System-Wide Performance Monitors and their Application to the Optimization of Coherent Memory Accesses

Jean-Francois Collard
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, California
jeff.collard@hp.com

Norm Jouppi
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, California
norm.jouppi@hp.com

Sami Yehia*
ARM Ltd
110 Furlbourn Road
Cambridge , CB1 9HR, UK
sami.yehia@arm.com

ABSTRACT

Inspired by recent advances in microprocessor performance monitors, this paper shows how a shared-memory multiprocessor chipset and interconnect can be equipped with performance monitors that associate performance events with the PCs of the individual instructions causing these events. Such monitors greatly simplify performance debugging of shared-memory programs—for example, they make finding pairs of instructions in false sharing straightforward. These monitors also enable precise feedback-directed compiler optimizations and, as a second contribution, we show how they can guide the code generator to use the version of the load instruction that makes the best use of the coherence protocol. Experiments show up to almost 10% coherence traffic reduction on SPLASH2 applications.

Categories and Subject Descriptors

C.5 [**Computer System Implementation**]: Miscellaneous

General Terms

Performance, Design

Keywords

Performance monitors, coherence traffic

1. INTRODUCTION

Performance monitors in microprocessors have proven to be invaluable. They enable performance debugging tools such as HP's Caliper and Intel's VTune, post-link binary optimizers [8] and feedback-directed compiler optimizations [2, 5]. In fact, they are so popular that a library for portable performance monitors was created [7]. In recent processors, such as Itanium, monitors not only report which performance events took place but also which instructions caused them. On Itanium, these monitors rely on hardware counters called Event Address Registers (EARs) that provide the program counter (PC) of the instruction causing a given event, making them a precious tool for performance analysis.

This paper shows how the same capability can be extended to shared-memory multiprocessors, that is, how chipsets and interconnect fabrics can be equipped with performance monitors that associate performance events with the PCs of the individual instructions that originate the events. We call these monitors SWIFT (System-Wide Information for Tuning) monitors. Our technique involves having processors send out a unique instruction ID together with each transaction. This ID comprises a processor ID, a thread ID, and the PC. Clearly, SWIFT monitors are only needed while tuning performance and can be turned off during production runs.

The benefits of correlating system-wide performance events to originating instructions is intuitive. For example, consider false sharing: it is very hard, even for the seasoned programmer, to detect pairs of program statements that access different data but compete for the same line. With SWIFT monitors, identifying such pairs is straightforward. This kind of performance analysis has been demonstrated, for instance, by Nagarajan et al. [11]. However, they had to resort to a software simulator for the lack of hardware support. This paper shows SWIFT monitors provide the same information "natively."

To further illustrate the benefits of SWIFT monitors, and as a second contribution of this paper, we show how they can guide instruction selection in a compiler or binary optimizer. So far, when tuning a shared-memory application for performance, the coherence protocol is often overlooked: modern ISAs do provide special instructions to optimize against a given protocol, but there is no mechanical way to know when to use them; in other words, these instructions are mostly used by expert library writers. One example of an instruction making use of the coherence protocol is Alpha's `wh64`; instructions that prefetch a line in exclusive state, as found on Alpha, PA-RISC, IA-64 and probably others, are another example. A third example is IA-64's `ld.bias`, which fetches data and, on a cache miss, requests a private (a.k.a. exclu-

*While with HP Labs.

PPoPP'05, June 15–17, 2005, Chicago, Illinois, USA.

sive) copy of the line. Clearly, inappropriate use of `ld.bias` causes more transactions since shared copies must typically be invalidated before providing the private copy, increasing the latency of the load. These invalidations can also entail higher cache miss rates, further degrading performance. As a consequence, the code generator's decision to select a regular load, a `ld.bias`, or one of the other instructions listed above should depend on the run-time behavior of references to shared data-structures. But today's compilers do not have this information, explaining why these instructions are found predominantly in hand-tuned libraries. SWIFT monitors tackle this problem and allow the compiler to precisely identify which static load instructions are better off being replaced by a `ld.bias`.

This paper is structured as follows. Section 2 describes the coherence protocol we assume and explains where instruction selection comes into play. Section 3 presents hardware extensions that enable SWIFT monitors. Section 4 then shows one application of this capability, specifically, how a compiler can unambiguously identify which loads should be replaced by `ld.bias` instructions. Section 5 describes our experiments, in particular the simulator we used to validate our architecture, and our experimental results. Finally, section 6 discusses related work.

2. CACHE COHERENCE PROTOCOL

We assume a cc-NUMA shared-memory system similar to the HP Superdome multiprocessor [14]. In particular, we assume a directory-based system where the chipset is responsible for converting snoop requests emitted by processors into directory-based requests, and vice-versa. To illustrate the optimizations we have in mind, we pick the Itanium ISA. As discussed earlier, this choice does not restrict our work to this processor family.

