Compile-Time Cache Hint Generation for EPIC Architectures

Kristof Beyls
Ghent University,
Sint-Pietersnieuwstraat 41,
Gent, Belgium
kristof.beyls@elis.rug.ac.be

Erik H. D'Hollander
Ghent University,
Sint-Pietersnieuwstraat 41,
Gent, Belgium
erik.dhollander@elis.rug.ac.be

ABSTRACT

One of the new possibilities offered by EPIC architectures is to allow the compiler to direct the cache placement and replacement policy through cache hints. In this work, a method to generate these cache hints at compile time is presented. The generation of appropriate cache hints is based on the locality of the instructions they apply to, which is quantified by the reuse distance metric. Next to the static selection of the most appropriate cache hints, a dynamic selection of cache hints by predicates is proposed. The generation of static hints is based on a simple profiling scheme, while the dynamic selection is based on an analytical model of the programs cache behavior.

The implementation of the static approach in the Open64-compiler shows a speedup of 7% on average on a set of pointer-intensive and regular loop-based programs. The dynamic approach shows that for loop kernels, up to 35% reduction in cache misses can be attained. Furthermore, the dynamic selection allows to remove almost twice as many cache misses as statically selected cache hints.

1. INTRODUCTION

It is well-known that data cache misses form a severe bottleneck in executing programs. Typically, Itanium programs spent about 50% of their execution time on resolving cache misses[3]. In the future, the speed gap between processors and memory will continue to grow, making cache misses an even larger bottleneck. Therefore, improving the cache behavior is essential to obtain good execution speeds. In the past, many compiler optimizations have been proposed to enhance the data cache behavior. However, in traditional processors, the hardware decides when and where data is placed and replaced in the cache hierarchy, and the compiler can only influence the cache behavior indirectly. With the advent of cache hints in EPIC architectures, for the first time, the compiler has the means to steer the cache behavior directly. The challenge is to decide in the compiler which cache hints to generate.

Cache hints are annotated to memory instructions, such as loads, stores and prefetches. They have two purposes: inform the compiler about the true latency of load and prefetch operations and inform the processor at which cache level data should be placed. Cache hints are further discussed in section 2.

In order to generate appropriate cache hints, it is required that the compiler has an idea about the locality of the instructions. In this paper, the locality is quantified by the reuse distance metric. It allows to accurately determine which memory instructions result in cache misses and whether the program will profit from retaining data at a given cache level. Section 3 further discusses the reuse distance and its properties as a locality metric.

In section 4, the selection of cache hints based on the reuse distance is presented. The cache hints are an integral part of the memory instruction. Therefore, all executions of the same instruction share the same cache hint. However, the different executions of the instruction can exhibit different amounts of locality, requiring different cache hints. The presented approach selects a single cache hint which is appropriate for all executions of the instruction.

The restriction that all executions of the same memory instructions must share the same cache hint is worked around in section 5. There, a program analysis and transformation is presented to make sure that all executions of the same instruction require the same cache hint. The program analysis calculates the exact reuse distances of the different executions of a memory instruction. After the analysis, the instruction is duplicated with different cache hints and predicates are used to dynamically select the version with the most appropriate hint.

In section 6, the implementation and the experimental evaluation of the cache hint selection scheme is presented. In section 7, related work is discussed, and in section 8, the conclusion follows.

2. SOFTWARE-CONTROLLED CACHING IN EPIC

One of the basic principles of the EPIC-philosophy is to let the compiler decide when to issue operations and which resources to use. This contrasts with the superscalar paradigm, where the processor is responsible for deciding e.g. which instructions to execute in parallel, how to predict branches and where to place data in the cache hierarchy. In an EPIC architecture, the responsibility for making these choices is mostly shifted to the compiler. In order to communicate the compiler decisions to the processor hardware, they must be representable in the instruction set architecture.

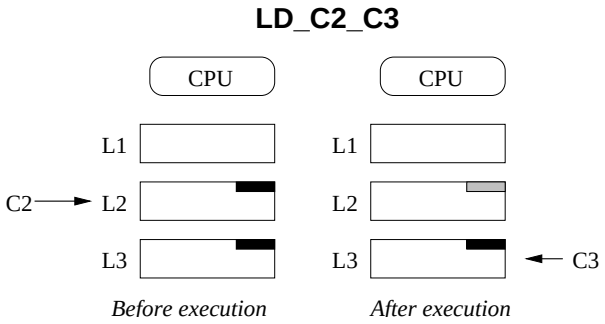


Figure 1: Example of the effect of the cache hints in the load instruction `LD_C2_C3`. The source cache specifier `C2` in the instruction suggests that the data resides in the L2-cache. The target cache specifier `C3` indicates that the data should be stored no closer than the L3-cache. As a consequence,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

The existing EPIC architectures (HPL-PD[14] and IA-64[11]) communicate the compiler decisions about the cache hierarchy management to the processor through cache hints. The semantics of cache hints on both architectures are similar. First, the cache hints in HPL-PD are presented, after which their counterparts in the IA-64 architecture are discussed.

In the HPL-PD architecture, cache hints are attachments to 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 fig. 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 determine the true memory latency of instructions. It uses this information to schedule the instructions explicitly in parallel. 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.

