Relative Performance of Hardware and Software-Only Directory Protocols Under Latency Tolerating and Reducing Techniques

Håkan Grahn
Department of Computer Science
University of Karlskrona/Ronneby
Soft Center, S-372 25 Ronneby, Sweden
Hakan.Grahn@ide.hk-r.se

Per Stenström
Department of Computer Engineering
Chalmers University of Technology
S-412 96 Göteborg, Sweden
pers@ce.chalmers.se

Abstract

In both hardware-only and software-only directory protocols the performance is often limited by memory access stall times. To increase the performance, several latency tolerating and reducing techniques have been proposed and shown effective for hardware-only directory protocols. For software-only directory protocols, the efficiency of a technique depends not only on how effective it is as seen by the local processor, but also on how it impacts the software handler execution overhead in the node where a memory block is allocated.

Based on architectural simulations and case studies of three techniques, we find that prefetching can degrade the performance of software-only directory protocols due to useless prefetches. A relaxed memory consistency model hides all write latency for software-only directory protocols, but the software handler overhead is virtually unaffected and now constitutes a larger portion of the execution time. Overall, latency tolerating techniques for software-only directory protocols must be chosen with more care than for hardware-only directory protocols.

1. Introduction

Private caches and a directory-based cache coherence protocol implemented in hardware, also called a *hardware-only directory protocol*, constitute an important approach to achieve high performance in shared-memory multiprocessors. During the last few years, several approaches have been proposed in order to reduce the hardware complexity and/or increase the protocol flexibility by migrating parts of the coherence protocol to software [1, 4, 7, 12, 13, 15]. In *software-only directory protocols* [4], the directory management is migrated from a hard-wired memory controller to software handlers executed on the compute processor. As a result, the hardware complexity can be reduced at the expense of lower performance; between 60% and 86% of the hardware-only protocol performance is reported in [7].

In both hardware-only and software-only directory protocols, performance is often limited by processor stall times resulting from *memory access latencies*. To reduce these stall times, and thus increase the performance, several *latency tolerating and reducing techniques* have been proposed and evaluated in the context of hardware-only directory protocols [6, 11, 14, 17]. In software-only directory protocols, the invocation of software handlers in can also prolong the execution time in two ways. First, the handler latency may end up on the memory access path and thus increase the memory latency seen by the requesting processor. In [7], we proposed strategies to remove or hide the handler latency from the memory access path for read misses. Second, the handler latency also burdens the compute processor in the node where a memory block is allocated. The latter protocol execution overhead, referred to as *p-time*, was found to have very limited possibilities to be overlapped by other stall times [7]. Therefore, p-time has a fundamental, and possibly large, impact on the effectiveness of latency tolerating and reducing techniques in software-only directory protocols. Such techniques can be divided into three classes depending on whether p-time *increases*, *decreases*, or is *invariant* when the technique is used.

In this paper we consider three latency tolerating and reducing techniques and compare the performance of hardware-only and software-only directory protocols under each of these techniques. We have chosen *prefetching* [5, 14], *migratory optimization* [17], a technique that dynamically detects migratory data blocks and optimizes their coherence protocol actions, and *release consistency* [6] as example techniques that increase, reduce, and do not affect p-time, respectively.

To evaluate the performance between hardware-only and software-only directory protocols we use architectural simulations and four applications from the SPLASH benchmarks [16]. We find that prefetching often *degrades* the performance of software-only directory protocols due to a large p-time overhead. By contrast, prefetching increases the performance for hardware-only directory protocols. Further, we find that migratory optimization has a potential to narrow the performance gap between software-only and hardware-only directory protocols. Finally, release consistency successfully hides the write latency and increases the performance of software-only directory protocols for all our applications, but the performance gap between software-only and hardware-only directory protocols usually increases.

We begin in Section 2 to describe the simulated hardware-only and software-only architecture organizations and their expected performance followed by our experimental assumptions in Section 3. In Sections 4 to 6 we present the experimental results regarding the performance of the three techniques described earlier. Finally, we conclude the study in Section 8.

2. Simulated protocols and their performance

2.1. HW-only and SW-only directory protocols

The architectural framework is a sequentially consistent cache-coherent NUMA architecture where a number of processor nodes are connected by a network. Each node consists of a

processor with its cache hierarchy, a memory module, a local bus, and a network interface connecting the processor node to the network as shown in Figure 1.

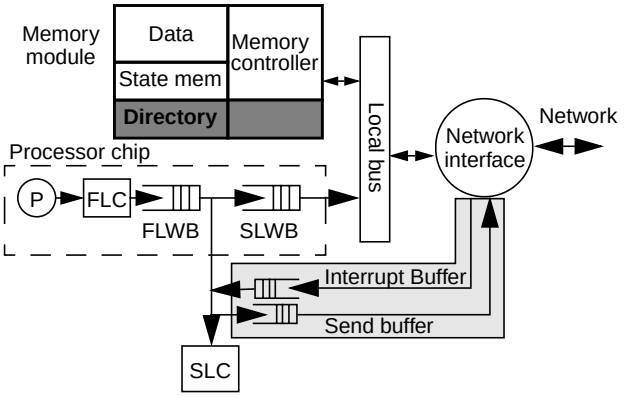

Figure 1. The organization of a processor node. The striped area only applies to HW-only directory protocols and the shaded area only applies to SW-only directory protocols.

