J Supercomput (2016) 72:770–788
DOI 10.1007/s11227-015-1612-8

$\mathrm{DS}_{\mathrm{spirit}}$: a data dependence and stride reference patterns profiling infrastructure

Hairong Yu[1] · Guohui Li[1] · LihChyun Shu[2]

Published online: 16 January 2016

Abstract Despite the widespread use of multi-core processors in modern computer systems, developing software tools so as to make best use of available computing resources has never been more urgent. This is because a considerable amount of spurious dependence and cache misses lurking in general-purpose applications restricts seriously the extraction of potential parallelism on the nowadays prevalent multi-core machines. Existing tools are limited in their ability to thoroughly detect data dependence and provide prefetched objects simultaneously. Further, some of the tools are unable to profile large-scale applications. To address this problem, we propose a novel profiler, called $\mathrm{DS}_{\mathrm{spirit}}$, that performs both data dependence and stride reference profiling. Data dependence profiling employs a hash-based scheme to detect actual data dependence while filtering out useless dependence via timestamps. Stride reference profiling employs value profiling to profile the stride pattern for each dynamic load and select the profitable loads as prefetched objects for compilers. To demonstrate the effectiveness of $\mathrm{DS}_{\mathrm{spirit}}$, we have evaluated it using several SPEC CPU2006, MPI2007 and OMP2012 benchmarks on an Intel i7-4700 machine. Experimental results show that $\mathrm{DS}_{\mathrm{spirit}}$ produces accurate profiling results, including expected data dependence and prefetched objects, which in turn contributes to more opportunities for extracting parallelism.

✉ LihChyun Shu
shulc@mail.ncku.edu.tw

Hairong Yu
hairongy@hust.edu.cn

Guohui Li
guohuili@hust.edu.cn

1 Huazhong University of Science and Technology, Wuhan, China

2 National Cheng Kung University, Tainan, Taiwan

Keywords Profiler · Profiling-based approach · Stride reference patterns · Data dependence profiling

1 Introduction

Multi-core processors are ubiquitous nowadays. To make best use of available computing resources, automatic parallelization, such as DSWP [23], PS-DSWP [22] and HELIX [4], is commonly used to transform single-threaded applications into concurrently executed multi-threaded applications. Despite so, these conservative static analysis approaches usually limit the opportunities of potential parallelism to be discovered in general-purpose applications, since a large amount of spurious dependence cannot be resolved accurately by compilers. Further, the poor space and temporal locality sometimes exhibited in multi-threaded applications also make static analysis difficult, since the compiler has limited ability to perform accurate reuse analysis for general-purpose applications at compile time. Consequently, some researchers have shifted their focus from static analysis to dynamic analysis.

Profiling is one of the most promising dynamic analysis techniques. A few popular profiling tools, such as Pin [13], ATOM [7], Valgrind [19], and DynamoRIO [39], are widely used in practice. To detect accurate data dependence, the techniques behind the tools can be simply classified into three categories: hash-based dependence profiling [5,35,38], stride-based dependence profiling [10], and objection-based dependence profiling [37]. Hash-based dependence profiling is the most common approach that utilizes a hash function to index the address values of memory references. Whenever a memory reference occurs, the address value of the memory reference is checked against the address values recorded in the hash function. Based on the order of memory references, the types of data dependence can be determined.

Stride-based dependence profiling is an extension of hash-based dependence profiling. The basic idea is that it computes a stride expression for each load (store) instruction, if all the memory addresses visited by the load (store) instruction can be inferred using the stride expression, the values of these addresses are not recorded in the hash function. This technique is usually used in cases when hash-based dependence profiling is not able to have enough memory set aside for profiling purpose, as it records dependence relationships in a compact way.

Objection-based dependence profiling employs type and alias information [1] to detect data dependence. This technique is preferable in speculating register promotion and dynamic data structures expansion [36]. While the above-mentioned profiling techniques have taken key steps towards producing highly efficient multi-threaded applications, cache misses exhibited in multi-threaded applications should by no means be neglected, because cache misses would disrupt pipeline execution, which in turn offsets the benefits gained from parallelization.

To improve cache performance, existing profiling tools, such as Valgrind [19], OProfile [12], and perf [14], usually provide some information related to cache miss rates. While studying the reasons of the high misses rates can inspire better cache organization, traditional optimization techniques, such as loop interchange, blocking and software prefetching [1], are usually beneficial to array-based applications, rather than general-purpose applications. The reason is because a considerable number of data structures in general-purpose applications use heap memory space during execution. As it is well known, heap memory can be allocated anywhere in the memory space, meaning that traditional optimization techniques are insufficient to perform reuse analysis accurately at compile time. For this reason, harnessing profiling to help the compiler to optimize the cache organization is a preferable alternative. Profiling techniques observe exactly the dynamic behavior of programs by tracking memory references. Further, they can collect stride reference information for each load instruction that may cause cache misses. Most importantly, profiling techniques help the compilers to identify profitable prefetched objects so as to improve prefetching efficiency. Therefore, it is essential for the profiling tools to provide in-depth analysis of cache effects and memory references simultaneously when it comes to monitoring the dynamic behavior of programs.

In this paper, we propose a novel profiler, called DS_{spirit}, that performs both data dependence profiling and stride reference profiling. Data dependence profiling employs a hash-based scheme to detect true dependence and loop-carried dependence while removing useless data dependence via timestamps. Stride reference profiling uses value profiling to compute the stride pattern for each dynamic load instruction. The dynamic loads with highly reliable stride patterns are selected as prefetched objects for compilers to improve prefetching efficiency. To demonstrate the effectiveness of DS_{spirit}, we have evaluated it using SPEC CPU2006, MPI2007 and OMP2012 benchmark suites on an Intel i7-4700 real machine. Experimental results show that our DS_{spirit} profiler produces accurate profiling results, including expected data dependence and prefetched objects. On average, our profiler DS_{spirit} contributes to 25 % of performance improvements when it is applied to assist automatic parallelization.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 describes the details of our DS_{spirit} profiler. Section 4 presents the experimental evaluation of DS_{spirit}, and finally Sect. 5 concludes the paper.

2 Related work

There is a wide range of research on profiling techniques. We mention some representative studies that are most relevant to our work.

Profiling is one of the most promising techniques to discover potential opportunities for automatic parallelization, including profiling-based parallelization [25,26,29,33] and speculative parallelization [6,21,28,31,32,40]. Among the many approaches, data dependence profiling is the most common technique used in the community of automatic parallelization. Chen et al. [5] proposed a data dependence profiling technique for speculative optimizations. To detect data dependence, a shadow space [18] was employed to store information, and a simple hashing scheme was introduced to facilitate address comparison. Further, to increase speculation efficiency, the probability of dependence edges was incorporated into the profiling results. Despite all the data structures and schemes used, profiling results may still be inaccurate, since the authors had employed sampling techniques to do profiling to reduce the profiling overhead. Different from Chen et al.'s work [5], our profiler DS_{spirit} offers accurate profiling results while keeping profiling overhead low. To achieve this, we omit data dependences that do not prevent parallelization, such as WAW and WAR. Further, we employ time stamps to filter out useless data dependences to reduce time and space complexity. The idea of using timestamps to handle data dependence is inspired by Xin and Zhang et al.'s work [35,38]. Different from their work, which produces incorrect results in the presence of recursions, our profiler DS$_{\text{spirit}}$ produces correct results for any data structures. This is because DS$_{\text{spirit}}$ uses circular buffers to deal with loop iterations, rather than index trees. Kim et al. [10] proposed a stride-based data dependence profiling. This approach is an extension of hash-based scheme, which stores data dependences in a compressed format. While DS$_{\text{spirit}}$ also computes stride information, there are two differences between SD^3 and DS$_{\text{spirit}}$. First, computing stride in SD^3 is to reduce profiling overhead, whereas computing stride in DS$_{\text{spirit}}$ is to perform reuse analysis for dynamic loads. Second, SD^3 focuses on all memory references, whereas DS$_{\text{spirit}}$ only emphasizes the dynamic loads in loops. Ketterlin et al. [9] developed a profiler, called Parwiz, that instruments binary code to guide parallelization. Since instrumenting binary code may not consider the code exactly when it is executed in the system calls, our profiler DS$_{\text{spirit}}$ instruments Immediate Representation (IR) for more actuate analysis. While data dependence profiling has made a great contribution to automatic parallelization, the problem of reducing cache misses exhibited in multi-threading applications has not received much attention in the design of the mainstream profiling tools, which leaves us room for improvement.