Similar cache hints are available in the IA-64 architecture. Since source cache hints are used inside the compiler, there's no need to communicate them to the processor. Therefore, IA-64 only defines the target cache hints `.t1`, `.nt1`, `.nt2` and `.nta`. `.t1` means that the memory instruction has temporal locality in all the cache levels. `.nt1` only has temporal locality in the L2 cache and below, but there might still be spatial locality in the L1 cache. Similarly, `.nt2` respectively `.nta` indicates that there's only temporal locality in L3 respectively no temporal locality at all.

3. REUSE DISTANCE

The reuse distance is the locality metric used in this work. It 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***[8].*

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 $\langle a_1, a_2 \rangle$ 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 2 shows three reuse pairs in a short memory access stream.

Reuse Distance Theorem. *In a fully associative LRU cache with n lines, an access with backward reuse distance $d < n$ will hit. An access with backward reuse distance $d \geq n$ will miss.*

In a fully associative LRU cache with n lines, the memory line accessed by a reference with forward reuse distance $d < n$ will stay in the cache until the next access of that memory line. A reference with forward reuse distance $d \geq n$ will be removed from the cache before the next access.

Proof. In a fully associative LRU cache with n cache lines, the n most recently referenced memory lines are retained. When a reference has a backward reuse distance d, exactly d different memory lines were referenced previously. If $d \geq n$, 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, the forward reuse distance d of a_1 equals the backward reuse distance of a_2, i.e. d. Therefore, the data will be found in the cache at access a_2, if and only if $d < n$. □

The theorem above indicates that the reuse distance can be used to precisely indicate the cache behavior of fully associative caches. However, previous research[10, 2, 4] indicates that also for lower-associative, and even for direct mapped caches, the reuse distance can be used to obtain a good estimation of the cache behavior. [10, 2, 4] independently measure the error made by the theorem above, when predicting cache behavior for low-associative caches. Both

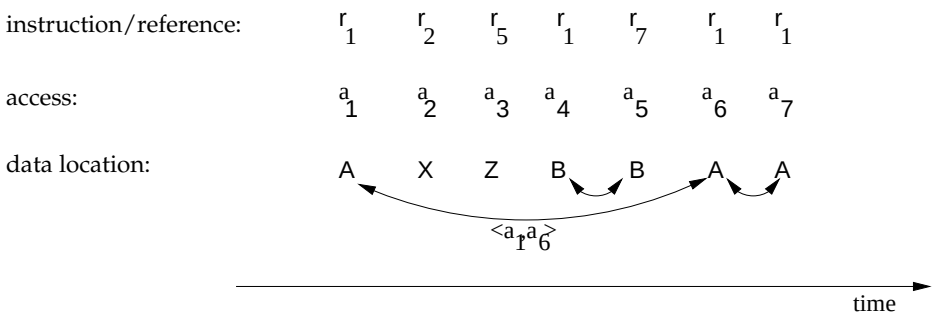


Figure 2: The top row indicates 7 memory reference instructions, generating a numbered memory access stream in the second row. The bottom row shows the corresponding memory locations A, B, X or Z. The accesses to X and Z are not part of a reuse pair, since they are accessed only once in the stream. $\mathsf{ADS}\langle a_1, a_6\rangle = \{B, X, Z\}$**, and** $\mathsf{RD}(\langle a_1, a_6\rangle) = |\mathsf{ADS}\langle a_1, a_6\rangle| = 3$**.** $\mathsf{RD}(\langle a_6, a_7\rangle) = 0$**.** $\mathsf{FRD}(a_1) = 3$**,** $\mathsf{BRD}(a_1) = \infty$**.** $\mathsf{FRD}(a_6) = 0$**,** $\mathsf{BRD}(a_6) = 3$**.**

statistical analysis and measurements based on a wide variety of program traces indicate that the relative error is low, typically less than 5%[10]. This is further confirmed by the cache behavior measurements performed on the SPEC2000 benchmark in [5]. The measurements show that for typical caches (associativity 2 or greater and cache size larger than 8KB), the capacity misses are responsible for at least 85% of all misses.

4. CACHE HINT SELECTION

In order to reduce the number of cache misses, the reuse distance of the memory accesses can be reduced so that they are smaller than the cache size, as is done by program transformations such as loop tiling[1]. However, due to dependences in the program, it is not always possible to reduce the reuse distance to be smaller than the cache size. In this paper, 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.

The cache hint selection is based on the reuse distance, which predicts fully associative cache behavior perfectly. For lower-associative caches, an exact simulation of the cache could be used to know whether data is found or retained at a given cache level. However this has two disadvantages when compared to reuse distance-based selection:

1. The cache behavior which is measured for a set-associative cache is dependent on the exact layout of data in the address space. The data layout and the data alignment can change between different executions of the program. Therefore, the optimal cache hints for the measured execution are not necessarily the optimal cache hints for other executions of the program. Since the reuse distance is independent of data layout, the metric doesn't change when the data layout changes.

2. Since cache hints indicate how data should be placed and replaced in multiple cache levels, different measurements of cache behavior for all cache levels would be needed. Furthermore, they would need to be combined to generate a single cache hint. Since the reuse distance is irrespective of cache size, a single measurement allows to easily select cache hints which take into account the different cache levels.

