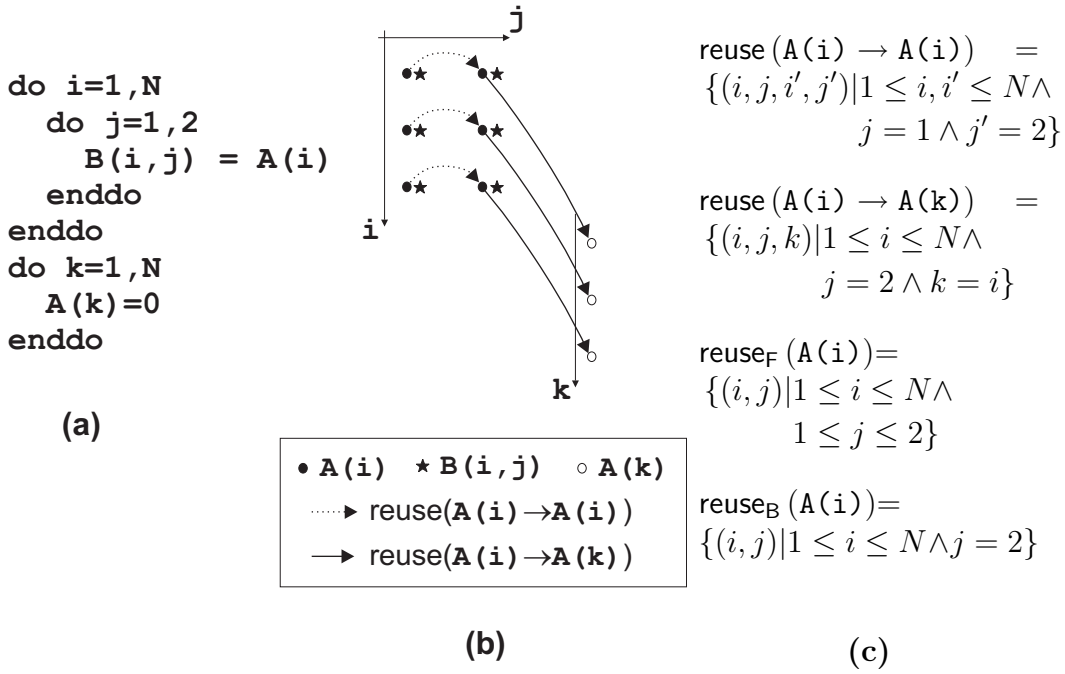


Fig. 6. Calculating intra- and inter-nest reuses. In (a), the example program is shown. In (b), the reuse pairs are shown as arrows between the iteration points of the three different references. There's only reuse between references accessing array `A`; each element of array `B` is accessed only once. In (c), the reuse pairs are described by integer polytopes.

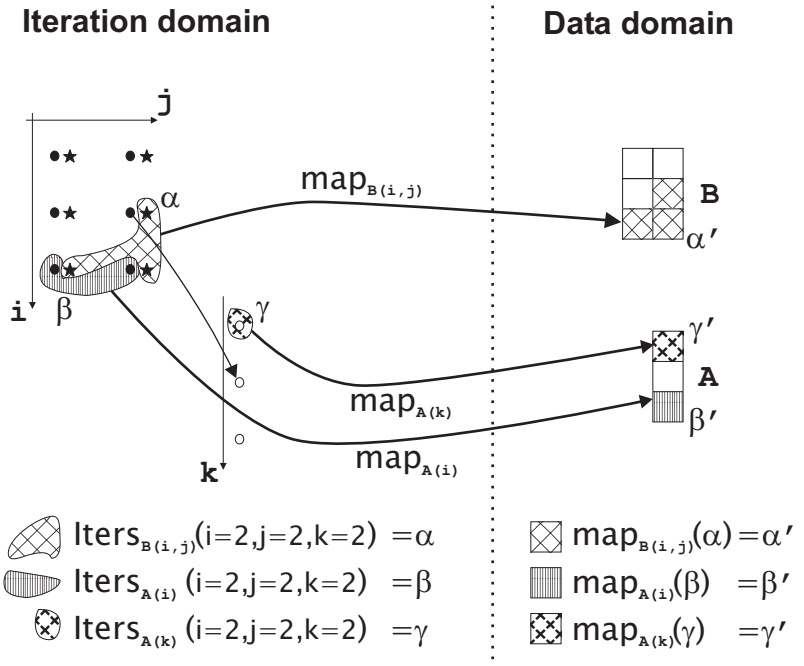
Equation (6) expresses that the ADS of a reuse pair can be found by first calculating the iterations between use and reuse. Then, the ADS is simply all the data locations which are touched by the accesses in the iterations between use and reuse. The calculation of the ADS for a reuse pair of the program in figure 6(a) is shown in figure 7.

5.2.3 Reuse distance of a reuse pair

In order to find the reuse distance of a reuse pair, the number of different memory locations in its ADS needs to be counted. This count is expressed as follows, assuming that the different arrays $A \in \mathcal{V}$ don't overlap:

$$\mathsf{RD}(r,s) = \sum_{A \in \mathcal{V}} \left| \mathsf{ADS}^A \left(\mathsf{reuse}\,(r \rightarrow s) \right) \right|_{\mathsf{param}(I_r, J_s, \mathcal{P})} \tag{7}$$

$\mathsf{ADS}^A\left(\mathsf{reuse}\,(r \rightarrow s)\right)$ is a set of parameterized integer polyhedra. $|p|_{\mathsf{param}(m,n)}$ means the number of integer points in polytope p, in function of m and n, i.e. the number of points when m and n are considered as a fixed parameter. Parametrically counting the number of points in such a set, can be performed by the methods discussed in [13], [36] and [49]. Besides calculating the reuse distance of a reuse pair, it is also possible to compute the forward and back-



(a)

$$\begin{aligned}\mathsf{ADS}^{\mathtt{A}}(\mathsf{reuse}(\mathtt{A(i)} \rightarrow \mathtt{A(k)})) =& \{(x) \mid 1 \leq x < i \leq N\} \\ & \cup \{(x) \mid k < x \leq N\} \\ \mathsf{ADS}^{\mathtt{B}}(\mathsf{reuse}(\mathtt{A(i)} \rightarrow \mathtt{A(k)})) =& \{(x,y) \mid x = i \wedge y = 2\} \\ & \cup \{(x,y) \mid 1 \leq y \leq 2 \wedge i < x \leq N\}\end{aligned}$$

(b)

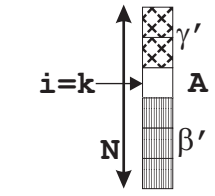
Fig. 7. Calculating $\mathsf{ADS}(\mathsf{reuse}(\mathtt{A(i)} \rightarrow \mathtt{A(k)}))$, for the program in figure 6(a). In (a), a single reuse pair is shown (from reference `A(i)` at iteration point (i=2) to the access made by reference `A(k)` at iteration point (k=2)). On the left hand side, $\mathsf{iters}_{\mathtt{A(i)}}(i{=}2, j{=}2, k{=}2)$, $\mathsf{iters}_{\mathtt{B(i,j)}}(i{=}2, j{=}2, k{=}2)$ and $\mathsf{iters}_{\mathtt{A(k)}}(i{=}2, j{=}2, k{=}2)$ are indicated in the iteration spaces as areas α, β and γ. After applying the mapping functions $\mathsf{map}^{\mathtt{A}}_{\mathtt{A(i)}}$, $\mathsf{map}^{\mathtt{B}}_{\mathtt{B(i,j)}}$ and $\mathsf{map}^{\mathtt{A}}_{\mathtt{A(k)}}$, the parts of arrays A and B that are accessed by the iterations between (i=2, j=2) and (k=2), are shown as α', β' and γ'. If the number of elements in $\alpha' + \beta' + \gamma'$ is less than the cache size, a cache hit results. In (b), the accessed data sets are represented by integer polytopes.

