A Study of the Performance Potential for Dynamic Instruction Hints Selection

Rao Fu[1], Jiwei Lu[2], Antonia Zhai[1], and Wei-Chung Hsu[1]

[1] Department of Computer Science and Engineering
University of Minnesota
{rfu, zhai, hsu@cs.umn.edu}
[2] Scalable Systems Group
Sun Microsystems Inc.
jiwei.lu@sun.com

Abstract. Instruction hints have become an important way to communicate compile-time information to the hardware. They can be generated by the compiler and the post-link optimizer to reduce cache misses, improve branch prediction and minimize other performance bottlenecks. This paper discusses different instruction hints available on modern processor architectures and shows the potential performance impact on many benchmark programs. Some hints can be effectively selected at compile time with profile feedback. However, since the same program executable can behave differently on various inputs and performance bottlenecks may change on different micro-architectures, significant performance opportunities can be exploited by selecting instruction hints dynamically.

1 Introduction

Cache misses and branch mispredictions have become the major bottlenecks of modern microprocessors. Attacking such performance issues has been a challenge for both hardware designers and software developers. Many modern architectures, including RISC, VLIW and EPIC, have paid much attention to the effective cooperation between the compiler and the hardware to achieve highly efficient execution. For instance, new instructions such as data and instruction cache prefetch have been introduced and they have been effectively used by the compiler and post-link optimizers (including runtime optimizers) to reduce cache miss penalties. Besides introducing new instructions, recent architectures also use *instruction hints* as another way to facilitate the communication between the compiler and the hardware. Unlike adding new instructions, using hints does not compromise binary compatibility. Instruction hints use a small number of available bits in the instruction encoding to allow programmers, compilers and other software tools to convey suggestions to the hardware. Since they are defined as hints, they do not pose correctness issues. Their presence can be simply ignored if the underlying micro-architecture does not support the needed feature.

Instruction hints are often used in architecture extensions and new architectures to expose new hardware features to software via some reserved bits.

Judiciously selecting the instruction hints can have very significant performance impact on applications. The selection of instruction hints relies on information such as working set, access patterns and effective memory latencies, which are not generally available at the compile time. Although profile-guided optimization (also known as profile-based or profile-directed optimization) can assist the selection process by using profile information collected via training runs, applications can behave differently on various inputs, and the profile collected from the training input may not be representative for the actual run. Furthermore, the runtime behavior of a program can change even within one run (i.e. execution phase changes). Although we have seen encouraging results from static hint selection, we believe there are significant performance potentials to be exploited with dynamic hint selection.

Dynamic binary optimizers [6][7] can monitor the execution of a program and perform the cost-effective optimization based on observed hot spots and respective performance bottlenecks. Dynamic hint selection requires relatively small amount of code analysis and binary modification and can be a good candidate for dynamic optimization. However, the extension of current dynamic binary optimization frameworks and the enhancement of current microprocessors are needed to support comprehensive dynamic hint selection.

The paper makes the following contributions,

- We show the performance impact of several architecture hints using the SPEC2000 CPU programs.
- We show the potential of using correct architecture hints over what have been done statically by the compiler.
- We discuss the limitations and difficulties associated with static hint selection.
- We discuss the current limitation on the hardware performance monitoring capability for exploiting dynamic hint selection.

The rest of the paper is organized as follows. Section 2 will provide a survey of instruction hints available on the mainstream architectures. Section 3 shows the performance impact of several instruction hints. In section 4, we discuss the selection of hints by some production compilers, the effectiveness of such selection, and the limitations. In section 5, we discuss the upside potential of selecting such hints at runtime, and the constraints and challenges for the dynamic optimizers. Related work is highlighted in section 6. Section 7 contains the conclusion and future work.

2 Instruction Hints

Most instruction hints are targeting at the two major performance bottlenecks, cache misses and branch mis-predictions. They can be divided into three main categories, branch prediction hints to improve branch prediction, memory locality hints to improve both data and instruction cache performance, strong/weak prefetch hints to improve the effectiveness of the data prefetch instructions.