Both the hardware-only and the software-only directory systems employ a write-invalidate protocol with a full-map directory [3]. A processor read that misses in the second-level cache (*SLC*) initiates a read miss request to the *home* node, i.e, the node where the memory block is mapped. If the memory copy is clean, home responds with a block copy to the *local* requesting node and updates the directory. Otherwise, if another cache, denoted *remote*, has an exclusive copy, home issues a write-back request to remote; remote updates home; and, home forwards a copy to local, updates the directory, and the block ends up clean in home.

A processor write to a non-exclusive block in the cache, results in an ownership request sent to home. Home inspects the directory and sends explicit invalidations to the other caches with a block copy. Each cache receiving an invalidation, sends an acknowledgment to home. When home has collected all acknowledgments, it grants ownership to local and the block ends up as dirty in home. During the time write-back requests and invalidations are pending, the block is in a transient state 'busy' and read and write requests to the block have to be retried.

In the hardware-only directory architecture, the memory protocol engine consists of three parts implemented in hardware: a memory controller, a directory, and a state memory. The controller processes incoming coherence requests, takes correct actions depending on the state of the memory block, and also manages the directory. By contrast, in a software-only directory protocol, the directory management is migrated to software handlers executed on the compute processor. These handlers process coherence requests and manage the directory which now is stored in main memory. To trigger the execution of a handler, the software-only directory architecture has an interrupt buffer in which coherence requests from the network are posted. If a handler triggers new coherence requests, e.g., write-back requests, it posts them in a send buffer.

The lower implementation cost of the software-only directory system has a performance cost. Upon each coherence request, the processor is interrupted and executes a software handler, thus interfering with the application execution which can prolong the total execution time. We showed in [7] that the handler latency can be removed from the critical memory access path of read requests by supporting request forwarding from home to remote for read miss requests to dirty blocks and also supporting data block transfers to local in parallel with the processor interrupt for directory updates. Finally, read misses to clean blocks allocated in the local node do not interrupt the processor.

2.2. A model for the relative execution time

To build intuition into the relative performance between hardware-only and software-only directory systems, we use a simple model of where the execution time is spent in a parallel application. In the left bar of Figure 2, the execution time of an application on a hardware-only system is broken down into the fraction of busy cycles, B_{hw}, and three stall time components: read (R_{hw}), write (W_{hw}), and synchronization stall (S_{hw}). R_{hw} is the time the processors are stalled due to read cache misses, and W_{hw} is the time the processors are waiting for ownership requests to complete. Finally, S_{hw} is the time the processors wait for, e.g., locks and barriers. The total execution time under a hardware-only directory protocol is $E_{hw} = B_{hw}+R_{hw}+W_{hw}+S_{hw}$.

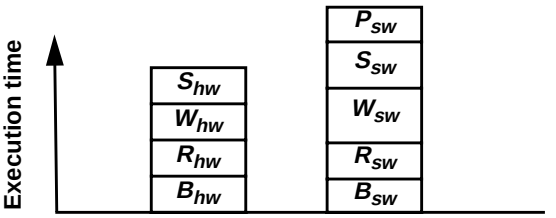

Figure 2. Execution time breakdown for HW-only (left) and SW-only (right) directory protocols.

For the software-only directory protocol, the execution time consists of the same components. In addition, the protocol execution overhead (P_{sw}) is present as shown in the right bar of Figure 2. P_{sw} arises when coherence requests to home interrupt the compute processor in home. When a software handler is invoked, the execution of it may only be partly, or in the worst case not at all, overlapped with other stall times. In this case, the handler execution prolongs the total execution time of the application. The sum of handler execution times that are not overlapped with other stall times is referred to as *p-time* and denoted P_{sw}. In [7], we found limited opportunities to overlap protocol execution with other stall times. The total execution time under a software-only directory protocol is $E_{sw} = B_{sw}+R_{sw}+W_{sw}+S_{sw}+P_{sw}$.

Throughout this paper, we use the *execution time ratio*, *ETR*, between hardware-only and software-only directory protocols as our primary measure of the relative performance to judge whether a certain latency tolerating and reducing technique will increase or reduce this ratio. We define *ETR* as E_{sw} / E_{hw}.

Comparing the components in the two systems, we first note that $B_{hw} = B_{sw}$ for all applications (all our benchmarks use static work distribution). Second, because the software-only architecture we use in this study employs the strategies proposed in [7], the read stall times are the same, i.e., $R_{hw} = R_{sw}$. The write and synchronization stall times, however, can be significantly higher in a software-only than in a hardware-only directory protocol [7], i.e., $W_{hw} < W_{sw}$ and $S_{hw} < S_{sw}$. Their impact together with P_{sw}

results in an *ETR*>1 for the baseline systems. *ETR*s between 1.16 and 1.68 were reported in [7]. The central question explored in this paper is whether the *ETR* can be reduced for three classes of latency tolerating and reducing techniques.