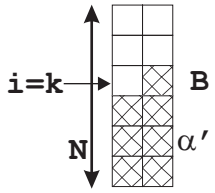
ward reuse distances of a memory reference r. These are denoted by $\mathsf{FRD}(r)$ and $\mathsf{BRD}(r)$:

$$\mathsf{FRD}(r) = \sum_{s \in \mathcal{R}, A \in \mathcal{V}} \left| \mathsf{ADS}^{A}(\mathsf{reuse}(r \rightarrow s)) \right|_{\mathsf{param}(I_r, \mathcal{P})} \tag{8}$$

$$\mathsf{BRD}(s) = \sum_{r \in \mathcal{R}, A \in \mathcal{V}} \left| \mathsf{ADS}^{A}(\mathsf{reuse}(r \rightarrow s)) \right|_{\mathsf{param}(J_s, \mathcal{P})} \tag{9}$$

Furthermore, the reuse distance theorem 1 can be used to calculate at which





(a)

- $\mathsf{RD}(\mathtt{A(i)}, \mathtt{A(k)}) = \left|\mathsf{ADS}^{\mathtt{A}}\left(\mathsf{reuse}\left(\mathtt{A(i)} \rightarrow \mathtt{A(k)}\right)\right)\right|_{\mathsf{param}(\mathtt{i,j,k,l,N})}$
  $+ \left|\mathsf{ADS}^{\mathtt{B}}\left(\mathsf{reuse}\left(\mathtt{A(i)} \rightarrow \mathtt{A(k)}\right)\right)\right|_{\mathsf{param}(\mathtt{i,j,k,l,N})}$
  $= |\{(x) \mid 1 \leq x < i \leq N\}|_{\mathsf{param}(\mathtt{i,j,k,l,N})}$
  $+ |\{(x) \mid k < x \leq N\}|_{\mathsf{param}(\mathtt{i,j,k,l,N})}$
  $+ |\{(x,y) \mid x = i \wedge y = 2\}|_{\mathsf{param}(\mathtt{i,j,k,l,N})}$
  $+ |\{(x,y) \mid 1 \leq y \leq 2 \wedge i < x \leq N\}|_{\mathsf{param}(\mathtt{i,j,k,l,N})}$
  $= i$ (when $1 \leq i \leq N$)
  $+ N - k$ (when $k \leq N$)
  $+ 1$
  $+ 2(N - i)$ (when $i \leq N$)
  $= 3N - i - k = 3N - 2i$ (when $1 \leq i = k \leq N$)
- $\mathsf{FRD}(\mathtt{A(i)}) = 3N - 2i$ (when $j = 2$)
- $\mathsf{BRD}(\mathtt{A(k)}) = 3N - 2i$
- $\mathsf{COLDM}(\mathtt{A(i)}) = \{(i,j) \mid 1 \leq i \leq N \wedge j = 1\}$
- $\mathsf{NOKEEP}(\mathtt{A(i)}) = \{(i,j) \mid 1 \leq i \leq N \wedge j = 2 \wedge 3N - 2i \geq CS\}$
- $\mathsf{CAPM}(\mathtt{A(k)}) = \{(k) \mid 1 \leq k \leq N \wedge 3N - 2k \geq CS\}$

(b)

Fig. 8. In figure **(a)**, the accessed data elements of array A between use and reuse for the reuse in figure 7 is shown graphically. The amount of data in this set is $3N - 2i$, which equals the reuse distance of that reuse. In figure **(b)**, the forward and backward reuse distance, the iteration points where cold misses and capacity misses occur, and the iteration points for which the data will not be retained in the cache are described in function of matrix size N, using the equations (7)–(12).

iteration points the data is not found in the cache and for which iteration points the data will not be retained in the cache. The iteration points where no backward reuse occurs are those where the data is fetched the first time, resulting in a cold miss. The iteration points at which a cold miss occurs for reference r are denoted by $\mathsf{COLDM}(r)$:

$$\mathsf{COLDM}(r) = \{I \mid I \in \mathsf{IS}(r) \wedge I \notin \mathsf{reuse}_{\mathsf{B}}(r)\} \tag{10}$$

Similarly, the iteration points at which the reuse is larger than the cache size exhibits *capacity misses*, denoted by $\mathsf{CAPM}(r)$:

$$\mathsf{CAPM}(r) = \{I \mid \mathsf{BRD}(r) \geq CS \wedge I \in \mathsf{reuse}_{\mathsf{B}}(r)\}, \tag{11}$$

where CS denotes the cache size. The iteration points at which the accessed data will not be retained in the LRU cache can be computed taking into account the reuse distance theorem 1, and is denoted by $\mathsf{NOKEEP}(r)$:

$$\mathsf{NOKEEP}(r) = \{I \mid \mathsf{FRD}(r) \geq CS \wedge I \in \mathsf{reuse}_{\mathsf{F}}(r)\}, \tag{12}$$

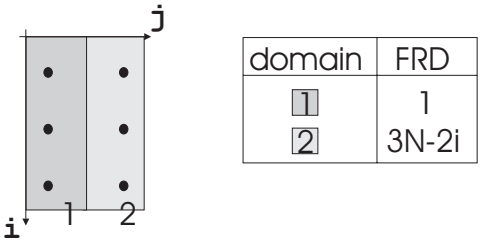


Fig. 9. The forward reuse distance of the iterations of reference `A(i)` are described by two different polynomials in two different domains of the iteration space.

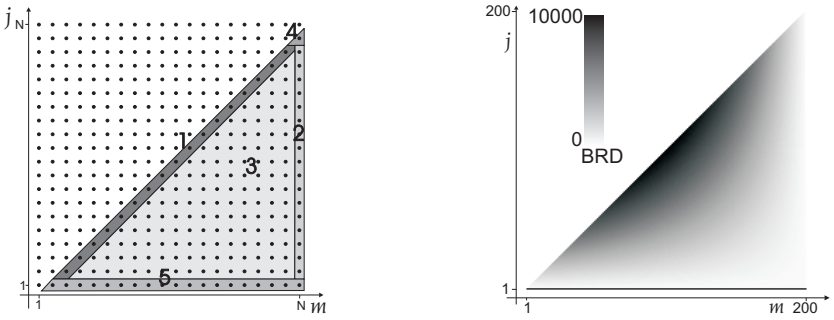
```
do j = 1, N
  do l = j, N
    do k = 1, j-1
       A(l,j) = A(l,j) - A(l,k) * A(j,k)
    enddo
  enddo
  A(j,j) = sqrt(A(j,j))
  do m = j+1, N
    A(m,j) = A(m,j) / A(j,j)
  enddo
enddo
```

Fig. 10. Cholesky factorization

The above formulas all work with polyhedra. Polyhedral tools such as the Omega library[36], the Polylib library[13] and the Barvinok library[48] are used to calculate, manipulate and count the polyhedra described above. Examples of the above equations are given in figure 8. The forward reuse distances of the different iterations of reference `A(i)` are shown in figure 9. It consists of two domains. One domain originates from the reuse pairs $\mathsf{reuse}\,(\texttt{A(i)} \rightarrow \texttt{A(i)})$, the other domain corresponds to the reuse pairs $\mathsf{reuse}\,(\texttt{A(i)} \rightarrow \texttt{A(k)})$.

