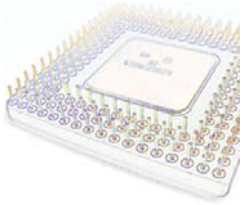


Second Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG-2)

January 25, 2008 Paphos, Cyprus

In Conjunction with the Fourth International Conference on High-Performance Embedded Architectures and Compilers (HIPEAC)

Message from the Organizers

Welcome to the Second Workshop on Programmability Issues for Multi-Core Computers.

We are delighted to present a very strong program composed of 7 high-quality papers, an informative Keynote talk on SARC by Georgi N. Gaydadjiev from Delft and a stimulating panel discussion moderated by Per Stenström from Chalmers.

As computer manufacturers are embarking on the multi-core roadmap, which promises a doubling of the number of processors on a chip every other year, the programming community is faced with a severe dilemma. Until now, software has been developed with a single processor in mind and it needs to be parallelized to take advantage of the new breed of multi-core computers. As a result, progress in how to easily harness the computing power of multi-core architectures is in great demand.

This workshop brings together researchers interested in programming models and their implementation and in computer architecture; who share a common interest in advancing our knowledge on how to simplify the task of parallelization of software for multi-core platforms. A wide spectrum of issues are central themes for this workshop; such as what the future programming models should look like to accelerate software productivity and how it should be implemented at the runtime, the compiler, and the architecture level. A special issue that contains selected papers is planned for the second issue of Transactions on HiPEAC in 2009.

We would like to thank the Program Committee and the additional reviewers for their effort; their hard work culminated in each submission having at least three reviews.

Eduard Ayguadé (BSC/UPC), Roberto Gioiosa (IBM Watson), Per Stenström (Chalmers) and Osman S. Unsal (BSC)

Organizing Committee

Program Committee

David Bernstein	IBM Research Lab in Haifa	Israel
Mats Brorsson	KTH	Sweden
Barbara Chapman	University of Houston	USA
Marcelo Cintra	University of Edinburgh	U.K.
Magnus Ekman	Sun Microsystems	USA
Pascal Felber	University of Neuchatel	Switzerland
Christof Fetzer	Dresden University of Technology	Germany
Matthew I. Frank	Univeristy of Illinois, Urbana-Chhampaign	USA
Guang Gao	University of Delaware	USA
Roberto Giorgi	University of Siena	Italy
Erik Hagersten	Uppsala University	Sweden
Mark Harris	Nvidia	Australia
Jay P. Hoeflinger	Intel	USA
Haoquiang Jin	NASA Ames	USA
Stefanos Kaxiras	University of Patras	Greece
Mikel Lujan	University of Manchester	UK
Ami Marowka	Shenkar College of Engineering and Design	Israel
Avi Mendelson	Intel	USA
Dieter an Mey	RWTH, Aachen	Germany
Andre' Seznec	IRISA	France
Peng Wu	IBM Watson Research	USA

Table of Contents

Power-Efficient Scaling of CMP Directory Coherence

Stefanos Kaxiras, Georgios Keramidas, Ioannis Economou
University of Patras, Greece

Abstract. Scalability of a parallel application in the era of multicores must account for both performance and power. A parallel application is power-efficient scalable if it maintains or improves its power efficiency (i.e., its EDP, or Energy per Instruction) as the core count allocated to run the application is increased. For an application to be EDP scalable to high core counts, both its core power (which increases roughly linearly with core count) and its network power consumption (which is a function of coherence traffic) must be commensurable to the attained performance. The goal of our work is to improve EDP scalability of shared-memory applications in CMPs by making directory coherence much more efficient in both power and performance. Our main idea is to have writer-only coherence and tear-off blocks to eliminate the bulk of the invalidation traffic, combined with writer prediction to eliminate much of the directory indirection (i.e., avoid going to the directory as much as possible for coherence operations). Instead, we make the writers responsible for maintaining coherence. This way, we achieve, at the same time, both power (network traffic reduction) and performance (read/write latency reduction) benefits. With writer prediction our protocols require an accuracy of 50% or greater to start yielding benefits. We optimize accuracy at run time by detecting the underlying sharing patterns and applying our protocols selectively. Our study show significant improvements in EDP for the SPLASH benchmarks for a wide range of core counts.

1 Introduction

To scale application performance in the era of multicores we must rely on explicit parallelism, beyond just ILP. An important issue for advances in this direction, is ease of parallel programming and to this end, the shared-memory programming model offers a good starting point. However, at the same time, we cannot ignore the issue of power efficiency. Poor power-efficiency —overblown power budgets for diminishing performance gains— killed the development of ever wider ILP architectures.

The danger of exploding power budgets for diminishing performance gains is also visible in multicores: when, for example, a parallel application experiences *sub-linear speed-up* and/or consumes power that increases *faster* than the number of cores allocated to the application. Hill describes the various notions of the term scalability [22], and, in the era of multicores, it is useful to think of the *scalability* of a parallel program in terms of *power-performance*. An existing metric, the Energy-Delay Product (EDP) [21] fits this purpose very well and in this work, we study the *EDP scalability* of parallel programs. The power efficiency (EDP) of a parallel application is shaped by three forces:

- the performance scalability (speedup) of the application: a sub-linear speedup means that as we increase the number of cores allocated to the application, its power efficiency, considering core power alone, is likely to worsen.

- the application's communication-to-computation growth rate: typically true sharing communication increases with the number of cores. This is inherent in the parallel applications; a good analysis appears in [1] and in [6]. Increasing true sharing communication, causes an increase in network power which worsens power-efficiency.
- the working set size and how it fits in the core caches —also analyzed in [1] and [6]. This is an opposing force to the two above that tends to *improve* power-efficiency with increasing core count until the working set fits in the caches.

EDP scalability, therefore, depends on the behavior of core and network power. Core power, at first approximation increases about linearly and leads to poor power efficiency if we employ more cores *for relatively less speed-up*. More interestingly, network power is an interplay between the capacity-miss traffic at low core counts and the increased coherence-communication traffic at higher core counts. Furthermore, network power is affected by the *distances* travelled by the messages. As we use larger networks (at higher core counts), the energy spent per message increases.

This paper focuses on the problem of improving *power-efficiency* for parallel applications in a shared-memory CMP. We assume a sufficient number of cores to warrant a network on chip (NoC), such as a mesh, rather than shared busses which do not scale well in terms of bandwidth. With point-to-point messaging, we assume that some form of a directory-based coherence protocol [9][10][11] is employed to keep distributed private L1 caches coherent. We assume that the directory is co-located with the shared on-chip L2 cache, i.e., the directory tracks only the on-chip cache blocks. In this setting, the central question we ask is: Is directory coherence **scalable** in terms of power-efficiency? If not, how can we make it so?

What we show is that: i) Directory coherence, due to the large number of messages it generates and sends via a power-hungry component of the CMP, the NoC, eventually does not scale well in terms of power/performance (EDP). ii) A significant source of inefficiency is the directory itself —specifically, the central role of the directory in all coherence operations necessitates indirections through it. We can significantly improve EDP by avoiding going to the directory as much as possible. We default to going to the directory only when we do not know what else to do. Our proposal combines, in a novel way, three approaches:

Our evaluation is based on execution-driven simulation using Simics/GEMS[29]. We model a CMP with a NoC and a Dir_iNB directory protocol [13], using power models for both the network [28] and the cores [27]. For SPLASH-2 benchmarks [1] our evaluation shows that we can significantly improve EDP across a wide range of core counts.

2 Transparent Reads and Tear-Off copies: Taking the Readers Out of the Directory

Main idea: *Transparent reads are reads that do not register in the directory. A transparent read promises that the cache block copy it fetches will be thrown away (self-invalidate) at the first synchronization event experienced by the processor which issued the transparent read*.