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 can exhibit different locality, requiring different cache hints. For example, in fig. 2, 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 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. It is clear that the accesses originating from the same instruction require different cache hints, even though only a single hint can be specified per instruction. As a first step, we consider which cache hint is most appropriate for every single *memory access*.

Selecting Cache Hints per Access

The selection of source and target cache hints from the backward and forward reuse distance of the access is performed as follows:

- The backward reuse distance indicates the 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 cache size needed for the access to be a hit for lower-associative caches with high probability. Therefore, as *source cache hint* we select *the smallest cache level that is larger than the backward reuse distance.* If a smaller (resp. larger) cache level would be selected, the compiler would assume a smaller (resp. larger) latency than the true latency. If the assumed latency is too small, the compiler doesn't try to hide the full latency of the load. If the assumed latency is too large, the compiler will generate a suboptimal schedule, because the target register of the instructions will be kept live longer than necessary.

- 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

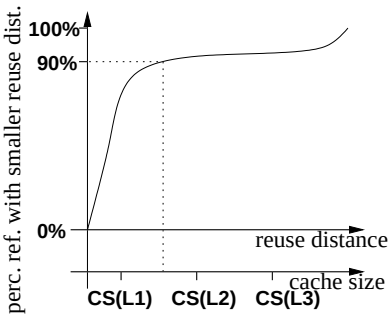


Figure 3: A cumulative reuse distance distribution for an instruction is shown and how a threshold value of 90% maps it to cache hint C2. CS(Lx) = cache size for cache level x.

to keep the data for lower-associative caches with high probability. Therefore, as *target cache hint* we select *the smallest cache level that is larger than the forward reuse distance*. In [12], it is proven that this cache hint choice per access is guaranteed to perform equal or better than the LRU replacement policy for a fully associative cache. If a smaller cache level would be selected, the data would be loaded into a small cache level where it is not retained until its reuse. Furthermore, by loading it into the small cache, that cache is polluted. If a larger cache level would be selected for the target hint, the data would not be loaded in all cache levels where it can exploit its locality.

Selecting a Single Appropriate Hint per Instruction

For every memory access, the most appropriate cache hint can be determined. However, a single memory instruction can generate multiple memory accesses during program execution. As described above, those accesses can require different cache hints. It is not possible to specify different cache hints for them, since the cache hint is specified on the instruction. As a consequence, all accesses originating from the same instruction share the same cache hint. Because of this, it is not possible to assign the most appropriate cache hint to all accesses. The following approach is used to obtain a single cache hint per instruction which is applicable for most accesses generated by the same instruction:

1. First, for every instruction, the cumulative distribution of the reuse distances of the memory accesses it generates are collected. An example of such a distribution is shown in fig. 3.
2. Based on the distribution, a source cache hint can be selected so that for at least x% of the accesses, the data will be found in that cache level. In fig. 3, it is shown 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 in that cache level.

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 are presented in section 6.

5. DYNAMIC CACHE HINT SELECTION

When implementing the cache hint selection scheme described above, two major issues pop up. The first problem is that a cache hint is tied to an instruction, and not to a single memory access. The second problem is that the locality of the memory accesses generated by an instruction can depend on the input of the program. For example, if the program performs a matrix computation, the size of the matrices can determine the cache level where data will be found. Therefore, the optimal cache hints are also dependent on information that is in general only known at runtime. In order to mitigate the above problems, cache hints should be selected at run-time, based on the actual reuse distance of the current access. In order to be able to select the best cache hint for every single access, the reuse distance is determined analytically. Below, the analytical calculation of the reuse distance for a class of loop-oriented programs is described.

5.1 Reuse Distance Calculation

5.1.1 Program model

The analytical calculation of the reuse distance applies to programs or program parts which satisfy the following conditions: The program consists of assignment, if and loop statements. The iteration spaces of the statements must be describable by a Presburger formula[17], which represent a set of integer points that are described by a combination of linear constraints. The variables in the program are either scalars or arrays. Scalars are treated as one-dimensional arrays with size 1. Loop induction variables are assumed to reside in registers and not to generate any memory accesses. The index expressions of the array variables must be affine functions of the loop induction variables and program parameters. In order to be consistent with the default terminology in analytical calculation of cache behavior, reference is used as a synonym for instruction in this section[8]. In order to calculate the reuse pairs, their accessed data sets ADS and the corresponding reuse distance, the set of references and their iteration spaces, the set of array variables, and the order of two accesses is needed. They are formally denoted as follows:

Definition 4. *The* **set of all the references** *in a program is denoted by* $\mathcal{R}$. *The* **set of variables** *in a program is denoted by* $\mathcal{V}$. *The* **iteration space** *of the statement in which a reference r occurs is denoted by* $\mathsf{IS}(r)$. *The* **memory location** *which is accessed by r at iteration point i is denoted by* $r@i$. *The fact that iteration point i of reference r is* **executed before** *iteration point j of reference s is expressed as* $i_r \lessdot j_s$.

Example 1. *In fig. 4, a small loop is shown which can be handled by our program model. To clarify the notations introduced in definition 4, some examples applied to the loop in fig. 4 are given here:*

- *The variable set* $\mathcal{V} = \{A\}$.

```
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
```

Figure 4: Example loop to demonstrate the program model.