5.3 Example: Cholesky Factorization

As an example of calculating the reuse distances of a real program, the Cholesky factorization code is discussed (see figure 10). In figure 11, the calculated backward reuse distances for one of the references is shown. The intermediate steps in this calculation are presented in detail in [3]. The result of the counting in equations (7)–(9) is a set of iteration space domains, each with a corresponding polynomial that describes the count in that domain[13]. The different reuse distance domains for the iteration space of reference `A(m,j)` are shown in the top left of figure 11. For each domain, the corresponding reuse distance is shown in the table in the middle. In the top right of figure 11, the actual backward reuse distance is shown for the different iterations. In comparison to the cumulative reuse distance distribution in the bottom left, the analytical calculation produces more detailed information: for every access, the exact



nr	in domain	BRD
1	$3 \leq m = j + 1 < N$	$(N - m)j + N - 1$
2	$m = N \wedge 2 \leq j \leq N - 2$	$m - j$
3	$2 \leq j < m - 1 < N - 1$	$(N - m)j + m - 1$
4	$j + 1 = m = N \geq 3$	1
5	otherwise	∞

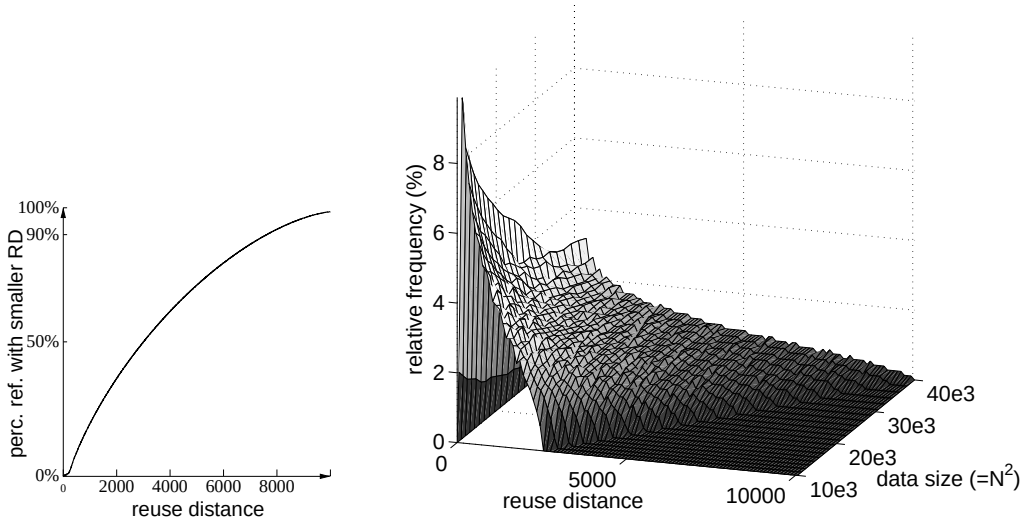
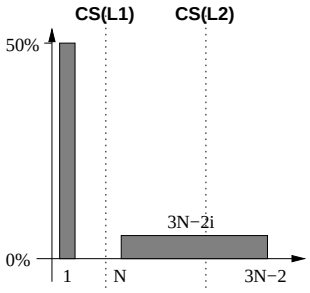


Fig. 11. The iteration space of read reference `A(m,j)` (see figure 10) is shown on the top left for $N = 20$ and is divided into 5 domains. The table shows the calculated parametric backward reuse distances for the 5 domains. In the top right, the backward reuse distance of the iteration points of the same reference are shown by color, for N=200. If the pixel representing the iteration is white, the BRD is 0, when it's black, the BRD is 10000. In comparison, the cumulative reuse distance distribution for the reference is shown in the bottom left. This is the most detailed information that can be obtained by profiling, but clearly contains less information than the information provided by the calculated reuse polynomials. Moreover, the polynomials in the table specify the reuse distances for all possible data sizes, indicated by N. This is graphically shown on the bottom right, where the relative reuse distance histogram (bin size=100) for matrix sizes $100 \times 100 \leq N \times N \leq 200 \times 200$ is shown. For a vertical slice at fixed N, the surface under the plot sums up to 100%. The histogram shows that when N increases, the backward reuse distances are distributed over a wider range and the average BRD increases.

```
do i=1,N
  do j=1,2
    B(i,j) = A(i)
  enddo
enddo
do k=1,N
  A(k)=0
enddo
```



iteration domain	FRD
$1 \leq i \leq N \wedge j = 1$	1
$1 \leq i \leq N \wedge j = 2$	$3N - 2i$

Fig. 12. An example of a reuse distance distribution for which it is impossible to statically select good hints for all accesses. On the left hand side, a program is shown, and on the right hand side, the reuse distance distribution of the reference `A(i)` in this program is shown. Half of the accesses generated by `A(i)` have reuse distance 1 requiring cache hint C1. The other half have reuse distance $3N - 2i$, requiring a cache hint dependent on the values of N and i.

reuse distance is known. Furthermore, the polynomials indicate the reuse distance for every possible data size, indicated by program parameter N. This is shown graphically at the bottom right.

6 Dynamic Cache Hint Selection

Two major issues arise when using the static cache hint selection scheme described in section 4. The first problem is that a cache hint is tied to an instruction and not to a single memory access. When the reuse distance distribution of the memory instruction is multimodal, different hints might be required for different iterations. An example is shown in the bimodal reuse distance distribution of figure 12. The second problem is that the locality of the memory accesses generated by an instruction depends on the input of the program. For example, if the program performs a matrix computation, the size of the matrices determines the cache level at which data is found. Therefore, the optimal cache hints also depend on information that is in general only known at run-time. In figure 12, the second peak of memory access `A(i)` is such an example, where the forward reuse distance $3N - 2i$ depends on the array size N.

In order to mitigate the above problems, cache hints may be selected at run-time, based on the actual reuse distance of the current access. In this section, the analytically calculated reuse distance for every memory access (see section 5) is used to generate dynamic hints. In general, for each reference, the

```
do i=1,N
 do j=1,2
 if (j.eq.1) then
  FRD_Ai = 1
 if (j.eq.2) then
  FRD_Ai = 3*N-2*i
 B(i,j) = A(i)
 enddo
enddo
do k=1,N
 A(k)=0
enddo
```

(a) exact target hints

```
do i=1,N
 FRD_Ai = 1
 B(i,1) = A(i)
 FRD_Ai = 3*N-2*i
 B(i,2) = A(i)
enddo
do k=1,N
  A(k)=0
enddo
```

(b) after loop peeling

```
FRD_A1 = 1
FRD_Ai = 3*N-2
do i=1,N
 B(i,1) = A(i), frd=FRD_A1
 B(i,2) = A(i), frd=FRD_A2
 FRD_A2 = FRD_A2-2
enddo
do k=1,N
  A(k)=0
enddo
```

(c) after optimization

Fig. 13. The program in figure 12 with dynamic hint calculation for reference A(i).

calculation results in a number of domains in the iteration space, where for each domain a polynomial represents the reuse distance. The variables in the polynomial can be the induction variables and the program parameters. For example, for reference `A(i)` in fig. 12, two different domains in the iteration space are discovered. The first domain exhibits reuse distance 1, while the second domain exhibits reuse distance $3N - 2i$.

6.1 Code Generation

In a first stage, it is determined to which iteration domain the current iteration belongs, by inserting if-tests. For the example code in fig. 12, the code for computing the forward reuse distance of reference `A(i)` is shown in fig. 13(a).

The value computed in variable `FRD_Ai` represents the forward reuse distance of reference `A(i)`, which is used to dynamically select the corresponding cache hint. We present two methods. The first method uses predicates. The second method requires a small extension to the IA-64 architecture:

(1) In order to select the most appropriate cache hint, the load instruction is replicated with different hints. The instruction with the appropriate hint is then selected using predicates. As an example, consider the following code. Assume that the reuse distance value for the given iteration is calculated and stored in register `r10`. The original load instruction loads to register `r5`. `CS1` and `CS2` are the cache sizes of the first level and second level cache. The IA-64 code executes a single load instruction with the appropriate cache hint, according to the calculated reuse distance:

```
     cmp.lt     p6, p7 = r10, CS1 ;;
(p7) cmp.ge.unc p8, p7 = r10, CS2
(p6) ld.t1      r5 = ...          ;;
(p7) ld.nt1     r5 = ...
(p8) ld.nta     r5 = ...
```

Exactly one of the predicate registers `p6`, `p7`, `p8` will be true after the two `cmp` instructions. `p6` is true if the reuse distance is smaller than the level 1 cache size, `p7` is true if it is between level 1 and level 2 cache size and `p8` is true if it is larger than level 2. A predicate between brackets before an instruction means that the instruction will only be executed if the predicate is true. Consequently, only the load instruction with the proper cache hint will be executed. The instructions between consecutive stop bits `;;` are executed in parallel.