With transparent reads, coherence in the "normal" sense, i.e., sending invalidations upon a write, is maintained only among writers. We call this *writer coherence*. Writers are still required to register in the directory so the latest value of the cache block can be

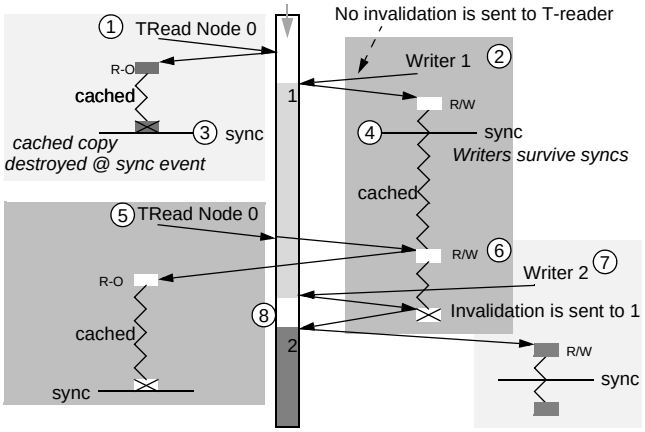


Figure 1. Basic operations. The node on the left reads with transparent reads and does not register in the directory (does not get invalidated by writers, but throws away its cache copy at synchronization events). The two nodes on the right, write and invalidate the previous writers (writer coherence).

safely tracked. A new write simply invalidates the previous writer of the block —there are no readers since they are transparent.

Figure 1 shows some of the basic operations of the Transparent Read protocol. The node on the left transparently reads a block (1) by going to the corresponding directory entry. The transparent read leaves no trace in the directory and gets a clean cache block in Read-Only mode (*RO*). The state of the cache block becomes *Tear-off RO* or *TRO*. Soon after the transparent read, a writer (on the right) goes to the directory (2) and obtains the block with Read/Write permissions (R/W). The writer does not invalidate the transparent reader, since the former cannot see the latter. However, the (transparent) reader promised to throw away the TRO block at the first synchronization point and does exactly that (3). The writer, who is registered in the directory does not have to throw away its copy (since it is registered in the directory) and the copy survives the synchronization point (4). In the mean time, the reader, who lost its copy in the last synchronization, is sending a new transparent read (5), which, after reaching the directory, proceeds to fetch the latest value from the writer (6). The writer *does not downgrade* with a transparent read. A second writer (7), however, invalidates the previous one (8) and registers its ID in the directory. Normal reads *downgrade* a writer from R/W permissions to RO. The downgraded copy however, can still survive a synchronization point (in contrast to the TRO copies brought in by Transparent Reads). The reason is that a writer's downgraded (RO) cache block copy is still registered in the directory and will be invalidated by the next writer. Lastly, note that the invalidation protocol is always active underneath the transparent read and tear-off mechanisms. This means that we can *freely mix* invalidation and tear-off copies, simultaneously, in the same directory entry. Minimal changes are required in the cores, protocol engines, and caches to implement transparent reads [2].

Tear-off cache copies were shown in [2] to be compatible with weak-ordered memory systems which requires correctly synchronized programs (all synchronization is clearly identified and exposed, unlike the example above) [7][8]. Weak ordering is the only relaxed memory consistency model that allows the re-arrangement of the program

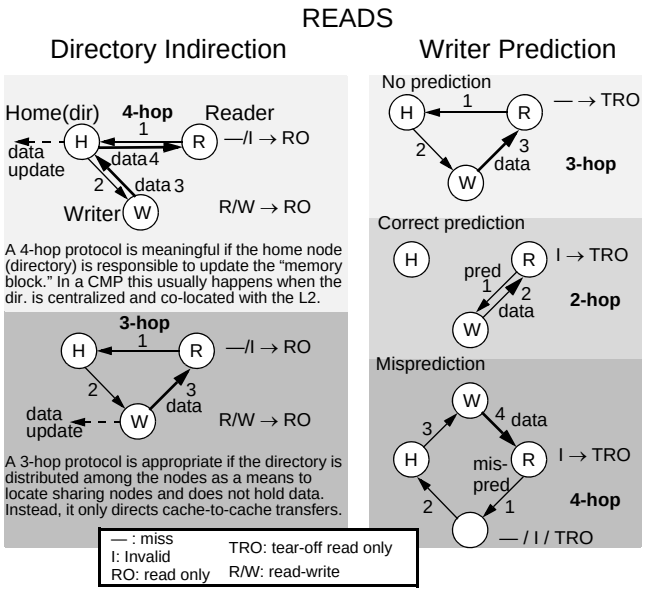


Figure 2. Writer prediction. Our protocol (on the right) aims to avoid the indirection via the directory (when possible) and go directly to the writer. In the first case the reader cannot predict and uses a standard 3-hop protocol. The second case is correct prediction (2-hop) and the third case is a misprediction where we have an additional indirection via a wrong node (4-hop). Note that the writer is downgraded when its copy is read in the invalidation protocols (left); this is not so with the transparent reads in our proposal.

order as long as this happens in-between synchronizations [6]. This allows, for example, compilers to optimize programs by re-arranging memory instructions [6]. Replacing flag synchronization with semaphore synchronization, (e.g., SIGNAL and WAIT which cannot use TRO copies) solves the problem. Thus, the only requirement for using transparent reads and tear-off copies is to have correctly synchronized programs.

Benefits: There are both performance and power implications from transparent reads and tear-off copies. First, in terms of performance writes become faster because reader invalidation (and its acknowledgement) is no longer required. However, since transparent reads and tear-off copies practically require a weakly-ordered consistency model, the improvement of write performance does not contribute significantly to the overall performance. In large part, weak consistency models are needed to hide the write latency, and its reduction is, therefore, unimportant. Secondary performance benefits come from removing the invalidation traffic from the network, reducing its congestion and average latency. On the other hand, significant power benefits come from the removal of the invalidation traffic.

However, there is also the danger of a negative impact on performance and power due to the discarding of all tear-off copies at synchronization events. This can increase a node's miss rate for items that should have survived the synchronization. We guard against this by detecting the type of sharing and switching to the invalidation protocol when necessary. This adaptation is discussed in Section 4.

3 Writer Prediction: Avoiding Directory Indirection

While transparent reads and tear-off copies reduce the invalidation traffic, a more significant source of inefficiency still remains in directory protocols: *the indirection through the directory*. The directory (whether centralized or distributed) is a fixed point of reference to locate and obtain the latest version of the data (i.e., the writer) or invalidate its sharers. It seems difficult not to have an indirection via the directory, which leads to the classic 3-hop, or 4-hop (invalidation) protocols.

Our main contribution is in exploiting the properties of transparent reads and tear-off copies to go directly to the writers (by predicting their identity), skipping the directory when possible. *In essence, we allow the writers to assume the role of the directory —the central role of coherence— but revert back to directory indirection when we are unable to locate the writers.*

3.1 Reads

In typical directory protocols, a read goes to the directory to find where the latest version of the data is. The correct data are then forwarded from the last writer with a 3-hop protocol (Figure 2, lower left) or back via the directory with a 4-hop protocol (Figure 2, upper left). Transparent reads, however, do not register in the directory. We exploit this property —to the best of our knowledge for the first time— to *avoid going to the directory altogether, i.e. avoid directory-indirection for reads*. Reads try to obtain the data directly from the writer, if they can locate it.

To locate the current writer (without peeking the directory) we use prediction. Based on history, a reader sends a direct request to a suspected writer (Figure 2, middle right). If this node happens to have *write permissions* (R/W) for the requested cache block then the prediction is correct: the node is the (one and only) writer of the block and has the latest copy of the data. If the node is in any other state (including knowing nothing about the block) it bumps the request to the directory (Figure 2, lower right, *misprediction* case). From there the request will be routed to the correct writer and back to the reader (i.e., the normal 3-hop protocol). The penalty for a misprediction is one extra message (indirection via the wrong node).

Prediction. While we can imagine arbitrary complex predictions to locate a writer, in this work we strive for simplicity. The prediction is carried by the cache blocks themselves (there are no separate prediction or history structures). Thus, in order for a read to make a writer prediction, it needs to have an invalidated —self- or externally-invalidated— cache block; otherwise the writer cannot be predicted and the request must be sent to the directory (Figure 2, upper right: *no-prediction* case). An invalid cache block carries with it the ID of the last known writer: *either* the node from where it got the correct data the last time around (if it self-invalidated), *or* the ID of the node that externally invalidated the cache block. The overhead for a reasonably sized 64-core CMP is just 6 bits per cache line, which is negligible. This simplistic prediction is surprisingly effective if it is restricted to the appropriate subset of producer-consumer sharing and *avoided* in the case of migratory sharing. We discuss this in Section 4.

Benefits. Avoiding directory indirection for reads is important in two ways. First, reads —in contrast to writes— are performance-critical, in the sense that a reduction of their latency directly reflects on the overall performance. Second, directory-indirection accounts for a significant part of the read traffic. Eliminating this (directory-indirection) traffic has an immediate impact on network power (and indirectly on network performance because it relieves congestion and improves network latency).