3 DS$_{\text{spirit}}$

In this section, we explain our DS$_{\text{spirit}}$ profiler in more detail. Figure 1 shows the framework of DS$_{\text{spirit}}$. It consists of three stages, viz. instrumentation, memory accesses monitoring, and post-processing. We describe each of the DS$_{\text{spirit}}$ stages in the following subsections, using a piece of C code in Fig. 2 as a running example.

3.1 Instrumentation

Instrumentation refers to the act of adding extra code to a program for observing the dynamic behavior of the program [17,19]. In our case, a pass is written to instrument

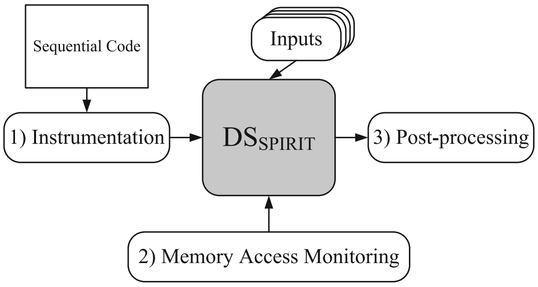
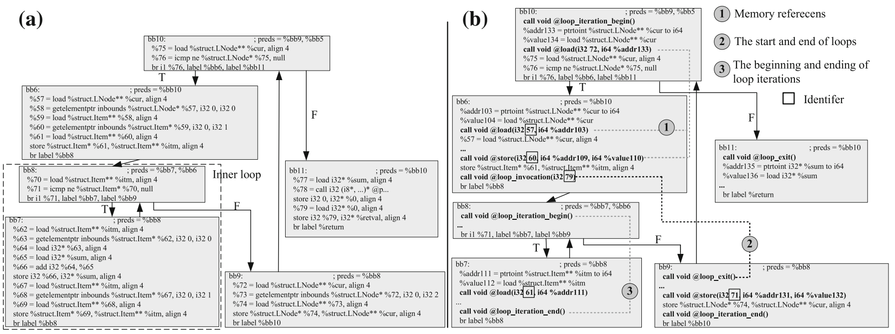


Fig. 1 The framework of DS$_{\text{spirit}}$

Fig. 2 Running example in C code

```
1: while (cur != NULL){
2:     struct Item *itm = cur->item->next;//skip
the head node
3:      while (itm != NULL){
4:          sum += itm->data;
5:          itm = itm->next;
       }
6:    cur = cur->next;
}
```


(a)
T
F
Inner loop
T
F
(b)
T
F
T
F
① Memory referecens
② The start and end of loops
③ The beginning and ending of loop iterations
☐ Identifer
1
2
3

```
bb10:                                   ; preds = %bb9, %bb5
  %75 = load %struct.LNode** %cur, align 4
  %76 = icmp ne %struct.LNode* %75, null
  br i1 %76, label %bb6, label %bb11

bb6:                                    ; preds = %bb10
  %57 = load %struct.LNode** %cur, align 4
  %58 = getelementptr inbounds %struct.LNode* %57, i32 0, i32 0
  %59 = load %struct.Item** %58, align 4
  %60 = getelementptr inbounds %struct.Item* %59, i32 0, i32 1
  %61 = load %struct.Item** %60, align 4
  store %struct.Item* %61, %struct.Item** %itm, align 4
  br label %bb8

bb8:                                    ; preds = %bb7, %bb6
  %70 = load %struct.Item** %itm, align 4
  %71 = icmp ne %struct.Item* %70, null
  br i1 %71, label %bb7, label %bb9

bb7:                                    ; preds = %bb8
  %62 = load %struct.Item** %itm, align 4
  %63 = getelementptr inbounds %struct.Item* %62, i32 0, i32 0
  %64 = load i32* %63, align 4
  %65 = load i32* %sum, align 4
  %66 = add i32 %64, %65
  store i32 %66, i32* %sum, align 4
  %67 = load %struct.Item** %itm, align 4
  %68 = getelementptr inbounds %struct.Item* %67, i32 0, i32 1
  %69 = load %struct.Item** %68, align 4
  store %struct.Item* %69, %struct.Item** %itm, align 4
  br label %bb8

bb11:                                   ; preds = %bb10
  %77 = load i32* %sum, align 4
  %78 = call i32 (i8*, ...)* @p...
  store i32 0, i32* %0, align 4
  %79 = load i32* %0, align 4
  store i32 %79, i32* %retval, align 4
  br label %return

bb9:                                    ; preds = %bb8
  %72 = load %struct.LNode** %cur, align 4
  %73 = getelementptr inbounds %struct.LNode* %72, i32 0, i32 2
  %74 = load %struct.LNode** %73, align 4
  store %struct.LNode* %74, %struct.LNode** %cur, align 4
  br label %bb10
```

```
bb10:                                   ; preds = %bb9, %bb5
  call void @loop_iteration_begin()
  %addr133 = ptrtoint %struct.LNode** %cur to i64
  %value134 = load %struct.LNode** %cur
  call void @load(i32 72, i64 %addr133)
  %75 = load %struct.LNode** %cur, align 4
  %76 = icmp ne %struct.LNode* %75, null
  br i1 %76, label %bb6, label %bb11

bb6:                                    ; preds = %bb10
  %addr103 = ptrtoint %struct.LNode** %cur to i64
  %value104 = load %struct.LNode** %cur
  call void @load(i32 57, i64 %addr103)
  %57 = load %struct.LNode** %cur, align 4
  ...
  call void @store(i32 60, i64 %addr109, i64 %value110)
  store %struct.Item* %61, %struct.Item** %itm, align 4
  call void @loop_invocation(i32 79)
  br label %bb8

bb11:                                   ; preds = %bb10
  call void @loop_exit()
  %addr135 = ptrtoint i32* %sum to i64
  %value136 = load i32* %sum
  ...
  br label %return

bb8:                                    ; preds = %bb7, %bb6
  call void @loop_iteration_begin()
  ...
  br i1 %71, label %bb7, label %bb9

bb7:                                    ; preds = %bb8
  %addr111 = ptrtoint %struct.Item** %itm to i64
  %value112 = load %struct.Item** %itm
  call void @load(i32 61, i64 %addr111)
  ...
  call void @loop_iteration_end()
  br label %bb8

bb9:                                    ; preds = %bb8
  call void @loop_exit()
  ...
  call void @store(i32 71, i64 %addr131, i64 %value132)
  store %struct.LNode* %74, %struct.LNode** %cur, align 4
  call void @loop_iteration_end()
  br label %bb10
```

Fig. 3 The disassembled code of the running example before and after instrumentation. **a** The control flow graph of the running example before instrumentation. **b** The control flow graph of the running example after instrumentation

memory references, nested loops, and function calls. We explain the detailed procedure below.

- Instrumenting memory references: a memory reference refers to one of the following instructions: store, load, allocation, de-allocation, and library call. The memory references are instrumented to keep track of memory accesses.
- Instrumenting nested loops: the elements associated with nested loops are one of the following: pre-header, exit block, header, and latch block. Instrumenting pre-header and exit block is to indicate the start and end of loops, while instrumenting header and latch block is to indicate the beginning and ending of loop iterations.
- Instrumenting function calls: the function calls are instrumented to indicate which caller calls this function, and where this function returns.