(2) A second method requires an extension to the IA-64 architecture, allowing more efficient and portable dynamic hints. The memory instructions, such as load, store and prefetch, are extended with an extra input register operand which contains the forward reuse distance of the memory access. An example of such a load instruction is `ld r5=[r6],frd=r7`, where `r7` contains the forward reuse distance. In the encoding of the IA-64 memory instructions, unused bits are available in which the extra input register operand can be encoded in a backwards compatible way[23].

At compile time, it is not necessary to know the machine dependent cache sizes of the different cache levels, since at run-time the processor will keep the data in the cache levels which are larger than the forward reuse distance. This makes recompilation of the binaries for processors in the IA-64 family with different cache sizes no longer necessary, as far as the target cache hint selection is concerned. Furthermore, it is easy to generate code without target cache hint selection. On IA-64, register `r0` always contains 0. So an instruction like `ld r5=[r6], frd=r0` would specify that the forward reuse distance is 0, leading to the default LRU replacement policy for all cache sizes, since every cache level is larger than 0. Finally, as opposed to the first method, the memory instruction doesn't need to be duplicated for every cache level in the memory hierarchy. The experiments in section 7 show that the extra calculations and increased register usage (for register `r7`) don't generate a significant execution time overhead, provided the overhead reduction techniques in section 6.2 are applied.

6.2 Overhead Reduction

The code and execution time overhead of calculating the forward reuse distances can be quite large. The reuse distance of a reference is described by

different polynomials for different iteration domains. If the reuse distance is described by d domains, before every execution of the reference, up to d if-tests need to be carried out to decide which polynomial should be evaluated. However, when performing experiments, we found that most references exhibit one single *dominant iteration domain*.

Definition 6. For a reference r, an iteration domain with a corresponding polynomial, describing r's reuse distance, is a *dominant domain* if and only if the dimension of that domain is the maximum of all iteration domains for that reference.

For example, consider the reuse distance of reference `a(m,j)` in figure 11. The reference has 5 domains, but domain 3 contains most iterations. Actually, domain 3 is 2-dimensional and all other domains are at most 1-dimensional. The corresponding *dominant polynomial* for reference `a(m,j)` is $(N - m)j + m - 1$.

The overhead is reduced by using only the dominant polynomial when calculating the reuse distance. As a result, no if-tests are executed. For most accesses, the correct reuse distance is calculated. Furthermore, for the benchmarks used in the experiments, we found that the non-dominant domains are only located at the borders of the loops, i.e. at the first or last iteration of a loop. Therefore, the iteration points in the non-dominant domains can be computed efficiently (without if-tests), by peeling of the first or last loop iteration. In the peeled iteration, the corresponding polynomial is used instead of the dominant polynomial. For example, the first iteration of the `j`-loop in fig. 12 can be peeled of (after which it becomes fully unrolled since it only has 2 iterations). The resulting code is shown in fig. 13(b).

Most reuse distance polynomial computations are easily optimized by standard compiler optimizations such as loop invariant code motion and strength reduction. Many of the computed polynomials are independent of the inner loop variable and their computation can be moved completely out of the inner loop. For the polynomials that depend on the induction variable of the inner loop, often strength reduction is able to reduce the polynomial computation to a single addition or subtraction instruction. The result of these optimizations on the running example is shown in fig. 13(c).

7 Experimental Results

7.1 Static Cache Hint Selection Implementation

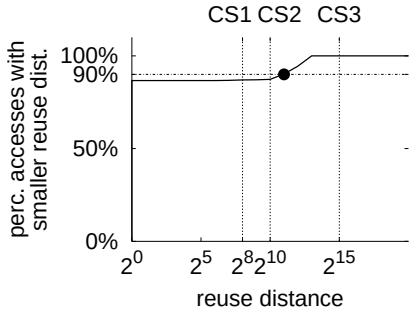
The static cache hint selection scheme presented in section 4 has been implemented in the Open64 compiler for the Itanium[34], which is based on SGI's Pro64 compiler. The reuse distance distributions for the memory instructions are obtained by instrumenting and profiling the program. After profiling, the cumulative reuse distance distribution for every instruction is stored. During compilation, the profile data is read by the compiler and based on the reuse distances, the appropriate source and target cache hints are calculated for each memory operation, in the way described in section 4.2.

The target hints were added to the intermediate representation by representing them as an extra operand to the memory instructions. When the compiler generates assembly code for memory instructions, the corresponding IA-64 cache hint is written out. The source hints are implemented as follows. The compiler back-end has a parametric description of the processor, similar to the machine description system MDES discussed in [37]. The machine description specifies, amongst others, the latency of a given instruction. When the compiler is retargeted to a new processor with different latencies, only the machine description needs to be changed. The source hints are incorporated into the compiler by adding new memory instructions in the machine description table. An example of such an instruction is `ldfd_C3` in figure 14. This new instruction has the same characteristics as the existing instruction `ldfd`, except that the machine description indicates the latency of a L3 cache access. In this way, the scheduler is able to take into account the longer latency.

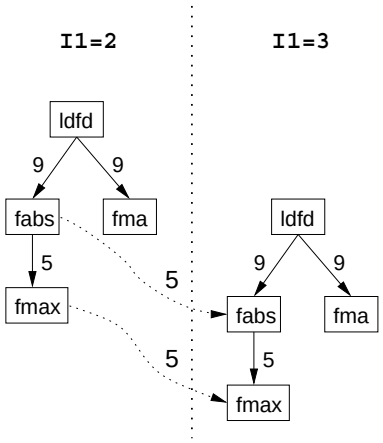
In figure 14, an example of the effect of source cache hints for a loop from the `mgrid` program is shown. The source code is shown in figure 14(a). The compiler generates one memory instruction for the loop nest: a load instruction `ldfd` which fetches `R(I1,I2,I3)`. In figure 14(b), the cumulative reuse distance distribution of the load instruction is shown. The granularity at which the reuse distance is measured is the memory line, which is 64 bytes on the Itanium, i.e. all accesses to the same consecutive 64 bytes in memory are considered to be accesses to the same data location. Since the loop fetches consecutive data elements from array `R`, and 8 elements fit in a single memory line, 7 out of 8 accesses are to the same memory line as the least recently touched. Therefore 7 out of 8 accesses (=87.5%) have reuse distance 0. The reuse distance of the 8th access, which fetches a new memory line, depends on where that memory line has last been accessed in other parts of the program. The reuse distance distribution shows that the reuse distance for these accesses are between 2^{10} and 2^{13}. The 90th percentile is at 2^{11} memory lines,

```
DO I1=2,N1-1
   S=S+R(I1,I2,I3)**2
   A=ABS(R(I1,I2,I3))
   IF(A.GT.RNMU)RNMU=A
ENDDO
```

(a) source code from MGRID



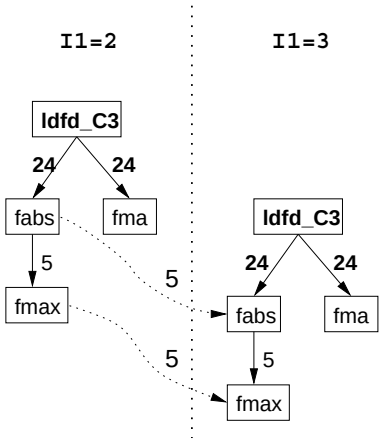
(b) cumulative backward reuse distance distribution




```
//<swps>   5 cycles per 1 iteration in steady state
//<swps>   4 pipeline stages

.Lt_7_17:
    (p18) fabs f34=f38              // [2*II+3]
    (p18) fma.d.s0 f39=f38,f38,f40  // [2*II+3]
 ;;
    (p16) ldfd f36=[r2],8
         // source hint C2 (latency=9) [0*II+4]
    (p19) fmax.s0 f41=f35,f42       // [3*II+4]
          br.ctop.dptk.few .Lt_7_17 // [3*II+4]
 ;;
```