2.1 Branch Prediction Hints

Many architectures use one or two bits in the conditional branches as a hint for static branch prediction. Itanium [14] uses one bit to indicate whether prediction resources should be allocated for the branch and the other bit to indicate the direction. Similarly Power4 [8] uses two previously reserved bits in conditional branch instructions for software branch prediction. Hardware branch prediction can be overridden by software branch prediction on Power4. One bit is used for that purpose while the other bit indicates the direction. PA-RISC 2.0 [15][17] does not have the luxury of one available bit but it defines the *branch prediction convention* to achieve the same effect. If the register numbers of the two operands in a conditional branch is in increasing order, the backward branch is predicted taken and the forward branch is predicted not taken; otherwise the branch is predicted the other direction. Compared with using a dedicated hint bit, this approach adds complexity to the instruction decoding.

Many microprocessors uses a return address stack to predict the target of a procedure return. When a procedure call is executed, the address of the next instruction is pushed onto the stack. The stack will be popped during the execution of a procedure return and the instruction fetching will start from the popped address. But in architectures such as Alpha [1], PowerPC [12] and PA-RISC [11][15], there are no dedicated instructions for procedure call and return. In Alpha [1], hints are introduced to push and pop procedure return addresses. PA-RISC 2.0 [15] and Power4 [8] adopted the same approach.

2.2 Memory Locality Hints

Memory locality hints are designed to achieve better cache performance by improving the allocation and replacement policy or initiating hardware prefetching. The temporal locality hints are used to indicate whether the data will be reused to help the hardware decide whether to allocate the data in a higher cache level. The temporal locality hints can be applied to all memory instructions including load, store and data prefetching. HP PA-RISC 1.1 architecture [11] defines a 2-bit cache control field, *cc*, which provides a hint to the processor on how to allocate data in the cache hierarchy. On PA-7200 [16], the processor will not allocate the cache line on the off-chip cache if the *cc* is specified to indicate poor temporal locality. The cache control field is also included in the prefetch instruction introduced in PA-RISC 2.0 [15].

Five variants of prefetch are defined in Sparc V9 [18], *read many*, *read once*, *write many*, *write once* and *prefetch page*[1]. The *once* and *many* hints are used

[1] *prefetch page* has not been implemented in any existing Sparc v9 microprocessors.

to indicate the temporal locality. UltraSparc III [20] implements a small prefetch cache (2KB) which can be accessed in parallel with the L1 data cache for floating-point loads. The *once/many* hint specifies whether the data should be brought into P-cache. However, no temporal hints are available for other memory instructions.

Itanium [14] provides locality control with finer granularity. Four completer (*t1*, *nt1*, *nt2* and *nta*) are used to specify whether the data has temporal locality at a given cache level. These completers will affect how cache lines are allocated in the cache hierarchy and whether the LRU bit should be updated. Using *t1* will cause the data to be allocated at all cache levels while using *nt1* suggests the data not to be allocated at L1. The Itanium 2 processor does not have a non-temporal buffer and L2 is used for that purpose. *nt2* accesses are still allocated in L2 but the LRU bit will not updated and thus the line has a high probability to be replaced. *nta* completer further causes the line not to be allocated in L3. Only *lfetch* instructions can use all four possible completers and the completers for different memory instructions may have different meanings.

Instruction references exhibit good sequential locality. Many microprocessors implement hardware prefetcher to sequentially prefetch instruction cache lines. Itanium [14] introduces the *sequential prefetch hint* to initiate instruction prefetching. The *sequential prefetch hint* on branches indicates how many cache lines the processor should prefetch starting at the branch target. On Itanium 2 [13], a branch with the *many* completer initiates the hardware streaming prefetching and the prefetch engine will continuously issue prefetch requests for subsequent instruction cache lines till a stop condition[2] happens.

2.3 Weak/Strong Prefetch Hints

The effectiveness of prefetching can be affected by whether micro-architecture implementations allow a prefetch to continue if it triggers a data TLB miss or there is not enough resource to handle the prefetch request. The UltraSparc IV+ processor [21] implements two more variants of prefetch instructions in addition to the five flavors defined in Sparc V9 [18]. Weak prefetches are dropped if the target address translation misses the TLB, while strong prefetches will generate software traps and be re-issued after the TLB entries are filled. The prefetch requests are tracked by an eight-entry prefetch queue. A strong prefetch will not be dropped even if the prefetch queue is full when it is issued and the pipeline will stall until one of the outstanding prefetches completes. The PCM (P-Cache Mode) in DCU (Data Cache Unit) control register provides further control on the behavior of weak prefetches under a prefetch queue full event. When the bit is on, a weak prefetch will also be recirculated if the prefetch queue is full.