From the perspective of a processor, a line can be in one of four states: Modified (a.k.a. dirty), Exclusive (a.k.a. clean), Shared or Invalid. Directory look-ups and updates for a given line are handled by a single node (typically, an element of the chipset) called the line's *home*. From the home's perspective, the line can be in Idle state (indicating the line is Invalid at all processors), Shared state, or Private state. The Private State indicates the line is Modified or Exclusive on a processor. Note that there is no need for home to contrast Modified and Exclusive (*i.e.*, to know whether the line in Private state at the home is dirty or not at the owner processor). We assume dirty sharing is not allowed, that is, that there is at any time at most one processor owning the line in Exclusive or Modified state.

The relevant part of our coherence protocol, which is an enhanced version of the Superdome protocol, is described below. On a read request from processor P1, the initiating transaction sent by P1 is called `READ_SHAR`. Assume that the cache line is currently Idle. From the target address, the CEC attached to P1 knows which CEC is the home (H) to which the transaction must be forwarded. H reads the directory entry for the line, and observes the line is Idle. It can therefore reply to P1 immediately with a `DATA_SHAR` transaction that gives the processor a Shared copy of the line. This is the best possible scenario from the load's standpoint since only two transactions are necessary to service its request.

Now assume the target cache line is initially owned by processor P2. This scenario is illustrated in Figure 1. P1's initial transaction T1 arrives at home H, which sends a `RECALL_PRIV` (T2) to the owner and waits for response T3. This response may be one of three kinds:

- If the line is Exclusive, P2 replies with `RESP_SHAR`, which indicates it will share the line (in read-only mode) with P1.
- If the line is Modified, P2 sends a `RESP_DATA` message to send back the up-to-date line content to H and relinquish ownership.

Upon reception of P2's response, H sends the data to P1 (in a `DATA_SHAR` transaction) and adds P1 to the set of sharers.

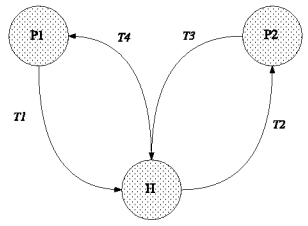

Figure 1: Example of a read request invalidating an exclusive copy

This protocol, however, is not best in all scenarios. To see why, consider the code generated to increment a shared variable: a load is followed by an add and a store to the same target address. With our protocol, the processor doing the increment may need to make two requests: one read-only request to service the load (a `READ_SHAR`), and a second (called `READ_PRIV`) to request a private copy to service the store.

This is where judicious instruction selection helps. The Itanium ISA comes with a special load instruction, `ld.bias`, that requests ownership at the same time as it fetches data. In other words, if the line is in the processor's caches, `ld.bias` behaves as a load; if the line isn't, `ld.bias` issues a `READ_PRIV`, that is, the same request a store that misses generates. The potential benefit of replacing a `ld` followed by a store to the same line by a `ld.bias` is to reduce the total number of transactions and therefore reduce bandwidth consumption. Clearly, if a processor issues a load followed by a store, and assuming the line is not in the cache before the load, two coherence requests are needed. This results in at least 6 transactions: two to service the load and at least four to service the store. In contrast, a `ld.bias` generates a single request and two transactions fewer than in the first scenario.

However, this replacement can negatively impact performance and should be applied with care: on a regular load and if the line is currently Shared, the requesting processor is added to the set of sharers in both versions of the protocol. On a `ld.bias`, however, shared copied are invalidated to honor the "bias" request, and the `ld.bias` can complete only after all copies are invalidated (remember that this only happens if the `ld.bias` missed in the requestor's cache). Performance may therefore suffer both from the invalidation of shared copies and from longer latency to service the `ld.bias` itself.

Our goal is to tell the compiler which loads are soon followed by a store so that it selects the `ld.bias` instruction

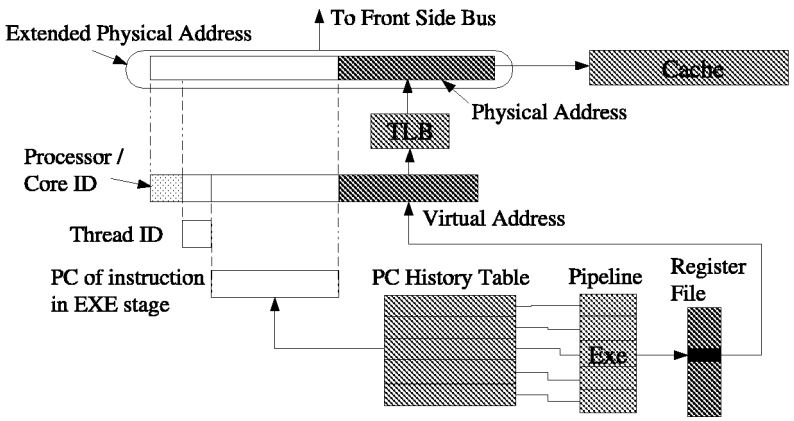

Figure 2: Processor microachitectural extensions to send instruction IDs to the rest of the system (typically, the front-side bus). Our extensions are in light gray. Elements shown in dark gray are already part of most modern microarchitectures.