(c) dependence graph and assembler code without source cache hints




```
//<swps>   5 cycles per 1 iteration in steady state
//<swps>   7 pipeline stages

.Lt_7_17:
    (p16) ldfd f39=[r35],8
        // source hint C3 (latency=24) [0*II+0]
    (p22) fmax.s0 f47=f38,f48       // [6*II+0]
 ;;
    (p20) fabs f36=f43              // [4*II+4]
    (p20) fma.d.s0 f44=f43,f43,f45  // [4*II+4]
          br.ctop.dptk.few .Lt_7_17 // [6*II+4]
 ;;
```

(d) dependence graph and assembler code with source cache hints

Fig. 14. Example of the effect of source hints on software pipelined loop scheduling. In (a), the source code is shown. In (b), the reuse distance distribution of the corresponding load operation is plotted. In (c), the dependence graph and the resulting assembly code without source hints is shown. In the dependence graph, the arcs are labeled with the latencies of the instructions. In (d), the dependence graph and assembly code after source hint selection is shown. The longer latency of the load instruction is hidden by a longer software pipeline (7 instead of 4 stages).

hence the source cache hint is chosen to indicate the first larger cache level which contains more than 2048 memory lines, which is the level 3 cache in this case.

In figure 14(c), the original intermediate representation and assembly code for the loop is shown. In figure 14(d), the same code is extended with a source hint. The source cache hint is selected by changing the instruction `ldfd` into `ldfd_C3`. The scheduler knows that the `ldfd_C3` instruction has a latency of 24 cycles. In order to hide this latency, the generated code consists of 7 pipeline stages instead of 4 pipeline stages in the original code. Therefore, in the code with hint C3, up to 7 iterations of the original loop execute simultaneously to overlap the longer latency. A drawback of the longer schedule is that more registers are needed (`f36-f48`) than in the original schedule (`f34-f42`). The advantage is that at run-time, the processor won't stall. In the code without source cache hints, the schedule is too tight, and the processor stalls.

The software pipelining scheduler can only generate code when there are enough registers to keep all temporary values, i.e. no spill or fill code needs to be generated. When using the source hint selection, we found that for a number of loops, the register pressure was increased too much by the hint to allow software pipelining. Software pipelining is one of the central optimizations for EPIC architectures like the Itanium, therefore we adapted the source hint selection. When a loop could not be software pipelined because of register pressure, all the hints C4 were replaced with C3. If the loop still required too much registers, the hints C3 were replaced with C2.

7.2 Dynamic Cache Hint Selection Implementation

The calculation of reuse distances, as described in section 5, has been implemented in the FPT[15] compiler. The Omega library[36] is used to simplify the formulas generated by equations (2)–(6) and the Barvinok library[48] is used to count the number of integer points in the polyhedra described by equations (7)–(9).

In order to verify the exactness of the equations (2)–(12), they have been calculated automatically for a number of loop-oriented programs, such as the matrix multiplication, Gauss-Jordan elimination, Cholesky factorization and the Tomcatv program from the SPEC-benchmark. Furthermore they were applied to a number of artificial loop nests which were constructed specifically to generate far more irregular reuse distances. The results of the analysis were compared to cache simulation, and analysis and simulation results were identical in all cases.

7.3.1 Static Hints Experiments

The static selection of cache hints was evaluated on a HP rx4610 multiprocessor, equipped with 4 733MHz Itanium processors. The data cache hierarchy consists of a 16KB L1, a 96KB L2 and a 2MB L3 cache. The hardware performance counters of the processor were used to obtain the execution time.

All compilations were performed at optimization level -O2, the highest level at which instrumentation and profiling is possible in the Open64 compiler. The existing framework doesn't allow to propagate the profile information through some optimizations phases at level -O3.

The programs were selected from the Olden and the Spec95fp benchmarks. The Olden benchmark contains programs which use dynamic data structures, such as linked lists, trees and quadtrees. The Spec95fp programs are numerical programs with mostly regular array accesses. For Spec95fp, the profiling was done using the training input sets, while the speedup measurements were done with the large input sets. For Olden, no separate input sets are available, and the training input was identical to the input for measuring the speedup. From each program, four executables were generated: one without hints, one with source hints, one with target hints and one with both source and target hints. Each program was run 100 times, to measure the speedup and the variation in execution time. Figure 15 shows the average speedup results with different combinations of hints. The error bars in the figure indicate the 99% confidence interval of the ratio of original execution time to optimized execution time (=speedup).

The figure shows that the programs run 9% faster on average, with a speedup of 56%. In the worst case, a slight performance degradation of 1% is observed. On average, the Olden benchmarks get a 3% speedup from the target hints, but no speedup from the source hints. To take advantage of the source hints, the instruction scheduler must find parallel instructions to fit in between a long latency load and its consuming instructions. In the pointer-based Olden benchmarks, the scheduler finds little parallel instructions, and cannot profit from its better view on the cache behavior. On the other hand, in the floating point programs, on average a 20% speedup is found, mainly resulting from source hints. The loop parallelism allows the compiler to find parallel instructions, mainly because it allows to software pipeline (SWP) the loops with long latency loads. Without software pipelining, the compiler doesn't find as many parallel instructions to schedule during long latency memory accesses. This can be seen in table 3, where the programs are shown which do profit from source hints. When SWP is enabled, adding source hints results in 12%

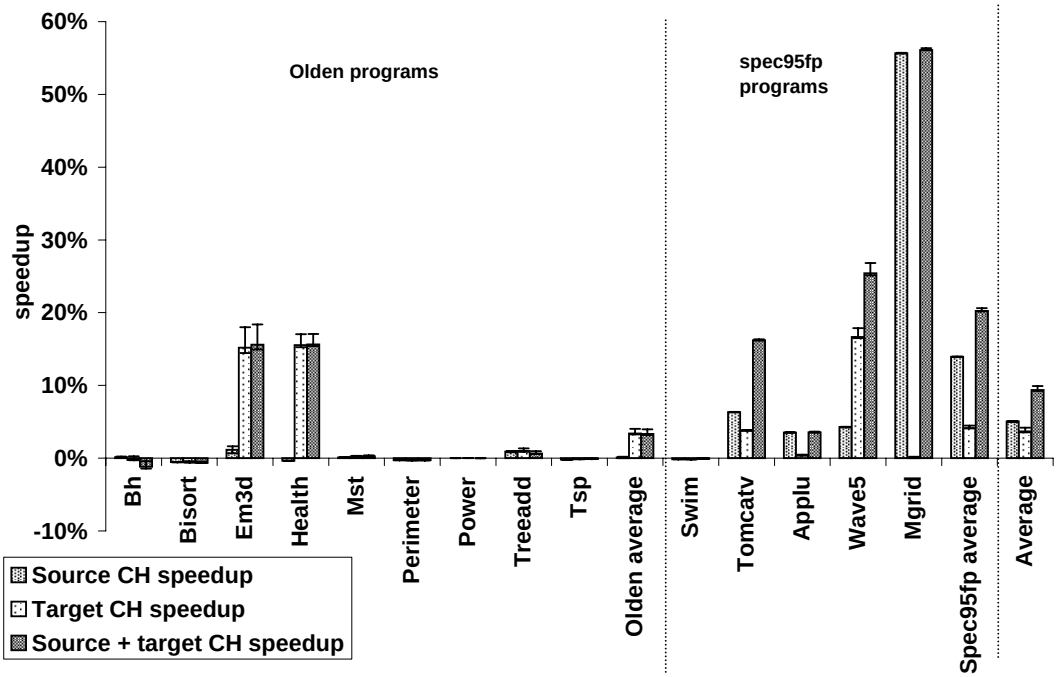


Fig. 15. Speedup after source and target hint generation. The error bars indicate the 99% confidence interval for the speedup, as measured from 100 program runs.

speedup on average. When SWP is disabled, adding source hints results in only 1% speedup on average, because the compiler cannot schedule parallel instructions during long latency memory accesses.