3. Simulation methodology

The simulation models are built on top of the CacheMire Test Bench [2], a simulation framework and programming environment. The framework consists of multiple SPARC processors simulated at the instruction level and an architectural simulator of the multiprocessor model. The processors issue memory references to the architectural simulator which delays the processors according to its timing model. Thus, the same interleaving as in the target system is obtained. Instruction and private data references are not fed into the architectural simulator since we assume they hit in cache and are carried out in a single cycle.

We simulate a multiprocessor with 16 nodes and the processor node organization is shown in Figure 1. If not stated otherwise, the same assumptions apply to both the hardware-only and the software-only directory systems. The two-level cache hierarchy consists of a 2 Kbytes on-chip write-through first-level cache (*FLC*) and a 64 Kbytes off-chip copy-back second-level cache (*SLC*). Both caches are direct-mapped with 64 bytes blocks, full inclusion is maintained, and the processor manages its own caches, i.e., the *SLC* controller is on-chip. A first-level write buffer (*FLWB*) with 16 entries connects the *FLC* and the *SLC* and allows processor writes to complete in a single cycle. The *SLC* is lockup-free and supports multiple pending requests which is essential under, e.g., release consistency and prefetching. A 16 entries large second-level write buffer (*SLWB*) buffers pending requests. The page size in the system is 4 Kbytes and the pages are allocated to the memory modules in a round-robin fashion. Moreover, synchronization operations, i.e., acquires and releases, are supported by a queue-based mechanism.

The network interface routes messages between the node and the network and is connected to the processor and the memory module through a 128 bits wide split transaction bus. The network interface also collects invalidation acknowledgments and notifies the memory-protocol engine when the last acknowledgment has arrived. The interrupt and send buffers are interfaced to the *SLC* bus, are accessible through memory mapped addresses, and have the same access time as the *SLC*. Directory entries are cached as all other data when the software handlers access them.

We assume that the SPARC processors and their FLCs are clocked at 100 MHz, i.e., 1 pclock = 10 ns. The SLC interface is 128 bits wide and the block access time is 60 ns. The memory module also has an 128 bits wide interface but with a 120 ns access time for a whole block. Both the local bus and the network interface are assumed to run at 100 MHz. The interconnection network has an infinite bandwidth and a constant latency of 300 ns, which approximately corresponds to the latency in a 50 MHz mesh network with 32-bit flits. Contention is correctly modeled within the processor node and the network interface.

The default software handler execution time is 50 pclocks excluding memory and buffer access times. To the default execution time, we add 6 pclocks for each message the handler sends, 13 pclocks for each read or write of the state of a memory block, and finally, 1 pclock or 16 pclocks for each directory access depending on whether a directory entry is cached or not. All handler functionality are simulated; only the timing is simplified.

In our experimental evaluation we use three parallel applications taken from the SPLASH suite [16] (Water, Ocean, and MP3D) and one that has been provided to us by Stanford University (LU). The applications are written in C using the ANL macros and compiled with `gcc` version 2.1 (optimization level `-O2`). Statistics are gathered in the parallel section of the applications to avoid initialization effects.

All applications are from the scientific and engineering domain. Water and MP3D are two applications with a high degree of migratory sharing, while producer-consumer sharing dominates in LU and Ocean. Water was run with 288 molecules for 4 time steps. LU used a 200x200 matrix and Ocean used a 128x128 grid with the tolerance factor set to 10^{-7}. Finally, MP3D was run with 10,000 particles for 10 time steps.

4. Performance of prefetching

4.1. Qualitative evaluation

The first latency tolerating technique we will study is prefetching, which appears in the literature both as software-controlled [14] and hardware-based techniques [5]. In this study we only consider non-binding, read-shared prefetching, i.e., the block is fetched in a shared mode.

The goal of prefetching is to bring data into the cache in advance so the processor encounters a cache hit instead of a miss, a so called *useful* prefetch. The data is still visible to the coherence protocol, and may be evicted from the cache due to, e.g., an invalidation, before the processor accesses it. If so, the processor still encounters a miss and the prefetch was *useless*. In addition, the prefetch scheme may also fetch blocks that the processor never accesses. All useless prefetches both cause unnecessary network traffic and increase the occupancy of the memory protocol engine in home. This is usually no problem in a hardware-only directory protocol since each prefetch occupies the controller only a short amount of time. By contrast, in a software-only directory protocol this occupancy is directly translated into protocol execution overhead, i.e., P_{sw} increases. We will refer to the ratio between the number of useful prefetches and the total number of prefetches as the *prefetch efficiency*, and to the fraction of read misses that is removed by useful prefetches as the *coverage*.