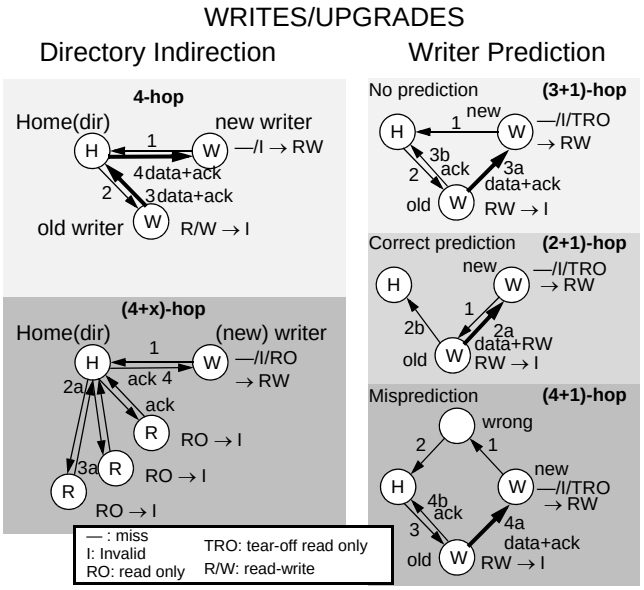


Figure 3. Writer (upgrade) optimization. Our protocol (on the right) relying on having a single registered node (writer) in the directory avoids data indirection via the directory with a 3-hop protocol. Furthermore, writer prediction can overlap going to the directory and getting the data from the last writer (middle and lower right).

Compared to the optimized 3-hop protocol, writer prediction yields a 2-hop transaction when correct, or a 4-hop transaction on a misprediction. A simple calculation reveals that any prediction accuracy over 50% yields both performance *and* power benefits.

3.2 Writes/Upgrades

Similarly to reads we also optimize the protocols for the writes and upgrades. Under an invalidation protocol, a (new) writer sends its request to the directory which is responsible to invalidate all other sharers. This is shown in Figure 3 on the left side; the top diagram shows the invalidation of a single "old" writer (4-hop), while the bottom the invalidation of multiple readers (4-hop critical latency, plus several overlapping transactions).

Data forwarding optimization. A straightforward optimization is to recognize that, with transparent reads and tear-off copies, there is always exactly one other node registered in the directory: the last writer. This makes it easy to directly transfer the data from the old writer to the new writer. In Figure 3, upper right, the request of the new writer (message 1) is routed via the directory to the last writer (message 2) which gets invalidated and forwards its latest value, *along with R/W permissions*, directly to the new writer (3a). Simultaneously, the old writer returns its acknowledgement to the directory (3b) exactly as it would in the normal 4-hop protocol (Figure 3, upper left). Because this is a 3-hop-latency protocol with an additional overlapping hop (3a and 3b overlap) we describe it as a **(3+1)-hop** protocol.

In both the normal 4-hop and the (3+1)-hop protocol, the directory points to the new writer *only after* it receives the acknowledgement from the old writer (message 3 in Fig-

ure 3 upper left, or message 3b in upper right). In the window between the initial request and the old-writer acknowledgment, other potential writers are NACKed, exactly as in the DASH [10] or the SGI Origin protocols [11]. While with NACKs there is always a potential starvation problem see Lenoski and Weber, p. 288, for solutions to such problems [11].

Writer prediction. Our contribution in this case is a more aggressive approach where the new writer *predicts* the old writer and sends a direct request to it. This is shown in Figure 3, middle and lower right. *The main idea here is that, the old writer assumes the role of the directory, passing out its R/W privileges. This is possible because it is the only node in the position to do this, other than the directory.*

The direct request from the new writer arrives at the predicted node (Figure 3, middle right). If the predicted node has R/W privileges then it is the (one and only) writer of the block holding the latest copy of the data. It returns the data to the requester, passing along its R/W privileges. At the same time, the old writer is responsible to inform the directory that it has relinquished its rights to the new writer and self-invalidated. This is a **(2+1)-hop** protocol because it overlaps the last two messages. Note that the only difference from the no-prediction case is that we have replaced the indirection via the directory (messages 1 and 2, upper right) with a single message (message 1, middle right) directly to the old writer. The acknowledgment of the old writer to the directory, message 2b, carries the identity of the new writer and plays the exact same role as message 3b in the no-prediction case.

In case of a misprediction (Figure 3, lower right), the incorrectly routed request bumps-off the wrong node —which is *not* the writer of the block— and is re-router to the directory. The penalty in this case is an extra request message indirection via the wrong node, resulting in a **(4+1)-hop** protocol. Besides the indirection via the wrong node the rest of the protocol is exactly the same as in the no-prediction case (Figure 3, upper right).

Benefits. Optimizing the write and upgrade protocols in (3+1)-hops does not reduce the message *count*. However, compared to the normal 4-hop protocol, where the latest value of the data must indirect via the directory to reach the new writer, the (3+1)-hop protocol saves one data transfer with an obvious power benefit. In addition, the critical latency drops from 4-hops to 3-hops offering a performance benefit for the writes.

Writer prediction, when correct, reduces the message count from 4 messages to 3 messages since it coalesces the two directory-indirection messages into one direct message to the writer. Besides the power benefit of eliminating a message, there is also a performance benefit because the critical latency further drops from 3 hops to 2 hops — hence called a (2+1)-*hop* protocol. However, a misprediction results in a *4-hop* 5-message protocol with negative impact on both power and performance. A simple calculation again reveals that a prediction accuracy of over 50% starts to yield benefits. To keep the accuracy high we only allow writer-prediction on stable sharing patterns with predictable writers. This is discussed in the next section.

4 Sharing Pattern Classification

While, in our proposal, there are potential power and performance benefits there is also a downside under certain conditions, for example when needlessly discarding treaoff copies that are not written by other nodes, or when unsuccessfully trying to predict the writers in *migratory sharing* or *unstructured sharing* (where the writers change in unpredictable ways).

To guard against these two undesirable situations, we only allow tear-off copies or writer prediction when the underlying cache line exhibits an acceptable sharing pattern. This approach is not new. Lebeck and Wood proposed the adaptation for tear-off copies [2], and Stenström et al. [17] and Cox et al. proposed adaptation for migratory sharing patterns [18]. We combine their mechanisms in the directory as follows:

- We first detect whether a cache block is read-mostly or frequently-written. If the block is requested as TRO repeatedly from the same nodes without any intervening writes the invalidation protocol is preferred for this block. This decision is taken at the directory (which can turn Transparent Reads into normal reads) and carried by all the writers.
- Further, writer prediction is allowed only for frequently written blocks (i.e., in conjunction with transparent reads). However, it is difficult to achieve good accuracy (using only the last-writer information in invalid cache copies) if the writers of a block change very frequently. In such a case, the directory (which sees all writers) disallows writer prediction. The directory's decision is also carried with the writers, so that transparent reads using writer-prediction —skipping the directory— are informed of what to do the next time.

Figure 4 summarizes the sharing pattern classification that takes place in our proposal, the detection mechanisms, and the decisions taken in each case. Finally, we note that *false sharing* can potentially mess up both steps of this classification. False sharing can give the impression of frequently-written to data that aren't, in which case, we make the wrong choice and use TRO where we should not. On the other hand, false sharing is likely to disable writer-prediction (e.g., when we have alternating false-sharing writers) which limits the damage. Fortunately, false sharing can be very effectively handled at higher levels as it was shown by Jeremiassen and Eggers [19].

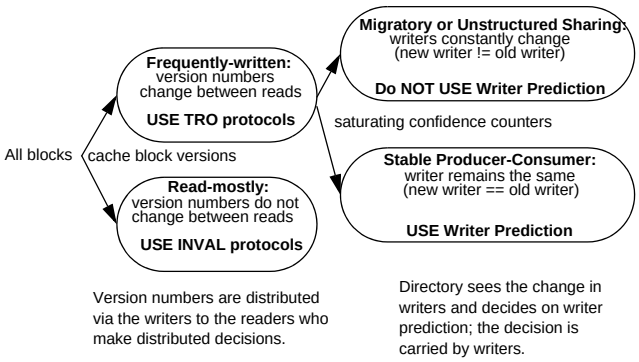


Figure 4. Sharing pattern adaptation

5 Related Work

Dynamic Self Invalidation. The closest work to our proposal is the work of Lebeck and Wood, Dynamic Self-Invalidation, that inspired our transparent read and TRO protocols [3]. However, Lebeck and Wood stopped short of optimizing the protocols with writer prediction to eliminate directory indirection. Because of writer prediction, the decision of whether to use TRO copies or not is *distributed* to both the directory and to the nodes. This means that our protocols must support mixed operations since we can have TRO and invalidation copies simultaneously for the same block. Further, at the time of

the Lebeck and Wood work, power was not an issue (especially for parallel architectures) and their study concerns only performance. Power-efficiency is the main driver in our study. Related to DSI is Lai and Falsafi's last-touch prediction which, however, carries additional H/W overhead (power) in the form of prediction structures.