Each operand is assigned to a unique identifier that is generated in sequence. Figure 3 shows the disassembled code of the running example before and after instrumentation. To be clear, the pieces of code associated with the instrumented memory references, the start and end of the loops, and the beginning and ending of loop iterations are marked in bold, and the corresponding identifiers are marked in square.

3.2 Memory accesses monitoring

We have designed several key data structures to manage profiling information, including Loop Nest Stack, Counter, Loop Iteration Queue, Last Write Table,

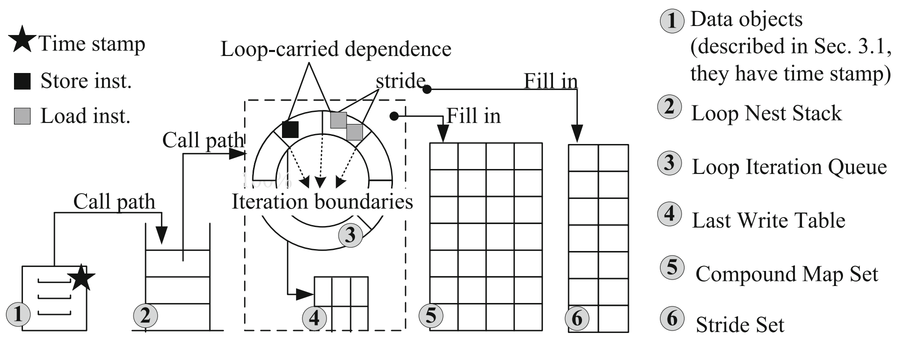


Fig. 4 Overview of profiling components

Compound Map Set and Stride Set, as shown in Fig. 4. We now describe the functionalities of the key data structures below:

- Loop nest stack (LNS): LNS is used to record the loop nesting information. Whenever a loop starts, the loop is pushed into LNS, and then its identifier is incremented by 1. Likewise, whenever a loop ends, the loop is popped off from LNS, and then its identifier is decremented by 1.
- Counter: Counter maintains a global timestamp for all elements of the key data structures. It is initialized to be 0, and incremented by 1.
- Loop iteration queue (LIQ): LIQ utilizes a circular buffer to maintain loop iterations for computing loop-independent dependence and loop-carried dependence. Combing with Counter and LNS, the two kinds of dependence are computed as follows. Firstly, compute the difference between $T_{(\text{LIQ1})}$ and $T_{(\text{LIQ2})}$, denoted by Δ_{T1}, where $T_{(\text{LIQ1})}$ is the timestamp of LIQ1 when the source of a dependence happens, and $T_{(\text{LIQ2})}$ is the timestamp of LIQ2 when the sink of the dependence happens. Secondly, compute the difference between $T_{(\text{LNS1})}$ and $T_{(\text{LNS2})}$, denoted by Δ_{T2}, where $T_{(\text{LNS1})}$ is the timestamp of LNS1 when the loop containing the source of a dependence is pushed into LNS1, and $T_{(\text{LNS2})}$ is the timestamp of LNS2 when the loop containing the sink of the dependence is pushed into LNS2. Thirdly, compute the difference between Δ_{T1} and Δ_{T2}, denoted by Δ_T. If Δ_T equals zero, then the pair of dependence is a loop-independent dependence. Otherwise, it is a loop-carried dependence. Figure 4 gives an illustration. Two squares marked in dark and gray colors across iteration boundaries represent a loop-carried dependence. Since loop-independent dependence only determines the order in which code is executed within loops, it does not prevent parallelization. Hence, we omit loop-independent dependence.
- Last Write Table (LWT): LWT uses the triple of $\langle addr, id, timestamp\rangle$ to maintain the last write access, where *addr* is the address of the last write access, *id* is the identifer, and *timestamp* is the timestamp of the last write. This data structure is used to compute true dependence. Since LWT records the last write access, when a read access is encountered, then the pair of dependence is a true dependence. Because output, input, and anti- dependences do not prevent parallelization, they are filtered out. Further, when the dependence distance between the source of a

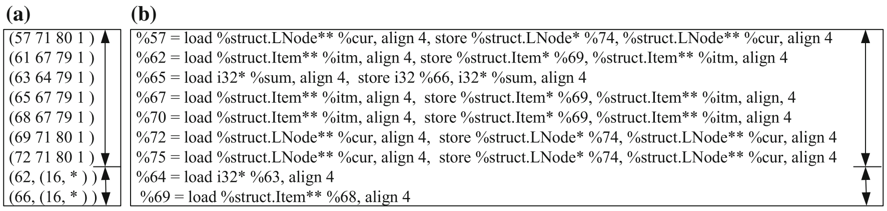
dependence and the sink of the dependence is greater than a threshold, the pair of dependence is also be filtered out.

- Compound Map Set (CMS): CMS uses the quadruple of ⟨*load_id*, *store_id*, *loop_id*, *type*⟩ to record pairs of data dependence, where *load_id*, *store_id*, and *loop_id* are the identifers of *load*, *store*, and *loop*, and *type* is the type of data decadence, including loop-carried dependence and true dependence.
- Stride Set (SS): SS uses the 2-tuple of ⟨*load*, ⟨*stride*, $*$⟩⟩ to record prefetched objects, where *load* is the identifer of prefetched objects, and ⟨*stride*, $*$⟩ records the dominate stride value while reflecting the stride patterns of prefetched objects. The prefetched objects refer to the *dynamic* loads in loops (the trip counts of the loop are at least greater than 200). A dynamic load in our terminology is a load whose input base address is produced by another load. The purpose of this definition is to distinguish from array loads. A *dynamic* load can be selected as a prefetched object only if its non-zero `stride` meets one of the following two conditions: (1) The frequency of a single stride is greater than 75. (2) The frequency of multiple dominate strides is greater than 30. Here, `stride` refers to the difference between two successive access addresses of a dynamic load, as illustrated in Fig. 4. For the first case, ⟨*stride*, $*$⟩ only records the most dominate stride value. For the second case, ⟨*stride*, $*$⟩ records the first two dominate stride values, i.e., *stride* records the first dominate stride value, and $*$ records the second dominate value.

Once profiling is completed, an annotation file that contains both data dependence and prefetched objects is dumped. Figure 5a shows the profiling results. The upper half represents the data dependence and the lower half represents the prefetched objects. For the data dependence, the mapping relationship is identical to the data structure CMS, i.e., ⟨*load_id*, *store_id*, *loop_id*, *type*⟩. For the prefetched objects, the mapping relationship is identical to the data structure SS, i.e., ⟨*load*, ⟨*stride*, $*$⟩⟩. One can see that the contents of the annotation file are recorded in the form of identifiers, hence they are meaningless for programmers. For this reason, we utilize an automatic approach to translate the profiling results into human-readable form, which is illustrated as below.

3.3 Post-processing

We write another pass to translate the profiling results into an Immediate Representation (IR) fashion. This pass performs an opposite process as instrumentation does.