When prefetching is applied to both hardware-only and software-only directory protocols, we are interested in whether the *ETR* will increase or not. Prefetching reduces the read stall time; ideally, the read stall time is equally reduced in both protocols, i.e., $R'_{hw} = R'_{sw} < R_{hw} = R_{sw}$. The write and synchronization stall times may increase slightly as a result of contention. The big difference between hardware-only and software-only protocols is P_{sw}, which *increases* as a result of useless prefetches. The important question is whether the read stall time reduction is large compared to the increase in P_{sw} or not.

Since useless prefetches are present to various degree in all prefetching schemes, we do not consider any specific scheme in this study. Instead we assume a scheme with a fix coverage and a fix prefetch efficiency. At the time a processor encounters a potential cache miss, we determine whether it should have been

covered by a useful prefetch or it results in a cache miss. If the miss is determined to be covered, a number of useless prefetches is generated depending on the prefetch efficiency. A potential drawback is as follows. If the prefetch efficiency is very low, i.e., below 10%, many useless prefetches are issued simultaneously, which may result in a clustering effect not present in a real scenario. However, since prefetch studies have reported higher efficiency numbers [5, 14], we believe that our approach is feasible. By varying the coverage and the prefetch efficiency in our simulations, we cover the behavior of a large portion of the prefetching schemes proposed in the literature.

4.2. Quantitative evaluation

In our evaluation of the relative performance between a hardware-only and a software-only directory protocol when prefetching is applied, we start with a default coverage and prefetch efficiency based on findings in other studies and then do a variation analysis. In [5], Dahlgren *et al.* reported coverage numbers between 2% and 80%, and prefetch efficiencies between 13% and 92% for hardware-based stride and sequential prefetching. For software prefetching, Mowry [14] presented coverage numbers between 75% and 98%, while the prefetch efficiency varied between 11% and 85%. As default numbers in our study, we have chosen a coverage of 50% and a prefetch efficiency of 25%, representing a conservative scheme.

The execution times of the four applications are shown in Figure 3. For each application, four bars are shown. The two left bars correspond to the execution time for a hardware-only directory protocol without (HW) and with (HW-P) prefetching, and the two right bars correspond to a software-only directory protocol without (SW) and with (SW-P) prefetching. Each bar is decomposed as suggested in Figure 2: the busy (B_x), the read stall (R_x), the write stall (W_x), and the synchronization stall times (S_x), where x is either *hw* or *sw*. For software-only directory protocols, the protocol execution overhead (P_{sw}) is shown at the top.

By comparing the relative execution times of SW and HW, we find that the execution time ratios (*ETR*s) between SW and HW are between 1.16 (Water) and 1.76 (MP3D), which is in accordance with the results presented in [7].

Prefetching aims at reducing the read stall times, thus obtaining a shorter execution time. As we see in Figure 3, the execution times is lower under HW-P than under HW for all applications. Simulation results show that prefetching reduces R_{hw} with 44%, 35%, 37%, and 50% for Water, LU, Ocean, and MP3D, respectively. By comparing R_{sw} under SW and SW-P, we find that prefetching reduces R_{sw} with 42%, 24%, 23%, and 47% for Water, LU, Ocean, and MP3D, respectively. Unfortunately, we observe a significant increase of P_{sw} under SW-P compared to SW; P_{sw} has increased by between 18% (Water) and 47% (Ocean). This higher protocol execution overhead is expected and stems from a higher number of coherence requests to home. Our simulation results show that SW-P generates between 16% (Water) and 46% (Ocean) more handler invocations than SW.

An examination of W_{sw} and S_{sw} gives that both increase by up to 15% (Ocean) and 21% (Ocean), respectively, under SW-P compared to SW. This is a result of longer queuing delays in the home node. The average number of messages in the interrupt

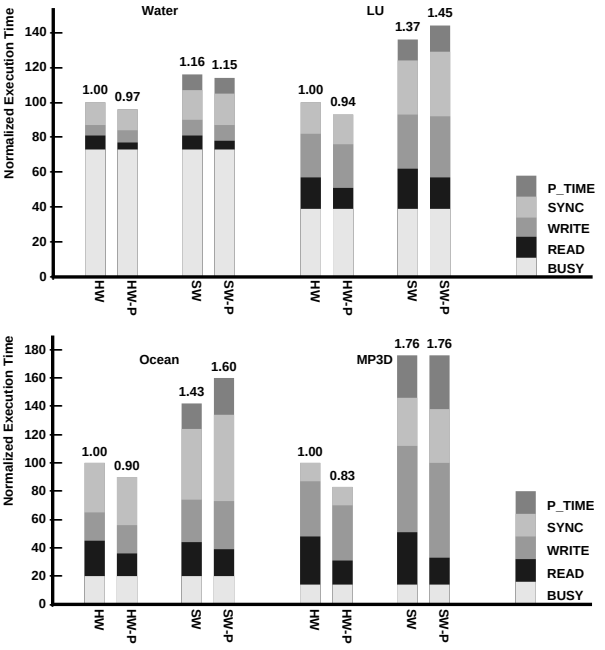

Figure 3. Normalized execution times for HW-only and SW-only protocols without (HW and SW) and with prefetching (HW-P and SW-P).

buffer increases under SW-P, e.g, for LU the average number of messages increases from three to almost five under SW-P.