On Itanium [14], a TLB miss will not necessarily generate a fault since it implements hardware page walker to reduce the latency of a TLB miss. If a lookup

[2] A stop condition can be a branch misprediction, the execution of an taken branch or the execution of a special instruction explicitly indicating the stop condition.

fails in both levels of the DTLB, hardware page walker can be triggered to resolve the miss by searching the page table. Slightly different with strong prefetch on UltraSparc IV+, *fault* completer is used to indicate whether a fault raised by an *lfetch* instruction should be handled by the processor. If the hardware page walker fails, only *lfetch.fault* will raise a software fault. Unlike UltraSparc IV+ [21], there is no dedicated resource for tracking data prefetching requests on Itanium. They share the same resource with the other memory requests. An *lfetch* instruction will not be dropped if there is not enough resource to handle it. Instead it will wait for the resource to be available.

3 Performance Impact of Instruction Hints

Though the instruction hints do not affect the correctness of a program's execution, they can have great impact on program performance. This section uses several instruction hints to show the compiler can improve program performance by judiciously using the available hints.

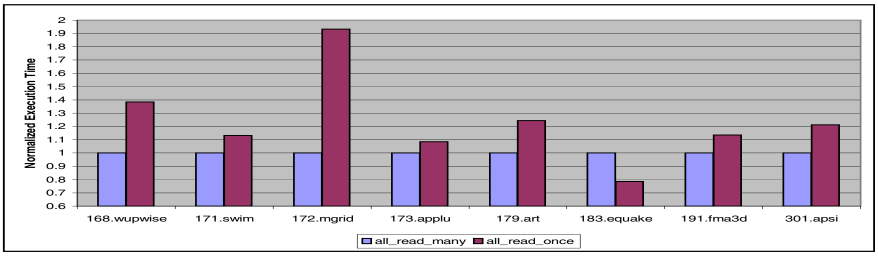
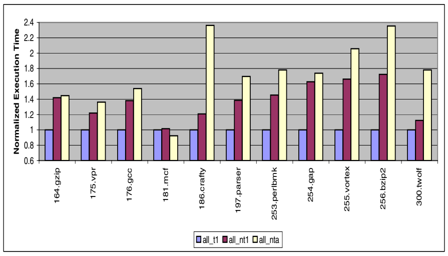


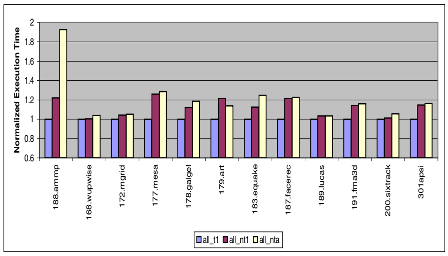
Fig. 1. Performance comparison of prefetch variants with different locality hints on UltraSparc III Cu for SPEC CPU2000 [19]. All binaries are compiled with the base option including PBO using Sun Studio 11 compiler and the data are collected on Sun Blade 1000. The execution time is normalized using the binaries generated by the compiler as the bases. The first bars are all 1 since the compiler only generates *many* hints.

Figure 1 shows the comparison of using two different locality hints for data prefetching on UltraSparc III Cu. By using the *read many* hint, the prefetched data are brought into both P-cache and L2 cache while the data are only brought to P-cache for *read once*. The compiler only generates *read many* hint for prefetches intended for data reads. Although only using the *read many* hint gives better performance in most cases, for 183.equake, only using *read once* actually has a 27% speedup.

The comparison of using different locality hints for load on Itanium 2 is shown in figure 2. For every possible completer allowed, we convert all loads into that single flavor and compare the performance with the binaries generated by the compiler. Two separate graphs are shown for SPEC CINT2000 and CFP2000



(a) CINT2000



(b) CFP2000

Fig. 2. Performance comparison of load variants with different locality hints on Itanium 2. All binaries are compiled with the base option including PBO using Intel C/C++ Compiler 9.0 and the data are collected on HP zx6000. The execution time is normalized using the binaries generated by the compiler as the bases. The first bars are all 1 since the compiler only generates *t1* hints.

[19] since *t1* and *nt1* have different meanings for floating point loads [3]. Intel compiler only uses *t1* for loads and using *t1* is clearly a better choice than using *nt1* or *nta* as shown in figure 2. But there is one exception that *mcf* benefits from only using *ld.nta*.