(a)
```
(57 71 80 1 )
(61 67 79 1 )
(63 64 79 1 )
(65 67 79 1 )
(68 67 79 1 )
(69 71 80 1 )
(72 71 80 1 )
(62, (16, * ) )
(66, (16, * ) )
```

(b)
```
%57 = load %struct.LNode** %cur, align 4, store %struct.LNode* %74, %struct.LNode** %cur, align 4
%62 = load %struct.Item** %itm, align 4, store %struct.Item* %69, %struct.Item** %itm, align 4
%65 = load i32* %sum, align 4,  store i32 %66, i32* %sum, align 4
%67 = load %struct.Item** %itm, align 4,  store %struct.Item* %69, %struct.Item** %itm, align, 4
%70 = load %struct.Item** %itm, align 4,  store %struct.Item* %69, %struct.Item** %itm, align 4
%72 = load %struct.LNode** %cur, align 4,  store %struct.LNode* %74, %struct.LNode** %cur, align 4
%75 = load %struct.LNode** %cur, align 4,  store %struct.LNode* %74, %struct.LNode** %cur, align 4
%64 = load i32* %63, align 4
 %69 = load %struct.Item** %68, align 4
```

Fig. 5 Profiling results of the running example before and after post-processing. **a** Profiling results. **b** Profiling results after post-processing

Figure 5b shows the profiling results after post-processing. For the results of data dependence profiling, the dependence relationships are the following: $6 \rightarrow 2, 5 \rightarrow 4, 4 \rightarrow 4, 5 \rightarrow 5, 5 \rightarrow 3, 6 \rightarrow 6,$ and $6 \rightarrow 1$, which corresponds to the statements of running example in Fig. 2. For the results of stride reference profiling, the prefetched objects are $itm- > data$, and $itm- > next$, which corresponds to the pointer itm (lines 4–5) in Fig. 2. Each prefetched object has only one dominate stride, and its stride value is 16.

Once the prefetched objects are identified, we first use the stride patterns of prefetched objects to compute their future addresses (denoted by $Addr_{future}$), and then insert prefetching intrinsic *llvm.prefetch*($Addr_{future}$, *r, 0*) before the prefetched objects. The future address of each prefetched object is computed as follows.

- For a *dynamic* load with one dominate stride value, its future address is computed using the following formula: $Addr_{future} = Addr_{base} + \alpha * stride$, where $Addr_{base}$ is the based address of the *dynamic* load, α is the prefetched distance determined by compilers, and *stride* is the dominate stride value.
- For a *dynamic* load with multiple dominate stride values, its future address is computed by the following steps:
  - (1) Insert a move instruction before the *dynamic* load, and save its address to a temporary register.
  - (2) Insert a subtract instruction after the move instruction to subtract the value in the temporary register from the current address value of the *dynamic* load. The difference is denoted by $stride_i$. Next, insert another move instruction after the subtract instruction, and save $stride_i$ to another temporary register.
  - (3) Compute the future address of the *dynamic* load using the following formula: $Addr_{future} = Addr_{base} + \alpha * stride_i$.

To be clear, we show the disassembled code of the running example before and after prefetching in Fig. 6. In Fig. 6b, each of the prefetching intrinsics *llvm.prefetch* is marked in rectangle, and two prefetched objects, $itm- > data$, and $itm- > next$, are marked in bold.

(a)

```
bb7:                                    ; preds = %bb8
  %62 = load %struct.Item** %itm, align 4
  %63 = getelementptr inbounds %struct.Item* %62, i32 0, i32 0
  %64 = load i32* %63, align 4
  %65 = load i32* %sum, align 4
  %66 = add i32 %64, %65
  store i32 %66, i32* %sum, align 4
  %67 = load %struct.Item** %itm, align 4
  %68 = getelementptr inbounds %struct.Item* %67, i32 0, i32 1
  %69 = load %struct.Item** %68, align 4
  store %struct.Item* %69, %struct.Item** %itm, align 4
  br label %bb8
```

(b)

```
bb7:                                    ; preds = %bb8
  %62 = load %struct.Item** %itm, align 4
  %63 = getelementptr inbounds %struct.Item* %62, i32 0, i32 0
  %ptrtoint1 = ptrtoint i32* %63 to i32
  %addition2 = add i32 %ptrtoint1, 16
  %inttoptr3 = inttoptr i32 %addition2 to i8*
  call void @llvm.prefetch(i8* %inttoptr3, i32 0, i32 0)
  %64 = load i32* %63, align 4
  %65 = load i32* %sum, align 4
  %66 = add i32 %64, %65
  store i32 %66, i32* %sum, align 4
  %67 = load %struct.Item** %itm, align 4
  %68 = getelementptr inbounds %struct.Item* %67, i32 0, i32 1
  %ptrtoint = ptrtoint %struct.Item** %68 to i32
  %addition = add i32 %ptrtoint, 16
  %inttoptr = inttoptr i32 %addition to i8*
  call void @llvm.prefetch(i8* %inttoptr, i32 0, i32 0)
  %69 = load %struct.Item** %68, align 4
  store %struct.Item* %69, %struct.Item** %itm, align 4
  br label %bb8
```

Fig. 6 The disassembled code of the running example before and after prefetching. **a** Before prefetching. **b** After prefetching

3.4 Profiling costs

Since profiling is an execution-driven approach, it must bear time and space overhead. Table 1 summarizes the time and space overhead. From the perspective of time overhead, the overhead of LNS and LIQ is $\mathcal{O}(1)$, since we employ linear data structure to manage LNS and LIQ. The overhead of the remaining data structures is $\mathcal{O}(log(N))$, $(N \in \theta, \eta,\ and\ \xi)$, since we employ RB-tree data structure to manage CMS, LWT and SS. From the perspective of space overhead, the space overhead of LNS, LIQ, CMS, LWT and SS is determined by their storage spaces, viz., the depth of the stack, the length of the queue, the number of the last write accesses, and the number of the dynamic loads.

4 Evaluation

In this section, we evaluate the performance of the proposed DS$_{\text{spirit}}$ profiler. Specifically, Sect. 4.1 describes the experimental setup, and Sect. 4.2 reports the experimental results.

4.1 Experimental setup

We describe our experimental platform, benchmarks used throughout for the evaluation and our methodology below.

Platform We consider a real system: Intel Core i7-4700 with a qual-core processor. The maximum number of available threads is 8. A brief description of the experimental platform is given in Table 2.

Table 1 Profiling cost

Main data structures	Time overhead	Space overhead
Loop nest stack (LNS)	$\mathcal{O}(1)$	$\mathcal{O}(\alpha)$
Loop iteration queue (LIQ)	$\mathcal{O}(1)$	$\mathcal{O}(\gamma)$
Compound map set (CMS)	$\mathcal{O}(log\theta)$	$\mathcal{O}(\theta)$
Last write table (LWT)	$\mathcal{O}(log\eta)$	$\mathcal{O}(\eta)$
Stride set (SS)	$\mathcal{O}(log\xi)$	$\mathcal{O}(\xi)$

Table 2 Detail description of platform Core i7

Processor	Intel(R) Core(TM) i7-4700
L1I cache:	32 KB∗8, 8-way set associative, 64-byte line size
L1D cache:	32 KB∗8, 8-way set associative, 64-byte line size
L2 cache:	256 KB∗8, 8-way set associative, 64-byte line size
Shared L3 cache	8192 KB, 16-way set associative, 64-byte line size
Main memory	DDR3 1333 Mhz, 16 GB
OS	Linux 2.6.32-28-generic-pae

Table 3 Benchmark details

Program	Description	Function name	% of runtime (%)	L1 cache miss rates (%)
429.*mcf*	Network algorithm	`global_opt`	90	35
456.*hmmer*	Hidden Markov model	`main_loop_serial`	90	2
470.*lbm*	Lattice Boltzmann method	`LBM_performStreamCollide`	90	17
462.*libquantum*	Shor algorithm	`quantum_qft`	80	2
401.*bzip*2	Compression algorithm	`handle_compress`	70	2
464.*h*264*ref*	Video compression	`code_a_picture`	85	1
104.*milc*	Quantum chromodynamics	`g_measure`	40	13
142.*dmilc*	Quantum chromodynamics	`g_measure`	45	7
352.*nab*	Molecular modeling	`md`	90	<1
358.*botsalgn*	Protein alignment	`align`	85	<1
359.*botsspar*	Sparse LU	`sparselu_par_call`	95	<1
372.*smithwa*	Pattern matching	`main`	100	<1