By adding all the execution time components under SW and SW-P, we conclude that the total execution time, E_{sw}, has increased for two applications (LU, +6% and Ocean, +12%), decreased for one application (Water, -1%), and is unchanged for one application (MP3D) when prefetching is applied. In total, the *ETR*s between SW-P and HW-P are higher than between SW and HW for all applications but one. The *ETR*s are 1.19, 1.54, 1.78, and 1.76 for Water, LU, Ocean, and MP3D, respectively.

One reason why prefetching incurs so much protocol execution overhead is the low prefetch efficiency. Therefore, a high prefetch efficiency is more important in software-only than in hardware-only directory protocols. An interesting question is then how high coverage and prefetch efficiency are necessary for prefetching to be effective, i.e, when does the reduction of R_{sw} outweigh the increase of P_{sw}. We have simulated coverage values between 25% and 90%, and prefetch efficiencies between 25% and 90%. All results are presented in [9], but due to space limitations, we only summarize them here.

Our results indicate that prefetching reduces R_{sw} for all coverage and prefetch efficiency numbers, although the reduction is quite small when the coverage is only 25%. Unfortunately, even though prefetching reduces R_{sw}, the performance gain is many times outweighed, or at least greatly reduced, by longer W_{sw} and S_{sw} times and higher protocol execution overhead. We have also observed some trends when the coverage and the prefetch efficiency varies. First, we have seen that a prefetch efficiency below or equal to 25% results in longer or equal execution times for all applications compared to without prefetching. By contrast, for hardware-only protocols we have seen execution time reductions for all application even with only 25% prefetch efficiency and

25% coverage. Second, we have seen that if the coverage is low, i.e., less or equal to 25%, a high prefetch efficiency is not that essential, as long as it is at least 25%. For a prefetch efficiency of 25% the total execution time is only slightly worse than without prefetching. As the coverage increases, a high prefetch efficiency becomes more important. For Ocean, which has the worst behavior, the prefetch efficiency needs to be 90% when the coverage is 90% in order to reduce the execution time compared to SW.

In summary, even though a prefetching scheme is efficient when applied under a hardware-only directory protocol, it might increase the execution time of a software-only directory protocol. Since software-only directory protocols are more sensitive to useless prefetches than hardware-only directory protocol, a low prefetch efficiency can have a devastating effect on the performance, especially if the coverage is high.

5. Performance of migratory optimization

5.1. Qualitative evaluation

In the previous section we studied prefetching and found that the main obstacle of prefetching in the context of software-only directory protocols was the increased p-time. Therefore, in this section we will evaluate a technique that *decreases* p-time.

Migratory sharing [10] is a program behavior not uncommon in parallel applications, e.g., data accessed in critical sections exhibits migratory sharing. First, one processor reads a data block and then modifies the block, i.e., it obtains exclusive ownership of the block. Then, another processor reads and modifies the block in the same way, and thus, the block *migrates* around among the processors. Migratory blocks are referenced by only one processor at a time but by many processors in the long run.

In a system with a write-invalidate protocol each 'migration' of the block between two processors incurs two global actions; first a read miss request and then an ownership request. In a sequential consistent system, both these actions stall the processor resulting in both read and write stall times. In [17], Stenström *et al.* proposed a solution to detect migratory blocks and optimize their coherence actions. This *migratory optimization technique* dynamically detects migratory blocks at the home node, which sees all read miss and ownership requests, by recognizing two subsequent read-write sequences by two different processors. The coherence protocol then handles migratory blocks with a single read-exclusive request instead of one read miss and one ownership request, thus avoiding the write stall.

The performance gain from the migratory optimization technique is the removal of global ownership requests for migratory blocks. As a result, W_{hw} and W_{sw} are reduced. Since the number of ownership requests are reduced, P_{sw} is also reduced, resulting in an additional gain for software-only protocols. The question is whether this reduction can actually make software-only directory protocols perform as well as hardware-only directory protocols.

5.2. Quantitative evaluation

In Figure 4, we present the resulting execution times when migratory optimization is applied to hardware-only and software-only directory protocols, referred to as HW-M and SW-M, respectively. We first conclude that the migratory optimization reduces the total execution times for all applications under both hardware-only and software-only directory protocols.