instead of the regular load. There are three hurdles, however: first, it is not clear what "soon" followed by a store means. Second, even though increments and decrements are cases where `ld.bias` is beneficial, other cases exist. For example, any load followed by a store, by the same processor, to different data in the same line may be a good candidate and probably should be granted ownership. And third, the transformation should be conservative: replacing a load that is never followed by a store to the same line would hurt performance; when in doubt, keeping the regular load is best.

Instead of using heuristics, our approach is systematic. An application is first run on extended hardware implementing SWIFT monitors, described in Section 3. The run creates a log that associates system transactions with the PCs of the instructions that originate them. This information allows the compiler to identify where it should generate `ld.bias` instructions instead of loads during the next compilation of the application.

3. SYSTEM-WIDE INSTRUCTION-LEVEL PERFORMANCE MONITORS

Implementing SWIFT monitors (*i.e.*, native hardware system-wide monitors at the granularity of instructions) requires extensions to both the processor (the sending end) and elements of the chipset (the receiving end) so that instruction identifiers are passed with each monitored transaction. These extensions are described in Sections 3.1 and 3.2, respectively. Please keep in mind that, in this paper, we focus on only one phenomenon related to performance: coherence traffic. However, many other events (*e.g.*, chipset queues filling-up) could be of interest to performance tuning. Finally, we address in Section 3.3 the issue of additional bandwidth consumption.

3.1 Microprocessor Extensions

A simplified view of today's processor microarchitecture is shown by the dark gray elements shown in Figure 2. The core pipeline, address translation (typically through a TLB), the register files and multiple cache levels are well-known. Also, to offer precise exceptions, some structure is needed to save the PCs of all non-retired instructions. This structure is implemented in various different ways, but even for microarchitectures that break instructions into micro-operations, it conceptually boils down to a table of PCs that we call the PC History Table. Executing a load instruction whose target address is contained in a register involves accessing the register file, reading the (virtual) address, and sending that address to the translation unit. Once translated into a physical address, the cache is accessed. (Translation and cache access can in fact overlap, but this is irrelevant to our discussion.) If the cache miss, the address is sent to the outside world. The outside world may be the front-side bus, or on-die circuitry if all the above occurred in a single core.

The parts that belong to our extension are shown in light gray in Figure 2. The goal of these extension is to create an Extended Physical Address (EPA) that uniquely identifies the instruction that originates the request throughout the multiprocessor. The first step is to identify the threads and the processor (and possibly the core) it is running on. As shown in the figure, we assume identifying up to 8 threads is enough. The thread ID thus takes 3 bits, which are under software control. Additional status bits, whose count is implementation-dependent, are also read to provide the processor/core ID; setting these bits is under firmware control.

The PC of the instruction currently in the execute stage of the pipeline is another element of the EPA. Finally, the processor/core ID, the thread ID, the PC and the virtual target address are concatenated to form the extended virtual address. The extended virtual address (and later the EPA) uses the address path that goes to the TLB and to the various cache, with the understanding that the high-order bits that constitute the extension should be ignored by the memory hierarchy before they reach the chipset. The address paths only needs to be widened.

Also of note, the EPA keeps all bits of the physical target address; in contrast, a few low-order bits are usually stripped since only the address of the cache line is passed to a cache, or when a memory read is initiated. Adding these bits do not change the operation of caches or memory systems, which should ignore these bits. These bits are only

used for performance monitoring.

Since these microprocessor extensions are mostly needed during performance tuning, a status bit under software control (not shown in Figure 2) is added to turn this feature on and off. When this status bit is on, transactions sent to the bus contain instruction IDs and the chipset is requested to monitor them in the way described in the following section.