Table 4 Instrumentation information and benchmark overhead statistics

Program	Loads/stores/ calls	Original runtime (s)	Runtime increased	Original space	Profiling space (G)
429.*mcf*	1508/588/63	16.65	436050×	15 M	150
456.*hmmer*	25334/7840/1663	147	8816×	12 M	200
470.*lbm*	2220/305/50	0.31	278709×	3 M	70
462.*libquantum*	3427/1618/158	3.56	159685×	11 M	60
401.*bzip*2	9999/3349/387	12.6	27348×	11 M	50
464.*h*264*ref*	11357/6930/385	31.1	27948×	42 M	70
104.*milc*	8654/3526/653	139	11810×	284 M	10
142.*dmilc*	9539/3941/738	168	10285×	482 M	15
352.*nab*	16129/5895/1536	155	6789×	11 M	1.5
358.*botsalgn*	1373/571/453	3.09	390193×	1 M	<1
359.*botsspar*	726/190/165	3.74	300320×	174 M	2
372.*smithwa*	3829/1139/597	0.12	15120000×	<1 M	2.5

Benchmarks To evaluate our profiling tool DS$_{\text{spirit}}$, we have chosen several parallelizable programs from SPEC2006 [8], MPI2007 [16] and OMP2012 [15] benchmarks, respectively. The basic characteristics of these benchmarks are given in Table 3. In the table, column 3 shows the function name containing the parallelizable loops, column 4 indicates the runtime of the loops normalized with respect to the total execution time of the program, and the last column shows the L1 cache misses rates (tested with *perf* [14]). Table 4 gives the basic information about instrumentation and profiling overhead. In the table, column 2 gives the numbers of loads, stores and function calls, respectively. Column 3 shows the original runtime of each benchmark program, followed by the increased runtime needed by $\mathrm{DS_{spirit}}$ in columns 4. Column 5 shows the original space needed to run each benchmark program, followed by the space needed by $\mathrm{DS_{spirit}}$ in columns 6.

Methodology $\mathrm{DS_{spirit}}$, an off-line profiling, is completely developed on top of the LLVM compiler infrastructure [11]. Our profiling results can be used in profiling-based parallelism [29,33] and speculative parallelism [21,27,31,40,41]. The two techniques can be divided into three parallelism paradigms: independence multi-threading (IMT) techniques, cyclic multi-threading (CMT) techniques and pipeline multi-threading (PMT) techniques, based on the patterns of inter-thread communication [3]. Our profiling results are used for profiling + IMT (CMT and PMT) techniques and speculative + IMT (CMT and PMT) techniques. We use DOALL [24], HELIX [4], and DSWP [20] as the baselines of IMT, CMT and PMT techniques, respectively. We use profiling + DOALL (HELIX and DSWP) and speculative + DOALL (HELIX and DSWP) as the enhanced versions of IMT, CMT and PMT, respectively. Each of these algorithms is written as a project added to the LLVM compiler.

We use a machine learning approach [30,34] to determine the parallelization scheme. We employ support vector machines (SVM) with a radical basis function as a kernel to classify the considered parallelization schemes, and we set parallelism with static analysis as the default parallelization scheme (i.e., the baseline). We say profiling-based parallelism (or speculative parallelism) is preferable only if it brings about at least 15 % performance gains than the default scheme. The reason is because both profiling-based and speculative techniques incur extra overhead. For convenience, we denote the default parallelization scheme as $\mathcal{S}$, profiling-based parallelism as $\mathcal{A}$, and speculative parallelism as $\mathcal{B}$. In this way, two groups of binary classifications are built, i.e., (class $\mathcal{S}$ vs. class $\mathcal{A}$) and (class $\mathcal{S}$ vs. class $\mathcal{B}$). Notice that we will separately predict whether the output belongs to (classes $\mathcal{S}$ vs. $\mathcal{A}$) or belongs to (classes $\mathcal{S}$ vs. $\mathcal{B}$).

We use the most common method, called "leave-one-out" cross-validation, to evaluate our parallelization scheme. The procedure is stated as follows. First of all, we leave one program out from all the benchmark programs, and then train a model on the remaining programs. Second, for the left-out program (also called the unseen program), we only extract its features (including instructions, loads/stores, branches, and so on), followed by presenting the features to the SVM predictor, with the parallelization scheme of the program produced as the prediction result. We repeat this procedure for each program in turn, meaning that each program will be predicted once. The prediction results of each program are shown in the fifth column in Table 5 (called parallelization).

4.2 Evaluation results

In this section, we discuss the implementations of our profiler $\mathrm{DS_{spirit}}$ in SPEC CPU2006, MPI2007 and OMP2012 benchmark suits.

4.2.1 Practical implementation in SPEC CPU2006

With our $\mathrm{DS_{spirit}}$ profiler, the simple flow group of the extracted pipeline for each selected CPU2006 benchmark programs is depicted Fig. 7. The speedups with different

Table 5 Profiling information statistics

Program	Total Dep.	True Dep.	Loop-carried Dep.	Parallelization		Stride patterns
				(Without prefetch)	(With prefetch)	
429.*mcf*	902	761	141	$\mathcal{S}$	$\mathcal{B}, \mathcal{S}$	Yes
456.*hmmer*	2965	2223	742	$\mathcal{A}, \mathcal{B}$	$\mathcal{A}, \mathcal{B}$	Yes
470.*lbm*	854	752	102	$\mathcal{A}, \mathcal{S}$	$\mathcal{A}, \mathcal{S}$	Yes
462.*libquantum*	1630	1252	378	$\mathcal{A}, \mathcal{S}$	$\mathcal{A}, \mathcal{S}$	No
401.*bzip*2	3648	2635	1013	$\mathcal{A}, \mathcal{B}$	$\mathcal{A}, \mathcal{B}$	Yes
464.*h*264*ref*	11,375	6930	385	$\mathcal{A}, \mathcal{B}$	$\mathcal{A}, \mathcal{B}$	Yes
104.*milc*	3874	2764	1110	$\mathcal{A}, \mathcal{S}$	$\mathcal{A}, \mathcal{S}$	Yes
142.*dmilc*	4752	3521	1231	$\mathcal{A}, \mathcal{S}$	$\mathcal{A}, \mathcal{S}$	Yes
352.*nab*	738	409	329	$\mathcal{S}$	$\mathcal{S}$	No
358.*botsalgn*	1032	547	485	$\mathcal{S}$	$\mathcal{S}$	No
359.*botsspar*	254	218	36	$\mathcal{S}$	$\mathcal{S}$	No
372.*smithwa*	586	439	87	$\mathcal{S}$	$\mathcal{S}$	Yes

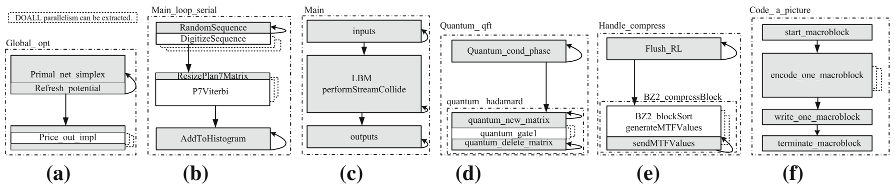


Fig. 7 Simple flow group of the extracted pipeline in SPEC CPU2006. **a** 429.mc. **b** 456.hmmer. **c** 470.lbm. **d** 462.libquantum. **e** 401.zip2. **f** 464.h264ref

parallelization techniques are given in Figs. 8 and 9. We present an in-depth discussion of DS$_{\text{spirit}}$ below.

1. *429.mcf.* This benchmark utilizes a network simplex algorithm to mange public mass transportation. We start from the highly parallelizable function, `global_opt`, to perform various analysis. This function is mainly composed of three subroutine calls, viz., `primal_net_simplex`, `refresh_potential`, and `price_out_impl`, as depicted in Fig. 7a. Because of pointer passing between functions, the inter-procedural dependence analysis [1] reveals that pointer *net* causes a considerable amount of loop-carried dependences and prohibits the potential opportunities for parallelization. Further, the program suffers from seriously cache misses (the L1 cache miss rates are 35 %), since the heap memory spaces are widely used in this program. In particular, function `primal_net_simplex` takes approximately 80 % of the cache misses.