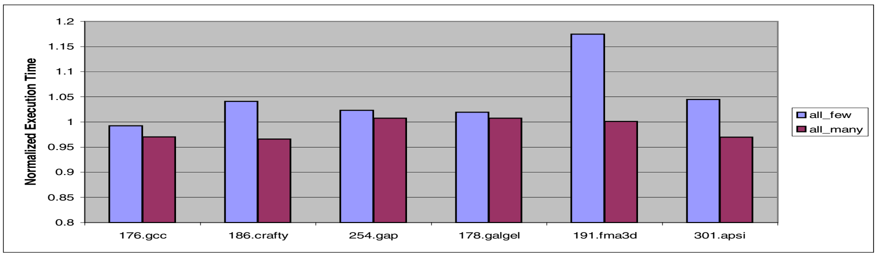


Fig. 3. Performance impact of streaming prefetch hint on Itanium 2. The same environment as specified in figure 2 is used for data collection.

The performance impact of streaming prefetch hint on Itanium 2 is shown in figure 3. The *many* completer is clearly preferable than the *few* completer. On three occasions (*gcc*, *crafty* and *apsi*), triggering streaming prefetches on every branch delivers better performance. Only using *few* completers can slow down a program as much as 17% in the case of *fma3d*.

Figure 4 shows the comparison for weak and strong prefetches on UltraSparc IV+. Again we show two extreme cases by converting all prefetches into weak or strong versions. In general, the compiler chooses strong versions for the majority

[3] For floating point loads, data are not allocated in L1 even *t1* is used and LRU bit is not updated for *nt1*.

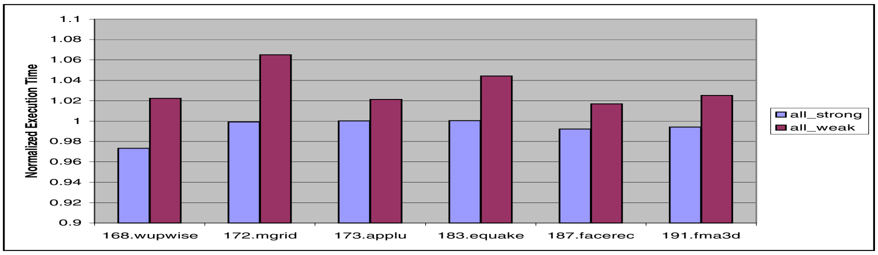


Fig. 4. Performance comparison of weak/strong prefetches on UltraSparc IV+. The data are collected on a Sun Fire E4900 sever and all binaries are compiled with the latest Sun Studio compiler [22] with the base option including PBO. The execution time is normalized using the binaries generated by the compiler as the bases.

of the prefetches and it yields better performance for six programs compared with only using the weak versions. Only using strong prefetches provides even slightly better performance overall. For *168.wupwise*, 3% speedup can be obtained by only using strong prefetches.

4 Static Selection of Instruction Hints

As shown in section 3, prudently using instruction hints can significantly improve program performance. In this section, we discuss the issues involved in static selection of these hints, including the branch prediction hints, instruction prefetching hints, data cache locality hints and weak/strong prefetch hints. We also show limitations of static selection using case studies for several benchmark programs.

4.1 Issues in Static Selection

Locality Hints for Data Prefetching. The cache hierarchies in modern processors are increasingly more complex. The cache hierarchy in the Itanium 2 has three levels of on-chip caches. They are non-blocking and can handle cache miss requests out-of-order. Therefore, it is difficult to estimate the precise cost of an *lfetch* instruction. In general, *lfetch* instructions with *t1* completers are more expensive than those with *nt* completers while *lfetch* instructions with *nt* completers (*nt1*, *nt2*, and *nta*) have similar costs.

On Itanium 2, every memory request that cannot be satisfied by L1D will be sent to L2 and must be scheduled within a 32-entry queue called OzQ. If the OzQ is full, the L1D pipeline must stall and it in turn causes the main pipeline to stall. Bank conflicts and multiple misses to the same cache line can increase the lifetime of the entries in the OzQ. An *lfetch* can be expensive if it cause either case to happen. Placing one of the *nt* completers mitigates those effects and reduces the cost of an *lfetch*. However, using the *nt* completer reduces the benefit of an *lfetch* since the prefetched data will only be brought up to the L2 cache. When deciding to use the *t1* completer, the compiler needs to be confident that the benefit outweighs the cost. The Intel compiler tends to use *nt* more often than *t1* for SPEC CINT2000 programs. But choosing between *t1* and *nt* relies largely on the application's working set as discussed in 4.2 and neither of them works best all the time.