For a number of programs, such as `tomcatv` and `wave5`, combining source and target hints leads to a larger speedup than the sum of the speedups from source and target hints alone. This may be explained by the fact that the hint selection is heuristic, and target hints influence the cache behavior. Source hints estimate the cache behavior of the instruction. Since the cache behavior of the instruction can be changed by target hints, it may be that after target hint insertion, the source hints predict the cache behavior of the instruction better, resulting in an increased speedup.

The relative code size of the programs with static cache hints is shown in figure 16. The target hints have no influence on the code size. For every memory instruction in the IA-64 architecture, 2 bits are reserved to encode the target hint. Traditional compilers just encode hint `.t1` in those 2 bits, whereas our compiler fills in these 2 bits to encode the selected target hint. Consequently, target hints do not increase code size. On the other hand, source hints might change code size, since the compiler creates a different instruction schedule. However, figure 16 shows that the code size increase due to source hints is always less than 1%. Therefore, these experiments indicate that the static hint selection scheme has no scalability issues when the size of the programs increase.

Table 3
Speedup after source hint insertion, as compared to a compilation without cache hints. The left column shows the speedup with software pipelining (SWP) and source hints enabled, compared to SWP without source hints. The right column shows the speedup without SWP but with hints, compared to no SWP and no hints.

program	source CH speedup with software pipelining	source CH speedup without software pipelining
em3d	1%	1%
treeadd	1%	0%
tomcatv	6%	3%
applu	4%	2%
wave5	4%	0%
mgrid	56%	1%
average	12%	1%

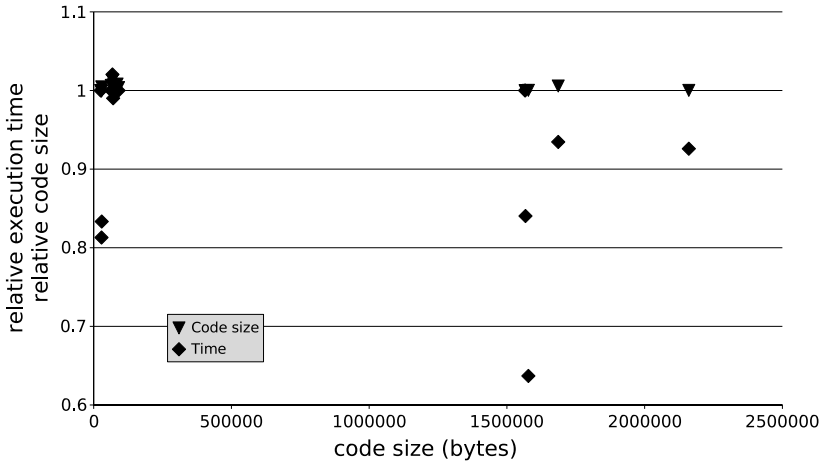


Fig. 16. Relative execution time and code size after static hint generation. For execution time speedup, also see figure 15. The changes in code size are only caused by source hints.

7.3.2 Dynamic Hints Experiments

The method based on profiling sets cache hints per instruction. The analytical calculation gives the exact forward reuse distance for every execution of a memory instruction. Here, the additional advantage of being able to select cache hints per access, instead of per instruction, is measured. In the experiments, we generated IA-64-like cache hints, using the second method described in section 6.1, based on the calculated FRD. A single cache level was simulated, which reacts similarly to cache hints as the Itanium and Itanium2 processors do. When the hint indicates temporal locality, the data is placed in the cache and marked as most recently used. When the hint indicates no temporal locality, the line is still brought into the cache, to exploit potential spatial locality. However, in order not to throw out too much data with temporal locality, the line is marked as the next to be replaced.

Table 4
The cache miss rates for a 4-way set associative LRU 16KB cache with 32 bytes per line. The first column indicates the program name and the cache miss rate for the program without hints. The second column indicates the percentage of cache miss reduction with cache hints, compared to the program without hints. The third column shows the cache miss reduction with dynamic hints.

	miss rate reduction	
program	**static hints**	**dynamic hints**
vpenta(miss-rate=31.56%)	19.00%	18.99%
mxm (miss-rate=3.20%)	0.87%	0.00%
liv18 (miss-rate=68.46%)	7.78%	9.57%
cholesky (miss-rate=19.81%)	-28.59%	9.43%
jacobi (miss-rate=14.32%)	0.02%	0.00%
gauss-jordan (miss-rate=11.90%)	24.82%	34.37%
tomcatv (miss-rate=9.22%)	12.05%	0.00%
average	5.14%	10.34%

Table 5
The overhead of dynamic computation of forward reuse distance and corresponding target cache hints, as compared to the original program without hints. The first two columns show the relative code size and execution time after inserting exact target hints. The third and fourth column show the relative code size and execution time, when only taking into account the dominant domains (see sect. 6.2). The last column shows the relative IPC (instructions per cycle), of the dominant domains-version as compared to the original program without cache hints.

program	exact		dominant domains		
	code size	exec. time	code size	exec. time	rel. IPC
vpenta	4.81	2.48	1.11	1.02	0.98
mxm	1.83	34.45	1.01	1.00	1.00
liv18	47.10	55.36	1.15	1.02	1.16
cholesky	2.17	5.77	1.34	0.98	1.22
jacobi	2.65	10.69	1.01	0.99	0.89
gauss-jordan	2.71	72.62	1.15	1.01	1.01
tomcatv	92.72	260.88	1.39	1.00	1.08
average	22.00	63.18	1.17	1.00	1.05

The data cache miss rates for a number of loop-oriented programs were measured. The relative miss rates for the programs compiled without hints, with static hints (per instruction) and with dynamic hints (per access) are shown in table 4. The table shows that on average static hints reduce the number of misses by 5.14% and dynamic hints reduce misses by 10.34%. The program which profits most from switching from static to dynamic hints is Cholesky. The reason is that it is the program with the most irregular reuse patterns. An example of the irregular reuse distances for this program can be seen in figure 11. Since the same instruction requires different hints, static hints cannot improve the cache behavior. On the other hand, dynamic hints result in a small cache miss reduction.

Table 6
Percentage of correct forward reuse distances and correct cache hints, when using only the polynomials in the dominant domains for all iteration points.

program	% wrong rd	% wrong hint
vpenta	0.31%	0.12%
mxm	0.50%	0.12%
liv18	0.21%	0.01%
cholesky	0.37%	<0.01%
jacobi	0.24%	0.03%
gauss-jordan	0.08%	<0.01%
tomcatv	2.52%	0.09%
average	0.60%	0.05%

As described in section 6.2, generating code so that with each memory access, the exact forward reuse distance is associated, can be costly. In table 5, the overhead of both exact target hints and hints based on the forward reuse distance of the dominant domains (see sect. 6.2) is shown. Since dynamic hints require a small ISA-extension (see second method in sect. 6.1), this code cannot directly be executed on an Itanium processor. In order to measure code size and execution time, we compiled the programs with reuse distance calculation for every memory access, but generated load instructions without the actual hint, i.e. `ld r5=[r6]` instead of `ld r5=[r6], frd=r7`. Exact hints lead to a code size increase of 22 times (as measured in byte-size of the compiled object-file) and an execution time increase of 63 times (as measured by performance counters). However, when only using the dominant domains, there's no execution time overhead on average, and the code size increases by only 17%. The average number of instructions executed per cycle increases by about 5%. Since the execution time remains the same on average, this indicates that the 5% extra instructions for dynamically calculating the hints are scheduled in unused instruction slots by the compiler, leading to no execution time overhead. For `jacobi`, the total number of executed instructions decreases ($0.99 \times 0.89 = 0.88$), even though more computations are performed in the source code. This counter-intuitive result is caused by the non-optimal instruction scheduler in the compiler. By inserting more instructions in the loop body of `jacobi`, the scheduler actually generated a tighter code sequence with less explicit `nop`-instructions.