3.2 Chipset Extensions

The shared address space is usually interleaved across the homes. Some address bits, typically the lowest-order ones, determine which memory controller is the line's home. In the proposed architecture, the line's home saves the EPA of each monitored transaction it receives. It saves this information in a buffer that could be on the controller chip, or in main memory. This buffer clearly has finite capacity, and data for a new transaction overwrites that of the oldest one in a circular fashion. It is the responsibility of software to sample this buffer when it sees fit. In this work, software concatenates buffer contents into a *request log*. Having multiple request logs, one per home, is OK since this would still ensure that requests corresponding to a given cache line are in the same log (since they are all directed to the same home memory controller) and that requests are logged in arrival order. As a result, running an application generates a log that contains, for the requests that were sampled, the EPA (originating processor, originating PC, and target address) and the type of the initial transaction. A snippet of a typical request log is shown in Table 1. For expository reasons, the logs shown in this paper also show line addresses following the EPA; this is clearly redundant, since line addresses can be derived from target addresses.

Observe that false and true sharing can easily be detected from these logs. The size of each request is not indicated in logs but is provided by each originating instructions, whose PC are provided.

3.3 Bandwidth Considerations

In the worst case, system-wide monitors require sending at most 73 additional bits with each request message initiated by a processor: 64 bits for the PC, 3 for the thread ID, and 6 for the processor ID assuming a 64-way multiprocessor. (Note that response messages do not need to be tagged with this information.) We believe that this overhead is acceptable: as a point of comparison, coherence messages in Superdome are 16 to 144 *bytes* long. Moreover, many bits on the Itanium 2 front-side bus are typically unused; for instance, out of the train of 128 bits carried at each transaction of the Itanium front-side bus, 23 bits are either unused, reserved for future use, or used for debugging. These bits could be used to carry part of the EPA at no cost to the front-side bus bandwidth.

Another possibility to dramatically reduce the additional payload is to hash the PC so as to leverage the sparsity of instruction addresses. PCs of interest are sparse because the relevant instructions are only those that may be seen by the rest of the system (typically, loads and stores). The structure of PCs can also be exploited by careful hashing: an instruction PC typically begins with a few (typically, 4) highest-order bits that differ depending on whether kernel, shared-library and application code is executing; for most applications, the bits are followed by several zeros before lower-order bits become significant. A possible hashing scheme would cut a 64-bit PC in three segments: the four high-order bits, the 51 middle bits coming next, and the 9 lowest-order bits. The high and low segments are kept unmodified. The middle segment is itself cut in three 17-bit vectors, and these three vectors are XORed bitwise. The 64-bit PC is thus hashed into a 30-bit vector, reducing the EPA overhead to 39 bits down from 73. This hashing scheme is illustrated in Figure 3.

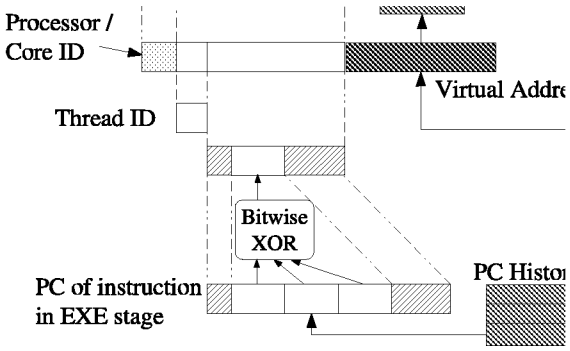

Figure 3: The PC of the instruction can optionnally be hashed. This figure details how the relevant part of Figure 2 is modified.

Software postprocessing is required to reconstruct the complete actual PCs. However, and assuming the hashing scheme described above, this reconstruction is straightforward if code size does not exceed 64 MB: If the first two 17-bit segments are known quantities (for example, they are all zeros), then the hashed PC immediately gives the value of the third 17-bit segment of the actual PC. That and the 9 lowest-order bits provide 26 bits, enough to uniquely encode all PCs without hashing collision up to 64 MB of text.

Let's now assume code size exceeds 64 MB. For a given hashed PC in the request log, software postprocessing has to consider all actual PCs that can be hashed into that value. If a collision occurs and an ambiguity arises between two loads or two stores[1], software may be able to lift the ambiguity by looking at other transactions immediately preceding or following the ambiguous one. In the worst case, minimal modification of the code (such as inserting a no-op) is enough to avoid that collision in a second run.

Finally, please keep in mind that sending out EPAs is only needed in performance tuning, not production runs.

4. COMPILER FEEDBACK

A compiler can leverage the hardware extensions of Section 3 by "mining" request logs, which hardware generated during a first execution, and by adapting its code generation accordingly. This section details one specific example, which allows the compiler to identify which `ld` instructions would benefit from being replaced by a `ld.bias`. Section 4.1 details the mining process, and Section 4.2 specifies how instruction selection is performed.