Prediction and Coherence. The use of prediction is also not new in coherence protocols. Muckherjee and Hill first published on using prediction to optimize coherence [14]. Kaxiras and Goodman introduced Instruction-based prediction that relates prediction and optimizations to code, not data [15]; Kaxiras and Young studied reader prediction [16]. The writer prediction we use is very simple and works only because it is restricted to stable (producer-consumer) sharing patterns. We note that more sophisticated predictors that have been proposed previously can expand the coverage for writer prediction and make it less reliant on directory guidance.

Sharing Pattern Detection and Optimization. Weber and Gupta identified several sharing patterns in parallel applications [18]. Soon after, many papers discussed targeted optimizations relying on the directory or prediction (addressed-based or instruction-based) to detect sharing patterns [17][18][15][16]. While our reference list is too small to include all relevant work, we note here our approach is based on the work or Stenström et al. for detecting migratory sharing but is part of a more complex classification scheme which identifies several patterns.

Optimizations to Coherence Protocols. Significant work also concerns the optimization of coherence. We note here two prominent approaches: Token Coherence [23] and In-Network Coherence [24], sharing tree structures [25][26] and others. However, all such work requires considerable message traffic making it difficult to approach the minimalistic messaging of our protocols.

6 Evaluation

In this section, we present our initial study for our proposal. We implemented our protocols in Gems [29] (which includes Wattch [27] and Orion [28] to model the power of the cores and the network respectively). Gems is configured to model a CMP with a mesh interconnect. The simulator parameters and the SPLASH-2 benchmarks with their inputs are shown in Table 1.

Table 1: Simulator configuration

Benchmark	Input	Chip configuration	
barnes	8K bodies, 4 timesteps	# of Processors	16
fft	64K complex doubles	Processor	Simics in-order blocking model
radix	2M keys	D & I L1 Caches	64KB, 4-way, 64-Byte Block size, 3-cycle latency, Pseudo-LRU
ocean-contig., ocean-non-contig.	66 x 66 grid	Shared L2 Cache	8 MB, 8 way, 64-Byte Block size, 30 cycle latency, Pseudo-LRU
water (N-Squared)	64 molecules, 3 timesteps	Interconnection Network	2D Mesh topology, 1-cycle link latency
water (Spatial)	512 molecules, 3 timesteps	Directory	Full bit vector sharers list, 6 cycles latency

In our configuration, most benchmarks show good speedups up to 16 cores, except ocean-ncont, and fft. We run the benchmarks with our protocols and we present results

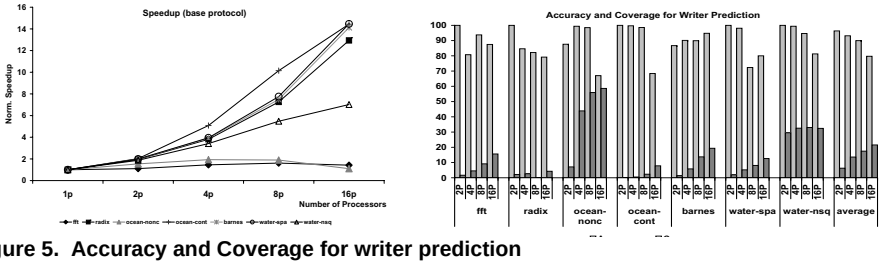


Figure 5. Accuracy and Coverage for writer prediction

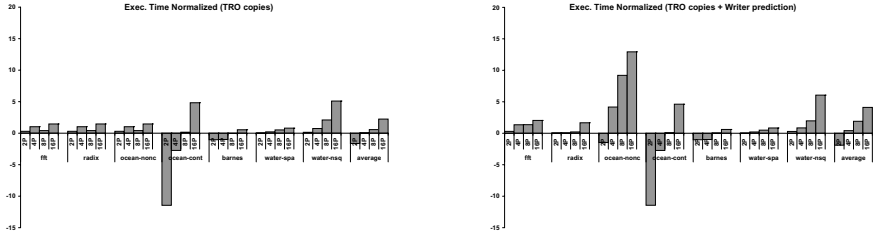


Figure 6. Normalized execution time

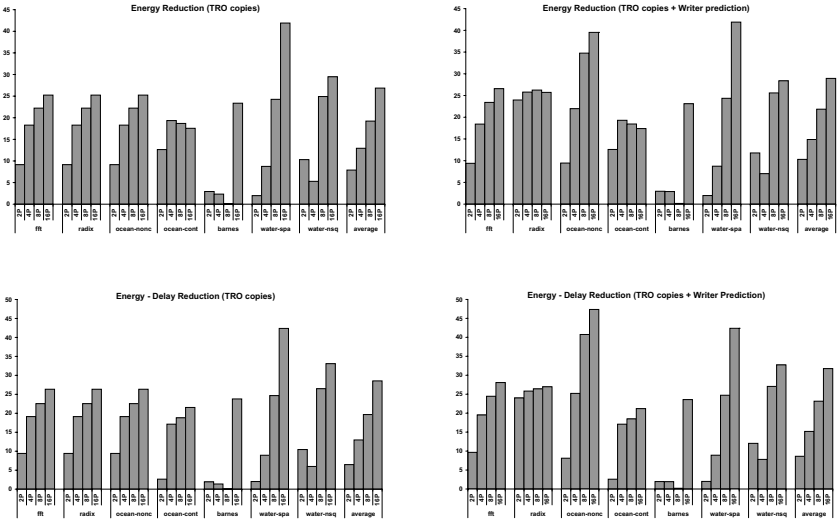


Figure 7. Normalized network energy and EDP

for both the TRO protocol and the TRO with writer prediction (TRO-WP). Results with writer prediction are better than TRO alone, although the improvement is proportional to the coverage of writer prediction. The accuracy and coverage of writer prediction are shown in Figure 5. In general, coverage is not very high (except in ocean-ncont) which indicates that there is considerable room for improvement with different prediction schemes. As expected, coverage increases with the number of cores: as cache capacity is increased with core count, more invalid TRO copies (which provide the predictions) survive in the caches. Accuracy, on the other hand, is more specific to the benchmark

and —in general— drops with higher core counts (where predicting the right core out of many more becomes harder). However, it does not drop below 80% even in 16 cores.

Figure 6 shows the effects of the TRO and TRO-WP protocols on execution time. The results are normalized to the base MESI protocol in each case. TRO provides improvements because of the elimination of invalidation but may also result in a slowdown because of increased L1 miss rates. The slowdown is especially pronounced when we also have significant capacity miss rates, for example in 2 or 4 cores for ocean-cont (and to a lesser extent in barnes and ocean-ncont). However, slowdown is restricted in low counts and as we use more cores the speedup obtained increases. More significant is the improvement in execution time which comes from writer prediction. The effect is more pronounced in ocean-ncont which has increased prediction coverage. But even with low prediction coverage the effects of writer prediction yield a noticeable improvement in execution time over TRO.

TRO and especially TRO-WP yield significant reductions in network energy (Figure 7) by eliminating a significant portion of the overall traffic. The reduction in network energy combined with the execution time from Figure 6 gives a reduction in EDP shown in the lower two graphs of Figure 7. Overall the reduction in EDP is 8%, 15%, 24% and 32% for 2,4,8 and 16 cores respectively. As our protocols reduce EDP more with higher core counts, they achieve better EDP scaling.

7 Conclusions

We present a novel and minimalistic approach to scalable directory coherence. Our view of scalability in the context of multicores is scalability in power-efficiency, meaning that as we increase the core count allocated to an application, the application scales both in performance and power in a manner that compromises EDP as little as possible.