- *The reference set* $\mathcal{R} = \{\mathtt{A(2i,1)}, \mathtt{A(i,j-i)}, \mathtt{A(3+j,i)}\}$.
- *The iteration space* $\mathsf{IS}(\mathtt{A(i,j-i)}) = \{(i,j) : 1 \leq i \leq N \wedge i+2 \leq j \leq N-i \wedge \neg(i=j)\}$.
- *The mapping of iteration to data space* $\mathtt{A(3+j,i)}@(i=3, j=6) = A(9,3)$.
- *The lexicographical order constraint* $(i)_{\mathtt{A(2i,1)}} \lessdot (i',j')_{\mathtt{A(i,j-i)}} \equiv i \leq i'$.

5.1.2 Formal Reuse Equations

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

1. The reuse pairs in the memory access stream are formulated. For every couple of *references* (r,s), a single formula $\mathsf{reuse}\,(r \rightarrow s)$ is generated. The formula 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 single formula is formed which symbolically 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[7].

The three steps are explained in further detail below. Examples of the formulas generated by the three steps can be found in section 5.1.3.

1. Reuse pair

Every memory access is uniquely defined by the reference r which generates the access, and the iteration point I_r at which the access occurs.

All reuse pairs $\langle x,y\rangle$ for which the first access x originates from reference r and the second access y 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:

$$\forall r,s \in \mathcal{R} : \mathsf{reuse}\,(r \rightarrow s) = \tag{1a}$$
$$\big\{(I_r,J_s) : I_r \in \mathsf{IS}(r) \wedge J_s \in \mathsf{IS}(s) \wedge \tag{1b}$$
$$I_r \lessdot J_s \wedge \tag{1c}$$
$$r@I_r = s@J_s \wedge \tag{1d}$$
$$\forall t \in \mathcal{R} : \neg\,(\exists K_t \in \mathsf{IS}(t) : I_r \lessdot K_t \lessdot J_s \wedge t@K_t = r@I_r)\,\big\} \tag{1e}$$

The above formula gives the constraints which must be satisfied before a reuse occurs between $r@I_r$ and $s@J_s$. Equation (1b) expresses that I_r and J_s are part of the iteration space of respectively r and s. (1c) demands that I_r must be executed before J_s; (1d) encodes that the same memory location must be accessed; and (1e) ensures that no intervening memory access touches the same memory location. Furthermore, the following formulas define the iteration points at which *forward* respectively *backward* reuse occurs:

$$\begin{aligned}&\mathsf{reuse_F}\,(r) = \\ &\{I_r : \exists s \in \mathcal{R}, J_s \in \mathsf{IS}(s) : (I_r,J_s) \in \mathsf{reuse}\,(r \rightarrow s)\} \\ &\mathsf{reuse_B}\,(s) = \\ &\{J_s : \exists r \in \mathcal{R}, I_r \in \mathsf{IS}(r) : (I_r,J_s) \in \mathsf{reuse}\,(r \rightarrow s)\}\end{aligned} \tag{2}$$

2. Accessed data set of a reuse pair

The second step is to calculate the accessed data set (ADS) of a reuse pair. First, the function map_r is defined, which maps an iteration space to the elements of array V which are accessed by r in that iteration space, see eq. (3). Furthermore, $\mathsf{iters}_t(I_r,J_s)$ is the set of iterations of reference t which are executed between iteration I_r and iteration J_s:

$$\mathsf{map}_r = \{I \rightarrow r@I : I \in \mathsf{IS}(r)\} \tag{3}$$

$$\mathsf{iters}_t(I_r,J_s) = \{K_t \in \mathsf{IS}(t) : I_r \lessdot K_t \lessdot J_s\} \tag{4}$$

With the help of equations (3) and (4), the function $\mathsf{ADS}_V\,(\mathsf{reuse}\,(r \rightarrow s))$ is defined, which indicates the elements of array V that are in the ADS of the reuse pairs in $\mathsf{reuse}\,(r \rightarrow s)$:

$$\mathsf{ADS}_V\,(\mathsf{reuse}\,(r \rightarrow s)) = \bigcup_{t \in \mathcal{R}} \mathsf{map}_t(\mathsf{iters}_t(\mathsf{reuse}\,(r \rightarrow s))), \tag{5}$$

where $\mathsf{map}_t(\mathsf{iters}_t(\mathsf{reuse}\,(r \rightarrow s)))$ means applying function map_t to the set of iterations $\mathsf{iters}_t(\mathsf{reuse}\,(r \rightarrow s))$. Equation (5) 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.