Streaming Prefetch Hints. The Itanium 2 processor has a relative small instruction cache (16K), from the perspective that it has a very strong issue bandwidth (up to 6 instructions can be issued per cycle). Overly aggressive streaming prefeching can cause instruction cache pollution and have negative impact on the pipeline front-end. The benefit of streaming prefetching can be determined by whether the lines brought into the L1I are used in the near future. A good indicator will be the number of instructions between the branch target to the first statically predicted taken branch. ISpike [4] defines this as *span* and uses a size of 128 bytes as a threshold to trigger streaming prefetching.

Intel compiler is rather conservative in selecting *many* completers. On average only one out of four branches uses the *many* completer for SPEC CPU2000 programs even we compile all programs with high optimization level (O3) and profile based optimization. Three programs (*gcc*, *crafty* and *apsi*) benefit as much as 4.5% from only using *many* completers as show in figure 3. All three programs have large instruction footprints and streaming prefetch can reduce the stalls when the pipeline front-end is unable to supply new instructions to the back-end.

Weak/Strong Prefetch Hints. As shown in section 3, strong prefetch can provide additional benefits over weak prefetchs on UltraSparc IV+, but a strong prefetch could be more expensive than a weak one. Firstly, a strong prefetch must wait when the prefetch queue is full while a weak prefetch can be simply dropped in this case. Secondly, a TLB miss triggered by a strong prefetch must be handled. The compiler must carefully use strong prefetches and make sure the performance gain from the prefetches is higher than the additional cost. The weak prefetches can be made "stronger" on UltraSparc IV+ by setting the PCM bit to 1 so that they will not get dropped when the prefetch queue is full. With the PCM bit set on, the difference between weak and strong prefetches becomes smaller, which makes it easier to select strong prefetch as the default. However, we have observed that setting the PCM bit on does not always yield better performance since programs may spend a significant portion of execution on waiting for available entries in the prefetch queue. The stall can be avoidable by providing flexible control over the PCM bit and relying on the compiler to more intelligently select the more suitable prefetch variants.

4.2 Limitations of Static Selection

As shown in section 3, though overall the compilers do well in selecting instruction hints, there are cases the compilers still leave significant performance opportunities on the table. This is evident when we blindly convert all instruction hints into one flavor. Static selection of instruction hints is also limited by lacking knowledge of a program's runtime behavior.

```
while (node) {
    ...
    temp = node;
    node = node->child;
}
```

(a) C code

```
(p17) adds r46=40,r37
        ...
(p17) ld8 r36=[r46]
        ...
(p17) cmp.eq p0,p16=r36,r0
        ...
(p16) br.wtop.dptk.few
```

(b) assembly code

Fig. 5. Code snippet from 181.mcf

Ambiguous Memory Access. The static analysis can be hindered by some programming language features. Figure 5 shows a code snippet from function *refresh_potential* in SPEC CINT2000 benchmark *181.mcf*. The loop is software pipelined but no prefetch instructions are generated by the compiler. The *ld8* instruction which tries to access *node* → *child* is delinquent and the program stalls on the *cmp* instructions. Since the data loaded by *ld8* are not reused, changing its completer to *nta* can reduce its latency without increasing the number of cache misses. The program can be sped up by 8.7% after this simple change. However, it is unlikely that the compiler can determine whether the data are reused with the presence of intensive dynamic memory objects and frequent pointer references.

```
void daxpy(double *x, int ix, double *y, int iy, int a, int n)
{
    int i;

    for (i = 0; i < n; i++)
        y[i * iy] += a * x[i * ix];
}
```

Fig. 6. DAXPY loop

Memory Access Pattern. The behavior of a program can change dramatically with different memory access patterns. Figure 6 shows a typical DAXPY loop with the strides for both arrays passed as the parameters. The Sun Studio compiler generates one strong prefetch for each array on UltraSparc IV+. As seen in the figure 7, the benefit of using weak prefetches is decreasing as the stride gets larger. When the memory stride (1024 for the arrays) is equal to