Our proposal strives to avoid directory indirection (the main source of inefficiency in directory coherence) by using transparent reads and tear-off copies to eliminate invalidation traffic and using writer-prediction to eliminate directory indirection. Overall, this approach is easy to implement and minimalistic in philosophy compared to other advanced coherence schemes. Our initial study shows a good prediction accuracy for writer-prediction albeit low coverage. Writer prediction enhances the TRO protocol offering a significant reduction in message traffic resulting in a reduction of network energy/power. Coupled with the performance improvements stemming from the elimination of directory indirection, our proposal provides significant benefits for EDP across a wide range of core counts.

8 References

[1] S. C. Woo et al. “The SPLASH-2 programs: characterization and methodological considerations,” Proceedings of the 22nd ISCA, 1995.

[2] AR Lebeck, DA Wood, “Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors,” Proceedings of the 22nd ISCA, 1995.

[3] AC Lai, B Falsafi “Selective, accurate, and timely self-invalidation using last-touch prediction,” Proceedings of the 27th ISCA, 2000.

[4] A. Lai, B. Falsafi “Memory Sharing Predictor: The Key to a Speculative Coherent DSM.” 26th ISCA, May 1999.

[5] Leslie Lamport, “How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs.” *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[6] J.P.Singh D. Culler “Parallel Computer Architecture: A Hardware/Software Approach,” Morgan-Kaufmann Publishers, 1998.
[7] SV Adve, MD Hill “Weak ordering-a new definition” Proceedings of the 17th ISCA, 1990.
[8] K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy, M. D. Hill, “Programming for Different Memory Consistency Models.” *J. of Parallel and Distributed Computing*, 15(4), 1992.
[9] L.M. Censier and P. Feautrier, “A New Solution to Coherence Problems in Multicache Systems.” IEEE Trans. Computers, 27(12):1112-1118, Dec. 1978.
[10] Daniel Lenoski *et al*., “The Stanford DASH Multiprocessor.” *IEEE Computer*, Vol. 25 No. 3, pp. 63-79, March 1992.
[11] J. Laudon, D. Lenoski. “The SGI Origin: A cc-NUMA Highly Scalable Server.” 24th ISCA, June 1997.
[12] D. Lenoski and W.-D. Weber, “Scalable Shared-Memory Multiprocessing,” Morgan-Kaufmann Publishers, 1995.
[13] A. Agarwal, M. Horowitz and J. Hennessy, “An evaluation of Directory schemes for Cache Coherence.” *Proceedings of the 15th ISCA*, pp. 280-289, June 1988.
[14] S. Mukherjee and M.D. Hill “Using Prediction to Accelerate Coherence Protocols.” 25th ISCA, June-July 1998.
[15] S. Kaxiras and J.R. Goodman, “Improving CC-NUMA Performance Using Instruction-based Prediction.” 5th HPCA, Jan. 1999.
[16] S Kaxiras, C Young “Coherence communication prediction in shared-memory multiprocessors,” Proc. of the 6th IEEE Symp. on High-Performance Computer Architecture, 2000.
[17] P. Stenström, M. Brorsson, L. Sandberg, “An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing,” 20th ISCA, 1993.
[18] Wolf-Dietrich Weber and Anoop Gupta, “Analysis of Cache Invalidation Patterns in Multiprocessors.” *Proc. of the 3rd ASPLOS*, pp. 243-256, April 1989.
[19] Tor E. Jeremiassen, Susan J. Eggers “Reducing false sharing on shared memory multiprocessors through compile time data transformations” In Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1995.
[20] S Kaxiras, Z Hu, M Martonosi “Cache decay: exploiting generational behavior to reduce cache leakage power,” Proc. of the 28 ISCA, 2001.
[21] R Gonzalez, M Horowitz “Energy dissipation in general purpose microprocessors,” Solid-State Circuits, IEEE Journal of, 1996.
[22] Mark Hill, “What is Scalability?” Computer Architecture News, Volume 18 , Issue 4, December 1990.
[23] MMK Martin, MD Hill, DA Wood “Token Coherence: decoupling performance and correctness” Proceedings. 30th Annual International Symposium on Computer Architecture, 2003.
[24] N Eisley, LS Peh, L Shang, “In-Network Cache Coherence,” International Symposium on Microarchitecture 2006.
[25] S. Kaxiras and J. R. Goodman, “The glow cache coherence protocol extensions for widely shared data,” in Proc. 10th int. conf. Supercomputing, May 1996, pp. 35–43.
[26] H. Nilsson and P. Stenstr¨om, “The Scalable Tree Protocol – A Cache Coherence Approach for Large-Scale Multiprocessors,” in Proc. 4th IEEE Symp. Par. and Dist. Processing, Dec. 1992, pp. 498–506.
[27] D. Brooks, V. Tiwari, M. Martonosi, “Wattch: a framework for architectural-level power analysis and optimizations,” 27th ISCA, 2000.
[28] H-S.Wang, X. Zhu, L-S. Peh, S. Malik, “Orion: A Power-Performance Simulator for Interconnection Networks,” 35th MICRO’02, 2002.
[29] Milo M.K. Martin, et al., “Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset,” Computer Architecture News, September 2005.

Memory-Communication Model for Low-Latency X-ray Video Processing on Multiple Cores

A.H.R. Albers*[1,2], E.A.L. Suijs[2], and P.H.N. de With[1,3]

[1] Eindhoven University of Technology, PO Box 513, 5600 MB, The Netherlands,
[2] Philips Healthcare, X-Ray, PO Box 10.000, 5680 DA Best, The Netherlands,
[3] CycloMedia Technology, PO Box 68, 4180 BB Waardenburg, The Netherlands.

Abstract. Despite the speed up of PC technology over the years, real-time performance of video processing in medical X-ray procedures continues to be an issue as the size and number of concurrent data streams is increasing steadily. Since the computing evolves quicker than memory technology, there is an increasing pressure on an efficient use of the off-chip memory bandwidth. Additionally, as a multitude of video functions is carried out in parallel, the memory-bandwidth problem is further stressed. In this paper, we present an architecture study for performance prediction and optimization of medical X-ray video-processing on multiple cores. By carefully modeling the critical stages of the architecture, bottlenecks are known in detail. Model descriptions for the video-processing algorithms are inserted into the architecture model, making explicit where data and functions needs to be partitioned to obtain higher throughput. For the application under study, we propose a combined 2-level data partitioning with functional partitioning scheme that result in a bandwidth and latency reduction of 40-70% compared to straightforward implementations.

Key words: performance modeling, video processing, multi-core, cache aware, data partitioning, memory bandwidth.

1 Introduction

In Cardiovascular minimal invasive interventions, physicians require low-latency X-ray imaging applications, as their actions must be directly visible on the screen. The video-processing system should enable the simultaneous execution of a plurality of functions, based on stream-oriented processing and data-dependent irregular processing. Because dedicated hardware platforms suffer from high investment costs and lack flexibility and reuse, there is an increasing interest in using off-the-shelf computer technology as a platform for real-time video processing. Simultaneously, algorithms show a continuous increase in complexity and the resolution of images is growing. These trends lead to a considerable increase of the required computation power and communication bandwidth between computation and memory in the signal processing.

The past years have shown that the discrepancy between processor and memory performance is rapidly increasing, making memory access and communication bandwidth the main bottleneck for many applications that access large amounts of data.

* a.h.r.albers@tue.nl; http://vca.ele.tue.nl

An X-ray image data stream can, for example, already have dimensions up to 2K×2K (16 bits/pixel) and frame speeds up to 60 Hz. Full storage of 2K×2K X-ray images is not possible inside the cache memory of the processor, thereby relying on a background off-chip memory. Since the computation technology evolves quicker than memory technology, there is an increasing pressure on an efficient use of the off-chip memory bandwidth [1]. Additionally, in many X-ray procedures, a multitude of imaging functions is carried out in parallel, which further stresses the memory-bandwidth problem. In this paper, we present an architecture study and focus on performance optimization methods for video-processing applications deployed on off-the-shelf chip-level multiprocessor systems.

A chip-level multiprocessor (CMP) has multiple cores on the same chip in a shared-memory configuration. Extracting performance from such a configuration requires the effective use of memory and cache performance. Beyls [2] described several optimization methods for better cache behavior and Jaspers [3] discussed a specific caching technique for increase of the performance of streaming video data. The performance optimization method, described in this paper, is based on the data-dependent storage organization of [3], but more generic in the sense that data is not reorganized to facilitate a detailed mapping. Instead, we optimize the data partitioning and distribution over the cache size without the need for specific data storage formats.