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:

$$\mathsf{RD}(\mathsf{reuse}\,(r \rightarrow s)) = \sum_{V \in \mathcal{V}} \big|\mathsf{ADS}_V\,(\mathsf{reuse}\,(r \rightarrow s))\big| \tag{6}$$

$\mathsf{ADS}_V\,(\mathsf{reuse}\,(r \rightarrow s))$ is a Presburger formula in general, which represents a set of integer points. Counting the number of elements in such a set (such as needed for $\big|\mathsf{ADS}_V\,(\mathsf{reuse}\,(r \rightarrow s))\big|$), can be performed by the methods discussed in [7] or [18]. Besides calculating the reuse distance of a reuse pair, it is also possible to compute the forward and backward reuse distances of a memory reference r. These are denoted by

$\mathsf{FRD}(r)$ and $\mathsf{BRD}(r)$:

$$\mathsf{FRD}(r) = \sum_{s \in \mathcal{R}, V \in \mathcal{V}} \left|\mathsf{ADS}_V\left(\mathsf{reuse}\left(r \rightarrow s\right)\right)\right| \quad (7)$$

$$\mathsf{BRD}(s) = \sum_{r \in \mathcal{R}, V \in \mathcal{V}} \left|\mathsf{ADS}_V\left(\mathsf{reuse}\left(r \rightarrow s\right)\right)\right| \quad (8)$$

Furthermore, the reuse distance theorem can be used to calculate at which iteration points the data will not be 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 first fetched, and results in a cold miss. The iteration points at which a cold miss occurs for reference r are denoted by $\mathsf{COLDM}(r)$:

$$\mathsf{COLDM}(r) = \{I : I \in \mathsf{IS}(r) \wedge I \notin \mathsf{reuse}_{\mathsf{B}}\left(r\right)\} \quad (9)$$

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

$$\mathsf{CAPM}(r) = \{I : \mathsf{BRD}(r) \geq CS \wedge I \in \mathsf{reuse}_{\mathsf{B}}\left(r\right)\}, \quad (10)$$

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, and is denoted by $\mathsf{NOKEEP}(r)$:

$$\mathsf{NOKEEP}(r) = \{I : \mathsf{FRD}(r) \geq CS \wedge I \in \mathsf{reuse}_{\mathsf{F}}\left(r\right)\}, \quad (11)$$

5.1.3 Example

As an example, the actual formulas generated during reuse distance calculation are presented for two different programs: the matrix multiplication and Cholesky factorization. The matrix multiplication is a simple loop kernel which is very well known in cache optimizations. The Cholesky factorization is a more complicated loop kernel, with multiple loop nests which are not perfectly nested.

Matrix Multiplication

The matrix multiplication code is shown in fig. 5(a). As an example, the backward reuse distance of reference `A(I,K)` at all its iteration points is calculated step by step here. The first step is to find the references that generate reuse pairs with `A(I,K)`. There are no other references in the program that can access the same data as `A(I,K)`, so only $\mathsf{reuse}\left(\mathtt{A(I,K)} \rightarrow \mathtt{A(I,K)}\right)$ needs to be calculated. Using equation (1a,...,1e), this leads to the following expression after simplification, where $I = (i, j, k)$ and $I' = (i', j', k')$:

$$\begin{aligned}&\mathsf{reuse}\left(\mathtt{A(I,K)} \rightarrow \mathtt{A(I,K)}\right) = \\ &\{(I, I') : (i, j+1, k) = (i', j', k') \text{ and } I \in \mathsf{IS}(A(I,K)) \text{ and} \\ &\quad I' \in \mathsf{IS}(A(I,K))\}\end{aligned}$$

In order to find the data that is accessed between reuses, the accesses that are executed between reuses must be calculated. For example, $\mathsf{iters}_{\mathtt{B(K,J)}}(\mathsf{reuse}\left(\mathtt{A(I,K)} \rightarrow \mathtt{A(I,K)}\right))$ is calculated as follows:

$$\begin{aligned}&\mathsf{iters}_{\mathtt{B(K,J)}}(\mathsf{reuse}\left(\mathtt{A(I,K)} \rightarrow \mathtt{A(I,K)}\right)) = \{(I'') : I \lessdot I'' \lessdot I' \\ &\quad \text{and } (i, j+1, k) = (i', j', k') \\ &\quad \text{and } I \in \mathsf{IS}(A(I,K)) \text{ and } I' \in \mathsf{IS}(A(I,K))\}\end{aligned}$$

```
DO I=1,N
 DO J=1,N
  DO K=1,N
   C(I,J) = C(I,J) + A(I,K)*B(K,J)
  ENDDO
 ENDDO
ENDDO
```

(a) The matrix multiplication code

```
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
```

(b) Cholesky factorization

Figure 5: source codes of matrix multiply and Cholesky factorization

which expands to

$$\begin{aligned}\{(i'', j'', k'') : &(1 \leq i'' = i' \leq N \wedge 1 \leq j' - 1 = j'' \leq N \\ &\wedge 1 \leq k' \leq k'' \leq N) \\ \vee &(1 \leq i'' = i' \leq N \wedge 1 \leq j' = j'' \leq N \\ &\wedge 1 \leq k'' < k' \leq N)\}\end{aligned}$$

In the equation above, the expression $k'' < k'$ shows that it is assumed that reference `A(I,K)` occurs before `B(K,J)` in the same iteration. Now, the data accessed between reuses can be calculated, using equation (5):

$$\begin{aligned}&\mathsf{ADS}_{\mathtt{B}}\left(\mathsf{reuse}\left(\mathtt{A(I,K)} \rightarrow \mathtt{A(I,K)}\right)\right) = \\ &\mathsf{map}_{\mathtt{B(K,J)}}\left(\mathsf{iters}_{\mathtt{B(K,J)}}(\mathsf{reuse}\left(\mathtt{A(I,K)} \rightarrow \mathtt{A(I,K)}\right))\right) = \\ &= \{(x, y) : (k' \leq x \leq N \wedge 1 \leq y = j' - 1 \leq N) \vee \\ &\quad (1 \leq x < k' \wedge 1 \leq y = j' \leq N)\}\end{aligned}$$