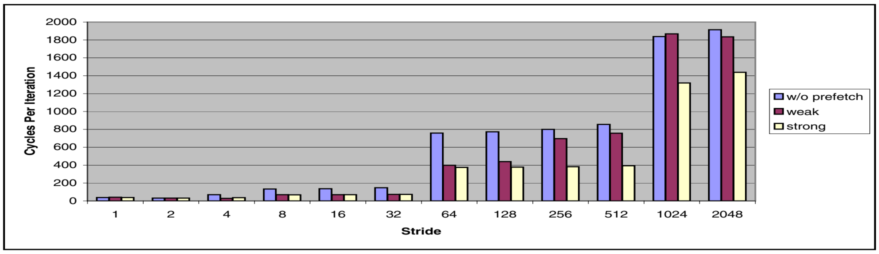


Fig. 7. Average cycles per iteration for DAXPY loop for different array stride (*ix* and *iy*.

the page size (8KB), we can see a sharp increase on average cycles spent on each iteration because of the TLB pressure. Using weak prefetches cannot provide better performance since most of the prefethes will cause TLB misses and get dropped. Strong prefetches clearly outperform weak prefetches for the large strides. But when the stride is no larger than 512 bytes (64 for *ix* and *iy*), using strong prefetches is hardly better than using weak prefetches. If the PCM bit is set to be off, weak prefetches may be more profitable for smaller strides because of their lower costs.

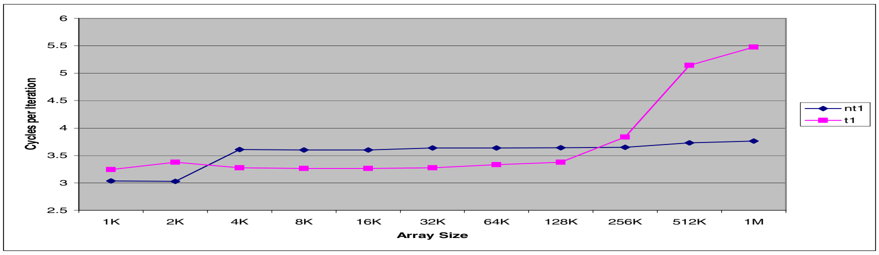


Fig. 8. Average cycles per iteration for IAXPY loop

Working Set Size. The runtime behavior of a program can largely rely on its working set. A static hint is unlikely to provide good performance across different working sets. To show the effect of *t1* completer, we change the DAXPY loop in figure 6 to IAXPY (i.e., both x and y are changed to integer arrays and the strides are fixed to be 1). The Intel C compiler generates a software-pipelined loop for IAXPY and one prefetch is included to prefetch both array x and y alternatively. Figure 8 shows the average cycles per iteration for the IAXPY on an Itanium 2 machine for two cases when the temporal completer of the prefetch is *nt1* or *t1*. When the working set of the loop is bigger than the size of L1D (16K) and but less than the size of L2 (256KB), using *t1* gives better performance. But if the size of array exceeds the size of L2, *nt1* will provide better performance and the performance gap is increasing as the work set increases.

5 Dynamically Selecting Instruction Hints

Static selection of instruction hints is limited by the lack of knowledge of program's runtime behavior and a static hint cannot adapt to behavior changes at runtime. Two ways can be used to select instruction hints dynamically. The first approach uses the compiler to generate multiple copies of an instruction with all possible hints, and have check instructions to select the desirable one based on the runtime performance information. The second approach is to use a dynamic binary optimizer, such ADORE [6], to adjust the hint bits at runtime.

5.1 Generating Multiple Copies

Compiler can generate multiple copies of an instruction with different hints and the corresponding code to select the hints at runtime as discussed in [2][3]. The selection can be based on the calculation on various runtime parameters (working set, stride and etc.). This scheme has two disadvantages which make it impractical. Firstly, it is known to cause severe code expansion since the compiler has to generate extra instructions to select the instruction with the wanted hint. Secondly, the cost of the additional calculations can offset the performance gain of using the right hints.

5.2 Adjusting Instruction Hints Using Dynamic Binary Optimizers

Using compiler to generate multiple copy of an instruction with different hints causes code expansion and has high runtime overhead. This approach has another limitation since the compiler can only generate the instruction hints with the knowledge of the target architecture. A binary compiled for an older micro-architecture cannot benefit from the additional instruction hints available on the newer micro-architecture. Recompilation is one possible solution but the source code for the legacy binaries may not be available.