Our approach is to carefully model the critical stages of the communication and memory architecture, so that bottlenecks are known in detail. Application-specific knowledge is covered by developing performance-estimation functions of the applied signal-processing functions. In an earlier paper [4], we have presented the modeling and design of an 1K×1K X-ray image-processing chain. Here, we extend this work with a solution for 2K×2K images and we validate the memory-communication model on a commercially available multi-core PC platform.

This paper is organized as follows. Section 2 presents related work. Subsequently, Section 3 introduces the architecture and application performance design. Section 4 presents the execution architecture including mapping and partitioning. In Section 5, experimental results are shown, combined with the validation of the prediction models. The last section gives conclusions.

2 Related Work

In the literature, several architecture-based approaches [5,6,7,8] aim at parallelizing chains of real-time video-processing functions on multiprocessor systems. Several proposals [5,6] describe a parallel function-level partitioning of tasks, whereas other approaches [7,8] deal with parallel data-level partitioning. As the techniques rely on generic approaches for improving the processing architecture, they do not consider detailed knowledge of the application for improved performance.

An extensive list of papers deal with the optimization and parallelization of image and video-processing applications in software, aiming at a real-time performance. To name a few, in [9], parallelization strategies are described for H.264 decoding. Similar approaches can be found for the Hough Transform [10], AdaBoost [11], and motion-

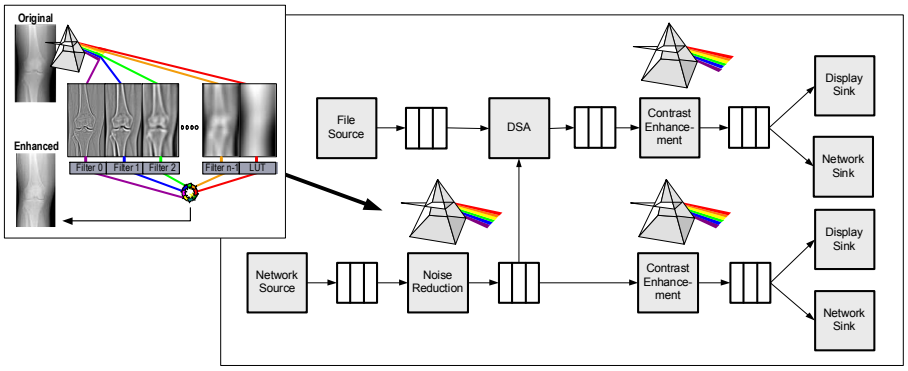


Fig. 1. Block diagram of the video-processing application.

compensated filtering [12]. As these papers describe methods to optimize performance by modeling the application in isolation, full system optimization remains a challenge.

Within performance computing, several techniques are reported in the literature for performance prediction of parallel applications [13][14][15]. Most of them are based on block-parallel or dataflow-like approaches. However, they rarely provide insights into how to improve the application performance.

In this paper, we study a mixture of the previous features and new aspects. Our approach is to carefully model the critical stages of the communication and memory architecture, so that bottlenecks are know in detail. The model allows us to compute the performance of the platform under the conditions of actual memory and bus-access parameters. By combining this with application-dependent data and functional partitioning, a more advanced approach for efficient mapping of the application is achieved.

3 Architecture and Application Performance Design

We describe a performance-design method, used in this paper, which is based on execution architecture development [16] and the Y-chart methodology [17]. The Y-chart methodology recognizes a clear separation between the application model, the platform architecture and an explicit mapping step, covering the execution architecture. The decoupling of application, architecture and execution models allow designers to exercise HW/SW mappings for different kind of optimization strategies.

3.1 Application model

One of the main devices in a Cardiovascular catheterization labs is the X-ray imaging system. Image quality is a critical system performance factor. Because X-ray radiation is not without harm for the human body, the system tries to deliver the best image quality at the lowest possible radiation dose. Therefore, advanced de-noising and contrast-enhancement techniques are employed to maintain an acceptable signal-to-noise ratio.

Because physicians must see their actions directly on the screen (eye-hand coordination), a low latency is a key requirement for the imaging application.

We have formed an advanced X-ray video-processing chain, consisting of two pipelines of processing algorithms. Fig. 1 provides an overview of the algorithm chain. We assume a double-pipeline processing approach, consisting of a combination of real-time imaging (fluoroscopy) and subtracted angiography. Spatio-temporal noise-reduction filtering takes advantage of a multi-resolution framework. At every scale, in local regions, a directional filter kernel is generated to preserve the local edges while removing noise. For temporal filtering, a recursive filter is used. Digital Subtraction Angiography (DSA) is an imaging technique that can show the contrast-filled vessels in the absence of an interfering anatomic background (e.g. bones and soft tissue). A contrast-enhancement function is applied before the images are sent to the display and network output. For application modeling, all processing functions are analyzed with respect to their memory, computation and communication primitives. Table 1 gives an overview of the requirements for each of the processing algorithms. The amount of computation is derived from micro-benchmarking the application software in isolation, with a data set size small enough to fit the level one (L1) cache memory of the processor, to avoid biased results. Memory neighborhood and communication bandwidth requirements are derived from the algorithm specifications. For the experiments, we employ a resolution of 2048×2048 pixels at 15 Hz. For a detailed application analysis, the reader is referred to [18].

3.2 Platform Architecture model

This section describes the creation of a model for platform architectures, based on general-purpose building blocks. The architecture used in the experiments is an off-the-shelf chip-level multiprocessor system[a]. We collect the critical performance and timing parameters by micro-benchmarking the five primary parts of the architecture (See Fig 2(a)). In order to describe the behavior of the memory and cache in more detail, Hristea [19] defined the memory performance metrics (1) 'Restart latency' and (2) 'Back-to-back latency', which will be used later.

[a] The system is commercially available as Xeon E5345, 2.33 GHz, 4 GB RAM.

Function	Operations	Computations	Memory	Communication Bandwidth
Network Source	Packet processing	5 cycles / pixel	Packet Size	2x Read + 2x Write
Gauss/Laplacian Pyramid	Filtering, up/downsample	10 cycles / pixel	40x40 up to 320x320 pixels	(3x Read + 3x Write) * 1.33
Spatial Filter	Structure-adaptive filtering	100 cycles / pixel	5x5 pixels	(3x Read + 3x Write) * 1.33
Temporal Filter	Recursive filtering	10 cycles / pixel	3x3 pixels, previous image	(2x Read + 1x Write) * 1.33
Contrast Enhancement	LUT processing	10 cycles / pixel	2 x 2^bits LUT	(2x Read + 1x Write) * 1.33
Automatic Pixel-shift	Parametric motion estimation	25 cycles / pixel	128x128 pixels, mask image	2x Read + 0x Write
Image Subtraction	Warping, Subtraction	5 cycles / pixel	2 x 2^bits LUT	4x Read + 3x Write
Network Sink	Packet processing	5 cycles / pixel	Packet Size	2x Read + 2x Write
Display Sink	Display texture processing	1 cycle / pixel	Image Size	1x Read + 1x Write

Table 1. Requirements for the set of processing functions under study.

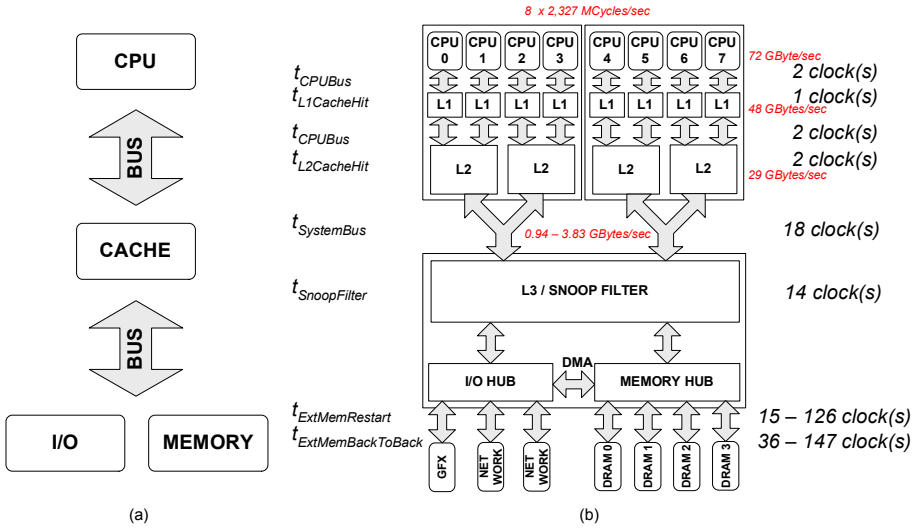


Fig. 2. Generic architecture model (a), instantiated architecture with parameters (b).