The equation above shows, for every iteration point (i', j', k') of `A(i,j)` where reuse occurs, the part of array `B` that is accessed between reuses. Similarly, the data of array `A` and array `C` accessed between reuses can be calculated as follows:

$$\begin{aligned}&\mathsf{ADS}_{\mathtt{A}}\left(\mathsf{reuse}\left(\mathtt{A(I,K)} \rightarrow \mathtt{A(I,K)}\right)\right) = \\ &\quad \{(x, y) : 1 \leq y \leq N \wedge 1 \leq x = i' \leq N \wedge y \neq k'\} \\ &\mathsf{ADS}_{\mathtt{C}}\left(\mathsf{reuse}\left(\mathtt{A(I,K)} \rightarrow \mathtt{A(I,K)}\right)\right) = \\ &\quad \{(x, y) : 1 \leq x = i' \leq N \wedge 1 \leq j' - 1 = y < N\} \\ &\qquad \cup \{(x, y) : 1 \leq x = i' \leq N \wedge 2 \leq y = j' \leq N\}\end{aligned}$$

To calculate the backward reuse distance for all the iteration points of `A(I,K)` with reuse, the number of array elements

accessed is counted, as described in equation (8):

$$\begin{aligned}\mathsf{BRD}(\mathtt{A(I,K)}) &= |\mathsf{ADS}_{\mathtt{A}}(\mathsf{reuse}(\mathtt{A(I,K)} \rightarrow \mathtt{A(I,K)}))| \\ &+ |\mathsf{ADS}_{\mathtt{B}}(\mathsf{reuse}(\mathtt{A(I,K)} \rightarrow \mathtt{A(I,K)}))| \\ &+ |\mathsf{ADS}_{\mathtt{C}}(\mathsf{reuse}(\mathtt{A(I,K)} \rightarrow \mathtt{A(I,K)}))| \\ &= N - 1 \ + \ N \ + \ 2 = 2N + 1\end{aligned}$$

The backward reuse distance of `A(I,K)` is $2N + 1$ at every iteration point with reuse. Therefore, using equations (10) and (9), the capacity and cold misses are calculated to be:

$$\begin{aligned}&\mathsf{CAPM}(\mathtt{A(I,K)}) = \\ &\{(i,j,k) : 2N + 1 \geq CS \ \wedge \ (i,j,k) \in \mathsf{reuse}_{\mathtt{B}}(\mathtt{A(I,K)})\} \\ &\mathsf{COLDM}(\mathtt{A(I,K)}) = \{(i,j,k) \in \mathsf{IS}(\mathtt{A(I,K)}) : j = 1\}\end{aligned}$$

Cholesky Factorization

The Cholesky factorization code is shown in fig. 5(b). It consists of non-perfectly nested loops. Due to space limitations, we show the calculated backward reuse distances for only one of the references. The different reuse distance domains for the iteration space of reference `A(m,j)` is shown on the left hand side of figure 6. In the middle of fig. 6, the actual backward reuse distance is shown for the different iterations. In comparison to the cumulative reuse distance distribution on the right hand side, the analytical calculation produces more exact information: for every single access, the exact reuse distance is known.

5.2 Dynamic Cache Hint Selection by Predicates

The analytical calculation allows to select the appropriate cache hints for every access at run-time. In contrast to a profile-based method, which records the reuse distance distribution, the analytical method generates a polynomial representing the reuse distance at a given iteration point. As an example, the result of a profile-driven measurement of the reuse distance is shown on the right hand side in fig. 6. For the same instruction, the analytical method generates the data represented in the table in the same figure. From the distribution, it is not possible to find out which executions of the instruction have which reuse distance. In contrast, the calculated polynomial allows to find out the reuse distance for every execution of the instruction. This extra information can be used to select the appropriate cache hint at run-time.