A dynamic binary optimizer can monitor a program's performance during the execution of the program. It can identify program hot spots as well as pin down the performance bottlenecks. Based on the observed performance bottleneck and hot spots, the dynamic optimizer can perform the most needed optimizations, and deploy the optimized code by patching the binary. It has been shown to effectively address runtime performance bottlenecks such as data cache misses. Compared with generating multiple copies at the compile time, using dynamic binary optimizers to adjust instruction hints can have very low overhead and adapt to different target micro-architectures and computing environments.

Compared with other optimizations currently implemented in dynamic binary optimizers, dynamically adjusting the instruction hints is less expensive. Optimizations such as partial dead code elimination requires flow analysis of the binary. Those optimizations also need to be carefully applied since they can change the architecture state and cause imprecise exceptions. Most optimizations require some free registers and acquiring them from the binary at runtime is very challenging. For dynamically adjusting instruction hints, if sufficient information can be obtained from hardware, the optimization only needs to patch one or two bits for some instructions. Trace formation and register acquisition, two of the most difficult tasks in dynamic optimizers, can be avoided.

However, similar to other runtime optimizations, dynamic hint selection needs proper support from software and hardware. The lack of appropriate performance counter information related to the instruction hints may limit the effectiveness of hint selection. Furthermore, the lack of comprehensive control flow information may also limit the code region where hint selection can be applied.

Hardware Support. Dynamic binary optimizers rely on the runtime performance monitoring features provided by recent architectures. Itanium 2 provides more than 400 different counters and advanced monitoring features such as Branch Trace Buffer (BTB) and Event Address Registers (EAR). Those features are very useful in the design of a dynamic binary optimizer. However, they are still insufficient for dynamically adjusting instruction hints. For example, to select the memory locality hints, *nta*, no temporal locality at all cache levels, requires cache reuse information. We need to know if the cache line referenced by one memory operation is not going to be reused, or the line may be replaced before it is used again. Current hardware performance counters do not provide this type of details. Furthermore, it is important that the cache line reuse information should be associated with the PC address of the memory instruction. One naïve hardware implementation is to tag the cache line with the full address of the instruction which requests the line. This may be too expensive to be practical. So using partial address (e.g. lower bits) may be a good compromise. A few bits like the LRU bits can also be added to track whether the line has been used recently.

Software Support. Data cache prefetching is the major optimization performed in current dynamic binary optimizers such as ADORE/Itanium [6] and ADORE/Sparc [7]. Therefore the trace selection and formation in these two systems focus on loops which are the best candidates for data cache prefetching. Dynamically adjusting instruction hints requires different type of traces. The effect of changing some instruction hints such as the temporal hints may not be visible immediately. For example, adjusting the temporal hints for a loop may not improve the performance of itself but the performance of another loop next to it. In such cases, we need a larger scope such as a complex loop nest in the trace selection in order to effectively apply hint selection. Secondly, self monitoring and dynamically undoing and redoing the optimization become critical. For example, the dynamic optimizer may initiate some hint selection to a loop, and monitor what performance change it may have. If the performance degrade in the monitored region, the optimizer should undo the selected hints.

6 Related Work

Even though there are quite a few instruction hints available on recent architectures, very limited research has been done to evaluate their performance impacts and no one has tried to select instruction hints using dynamic binary optimizers.

Memory Locality Hint: Wang et al. [10] propose to add an *evict-me* bit to each cache line, which indicates a cache line is a good candidate for replacement. Compilers set this bit for memory instructions based on locality analysis. Their study shows that using the evict-me algorithm in both L1 and L2 caches can improve the performance of a set of scientific programs over LRU policy by increasing the cache hit rate. Yang [5] et al. has a detailed study on the compiler algorithms to generate cache hints. Beyles and D'Hollander [2][3] proposes a compiler framework using reuse distance profile to generate temporal hints for memory instructions. Their study is based on the temporal completers available on Itanium architecture [14] and they used a physical Itanium server for their experiment. They also propose to use prediction or extending the format of the memory instructions to support dynamic cache hint for an individual access.