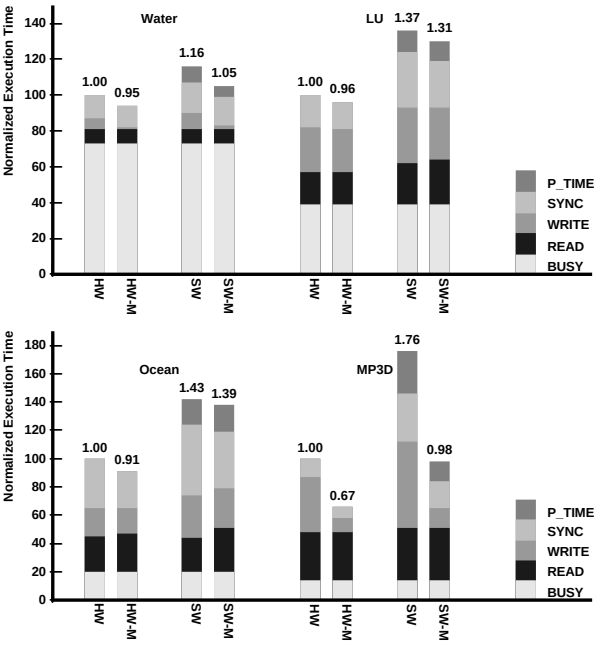

Figure 4. Normalized execution times for HW-only and SW-only protocols without (HW and SW) and with (HW-M and SW-M) migratory optimization.

By looking at the write stall times in Figure 4, we see that both W_{hw} and W_{sw} are significantly reduced for Water and MP3D under HW-M and SW-M, respectively. This is expected and in accordance with the results presented in [17]. Our results show that W_{hw} is reduced by 73% and 75% for Water and MP3D, respectively, while the corresponding numbers for W_{sw} are 78% for both Water and MP3D. W_{sw} is reduced relatively more than W_{hw} as a result of shorter queuing delays in home; the lower number of coherence requests reduces the average interrupt buffer size. For LU and Ocean, only small decreases in W_{hw} and W_{sw} are observed. This corresponds well to the application behavior; in Water and MP3D migratory sharing dominates, while producer-consumer sharing dominates in LU and Ocean.

We find that S_{hw} and S_{sw} also are reduced when the migratory optimization is applied. This effect stems mainly from more efficient barrier synchronizations. In the barrier, a counter variable keeps track of the current number of processors waiting at the barrier. This counter exhibits migratory sharing. For Ocean, S_{sw} is reduced relatively less than S_{hw}, 19% and 26%, respectively, when the migratory optimization is applied. In [7], S_{sw} was found to be larger than S_{hw} due to imbalance in the number of coherence interrupts; some processors encounter up to ten times as many interrupts as other processors. Even though the barrier itself is more efficient with the migratory optimization, the inherent larger overhead due to the interrupt imbalance in software-only directory protocols are not affected.

P_{sw} is reduced by 35% and 56% for Water and MP3D, respectively. This reduction stems from significantly fewer coherence interrupts as we predicted in Section 5.1. For Water, the number

of interrupts has decreased by 34% and for MP3D by 52%. For LU and Ocean, P_{sw} is virtually unaffected.

Finally, both R_{hw} and R_{sw} are virtually unaffected for Water, LU, and MP3D when the migratory optimization is applied. For Ocean, however, we observe that R_{hw} and R_{sw} have increased by 11% and 27%, respectively. Ocean has a non-negligible amount of false sharing, which causes many blocks to be deemed as migratory even though they are not. As a result, the miss rate increases with 12% under HW-M and 29% under SW-M. This effect was also observed in [8].

The resulting *ETR*s between SW-M and HW-M are 1.11, 1.36, 1.53, and 1.46 for Water, LU, Ocean, and MP3D, respectively. Migratory optimization results in lower *ETR* for applications with a high degree of migratory sharing (Water and MP3D). The lower *ETR*s result both from a relatively larger reduction of W_{sw} than of W_{hw} and from a reduction of P_{sw}. For LU, we find that the *ETR* is virtually the same with and without migratory optimization. Finally, the *ETR* increases for Ocean, mainly as a result of a smaller reduction of S_{sw} than S_{hw} and a larger increase of R_{sw} than of R_{hw} when migratory optimization is applied.

In summary, we have found that migratory optimization effectively reduces both W_{sw} and P_{sw} for applications with a high degree of migratory sharing. Since the write and synchronization stall times are relatively more reduced for SW-M than for HW-M, the *ETR* decreases. Further, migratory optimization can potentially reduce S_{hw} and S_{sw} also for other applications as a result of more efficient barrier synchronizations. Finally, migratory optimization can increase the *ETR* for applications with a high degree of false sharing which penalizes SW more than HW.

6. Performance of release consistency

6.1. Qualitative evaluation

The third technique we will study is release consistency, a technique that not affects the protocol execution time. Under sequential consistency, SC, which is our default memory consistency model, the processor stalls on each access to shared data in order to enforce a global access order. By contrast, under *release consistency* [6], *RC*, ordering is only enforced on special, hardware-recognizable synchronization primitives. RC distinguishes between *acquires* and *releases*. The most important implication of RC in our framework is that the processor does not stall on write and release requests, i.e., all write stall time can be hidden and the synchronization stall time may decrease.