In order to select the most appropriate cache hint, the load instruction is duplicated with different cache hints. The instruction with the appropriate hint can then be selected using predicates. The following code is an example of this. 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 following 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 = RD<CS1
        // p8 = RD>=CS2; p7 = CS1<=RD<CS2
(p6) ld.t1      r5 = ...          ;;
(p7) ld.nt1     r5 = ...
(p8) ld.nta     r5 = ...
```

6. RESULTS

6.1 Compiler Implementation

The static cache hint selection scheme presented in section 4 has been implemented in the Open64 compiler, 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 is saved to disk. During the feedback compilation, this profile data is read in by the compiler. The source and target cache hints are annotated to the memory instruction, based on the profile data. The instruction scheduler was adapted so that it assumes the latency indicated by the source cache hint. After instruction scheduling, the compiler produces the IA-64 assembly code with target cache hints. 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 feedback information through some optimizations phases at level -O3.

6.2 Predicated Cache Hint Selection

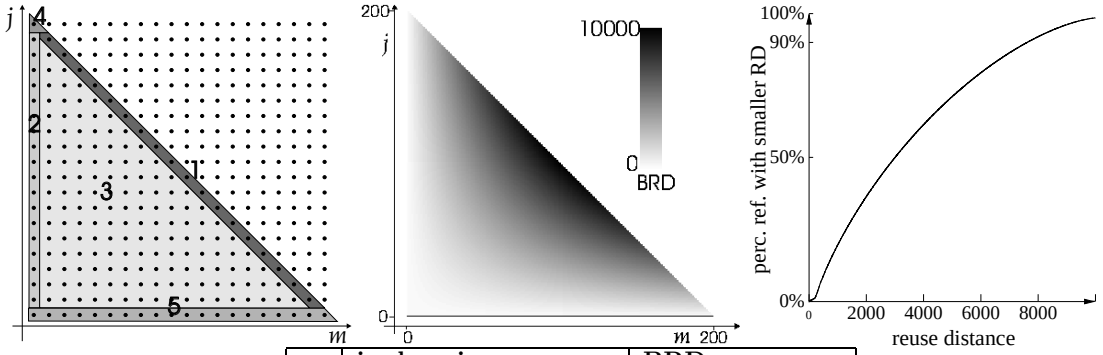
The calculation of reuse distances, as described in section 5.1, has been implemented in the FPT[26] compiler. The Omega library[18] is used to simplify the formulas and Polylib[7] is used to count the number of integer points in the sets described by the formulas.

In order to verify the exactness of the proposed equations for calculating reuse distances and cache behavior, they have been calculated automatically for a number of loop-oriented programs, such as the matrix multiplication, Gauss-Jordan elimination and Cholesky factorization. Furthermore they were applied to a number of artificial loop nests which were constructed specially to lead to far more irregular reuse distances. The results of the analysis were compared to cache simulation. The analytical results and the simulation results were identical in all cases.

6.3 Experiments

The static selection of cache hints was evaluated on a HP rx4610 multiprocessor, equipped with 733MHz Itanium processors. The data cache hierarchy consists of a 16KB L1, 96KB L2 and a 2MB L3 cache. The hardware performance counters of the processor were used to obtain detailed micro-architectural information, such as processor stall time because of memory latency and cache miss rates.

The programs were selected from the Olden and the Spec95fp benchmarks. The Olden benchmark contains programs which uses dynamic data structures, such as linked lists, trees and quadtrees. The Spec95fp programs are numerical programs with mostly regular array accesses. For the Spec95fp, the profiling was done using the train input sets, while the speedup measurements were done with the large input sets. For Olden, no separate input sets are available, and the



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

Figure 6: The iteration space of reference `=A(m,j)` (see fig. 5(b)) is shown on the 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 middle, the backward reuse distance of the points in the iteration space 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. On the right hand side, the cumulative reuse distance distribution is shown for this reference.

training input was identical to the input for measuring the speedup. The results of the measurements can be found in table 1.

The table shows that the programs run 7% faster on average, with a maximum execution time reduction of 36%. In the worst case, a slight performance degradation of 2% is observed. On average, the Olden benchmarks do not profit from the source cache specifiers. To take advantage of the source cache specifiers, the instruction scheduler must be able to 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 10% speedup is found because of the source cache hints. Here, the loop parallelism allows the compiler to find parallel instructions, mainly because it allows it to software pipeline the loops with long latency loads. In this way, the latency is overlapped with parallel instructions from different loop iterations. Some of the floating point programs didn't speedup a lot when employing source cache specifiers. The scheduler couldn't generate better code since the long latency of the loads demanded too many software pipeline stages to overlap it. Because of the large number of pipeline stages, not enough registers were available to actually create the software pipelining code.

The table also shows that the target cache specifiers improve both kinds of programs by the same percentage.

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 quantified. IA-64-like cache hints were generated from the calculated FRD, assuming a cache line size of 1 element, so that the forward temporal locality is measured. A single cache level was simulated, which reacts similar 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.

The data cache miss rates for a number of loop-oriented programs were measured. The loop kernels and the relative miss rates for the programs compiled without cache hints, with static cache hints (per instruction) and with dynamic cache hints (per access) are shown in table 6.3.

The table shows that on average the number of misses is reduced by 6.7% by static cache hint selection and by 11.7% by dynamic cache hint selection. The program which profits most from switching from static to dynamic cache hints is Cholesky. The reason for this 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 fig. 6, which shows the different reuse distances that are generated by a single instruction. Since the same instruction requires different cache hints, static hints cannot improve the cache behavior. On the other hand, dynamic hints allow to provide every access with the most appropriate cache hint, which results in a slight cache miss reduction.

7. RELATED WORK

The speed gap between processor and memory has been doubling every two years since 1980. This observation has attracted a lot of researchers to come up with improved caching schemes. Most proposed schemes can be categorized into either hardware modifications, loop transforma-

	program	mem. stall	mem. stall reduction	source CH speedup	target CH speedup	missrate reduction L1	missrate reduction L2	missrate reduction L3	overall speedup
Olden	bh	26%	0%	0%	-1%	1%	-20%	-3%	**-1%**
	bisort	32%	0%	0%	0%	0%	6%	-5%	**0%**
	em3d	77%	25%	6%	20%	-28%	-3%	35%	**23%**
	health	80%	19%	2%	16%	0%	-1%	15%	**20%**
	mst	72%	1%	0%	0%	-10%	1%	2%	**1%**
	perimeter	53%	-1%	-1%	-1%	-11%	-56%	-6%	**-2%**