Our profiler DS$_{\text{spirit}}$ aims to overcome these limitations from two aspects. First, data dependence profiling tries its best to analyze exactly data dependence. By studying the results of data dependences, we note that spurious dependences account for a small portion of the total dependences. Thus, harnessing profiling-based parallelism

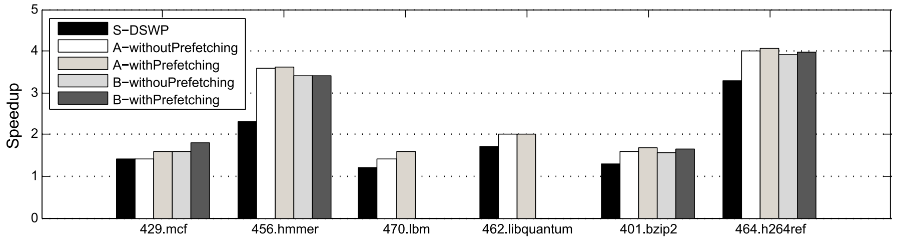


Fig. 8 Speedup with PMT parallelization techniques in SPEC CPU2006

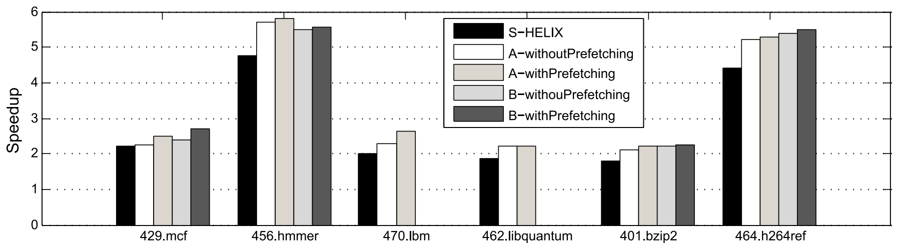


Fig. 9 Speedup with CMT parallelization techniques in SPEC CPU2006

in this program may not be an optimal solution. By contrast, speculative parallelism is a preferable alternative. This is because there are several dependences that rarely manifest, e.g., loop-carried dependences are caused by node $node- > potential$ in function `refresh_potential`. Using speculation, such dependences can be ignored safely by compilers, allowing more potential parallelism to be discovered. Nevertheless, speculative parallelism assisted using the results of data dependence profiling alone cannot outperform the default parallelization scheme in terms of speedup by 15 %, as demonstrates in the 5th column of Table 5.

Second, we employ the stride reference profiling to create new opportunities for improving cache efficiency. As it is well known, when a perfect cache is combined with pipeline parallelism, it can improve the efficiency of parallel execution significantly. According to the results of stride reference profiling, functions `primal_net_simplex`, `refresh_potential` and `price_out_impl` contain prefetched objects, such as pointers *bea*, *acrin* and *tail*. With prefetching, both speculative parallelism and profiling-based parallelism achieve significant performance improvements over the defaults parallelization scheme, as Figs. 8 and 9 demonstrate.

2. *456.hmmer*. This benchmark profiles the hidden Markov models for researching the pattern of DNA sequences. The considered target for parallelization is function `main_loop_serial`. Figure 7b shows the primary three stages: (1) Generate random numbers. (2) Perform the Viterbi algorithm. (3) Add to the histogram for getting a score. To maximize parallelism, we use commutative annotations [2] in the first stage.

Based on the $\text{DS}_{\text{spirit}}$ results, we also analyze the program from two aspects. First, data dependence profiling plays a key role in eliminating spurious dependence. For example, pointer operations, such as $xmx = mx->xmx, mx->xmx\_mem[0] = mx->xmx\_mem, and\ mx->xmx\_mem[i] = mx->xmx\_mem + padding$, can be eliminated in function `P7Viterbi`. Then, DOALL parallelization [24] is amenable to the second stage. Compared to speculative parallelism, profiling-based parallelism is more preferable. The reasons are twofold: (1) Speculating loop-carried dependences would lead to high mis-speculation rates. (2) The percentage of infrequent basic blocks is small, according to the frequency of dependence edges. Due to the fact that speculating the frequent dependence edges would offset the benefits from parallelization, speculative parallelism is less efficient than profiling-based parallelism.

By observing the results of stride reference profiling, we found that the prefetched objects mainly appear in function `P7Viterbi`, such as variables *hmm* and *xmx*. In practice, the L1 cache miss rates are not manifested in this program, prefetching only contributes to approximately 2 % performance improvements, as Figs. 8 and 9 demonstrate.

3. *470.lbm*. This benchmark simulates incompressible fluid in 3D using a Lattice Boltzmann Method. It comprises three stages: (1) Handling input (output) flow. (2) Performing stream collision. and (3) Outputs. Figure 7c gives an illustration. This program extensively utilizes macros to hide the details of the data access, such as X(*dstGrid*) = Y(*srcGrid*), which copies pointer from *srcGrid* to *dstGrid*.

We emphasize on function `LBM_performStreamCollide`, since it possesses a considerable amount of opportunities for parallelization. Because *srcGrid* and *dstGrid* cause pointer alias [1], they bring a big challenge to the compiler, since pointer alias cannot be resolved exactly at compile time. Profiling, however, provides a means to analyze accurate pointer alias and remove spurious data dependence. In our experiments, profiling-based parallelism brings about nearly 20 % of performance gains, compared to the default parallelization scheme. However, for speculative parallelism, this approach is impractical. The reason is that the dependences caused by *srcGrid* and *dstGrid* are too manifest to benefit from speculation. According to the cache miss rates, we learn that this program suffers from poor cache locality (the L1 cache miss rates are up to 17 %). Prefetching is an effective way to improve cache effects. With prefetching, parallelization techniques benefit from on average 15 % of more performance gains. Figures 8 and 9 show the overall performance of this benchmark program.

4. *462.libquantum*. This benchmark utilizes the Shor's algorithm to solve the problem of quantum computation in polynomial time. The key part occurs in function `quantum_qft`. Figure 7d shows the simplified flow graph.

According to the results of data dependence profiling, a considerable number of dependences are caused by pointer variable reg. With the help of profiling, a small portion of code in function `quantum_gate1` can be parallelized in DOALL, as illustrated in Fig. 7d. Since dependences caused by reg are remarkable, speculative parallelism is less effective. Notice that while function `quantum_gate1` suffers from cache misses, stride-based prefetching is not an optimal solution, since there are no prefetched objects to be detected during profiling. To avoid useless prefetching, which would result in the premature eviction of useful cache lines, and increase the needlessly memory traffic, this program does not provide prefetching. The overall performance of this benchmark program is shown in Figs. 8 and 9.

5. *401.bzip2*. This benchmark performs a block-based compression algorithm. It comprises two parts: compression and decompression. We mainly consider compression part, which occurs in function `handle_compress`. Figure 7e shows the simple flow graph. According to the DS$_{\text{spirit}}$ results, function `BZ2_compressBlock` except `sendMTFValues` is amenable to DOALL parallelism, as depicted in Fig. 7e. Further, prefetching contributes to approximately 5 % of performance improvements. The overall performance with different parallelization techniques is given in Figs. 8 and 9.

6. *464.h264ref*. This benchmark is a video compression program. We focus on function `code_a_picture`. Figure 7f depicts the flow graph of the extracted pipeline. Function `encode_one_macroblock` contains the opportunities of DOALL parallelization. Further, this function has various stable access patterns, e.g., $X+ = Y[refptr++ \oplus orgptr++]$, where $orgptr$ and $refptr$ are the pointers of the original and reference blocks, respectively. In this function, both profiling-based parallelism and speculative parallelism are profitable. Prefetching brings about nearly 2 % of improvements. The overall performance with different parallelization techniques is given in Figs. 8 and 9.