4.1 Postprocessing of Request Logs

[1]If the PCs of a load and a store are hashed to the same value, the type of the transaction seen by the chipset will disambiguate them.

```
109,1,1,0x4008b80,0xab02380,0xab02380,READ_SHAR
110,3,1,0x4008120,0xab02408,0xab02400,READ_SHAR
111,1,1,0x4008b90,0xab0238c,0xab02380,READ_PRIV
```

Table 1: Fragment of an unsorted request log. Each row corresponds to a request. The first entry is the request number, which is shown only for expository reasons. Then come the ID of the processor placing the request, the thread ID, the instruction PC and the target address, these three entries making up the EPA. The fifth entry is the cache line address, followed by the initiating transaction.

```
109,1,1,0x4008b80,0xab02380,0xab02380,READ_SHAR
111,1,1,0x4008b90,0xab0238c,0xab02380,READ_PRIV
..
110,3,1,0x4008120,0xab02408,0xab02400,READ_SHAR
```

Table 2: Request log of Table 1, after sorting. Transaction 111 is now right after 109. Transaction 110 is now in the portion of the log related to the line at address 0xab02400.

To identify load instructions that are candidate for replacement by a `ld.bias`, the compiler (or any other software) sorts the request log by address and, as the second criterion, by request number. This is illustrated in Table 2: the rows for Requests 109 and 111, that were apart in Table 1, are now consecutive. The compiler then looks for pairs of requests such that:

- The second request immediately follows the first in the sorted request log,
- Both request refer to the same cache line,
- Both requests come from the same processor,
- The first request is a `READ_SHAR` and the second a `READ_PRIV`.

This filter identifies which loads the next compiler run should consider for replacement: these loads are immediately followed by a request for ownership (a store) issued by the same processor. For any such pair of requests, the PC recorded with the `READ_SHAR` request is that of a load that the compiler may want to convert into a `ld.bias`. Looking again at Table 2, the filter identifies Requests 109 and 111 as being such a pair because the processor that issues them is the same (processor 1, second entry), their cache line addresses are equal, and their types match the last filter criterion.

4.2 Code Generation

Given the PC of an instruction, the relevant load instruction can easily be identified and replaced by a `ld.bias`. The target address, predicate, and possibly the post-increment of the original load do not change. We call this optimization LDBIAS.

Note that another approach could be to generate prefetch instructions with request for exclusive access (*i.e.*, using the IA-64 `lfetch.excl` instruction). However, we want in this paper to carefully separate the benefits of prefetching with that of our optimizations.

However, this simple strategy does not work when the load is a control- or data-speculative load (`ld.s` and `ld.a`, resp.), a load check (`ld.c`), a load with acquire semantics (`ld.acq`), or any of their architected combinations. This is because the IA-64 architecture does not provide a "bias" version of these instructions. In such cases, one option would have been to modify other aspects of the compiler, possibly overturning some of its decisions, such as the decision of speculating a load. But again, this would defeat the purpose of evaluating our technique in isolation from other optimizations. Therefore, even though these load flavors are often identified as targets for substitution by a `ld.bias`, we leave them untouched, knowingly leaving some performance on the table.

The situation is more complicated for floating-point loads, because the IA-64 architecture does not provide a "bias" version of these loads. Therefore, when a floating-point load is identified by the request log, we insert a dummy `ld.bias` using the same address register as the floating-point load, right before that floating-point load; it's a dummy load in the sense that the content of its destination register is discarded[2]. We call this optimization FPBIAS.

5. EXPERIMENTS

We first describe our methodology, in Section 5.1, then present in Section 5.2 the experimental results.

5.1 Simulation Infrastructure

We simulated the hardware extensions of Section 3 on HP's reference multiprocessor simulator, which has been validated in numerous internal studies. This system simulator faithfully simulates each chip in a Superdome multiprocessor, including each processor. That is, each processor implementation has its corresponding processor simulator, which is microarchitecture-accurate. The system simulator runs any piece of software the actual hardware can; in particular, it executes unmodified Superdome firmware and boots the various available operating systems. In our simulations, we run HP-UX 11.23 running on top of Itanium 2 processors with 256kB second-level caches and 3MB third-level caches.

The system simulator is a functional simulator and thus does not provide cycle counts. However, it does provide good performance indicators for system-wide events, which are those of interest in this paper, such as counts of transactions generated by each request. This methodology has been successfully used in other papers using functional simulators—see for instance [10].

5.2 Experimental Results

For our experiments, we use the SPLASH-2 suite [16]. For each benchmark, we had to define which input set we use to generate the request log, and which one we use to measure

[2]Note that we cannot insert a dummy `ld.bias` before a `ld.s`, `ld.a`, `ld.c`, or acquire loads instruction (or any floating-point equivalent) since the new instruction would change the program's semantics even if both instructions target the same address.

performance improvement. Typically, the first data set is small and used to train the compiler and the second is a large set representing a more realistic workload. Reusing the SPEC terminology, we'll call them the "train" and "ref" sets, respectively. For each of the benchmarks, the ref data set we use is the default input file or the default values of the parameters (except for the number of processors, which is always four in our experiments). The train sets we selected are given in Table 3, together with a rationale for these choices.