	power	15%	0%	0%	0%	-14%	2%	0%	**0%**
	treeadd	48%	0%	-2%	-1%	-2%	26%	17%	**0%**
	tsp	20%	0%	0%	0%	2%	7%	7%	**0%**
	Olden avg.	47%	5%	0%	4%	-6%	-6%	7%	**5%**
Spec95fp	swim	78%	0%	0%	1%	32%	0%	0%	**0%**
	tomcatv	69%	33%	7%	4%	-11%	-43%	6%	**9%**
	applu	49%	10%	4%	1%	-9%	-1%	-1%	**4%**
	wave5	43%	-9%	4%	15%	-26%	-7%	-5%	**5%**
	mgrid	45%	13%	36%	0%	13%	-24%	25%	**36%**
	Spec95fp avg.	57%	9%	10%	4%	0%	-15%	5%	**10%**
	overall avg.	51%	7%	4%	4%	-5%	-8%	6%	**7%**

Table 1: Table with results for programs from the Olden and the SPEC95FP benchmarks: mem. stall=percentage of time the processor stalls waiting for the memory; mem. stall reduction=the percentage of memory stall time reduction after optimization; source CH speedup=the speedup if only source cache specifiers are used; target CH speedup=speedup if only target cache specifiers are used; missrate reduction=reduction in miss rate for the three cache levels; overall speedup=speedup resulting from reuse distance-based cache hint selection.

	miss rate reduction	
program	**static hints**	**dynamic hints**
vpenta(miss-rate=29.4%)	6%	6%
mxm (miss-rate=1.3%)	0%	0%
liv18 (miss-rate=15.62%)	0%	0%
cholesky (miss-rate=5.75%)	-22%	1%
jacobi (miss-rate=21.02%)	32%	32%
gauss-jordan (miss-rate=11.9%)	25%	34%
tomcatv (miss-rate=9.67%)	6%	9%
average	6.7%	11.7%

Table 2: The cache miss rates for a 4-way set associative 16KB cache with 32 bytes per line. The first column indicates the program under consideration. Next to the program name, between brackets, the cache miss rate for the program without cache hints is indicated. The second column indicates the relative number of cache misses for the program with cache hints per instruction, compared to the program without hints. The third column shows the relative number of cache misses with cache hints per access.

tions, data layout optimizations or hardware or software prefetching schemes[22, 21, 15, 13]. Most of the hardware modifications, loop transformations and data layout transformations aim at improving the cache locality of the program. The prefetching schemes aim at hiding the latency of cache misses with parallel instructions. The proposed cache hint selection scheme aims to do both: 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.

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 focusses 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 [9] and [16]. In [9] 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 [16], 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 very 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 target cache hints is found in [12], [20],[25] and [24].

In [12], keep and kill instructions are proposed. The keep instruction locks data into the cache, while the kill instruction indicates it as the first candidate to be replaced. Jain et al. also proof under which conditions the keep and kill instructions improve the cache hit rate. In [24], it is proposed 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 [20], 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 present day architectures. The 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.

The analytical approach to calculate 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[8, 19, 23], which do not capture all reuses in loops. Therefore, these methods result in an inexact estimate of the cache behavior. Recently, Chatterjee et al.[6] 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). In contrast, the calculation of reuse distances allows to calculate the cache behavior of fully associative caches. Furthermore the proposed cache equations can be extended to also handle arbitrary caches.

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 from 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. In this paper, this property is exploited to select appropriate cache hints for multiple levels of cache.

Two methods were proposed to determine the reuse distances in the program, one based on profiling which statically assigns a cache hint to a memory instruction and one based on analytical calculation which allows to dynamically select the most appropriate hint. 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 and for every single memory access. Furthermore, the analytical calculation allows to generate dynamic cache hints, i.e. different cache hints can be generated for different executions of the same original memory instruction. 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 7% with a maximum of 36%. The pointer-intensive programs profit most from the target cache hints, while for the numerical programs the source and target cache hints are of equal importance. 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. On average, the static hints reduce the number of cache misses by 7%, while the dynamic cache hints reduce the cache misses by 12%. Dynamic target cache hints are especially valuable in programs where the same instruction can exhibit wildly varying locality. Therefore, in the future, we will further investigate ways to efficiently select cache hints dynamically, also for non-loop oriented programs.

9. REFERENCES

[1] 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.

[2] 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.

[3] I. R. Bratt, A. Settle, and D. A. Connors. Predicate-based transformations to eliminate control and data-irrelevant cache misses. In *Proceedings of EPIC-1*, pages 15–22, 2001.

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

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

[6] 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.

[7] 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.

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

[9] 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.

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

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

[12] 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.

[13] 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*, pages 500–509, 1999.

[14] 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.

[15] 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.

[16] 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.

[17] M. Presburger. Über die Volständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du Premier Congrès des Mathématiciens des Pays Slaves*, pages 91–101, 1929. Warsaw, Poland.

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

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

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

[25] 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.

[26] F. Zhang. *The FPT Parallel Programming Environment*. PhD thesis, Ghent University, 1996.