4.2.2 Practical implementation in MPI2007

With our DS$_{\text{spirit}}$ profiler, the simple flow group of the extracted pipeline for each selected MPI2007 benchmark programs is depicted Fig. 10a, b. The speedup with different parallelization techniques is given in Fig. 11. We present an in-depth discussion of DS$_{\text{spirit}}$ below.

7. *104.milc*. This benchmark performs quantum Chromodynamics. We emphasize on function `g_measure`. Figure 10a shows the simple flow graph for the extracted pipeline. With our profiling results, we perform various analysis on this function. First of all, data dependence profiling shows that a majority of loop-carried dependence is caused by structure pointer s, which mainly appears in function `path_product`. For this reason, speculative parallelism is not applied, since speculating frequent dependence edges would incur high mis-speculation rates. By contrast, profiling-based parallelism is beneficial to more aggressive parallelism, since profiling can disambiguate pointer alias and remove spurious dependence. On the other hand, according

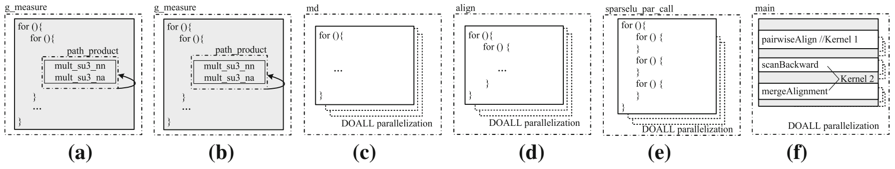


Fig. 10 Simple flow group of the extracted pipeline in SPEC MPI2007 and OMP2012. **a** 104.milc. **b** 142.dmilc. **c** 352.nab. **d** 358.botsalgn. **e** 359.botsspar

to the results of stride reference profiling, two subroutines, viz., `mult_su3_nn` and `mult_su3_na`, contain prefetched objects. Prefetching contributes to approximately 5 % of performance improvements, since function `g_measure` only takes a small portion of overall execution time. Figure 11 shows the overall performance with different parallelization schemes.

8. *142.dmilc*. This benchmark also performs quantum Chromodynamics. It is almost identical to 104.milc, with some minor differences, as shown in Fig. 10b. Likewise, this program is also suitable for profiling-based parallelism. Figure 11 shows the overall performance.

4.2.3 Practical implementation in OMP2012

With our DS$_{\text{spirit}}$ profiler, the simple flow group of the extracted pipeline for each selected OMP2012 benchmark programs is depicted Fig. 10c–f. The speedup with DOALL parallelization techniques is given in Fig. 12. We present an in-depth discussion of DS$_{\text{spirit}}$ below.

9. *352.nab*. This benchmark is a molecular modeling, floating point intensive application. The key part occurs in function `md`. The simplified flow graph is shown in Fig. 10c. According to our DS$_{\text{spirit}}$ results, the loops in function `md` are amenable to DOALL parallelization, since there are no loop-carried dependences. There are no prefetched objects to be detected by our profiling tool. This is because this program mainly adopts array-based accesses, hence a majority of loads do not meet the requirements of our stride reference profiling. Figure 12 shows the overall performance.

10. *358.botsalgn*. This benchmark aligns protein sequences using the Myers and Miller algorithm. We mainly focus on function `align`. Figure 10d depicts the simple flow graph of the extracted pipeline. Same as 352.nab, this program can also be parallelized

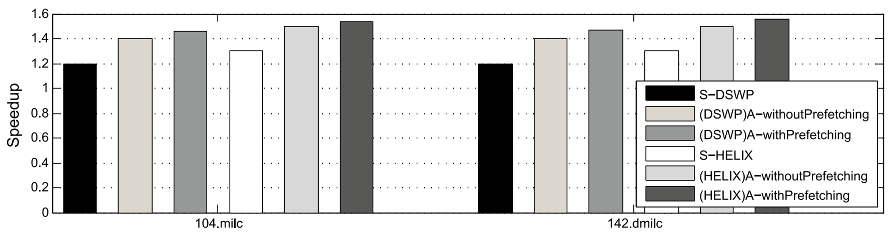


Fig. 11 Speedup with PMT and CMT parallelization techniques in SPEC MPI2007

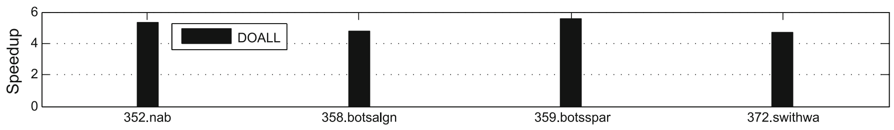


Fig. 12 Speedup with IMT parallelization techniques in SPEC OMP2012

in DOALL, since there is no parallelism-preventing dependence. Likewise, there are no prefetched objects. The overall performance is given in Fig. 12.

11. *359.botsspar*. This benchmark is a sparse LU matrix factorization. The considered target for parallelization is function `sparselu_seq_call`. The simplified flow graph of the extracted pipeline is depicted in Fig. 10e. According to the profiling results, functions `lu0`, `fwd`, `bdiv`, and `bmod` contain loop-carried dependences. But, by further studying the dependences, we found that these dependences do not prevent parallelization, since they can be eliminated by privatization [1]. Then, function `sparselu_seq_call` can be parallelized in DOALL. There are no prefetched objects, since a large amount of loads is array loads. Figure 12 shows the overall performance.

12. *372.swithwa*. This benchmark is a pattern matching program. We consider `main` function that is composed of two key kernels occurring functions `pairwiseAlign`, `scanBackward`, and `mergeAlignment`, as depicted in Fig. 10f. While loop-carried dependences are detected in these three functions, they can be removed by privatization. Hence, the program is amenable to DOALL parallelization. Further, according to the results of stride reference profiling, prefetched objects are detected in function `insertValidation`. Since prefetching plays a marginal role in the overall performance, the benefits obtained from prefetching are omitted. Figure 12 shows the overall performance of this program.

In summary, the results produced by DS$_{\text{spirit}}$ can be used to do profiling + CMT (IMT and PMT) and speculative + CMT (IMT and PMT) in SPEC CPU2006 benchmark programs, and to do profiling + CMT (and PMT) in MPI2007 benchmarks. Further, programs except 462.libquantum in both CPU2006 and MPI2007 benchmarks benefit from prefetching. The OMP2012 benchmarks can be parallelized using IMT techniques. Except for 372.swithwa, our profiler DS$_{\text{spirit}}$ does not detect prefetched objects from 352.nab, 358.botsalgn, and 359.botsspar. The main reason is that we do not care about array loads, since cache misses from array-based programs can be resolved by traditional cache optimization techniques, such as loop interchange and blocking [1].

5 Conclusions

In this paper, we propose a novel profiler, called DS$_{\text{spirit}}$ that performs both data dependence profiling and stride reference profiling in general-purpose programs. Data dependence profiling employs a hash-based scheme to detect true dependence and loop-carried dependence while filtering out useless dependences via the timestamps. Stride reference profiling employs value profiling to compute stride reference patterns for each dynamic load and select the profitable loads as the prefetched objects for compilers to do prefetching. To demonstrate the effectiveness of DS$_{\text{spirit}}$, we have evaluated it using SPEC CPU2006, MPI2007, and OMP2012 benchmark programs on an Intel i7-4700 machine. Experimental results show that our profiler DS$_{\text{spirit}}$ produces accurate profiling results. On average, DS$_{\text{spirit}}$ contributes to 25 % of more performance improvements when it is applied to assist parallelization.