Table 4 reports absolute counts of transactions generated to service all application requests, in three different cases: (A) the unmodified binary run on (simulated) hardware equipped with EPAs and SWIFT monitors; (B) the binary modified by applying LDBIAS (*i.e.*, after integer loads identified by run A are replaced by `ld.bias` instructions); and (C), where both LDBIAS and FPBIAS are applied. Runs B and C don't need SWIFT monitors. As can be seen, application transaction counts are reduced by at least 2.5% and up to almost 10% on real applications. These gains are not due to the EPAs being off in runs B and C since Table 4 reports absolute transaction counts, not bandwidth consumption—these counts are the same with EPAs on.

To appreciate the performance improvement brought by such a traffic reduction, please keep in mind that each transaction corresponds a processor-to-home or home-to-processor communication, and that in large multiprocessors (64-way and up) the latency of each of these communications is in the order of a hundred processor cycles. Latencies as high as those are difficult to hide by the compiler or the processor. Even worse, the latency of coherence traffic on synchronization variables cannot be reduced much by prefetching, because prefetching won't prevent data races. To top it off, synchronizations block the processor from modifying shared data and, in practice, often stall it.

Second, we observe that floating-point benchmarks benefit from our optimization even when only integer loads are replaced. This is because most of these loads access synchronization variables, such as those used to implement barriers, and therefore have a large impact on coherence traffic.

As mentioned earlier, we must keep an eye on other performance indicators to make sure nothing went wrong elsewhere. To that purpose, Table 5 reports absolute counts of transactions needed to service application loads (*i.e.*, a subset of the transaction counted in Table 4). Transactions generated by `ld.bias` instructions are counted in this table. The risk was indeed that too many `ld.bias` instructions invalidate other processors' copies and generate more load requests. As Table 5 shows, load transaction counts are in fact reduced more often than they are increased, and are never increased by more than 1.2%. Another performance indicator is cache misses. Table 6 reports the absolute counts of L3 misses in each case. As can be seen, miss rates are not impacted much.

5.3 Code Analysis

Looking at code snippets sheds light on why the feedback provided by SWIFT monitors guide compilers beyond the obvious case of increments/decrements to shared variables.

Consider the following loop from RADIX, found at lines 518 through 522 in file radix.C. (In the Splash2 suite, files with .C extensions are preprocessed into .c files after expansion of platform-dependent macros.)

```
for (i=key_start;i<key_stop;i++) {
  my_key = key[0][i] & bb;          // Line 519
  my_key = my_key >> shiftnum;
  rank_me_mynum[my_key]++;
}
```

Some instances of the load at line 519 are identified by SWIFT monitors as followed by a store to the same cache line, in line 641:

```
for (i = key_start; i < key_stop; i++) {
  this_key = key[0][i] & bb;
  this_key = this_key >> shiftnum;
  tmp = rank_ff_mynum[this_key];
  key[1][tmp] = key[0][i];          // Line 641
  rank_ff_mynum[this_key]++;
}
```

To show the impact our method can have, we replaced a single load, that of `key[0][i]` at line 519, by a `ld.bias` and ran that binary. We observed 103,550 total application transactions (a 1.8% improvement over the unmodified application) and 8,928 load transactions. Replacing this single static load therefore accounts for most of the gains we got. We believe that static compiler analyses would have a hard time identifying the interaction between the load at Line 519 and the store at line 641. In contrast, the feedback provided by SWIFT monitors makes it straightforward.

We also wanted to know if run-time feedback was really necessary to identify loads that should be turned into `ld.bias` instructions. To that purpose, we evaluated a purely static approach: whenever a field of RADIX's `global` data structure was syntactically incremented or decremented using the `++` or `--` operator, we replaced the corresponding load by a `ld.bias`. We didn't change any other `ld` instruction. The result was insignificant improvements in either transaction counts or counts of transactions servicing loads. This indicates that, in RADIX at least, statically obvious increments/decrements account for relatively few `READ_SHAR-READ_PRIV` sequences and thus offer little opportunity for improvement.

6. RELATED WORK

Some performance monitors are provided in Superdome and other platforms, such as the Sun Fire interconnect [12], but the information they provide is not at the granularity of individual instructions. As noted earlier, the goal of this work is to extend these multiprocessor performance monitors with features closer to that of Event Address Registers (EARs) found on Itanium. EARs can guide the compilation of sequential code [2, 5], and even though these papers tackle performance issues that are totally different from ours, they all rely on performance monitors at the granularity of instructions.