Table 6 shows that using only the dominant domains has a very low number of wrongly predicted reuse distances: only for 0.6% of the accesses, the calculated reuse distance was wrong. Even if the reuse distance is wrong, the correct hint can result. For the examples, we measured that only 0.05% of the accesses had a wrong cache hint.

Figure 17 plots relative code size and execution time overhead of dynamic target hints against the size of the original programs. The plot shows that the execution time overhead remains below 3%, independent of the size of the

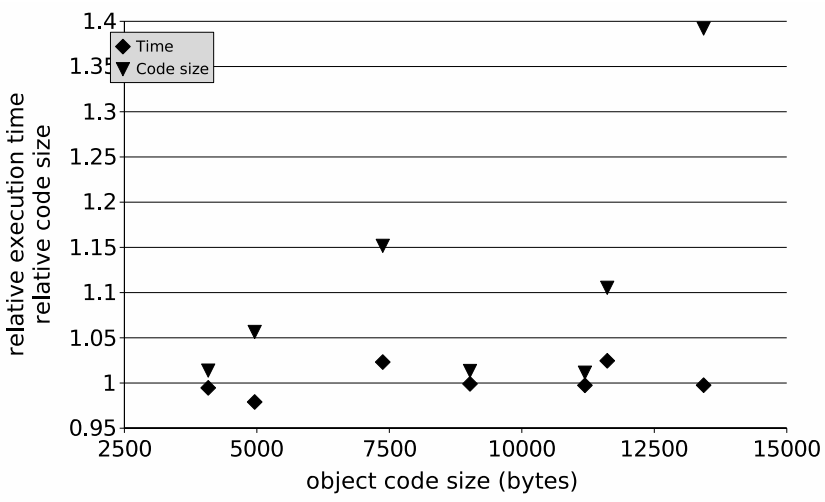


Fig. 17. Code size and execution time overhead for dynamic target hints based on the dominant domains. In contrast to fig. 16, this plot only shows execution time *overhead*. It doesn't include execution time reduction due to improved cache behavior.

Table 7
Comparison of profile-based static hints versus analysis-based dynamic hints.

profile-based static hints	analysis-based dynamic hint
applicable for all programs	only applicable to programs representable in the polyhedral model
Optimized for a single data size	optimized for all possible data sizes
Only a single hint per instruction	The hint is computed for each individual memory access.
Tiny code overhead (<1%)	17% code size increase on average
Compatible with IA-64 ISA	Small extension to IA-64 ISA is necessary for generating efficient code

program, resulting in good scalability. The plot also shows that the largest program gives the largest code size overhead.

7.4 Discussion

The results of the experiments indicate that both static and dynamic hints reduce the number of cache misses. In table 7, the main differences between the static hint and the dynamic hint selection are given. For programs in the polyhedral model, either static or dynamic hints can be generated. When an ISA is available which allows an efficient selection of dynamic cache hints (see sect. 6.1, second method), the analytical method is preferred, since (a) it

optimizes for all possible input; (b) for every memory access the best hint is indicated; (c) the resulting binary is optimized for all possible cache sizes since the actual cache size isn't used in the compiler. On the other hand, when code size is of primary importance, e.g. in embedded systems, static hints might be preferred. Furthermore, in embedded systems the data size and the cache size is often fixed at compile time, so dynamic hints loose some of their advantages in that context.

Both static and dynamic hints are selected based on the reuse distance. One might wonder what the limitations are of reuse-distance based selection. A first limitation of the reuse distance is that it measures temporal locality, whereas caches also exploit spatial locality. The spatial locality is taken into account in the following ways:

- For the profile-based method, the term "memory location" in the reuse distance definition is interpreted as being a "memory line". As such, the profile-based method measures reuse distance as the number of memory lines accessed between two accesses to the same memory line. In this way, spatial locality is taken into account, since two consecutive accesses to the same cache line have reuse distance 0.
- For the analytical method, the term "memory location" in the reuse distance definition is interpreted as being an array element. As such, spatial locality is not explicitly taken into account. Taking spatial locality into account in the formulas would result in a number of difficulties:

(1) The formulas become too complex for the current polyhedral tools to process and simplify them.
(2) If the size of the arrays is only known by parameters (e.g. `DIMENSION A(N,M)` in Fortran), the formulas would require quadratic expressions, resulting in formulas that don't fit in the polyhedral model.
(3) Even if the above two difficulties could be solved, then the exact solution of the reuse distance equations might become too complex to handle effectively, i.e. the number of iteration domains with different polynomials might increase sharply.

The analytical calculation is used to generate dynamic hints, which indicate at which cache levels temporal locality is lacking (e.g. `.nta` means "no temporal locality at all cache levels"). In order to capture potential spatial locality exhibited by these memory accesses, the complete memory line is always fetched in the cache even if the hint specifies no temporal locality. The difference with LRU is that the line is indicated as next to be replaced. The effectiveness of this method is indicated by the cache miss reductions shown in table 5, where this method is applied to numerical programs in which the arrays are mostly traversed with stride 1, i.e. with very good spatial locality.

The basic assumption of reuse distance-based cache hint selection is that all

data between use and reuse is fetched into the cache (see lemma 1), and is replaced with LRU policy. However, after hints have been introduced, intervening data with a non-temporal hint is replaced earlier. Even though this effect is not taken into account by the reuse distance, selecting hints based on the reuse distance assures that cache hit rates are at least as good as under the LRU replacement policy[25]. A selection scheme which improves over reuse distance based selection would need to measure, not only the amount of data accessed between use and reuse, but also what data is accessed, and what cache hints are assigned to it. For the profile-based method, this would result in having to record a huge amount of profile information. Furthermore, even if it is recorded which references and accesses occur between any two accesses to the same data, another problem arises: The selection of a cache hint for reference a might influence the cache hint selection for reference b, which in turn might influence the cache hint selection of reference a, resulting in a hard optimization problem. In contrast to a method which would take these effects into account, the reuse distance based selection is simple and effective. Nonetheless, it would be interesting to look for practical methods which also take the interactions of cache hints into account.

Target hints induce a compiler-steered replacement policy. All previously proposed methods[25,40,44,45,50,54] to influence the replacement policy from the compiler are similar to static hints, i.e. there is a single replacement hint per reference. In contrast, the reuse distance equations allow to generate efficient code with replacement decisions optimized for each individual memory access, i.e. optimized hints for each access. As such, the presented dynamic hint generation allows memory replacement tuned to individual memory accesses, leading to less cache misses (see table 4).

8 Conclusion

EPIC architectures provide cache hints which allow the compiler to have more control and a better view on the data cache behavior. The source cache hints inform the compiler to hide long latency loads with parallel instructions, while the target cache hints enable an adapted replacement policy for data with low locality. In this work, a framework is proposed to generate both source and target cache hints based on the reuse distance metric. Since the reuse distance indicates cache behavior irrespective of the cache size and associativity, it can be used to make caching decisions for all levels of cache simultaneously. This paper describes how this property is exploited to select appropriate cache hints for multiple levels of cache.

Two methods are proposed to determine the reuse distances in the program: one based on profiling which statically assigns a cache hint to a memory in-

struction and one based on an analytical calculation that allows to generate efficient dynamic hints. The advantage of the profiling-based method is that it works for all programs. The analytical calculation of reuse distances is applicable to loop-oriented code and has the advantage that the reuse distance is calculated independent of program input, with the data sizes as parameters and for individual memory accesses. This allows to dynamically select cache hints per access. The experimental evaluation of the profile-based source and target cache hint selection on a number of pointer-intensive and numerical programs shows an average speed up of 9% with a maximum of 56%. The pointer-intensive programs gain most from the target cache hints, while the numerical programs profit most from the source hints.