References

1. Allen R, Kennedy K (2002) Optimizing compilers for modern architectures. Morgan Kaufmann, San Francisco
2. Bridges M, Vachharajani N, Zhang Y, Jablin T, August D (2007) Revisiting the sequential programming model for multi-core. In: Proceedings of the 40th Annual IEEE/ACM international symposium on microarchitecture. IEEE Computer Society, pp 69–84
3. Bridges MJ (2008) The velocity compiler: extracting efficient multicore execution from legacy sequential codes. Ph.D. thesis, Princeton University
4. Campanoni S, Jones T, Holloway G, Reddi VJ, Wei GY, Brooks D (2012) Helix: automatic parallelization of irregular programs for chip multiprocessing. In: Proceedings of the tenth international symposium on code generation and optimization. ACM, pp 84–93
5. Chen T, Lin J, Dai X, Hsu WC, Yew PC (2004) Data dependence profiling for speculative optimizations. In: Compiler construction. Springer, pp 57–72
6. Ding C, Shen X, Kelsey K, Tice C, Huang R, Zhang C (2007) Software behavior oriented parallelization. In: ACM SIGPLAN Notices, vol 42. ACM, pp 223–234
7. Eustace A, Srivastava A (1995) Atom: a flexible interface for building high performance program analysis tools. In: Proceedings of the USENIX technical conference proceedings. USENIX Association, pp 25–25
8. Henning JL (2006) Spec cpu2006 benchmark descriptions. ACM SIGARCH Comput Archit News 34(4):1–17
9. Ketterlin A, Clauss P (2012) Profiling data-dependence to assist parallelization: framework, scope, and optimization. In: Proceedings of the 2012 45th annual IEEE/ACM international symposium on microarchitecture. IEEE Computer Society, pp 437–448
10. Kim M, Kim H, Luk CK (2010) Sd3: a scalable approach to dynamic data-dependence profiling. In: Proc. of the 43rd annual IEEE/ACM international symposium on microarchitecture (MICRO). IEEE, pp 535–546
11. Lattner C, Adve V (2004) Llvm: a compilation framework for lifelong program analysis & transformation. In: Proc. of the international symposium on code generation and optimization (CGO). IEEE, pp 75–86
12. Levon J (2004) Oprofile manual. Victoria University of Manchester
13. Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K (2005) Pin: building customized program analysis tools with dynamic instrumentation. In: ACM SIGPLAN notices, vol 40. ACM, pp 190–200
14. de Melo AC (2010) The new linuxperftools. In: Slides from Linux Kongress
15. Müller MS, Baron J, Brantley WC, Feng H, Hackenberg D, Henschel R, Jost G, Molka D, Parrott C, Robichaux J, et al (2012) Spec omp2012 an application benchmark suite for parallel systems using openmp. In: OpenMP in a heterogeneous world. Springer, pp 223–236
16. Müller MS, van Waveren M, Lieberman R, Whitney B, Saito H, Kumaran K, Baron J, Brantley WC, Parrott C, Elken T et al (2010) Spec mpi2007 an application benchmark suite for parallel systems using mpi. Concurr Comput Pract Exp 22(2):191–205
17. Nethercote N (2004) Dynamic binary analysis and instrumentation. Ph.D. thesis, PhD thesis, University of Cambridge
18. Nethercote N, Seward J (2007) How to shadow every byte of memory used by a program. In: Proceedings of the 3rd international conference on virtual execution environments. ACM, pp 65–74
19. Nethercote N, Seward J (2007) Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan Not 42(6):89–100
20. Ottoni G, Rangan R, Stoler A, August DI (2005) Automatic thread extraction with decoupled software pipelining. In: Proceedings of the 38th Annual IEEE/ACM international symposium on microarchitecture (MICRO). IEEE, p 12
21. Raman A, Kim H, Mason TR, Jablin TB, August DI (2010) Speculative parallelization using software multi-threaded transactions. In: ACM SIGARCH computer architecture news, vol 38. ACM, pp 65–76
22. Raman E, Ottoni G, Raman A, Bridges MJ, August DI (2008) Parallel-stage decoupled software pipelining. In: Proceedings of the 6th annual IEEE/ACM international symposium on code generation and optimization. ACM, pp 114–123

23. Rangan R, Vachharajani N, Vachharajani M, August DI (2004) Decoupled software pipelining with the synchronization array. In: Proceedings of the 13th International conference on parallel architectures and compilation techniques. IEEE Computer Society, pp 177–188
24. Rauchwerger L, Padua D (1994) The privatizing doall test: a run-time technique for doall loop identification and array privatization. In: Proceedings of the 8th international conference on supercomputing. ACM, pp 33–43
25. Rul S, Vandierendonck H, De Bosschere K (2008) Extracting coarse-grain parallelism in general-purpose programs. In: Proceedings of the 13th ACM SIGPLAN Symposium on principles and practice of parallel programming. ACM, pp 281–282
26. Rul S, Vandierendonck H, De Bosschere K (2010) A profile-based tool for finding pipeline parallelism in sequential programs. Parallel Comput 36(9):531–551
27. Steffan JG, Colohan C, Zhai A, Mowry TC (2005) The stampede approach to thread-level speculation. ACM Trans Comput Syst (TOCS) 23(3):253–300
28. Tian C, Feng M, Nagarajan V, Gupta R (2008) Copy or discard execution model for speculative parallelization on multicores. In: Proceedings of the 41st annual IEEE/ACM International symposium on microarchitecture. IEEE Computer Society, pp 330–341
29. Tournavitis G, Franke B (2010) Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In: Proceedings of the 19th international conference on parallel architectures and compilation techniques. ACM, pp 377–388
30. Tournavitis G, Wang Z, Franke B, OBoyle M (2009) Towards a holistic approach to auto-parallelization. In: 2009 Conference on programming language design and implementation (PLDI)
31. Vachharajani N, Rangan R, Raman E, Bridges MJ, Ottoni G, August DI (2007) Speculative decoupled software pipelining. In: Proceedings of the 16th international conference on parallel architecture and compilation techniques. IEEE Computer Society, pp 49–59
32. Vachharajani NA (2008) Intelligent speculation for pipelined multithreading. Princeton University
33. Vandierendonck H, Rul S, De Bosschere K (2010) The paralax infrastructure: automatic parallelization with a helping hand. In: Proceedings of the 19th international conference on parallel architectures and compilation techniques. ACM, pp 389–400
34. Wang Z, O'Boyle MF (2009) Mapping parallelism to multi-cores: a machine learning based approach. In: ACM Sigplan notices, vol 44. ACM, pp 75–84
35. Xin B, Sumner WN, Zhang X (2008) Efficient program execution indexing. In: ACM SIGPLAN notices, vol 43, no 6. ACM, pp 238–248
36. Yu H, Ko HJ, Li Z (2013) General data structure expansion for multi-threading. In: Proceedings of the 34th ACM SIGPLAN conference on programming language design and implementation. ACM, pp 243–252
37. Yu H, Li Z (2012) Fast loop-level data dependence profiling. In: Proceedings of the 26th ACM international conference on supercomputing. ACM, pp 37–46
38. Zhang X, Navabi A, Jagannathan S (2009) Alchemist: a transparent dependence distance profiling infrastructure. In: Proceedings of the 7th annual IEEE/ACM international symposium on code generation and optimization, pp 47–58
39. Zhao Q, Sim JE, Wong WF, Rudolph L (2006) Dep: detailed execution profile. In: Proceedings of the 15th international conference on parallel architectures and compilation techniques. ACM, pp 154–163
40. Zhong H, Mehrara M, Lieberman S, Mahlke S (2008) Uncovering hidden loop level parallelism in sequential applications. In: Proc. of IEEE 14th international symposium on high performance computer architecture (HPCA). IEEE, pp 290–301
41. Zilles C, Sohi G (2002) Master/slave speculative parallelization. In: Microarchitecture, 2002. (MICRO-35). Proceedings 35th Annual IEEE/ACM international symposium on. IEEE, pp 85–96