To improve the performance of directory-based protocols, most of the literature focuses on destination-set prediction [1, 4, 9]. We clearly follow a totally different approach that has, to the best of our knowledge, never been explored. We expect the LDBIAS and FPBIAS optimizations are compatible with destination set prediction.

In the Check-In Check-Out (CICO) programming model [3], programmer-supplied annotations tell the protocol when exclusive or share access to a line should be expected (Check-Out), or when a line can be relinquished (Check-In). Check-Out for exclusive access is similar to a `ld.bias`; however,

Benchmark	Ref Set	Train Set	Comments
FFT	-m10	-m4	P=4 processors and M must be such that $2^{M/2} \geq P$.
LU	-n128 -b16	-n16 -b16	16 is smallest possible value for N given B.
CHOLESKY	tk29.O	tk23.O	tk23.O is smallest provided input file except wr10.O, but wr10.0 gave an "Overflow" message.
RADIX	-n262144	-n16384	
BARNES	16384 particles	1024 particles	
RAYTRACE	balls4.geo	teapot.geo	teapot smallest provided geometry file.
WATER-SPATIAL	512 molecules	64 molecules	Fewer molecules would require reducing cutoff value of 6.2Å.
VOLREND	head.den	head-scaleddown4.den	

Table 3: Ref and train input sets for the different benchmarks

Benchmark	Unmodified Binary	LDBIAS	Improvement	LDBIAS + FPBIAS	Improvement
FFT	4,108	3,936	4.2%	3,821	7.0%
LU	196,530	192,622	2.0%	188,732	4.0%
CHOLESKY	385,186	365,010	5.2%	360,475	6.4%
RADIX	105,468	102,256	3.0%	102,256	3.0%
BARNES	1,971,764	1,944,422	1.4%	1,889,776	4.2%
RAYTRACE	847,122	768,800	9.2%	768,800	9.2%
WATER-SPATIAL	128,766	125,818	2.3%	125,502	2.5%
VOLREND	543,110	489,666	9.8%	489,666	9.8%

Table 4: Counts of application transactions. No opportunity for FPBIAS were identified in RADIX, RAYTRACE and VOLREND.

Benchmark	Unmodified Binary	LDBIAS	Variation	LDBIAS + FPBIAS	Variation
FFT	1,506	1,468	2.5%	1,462	2.9%
LU	86,056	85,630	0.5%	85,431	0.7%
CHOLESKY	105,839	104,361	1.3%	105,003	0.8%
RADIX	9,070	8,910	1.7%	8,910	1.7%
BARNES	942,122	932,842	0.9%	930,506	1.2%
RAYTRACE	480,134	486,350	-1.2%	486,350	-1.2%
WATER-SPATIAL	70,026	70,740	-1.0%	70,756	-1.0%
VOLREND	473,712	478,878	-1.0%	478,878	-1.0%

Table 5: Counts of application transactions servicing loads. Positive variations are improvements.

Benchmark	Unmodified Binary	LDBIAS	Variation	LDBIAS + FPBIAS	Variation
FFT	943	962	-2.0%	943	0.0%
LU	45,400	45,129	0.6%	45,334	0.1%
CHOLESKY	56,912	57,384	-0.8%	56,998	-0.1%
RADIX	25,676	25,442	0.9%	25,442	0.9%
BARNES	473,565	468,296	1.1%	474,401	-0.2%
RAYTRACE	318,620	313,699	1.5%	313,699	1.5%
WATER-SPATIAL	36,111	35,450	1.8%	35,605	1.4%
VOLREND	224,517	222,057	1.0%	222,057	1.0%

Table 6: L3 miss counts. Each L3 miss results in two or more transactions. Positive variations are improvements.

CICO requires user annotations, whereas our method is entirely automatic. Finding beneficial locations for a Check-Out may be straight-forward in simple cases (such as increments of shared data structures) but hard when, for instance, false sharing occurs.

The Write-Through primitive of [6] is similar to Check-Out. Moreover, even though [6] does not perform optimizations similar to using `ld.bias`, it does rely on a compiler to insert Write-Through's. However, their paper relies on static analysis of references (typically, affine array subscripts), whereas our method relies on more precise run-time feedback.

In [10], McCurdy and C. Fischer show how new types of load instruction (called `ldp` and `ld+`) together with a modified MSI protocol significantly reduce coherence traffic. Our use of `ld.bias` is similar in spirit, but we showed it can be automated using SWIFT monitors.

Closer to our work is a recent paper by Nagarajan et al. [11]. In that work, a system simulator is used to collect run-time information and identify performance issues. Using a simulator provides at least as much information as SWIFT monitors do, at the cost of lower accuracy and much slower speed.