Relating the expected performance effects of RC to Figure 2, we note that $W_{hw} < W_{sw}$ under SC. Since $W'_{hw} = W'_{sw} = 0$ under RC, we expect the software-only directory protocol to gain relatively more from RC than the hardware-only does. Further, we expect both S_{hw} and S_{sw} to decrease, but a slight increase in R_{hw} and R_{sw} may occur as a result of higher contention. Finally, since RC does not change the number of coherence requests in the system, we expect P_{sw} to be unaffected.

6.2. Quantitative evaluation

In Figure 5, we show the simulated execution times of hardware-only and software-only directory protocols under SC (HW and SW, respectively) and RC (HW-RC and SW-RC, respectively). We start with a comparison between the execution times of HW and HW-RC, and between SW and SW-RC. The results in Figure 5 show that RC gives a large and consistent performance improvement for both hardware-only and software-only directory protocols, which is consistent with the results in [7].

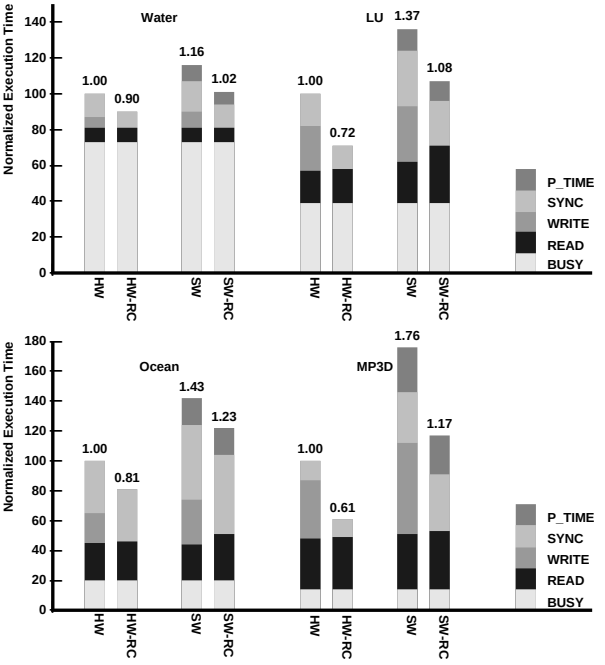

Figure 5. Normalized execution times for HW-only and SW-only protocols under sequential (HW and SW) and release consistency (HW-RC and SW-RC).

By comparing the protocol execution overhead for SW and SW-RC, we see that the intuition from Section 6.1 seems correct; since the number of coherence requests are virtually the same under SC and RC, P_{sw} is virtually unaffected for all applications.

As we can see in Figure 5, RC removes W_{hw} and W_{sw} for all applications. For the hardware-only directory protocol, this is achieved with virtually no increase in R_{hw} and S_{hw}. By contrast, R_{sw} has increased for LU and Ocean. For LU, the main reason is longer delays in the *FLWB*. Under SC, a *FLC* read miss gets served by the *SLC* almost at once, while under RC it has to wait in average seven cycles. In a software-only directory protocol, each global write request takes a longer time than in a hardware-only directory protocol. Therefore, the *SLWB* is filled up more often in a software-only directory protocol which blocks requests from the *FLWB*. For Ocean, the longer R_{sw} originates from a combination of longer delays in the *FLWB* and higher contention in the network interfaces.

The resulting *ETR*s between SW-RC and HW-RC are 1.13, 1.50, 1.52, and 1.92 for Water, LU, Ocean, and MP3D, respectively. For LU, Ocean, and MP3D, the *ETR* increases under RC. The relatively larger contribution from P_{sw} to the execution time under RC results in a higher *ETR* for MP3D. For LU and Ocean, a combination of a relatively larger P_{sw} and longer R_{sw} increases *ETR*. The *ETR* slightly decreases for Water under RC, as a result of slightly shorter S_{sw} and P_{sw} under RC than under SC.

In summary, RC hides all write latency for both hardware-only and software-only directory protocols. As expected, the

ETR between software-only and hardware-only directory protocols increases for three of the application under RC. For the fourth application, Water, an application with high computation-to-communication ratio, the *ETR* slightly decreases under RC.

7. Conclusions

In both hardware-only and software-only directory protocols, i.e., protocols where the directory management is done by software handlers executed on the compute processor, performance is often limited by processor stall times due to memory accesses latencies. For software-only directory protocols, the total execution time may be prolonged by protocol execution overhead resulting from the handler invocations, referred to as p-time. To increase the performance, many latency tolerating and reducing techniques have been proposed and evaluated for hardware-only protocols. However, the effectiveness of such techniques for software-only directory protocols has been unexplored.

In this study we have evaluated the relative performance between hardware-only and software-only directory protocols when different latency tolerating and reducing techniques are applied. Such techniques can be divided into three classes depending on how they impact p-time; a technique either increases, decreases, or does not affect p-time. As representatives for the three classes we have chosen prefetching, a migratory optimization technique, and release consistency, respectively.