Furthermore, it is shown for a number of loop-oriented programs that dynamic target cache hint selection can reduce cache misses better than static hints. By only considering the reuse distances of the dominant domains, the execution time overhead of computing dynamic hints at run-time is eliminated, and code size overhead is reduced from 2200% to 17%. Dynamic target hints are especially valuable in programs where the same instruction exhibits wildly varying locality. On average, the static hints reduce the number of cache misses by 5.14%, while the dynamic hints reduce the cache misses by 10.34%.

Acknowledgments

This research was funded by Ghent University (contract No. GOA-12051002) and a Ph.D grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

References

[1] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *ICS*, pages 141–152, 2000.

[2] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[3] K. Beyls. *Software Methods to Improve Data Locality and Cache Behavior*. PhD thesis, Ghent University, June 2004.

[4] K. Beyls and E. D'Hollander. Compiler generated multithreading to alleviate memory latency. *Journal of Universal Computer Science, special issue on Multithreaded Processors and Chip-Multiprocessors*, 6(10):968–993, oct 2000.

[5] K. Beyls and E. H. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of PDCS'01*, pages 617–662, Aug 2001.

[6] M. Brehob and R. J. Enbody. An analytic model of locality and caching. Technical Report MSU-CSE-99-31, Michigan State University, August 1999.

[7] J. F. Cantin and M. D. Hill. Cache performance for selected SPEC CPU2000 benchmarks. *Computer Architecture News (CAN)*, September 2001.

[8] G. C. Caşcaval. *Compile-time performance prediction of scientific programs.* PhD thesis, University of Illinois at Urbana-Champaign, 2000.

[9] K. K. Chan, C. C. Hay, J. R. Keller, G. P. Kurpanek, F. X. Schumacher, and J. Zheng. Design of the HP PA 7200 CPU. *Hewlett-Packard Journal: technical information from the laboratories of Hewlett-Packard Company*, 47(1):25–33, Feb. 1996.

[10] S. Chatterjee, V. Jain, A. R. Lebeck, S. Mundhra, and M. S. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *13th ACM International Conference on Supercomputing (ICS '99)*, June 1999.

[11] S. Chatterjee, E. Parker, P. Hanlon, and A.R.Lebeck. Exact analysis of the cache behavior of nested loops. In *PLDI*, pages 286–297, 2001.

[12] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 1–12, May 1999.

[13] P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *ACM Int. Conf. on Supercomputing*, pages 278–285. ACM, May 1996.

[14] P. Clauss and V. Loechner. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal Processing*, 19:179–194, 1998.

[15] E. D'Hollander, F. Zhang, and Q. Wang. The Fortran parallel transformer and its programming environment. *Journal of Information Science*, 106:293–317, 7 1998.

[16] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *PLDI'03*. ACM, 2003.

[17] C. Fang, S. Carr, S. Onder, and Z. Wang. Reuse-distance-based miss-rate prediction on a per instruction basis. In *Workshop on Memory System Performance (MSP)*, 2004.

[18] P. Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, pages 79–103, 1996.

[19] B. B. Fraguela, R. Doallo, J. Touriño, and E.L.Zapata. A compiler tool to predict memory hierarchy performance of scientific codes. *Parallel Computing*, 30:225–248, 2004.

[20] S. Ghosh. *Cache Miss Equations: Compiler Analysis Framework for Tuning Memory Behaviour.* PhD thesis, Princeton University, November 1999.

[21] P. Grun, N. Dutt, and A. Nicolau. MIST: An algorithm for memory miss traffic management. In *International Conference on Computer Aided Design*, pages 431–437, 2000.

[22] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, Dec. 1989.

[23] *IA-64 Application Developer's Architecture Guide*, May 1999.

[24] *Intel Itanium 2 Processor Reference Manual*, June 2002.

[25] P. Jain, S. Devadas, D. Engels, and L. Rudolph. Software-assisted replacement mechanisms for embedded systems. In *International Conference on Computer Aided Design*, pages 119–126, nov 2001.

[26] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *The 17th ISCA*, pages 364–373, May 1990.

[27] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and E. Ayguade. An integer linear programming approach for optimizing cache locality. In *Proceedings of the 1999 Conference on Supercomputing*, ACM SIGARCH, pages 500–509, N.Y., jun 1999. ACM Press.

[28] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A matrix-based approach to the global locality optimization problem. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 306–313, Paris, France, Oct. 12–18, 1998. IEEE Computer Society Press.

[29] V. Kathail, M. S. Schlansker, and B. R. Rau. HPL_PD architecture specification: Version 1.1. Technical Report HPL-93-80(R.1), Hewlett-Packard, February 2000.

[30] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *1993 Workshop on Languages and Compilers for Parallel Computing*, number 768, pages 301–320, Portland, Ore., 1993. Berlin: Springer Verlag.

[31] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *PLDI*, pages 346–357, 1997.

[32] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[33] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. *ACM SIGPLAN Notices*, 27(9):62–73, Sept. 1992.

[34] Open64 compiler. http://sourceforge.net/projects/open64.

[35] T. Ozawa, Y. Kimura, and S. Nishizaki. Cache miss heuristics and preloading techniques for general-purpose programs. In *MICRO'95*, pages 243–248, Ann Arbor, Michigan, Nov. 29–Dec. 1, 1995. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[36] W. Pugh. Counting solutions to Presburger formulas: How and why. *ACM SIGPLAN Notices*, 29(6):121–134, jun 1994.

[37] B. Rau, V. Kathail, and S. Aditya. Machine-description driven compilers for EPIC and VLIW processors. *Design Automation for Embedded Systems*, 4:71–118, 1999.

[38] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 38–49, June 1998.

[39] J. Sánchez and A. González. Fast, accurate and flexible data locality analysis. In *PACT '98*, pages 124–129, Paris, France, Oct. 12–18, 1998. IEEE Computer Society Press.

[40] J. Sanchez and A. Gonzalez. A locality sensitive multi-module cache with explicit management. In *Proceedings of the 1999 Conference on Supercomputing*, ACM SIGARCH, pages 51–59, N.Y., June 20–25 1999. ACM Press.

[41] M. S. Schlansker and B. R. R. Cover. EPIC: Explicitly parallel instruction computing. *IEEE Computer*, 33(2):37–45, Feb. 2000.

[42] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, New York, 1986.

[43] A. Seznec and F. Bodin. Skewed-associative caches. In *Proceedings of PARLE '93*, Lecture Notes in Computer Science, pages 305–316, Munich, Germany, June 14–17, 1993. Springer-Verlag.

[44] R. Sree, A. Settle, I. Bratt, and D. A. Connors. Compiler-directed resource management for active code regions. In *Proceedings of the 7th Workshop on Interaction between Compilers and Computer Architecture*, February 2003.

[45] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *MICRO'95*, pages 93–103, Ann Arbor, Michigan, Nov. 29–Dec. 1, 1995. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[46] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, June 2000.

[47] X. Vera and J. Xue. Let's study whole-program cache behavior analytically. In *High-Performance Computer Architecture (HPCA'02)*, pages 175–186, Feb 2002.

[48] S. Verdoolaege. The Barvinok library. http://freshmeat.net/projects/barvinok.

[49] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Analytical computation of Ehrhart polynomials: Enabling more compiler analyses and optimizations. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES*, September 2004.

[50] Z. Wang, K. McKinley, A. Rosenberg, and C. Weems. Using the compiler to improve cache replacement decisions. In *PACT'02*, September 2002.

[51] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, 1991.

[52] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[53] W. A. Wong and J.-L. Baer. Modified LRU policies for improving second-level cache behavior. In *HPCA-6*, pages 49–60, Jan. 8–12, 2000.

[54] H. Yang, R. Govindarajan, G. R. Gao, and Z. Hu. Compiler-assisted cache replacement: Problem formulation and performance evaluation. In *16th International Workshop on Languages and Compilers for Parallel Computing(LCPC'03)*, October 2003.

[55] F. Zhang and E. D'Hollander. Using hammock graphs to structure programs. *IEEE Transactions on Software Engineering*, 30(4):231–245, April 2004.