Finally, let us note that system-wide performance monitoring has also been recently investigated in [13, 15], although their work does not correlate events to instructions.

7. CONCLUSION

We introduced SWIFT monitors, which capture performance events in a shared-memory system and associate them with unique identifiers of the instructions that generate them. In this paper, we limited ourselves to SWIFT monitors reporting coherence traffic, but many other performance-impacting events could be monitored the same way. Because the granularity of SWIFT monitors is that of individual instructions, performance debugging is eased tremendously; this granularity also enables run-time information to be fed back to compilers, and we introduced a new optimization based on such a feed-back. This compiler optimization leverages SWIFT monitors to decide, at each individual memory reference, which version of the load instruction makes the best use of the coherence protocol. Experiments on the HP reference simulator showed that this optimization reduces coherence traffic by up to almost 10% on real applications, the smallest observed improvement being 2.5%.

SWIFT monitors do require extending processor and chipset microarchitectures and the front-side bus protocol. However, each party (microprocessor or chipset) is free to honor this extension or not. For example, an entry-level processor may not offer this feature, while a server-class processor might. Symmetrically, a chipset manufacturer could differentiate its different chipsets partly based on the extent of their support for SWIFT monitors. An alternative solution could be to use a software simulator to gain similar information, but this approach may not be palatable in an industrial environment. In contrast, for middle- and high-end servers, we believe hardware SWIFT monitors would tremendously simply the fine-tuning of critical, complex applications.

8. REFERENCES

[1] M. E. Acacio et al. Owner prediction for accelerating cache-to-cache transfer misses in a cc-NUMA architecture. In *Proc. of Supercomputing SC02*, pages 1–12, 2002.

[2] Y. Choi, A. Knies, G. Vedaraman, and J. Williamson. Design and experience: Using the Intel Itanium2 processor performance monitoring unit to implement feedback optimizations. In *EPIC2 Workshop*, 2002.

[3] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative shared memory: Software and hardware for scalable multiprocessors. *ACM Trans. on Comp. Sys.*, 11(4):300–318, Nov. 1993.

[4] S. Kaxiras and C. Young. Coherence communication prediction in shared-memory multiprocessors. In *Proc. 6th Int'l Symp on High-Performance Architecture*, pages 156–167, 2000.

[5] D. Kim et al. Physical experimentation with prefetching helper threads on Intel's hyper-threaded processors. In *Proc. 2nd Symp. on Code Gen. and Optim (CGO)*, pages 27–38, Palo Alto, CA, Mar. 2004.

[6] D. Koufaty and J. Torrellas. Compiler support for data forwarding in scalable shared-memory multiprocessors. In *Intl. Conf. on Parallel Proc.*, 1999.

[7] K. London et al. End-user tools for application performance analysis using hardware counters. In *Intl. Conf. on Parallel and Distributed Computing Systems*, Aug. 2001.

[8] C. Luk et al. Ispike: A post-link optimizer for the Intel Itanium2 architecture. In *Proc. 2nd Intl. Symp. on Code Generation and Optimization (CGO)*, pages 15–26, Palo Alto, CA, Mar. 2004.

[9] M. M. K. Martin et al. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors. In *Proc. 30th Intl. Symp. on Computer Arch. (ISCA)*, pages 206–217, 2003.

[10] C. McCurdy and C. Fischer. User-controllable coherence for high performance shared memory multiprocessors. In *Proc. of Symp. on Prin. and Practice of Parallel Prog. (PPoPP)*, pages 73–83, San Diego, June 2003.

[11] A. Nagarajan, J. Marathe, and F. Mueller. Detailed cache coherence characterization for OpenMP benchmarks. In *Proc. Intl. Conf. on Supercomputing (ICS)*, pages 287–297, Saint-Malo, France, June 2004.

[12] L. Noordergraaf and R. Zak. SMP system interconnect instrumentation for performance analysis. In *Proc. of Supercomputing, SC-2002*, Baltimore, Maryland, Nov. 2002.

[13] M. S. others. Owl: Next generation system monitoring. In *Proc. of Computing Frontiers 2005*, Ischia, Italy, May 2005.

[14] M. E. Shaw. Superdome. Interex Enterprise Solutions. http://www.interex.org/pubcontent/enterprise/sep01/frame_usr.html, Sept. 2001.

[15] T. Suh et al. Evaluating system-wide monitoring capsule design using Xilinx Virtex-II Pro FPGA. In *Workshop on Arch. Res. using FPGA Platforms, in conj. with HPCA05*, San Francisco, CA, Feb. 2005.

[16] S. C. Woo et al. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. 22nd Intl, Symp. on Computer Arch.*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.