When multiple processors are allowed to maintain local copies of shared-memory locations, a cache-coherence protocol is required to ensure that the processors have consistent views of the memory contents. Every time a processor has a cache miss, a coherency protocol broadcasts requests and receives all the responses prior to obtaining the necessary data. A snoop filter is included to avoid most of the expensive cache-coherence traffic on the system buses. The latency penalty of cache-coherence traffic is defined as $t_{CacheCoherence}$ and snoop filtering as $t_{SnoopFilter}$. The parameters are derived from hardware specifications. Large-scale parallel applications inevitably induce contention when executed on shared-memory multiprocessors. Contention occurs if several processes simultaneously try to access a shared hardware resource. Stochastic models of interference among processors in multiprocessor systems can be found in [20]. In our model, we include the time fraction for bus and memory contention and classify this into three levels (A = low, B = medium, C = high).

We will now construct the equations in the form of timing formulas to involve the above-mentioned factors. Let us begin with the definition of the average memory access time, which is an important performance parameter in our model. For a two-level cache, taken from literature [21], this can be given by

$$AverageMemoryAccessTime = t_{L1CacheHit} + MissRate_{L1} \times t_{L1CacheMiss}. \quad (1)$$

Similarly, an L1 cache miss is defined by:

$$t_{L1CacheMiss} = t_{CPUBus} + t_{L2CacheHit} + MissRate_{L2} \times t_{L2CacheMiss}. \quad (2)$$

The latency penalty for an L2 Cache Miss to external memory can be stated more precisely as:

$$t_{L2CacheMiss} = \max(t_{CacheCoherence}, t_{SnoopFilter}) + t_{ExtMemRestart} + t_{SystemBus}. \quad (3)$$

Component	Parameter	Value	Unity	Component	Parameter	Value	Unity
Central Processing Unit	Clock Speed	2327	MHz	System Bus	Clock Speed	333	MHz
	Bus Width	32	Bytes		Address Bus Width	4	Bytes
	Bandwidth	72	GByte/sec		Data Bus Width	8	Bytes
	t_{CPUBus}	2	clock tick(s)		Pipelined Bandwidth	3.83 (A) / 3.24 (B) / 0.94 (C)	GByte/sec
Level 1 Cache	Clock Speed	2327	MHz		$t_{AddressBus}$	0.5	clock tick(s)
	Capacity	32	KBytes		$t_{DataBus}$	2	clock tick(s)
	Bandwidth	48	GByte/sec		$t_{SnoopReply}$	1	clock tick(s)
	$t_{L1CacheHit}$	1	clock tick(s)		$t_{Cache2CacheL2}$	80 (A) / 110 (B) / 200 (C)	clock tick(s)
Level 2 Cache	Clock Speed	2327	MHz		Contention	low (A) / med. (B) / high (C)	
	Capacity	4096	KBytes	Memory Hub	Clock Speed	667	MHz
	Bandwidth	29	GByte/sec		Max. Read Bandwidth	20.8	GByte/sec
	$t_{L2CacheHit}$	2	clock tick(s)		Max. Write Bandwidth	10.4	GByte/sec
	$t_{L2CacheMiss}$	40 (A) / 70 (B) / 160 (C)	clock tick(s)		$t_{MemBuffer}$	3.2 (A) / 12.8 (B) / 51.1 (C)	ns
Level 3 Cache	Clock Speed	667	MHz		$t_{DRAMRestart}$	3	ns
(Snoop Filter)	Capacity	8192	KBytes		$t_{DRAMBackToBack}$	12	ns
	$t_{SnoopFilter}$	4 / 8	clock tick(s)		Contention	low (A) / med. (B) / high (C)	

Table 2. Parameters extracted from specifications and micro-benchmark experiments.

Cache-miss latency due to cache lines resident in other processor caches is defined as the Cache-to-Cache latency (Cache2Cache) [22]. For L2 cache-line transfers, this leads to the following equation

$$t_{Cache2CacheL2} = t_{L2CacheHit} + \max(t_{CacheCoherence}, t_{SnoopFilter}) + 1.5^{\text{b}} \times t_{SystemBus} + t_{L2CacheMiss}{}^{\text{c}}. \quad (4)$$

Formula parameters (See Table 2) are derived from the specifications and micro benchmarking at several levels in the architecture following the approaches from literature [23,19,24,22,25]. The above equations form the platform-architecture model. For the experiments, we used a dual-socket, quad-core, processor architecture with a two-level cache, denoted as Level-1 (L1) and Level-2 (L2). In Fig. 2(b), the processor architecture is shown. In total, the system consists of 8 processors of 2.33 GCycles/s, 8 L1 caches of 32 KB and 4 L2 caches of 4 MB. The architecture contains a snoop filter (8 MB) to avoid most of the expensive cache-coherence traffic on the system buses. The system is equipped with 4 GB of FB-DIMM memory. For more details about the instantiated architecture, we refer to literature [26]. As our model is not restricted to architectures based on general-purpose CPU cores, in principle, graphics-processor units (GPUs) can be described in the same way. However, the careful modeling of the critical parts of the architecture and micro-benchmarking has to be redone.

4 Execution Architecture

In the previous section, we have carefully modeled the video-processing application on a DSP native level. Furthermore, the performance characteristics and possible bottlenecks in the platform architecture are described in detail, by modeling the critical stages of the communication and memory architecture. Our modeling is based on a double pipeline X-ray application. We use an optimization strategy to match the size of

[b] Half of the cache-line transfer is send in parallel on the two system buses.

[c] If the requested data is modified, it has to be written back to external memory first, adding an $t_{L2CacheMiss}$ delay to the overall latency.

the data partitions and the required computing to minimize communication bandwidth to external memory. This approach should maximize throughput while preserving a low latency. Performance and resource budgets are used to guide the optimization.

4.1 Budgeting

Next, the computation, memory and bandwidth budgets are analyzed for the application under study for each individual processing step. Our main requirement is low latency, so the communication has to be always fluent without severe interruptions. Let us start with the calculation of required budgets for computation, memory and bandwidth, following the application as described in detail in Section 3.1. It consists of 10 functions. Three functions are build around a multi-resolution image pyramid. The computation budget for the application on a per pixel basis (following the dataflow from Fig. 1 and the requirements from Table 1) is defined by:

5+1.33×(10+100+10)+(25+5)+ 2×(1.33×(10+10)+5+1) = 260 (cycles/pixel) .

For our experiments (2048×2048 pixels; 15Hz), the required computing budget is 16.36 GCycles/s. The required memory budget per pixel when counting only write actions (Table 1) can be computed from:

2+1.33×(3+3+2)+(2+3)+2×(1.33×(3+1)+2+1) = 34.3 $\cdot N_{pixels} \cdot bytes/pixel$ (bytes).

Without data partitioning, the required memory budget per image becomes 274.4 MB in our experiments. In the same way, the communication bandwidth budget per pixel counting all input-output transfers (Table 1) is specified by:

4+1.33×15+9+2×(1.33×9+4+2) = 68.9$\cdot N_{pixels} \cdot bytes/pixel$ (bytes).

Without data partitioning, a communication bandwidth budget is required of 8.27 GB/s.

4.2 Mapping and partitioning for low latency

In general, there are two ways to partition the application over a multiprocessor or multicore environment, namely *functional partitioning* and *data partitioning* [27]. With functional partitioning, a function is decomposed into a set of independent tasks, which can be executed in parallel on the architecture. Data partitioning implies the division of an image into sub-images (stripes, blocks or lines) and then process each partition on a single processor. Our approach starts with inspecting the required computation, memory and bandwidth budgets for the application and apply data partitioning on the available resources of the platform architecture. In our experiments, a minimum of 7 processor cores is required for execution.