Based on architectural simulations we have found that software-only directory protocols are more sensitive to the prefetch efficiency than hardware-only directory protocols are. Each useless prefetch, i.e., a prefetch that fetches a block that is evicted from cache before the processor accesses it, increases p-time which potentially prolongs the total execution time. Our results show that even though prefetching reduces the read stall time in a software-only directory protocol, the execution time may be prolonged by to many useless prefetches. Therefore, the prefetch efficiency is more important when choosing a prefetch scheme for a software-only than for a hardware-only directory protocol.

In contrast, software-only directory protocols generally gain relatively more than hardware-only directory protocols when techniques that reduce the number of coherence actions are applied. This was confirmed with the migratory optimization technique. For three of four applications, the performance gap between software-only and hardware-only directory protocols decreases when migratory optimization is used.

Release consistency, a technique that not affects p-time, hides all write stall time for both software-only and hardware-only directory protocols. However, since release consistency does not reduce p-time, it becomes a relatively larger part of the total execution time. For three of our applications, the relative performance between hardware-only and software-only directory protocols increases when release consistency is applied.

Overall, this study shows that the efficiency of a latency tolerating technique not only depends on how well it tolerates the latency as seen by the local node, but also on how it interacts with the node where a memory block is allocated. Therefore, more care has to be taken when choosing an appropriate latency tolerating and reducing technique for software-only directory protocols than needed for hardware-only directory protocols.

Acknowledgments

This research has been funded in part by the Swedish National Board for Industrial and Technical Development (NUTEK) under contract number P855. The main part of this work was carried out while the authors were at the Department of Computer Engineering at Lund University.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B-H. Lim, K. Mackenzie, and D. Yeung, “The MIT Alewife Machine: Architecture and Performance,” *Proc. 22nd Int’l Symp. on Computer Architecture*, pp. 2-13, Jun. 1995.

[2] M. Brorsson, F. Dahlgren, H. Nilsson, and P. Stenström, “The CacheMire Test Bench — A Flexible and Effective Approach for Simulation of Multiprocessors,” *Proc. 26th Ann. Simulation Symp.*, pp. 41-49, Mar. 1993.

[3] L.M. Censier and P. Feautrier, “A New Solution to Coherence Problems in Multicache Systems,” *IEEE Trans. on Computers*, C-27(12):1112-1118, Dec. 1978.

[4] D. Chaiken and A. Agarwal, “Software-Extended Coherent Shared Memory: Performance and Cost,” *Proc. 21st Int’l Symp. on Computer Architecture*, pp. 314-324, Apr. 1994.

[5] F. Dahlgren and P. Stenström, “Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors,” *IEEE Trans. on Parallel and Distributed Systems*, 7(4):385-398, Apr. 1996.

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, “Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors,” *Proc. 17th Int’l Symp. on Computer Architecture*, pp. 15-26, May 1990.

[7] H. Grahn and P. Stenström, “Efficient Strategies for Software-Only Directory Protocols in Shared-Memory Multiprocessors,” *Proc. 22nd Int’l Symp. on Computer Architecture*, pp. 38-47, Jun. 1995.

[8] H. Grahn and P. Stenström, “Evaluation of a Competitive-Update Cache Coherence Protocol with Migratory Data Detection,” *J. of Parallel and Distributed Computing*, 1997. (To appear)

[9] H. Grahn and P. Stenström, “Performance Evaluation of Latency Tolerating and Reducing Techniques for Software-Only Directory Protocols,” Technical Report, Dept. of Computer Science, University of Karlskrona/Ronneby, Jan. 1997.

[10] A. Gupta and W-D. Weber, “Cache Invalidation Patterns in Shared-Memory Multiprocessors,” *IEEE Trans. on Computers*, 41(7):794-810, Jul. 1992.

[11] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W-D. Weber, “Comparative Evaluation of Latency Reducing and Tolerating Techniques,” *Proc. 18th Int’l Symp. on Computer Architecture*, pp. 254-263, May 1991.

[12] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, “The Stanford FLASH Multiprocessor,” *Proc. 21st Int’l Symp. on Computer Architecture*, pp. 302-313, Apr. 1994.

[13] 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 Computer Systems*, 11(4):300-318, Nov. 1993.

[14] T. Mowry, “Tolerating Latency Through Software-Controlled Data Prefetching,” Ph.D. Thesis, Stanford University, Mar. 1994.

[15] S. K. Reinhardt, J. R. Larus, and D. A. Wood, “Tempest and Typhoon: User-Level Shared-Memory,” *Proc. 21st Int’l Symp. on Computer Architecture*, pp. 325-336, Apr. 1994.

[16] J-P. Singh, W-D. Weber, and A. Gupta. “SPLASH: Stanford parallel applications for shared-memory,” *Computer Architecture News*, 20(1):5-44, Mar. 1992.

[17] P. Stenström, M. Brorsson, and L. Sandberg, “An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing,” *Proc. 20th Int’l Symp. on Computer Architecture*, pp. 109-118, May 1993.