Weak/Strong Prefetch: Song et al. [9] briefly describe the weak and strong prefetch on UltraSparc VI+ [21]. They only use strong prefetches in the statically generated helper thread by the compiler and they claim the benefit of helper thread will be greatly reduced if prefetches are dropped on TLB misses. In [7], Lu et al. evaluate the performance impact of using strong prefetches in their dynamic helper threaded prefetching on UltraSparc IV+ [21]. Even though they conclude using strong prefetches in the helper thread code is generally a preferable strategy, they also find cases when weak prefetches yield better performance.

Sequential Prefetch Hint: Luk et al. [4] study the performance potential of streaming prefetching on Itanium [14] using a post-link optimizer (Ispike). They find streaming prefetching helps a little for SPEC CPU2000 Int [19] programs but they observe larger speedup on a commercial database application with a much bigger code footprint.

7 Conclusions and Future Work

Modern processors have increasingly relied on using hints associated with instructions to pass performance related information from software to hardware. We have shown the use of such hints could have significant performance impact on recent Itanium and Sparc processors. The statically hint selection by the compiler cannot address the performance opportunities created by dynamic program behavior changes and has room for improvement. With appropriate software and hardware support, we believe a dynamic optimizer can make more effective use of instruction hints for future systems.

Our future work will focus on the software and hardware support for dynamic selecting instruction hints. We want to enhance the current dynamic binary optimizer to handle more complex trace types other than loops. We also plan to improve the self-monitoring ability and add support for undoing and redoing optimizations. Finally we would like to have more detailed study and evaluation for possible hardware support to assist future dynamic selection of instruction hints.

Acknowledgment This work is partly supported by grant EIA-0220021 and grants from Intel and Sun Microsystems. The authors want to thank Abhinav Das and Jinpyo Kim for their suggestions and help. We also thank all of the anonymous reviewers for their valuable comments.

References

1. Alpha architecture handbook, Oct 1998.
2. K. Beyls and E. D'Hollander. Compile-time cache hint generationfor epic architectures. In *EPIC-2*, Nov 2002.
3. K. Beyls and E. H. D'Hollander. Generating cache hints for improved program efficiency. *J. Syst. Archit.*, 51(4):223–250, 2005.
4. C. K. Luk et al. Ispike: a post-link optimizer for the intel®itanium®architecture. In *CGO 2004*, pages 15–26, 2004.
5. H. Yang et al. Compiler-assisted cache replacement: Problem formulation and performance evaluation. *Lecture Notes in Computer Science*, 2958:77–92, 2004.
6. J. Lu et al. Design and implementation of a lightweight dynamic optimization system. *JLPT*, 6(1), 2004.
7. J. Lu et al. Dynamic helper threaded prefetching on the sun ultrasparc cmp processor. In *MICRO '05*, pages 93–104, 2005.
8. J. M. Tendler et al. Power4 system microarchitecture, Oct 2001.
9. Y. Song et al. Design and implementation of a compiler framework for helper threading on multi-core processors. In *PACT '05*, pages 99–109, 2005.
10. Z. Want et al. Using the compiler to improve cache replacement decisions. In *PACT '02*, pages 199–208, 2002.
11. Hewlett-Packard Company. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 3rd edition, Feb 1994.
12. IBM. *PowerPC User Instruction Set Architecture*, Sep 2003.
13. Intel Corp. *Intel®Itanium®2 Processor Reference Manual for Software Development and Optimization*, May 2004.
14. Intel Corp. *Intel®IA-64 Architecture Software Developer's Manual*, Jan 2006.
15. Gerry Kane. *PA-RISC 2.0 Architecture.* Prentice Hall, 1995.
16. G. et al. Kurpanek. Pa7200: a pa-risc processor with integrated high performance mp bus interface. In *Compcon Spring '94, Digest of Papers*, pages 375–382, 1994.
17. R. Lee and J. Huck. 64-bit and multimedia extensions in the pa-risc 2.0 architecture. In *Compcon '96*, pages 152–160, Feb 1996.
18. SPARC International, Inc. *The SPARC Architecture Manual Version 9*, 1994.
19. Standard Performance Evaluation Corp., http://www.spec.org/cpu2000.
20. Sun Microsystems Inc. *UltraSPARC®III Processor User's Manual*, Jan 2004.
21. Sun Microsystems Inc. *UltraSPARC®IV+ Processor User's Manual Supplement*, Oct 2005.
22. Sun Studio Compilers and Tools, http://developers.sun.com/prodtech/cc/-index.jsp.