With the above approach, the memory requirements for some of the processing functions will exceed the available cache resources of the platform, causing the function to communicate internally via the slower external memory. The performance of the application will be hampered by the latency penalty of retrieving data from external

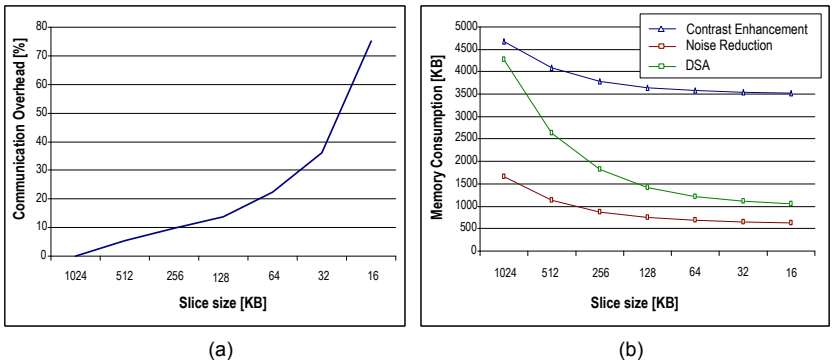


Fig. 3. Slice size vs. (a), communication overhead and (b), total memory consumption.

memory due to the limited cache capacity[d]. As a possible solution, we will look for combinations of functional partitioning and data partitioning in the following way.

1. *Intra-function partitioning*. We partition the image block into smaller slices for each function locally. We have to take care that the minimum memory requirement of each function remains available at the input. The input and output data-block granularity remain unchanged so no additional communication is introduced.
2. *Inter-function partitioning*. We further divide the globally partitioned image blocks into smaller slices. In order to decide what the optimal division is of the image blocks into slices, we investigate the communication time overhead versus the data-block size to argue about the optimal data granularity. The results are shown in Figure 3.
3. *Verify locality of data against cache size*. The memory consumption, resulting from the previous step is accumulated from different tasks and compared to the available cache size. If there is an overflow, the previous step is repeated until the accumulated task-memory consumption fits in the cache, or the slice reached the minimum required size for correct operation.

As can be noticed from Figure 3(a), choosing a very small slice size is not preferred, since the communication (cache coherence) overhead percentage will start to increase exponentially. This is further explained in [22]. From Figure 3(b), one can see that the total memory consumption of the video-processing functions decreases only marginally for slice sizes smaller than 128 KB. This can be explained by the minimum required amount of input and intermediate data that is required for each function. We therefore conclude that with the additional decomposition of the image blocks into slices, the optimal slice size should be chosen such that the cache memory usage is maximized with respect to capacity without task partitioning (cache-aware partitioning). As a con-

[d] Apart from the cache capacity, cache conflict misses [21] can also contribute to the overall performance. For our case, they contribute only on a very small scale to the overall number of cache misses.

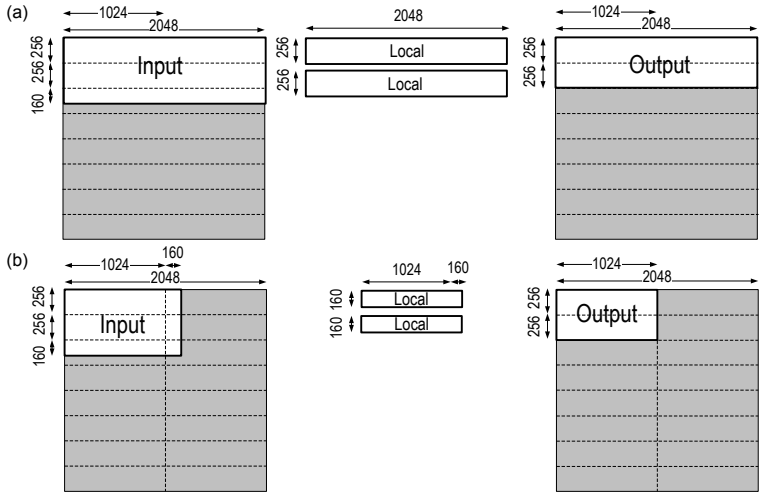


Fig. 4. Memory communication for straightforward data partitioning (a) and cache-aware algorithmic data partitioning (b) (numbers apply for contrast enhancement).

sequence, when observing the cache contents at a given time moment, the amount of tasks contained is minimized, leading to lower bandwidth requirements.

5 Experimental Results

In this paper, we explore a case, with 2K×2K images mapped onto a multi-core platform using 8 cores in parallel. The data is processed by a noise-reduction function and in two branches contrast enhancement is applied using pyramidal processing, and further quality improvement measures. This solution has been compared to a straightforward data-parallel mapping, where image stripes of $2,048 \times 256$ pixels are used at all stages of the processing. In Fig. 4, the memory communication is shown for the two mappings. As a first step, the image is partitioned into stripes to map the computation budget onto the available processor cores. Storage of extra required filter neighborhood is only needed for image parts that are not resident in the same L2 cache (cache-aware partitioning). Subsequently, intra-function partitioning is applied for each function locally,

Approach	Memory consumption / L2 Cache (of 4,096 KB)			Slice Size [kB]	Slices / Image
	Min. [KB]	Average [KB]	Max. [KB]		
Straightforward data partitioning	11,668	19,136	24,164	1,024	8
Cache-aware partitioning (iter. 1)	1,160	3,538	4,683	1,024	8
Cache-aware partitioning (iter. 2)	1,138	2,622	4,091	512	16

Table 3. Memory requirements for the analyzed partitioning strategies.

Processor	Function	Computation Latency [ms]	I/O Latency [ms]	Bandwidth [GB/s]	Stall Latency [ms]			Extra Bandwidth [GB/s]
		-1-	-2-	-3-	-4a-	-4b-	-4c-	-5-
4	Network Source	1.1	0.8	0.59				
0 - 7	Noise Reduction	35.5	1.1	0.20	11.6	19.3	43.7	2.30
0 - 7	DSA	6.8	1.3	0.23	4.4	7.3	16.5	0.87
0 - 7	Contrast Enhancement	6.0	1.3	0.23	10.1	16.8	16.8	2.00
4	Network Sink	1.1	1.1	0.59				
4	Display Sink	0.9	3.3	0.12				
0 - 7	Contrast Enhancement	6.0	1.3	0.23	10.1	16.8	16.8	2.00
4	Network Sink	1.1	1.1	0.59				
4	Display Sink	0.9	3.3	0.12				
Latency	(1 + 2) + (4a or 4b or 4c)	57.4	10.2		36.1	60.1	93.7	
Bandwidth	(3 + 5)			2.89				7.16

Table 4. Simulated latency and bandwidth figures for the video-processing under study (measured values differ within 10-20%).

implemented as a sliding window buffer, to avoid the still occurring cache overflow. An overview of the memory requirements can be found in Table 3. After the first iteration of the proposed cache-aware memory-communication model, some of the functions still overflow the L2 cache. By applying a second inter-function partitioning for those functions, the cache overflow is almost negligible[e]. Compared to a straightforward data-partitioning approach, the memory requirements of the new approach are an order of an magnitude smaller. In Table 4, the results are shown for the video-processing application under study (2K$\times$2K images; 15Hz). The difference between the performance prediction and measured performance is within 10-20% in all cases.
The following rules have been adopted to find the presented results:

$$\begin{aligned}
&\text{Computation latency} = (\text{Comp. Budget} \times \text{Block Size}) \,/\, \text{Resource Budget}\\
&\text{I/O latency} = (\text{Input+Output Block Size}) \times 16 \times (t_{L2CacheMiss} \text{ or } t_{Cache2CacheL2})\\
&\text{Stall latency} = (\text{Input+Inter+Output Bl Size} - \text{L2-Cache Cap.}) \times 16 \times t_{L2CacheMiss}\\
&\text{Bandwidth} = 2 \times \sum_{\forall Datablocks} (\text{Block Size} \times \text{Frame rate})\\
&\text{Extra bandwidth} = 2 \times \sum_{\forall Datablocks} ((\text{Input+Inter+Output Bl Size} - \text{L2-Cache Cap.}) \times \text{Fr rate})
\end{aligned}$$

Results show a memory bandwidth for cache-aware algorithmic data partitioning of 2.89 GB/s, as nearly all intermediate communication is arranged locally via the local cache memory. For straightforward data partitioning, this is not the case and the bandwidth requirements increase with 7.16 GB/s, to a total bandwidth of 10 GB/s, effectively. The total latency of the video-processing application for cache-aware algorithmic data partitioning is 57.4 + 10.2 = 67.6 ms. For straightforward data partitioning, our modeling states that between 36.1 and 93.7 ms are added due to stalled CPU cycles where the processor is waiting for memory. In practice, the instantiated platform architecture is unable to cope with this high bandwidth requirements, and the system completely stalls.

[e] A small amount of cache memory is occupied by instructions and scalar variables.

6 Conclusions

We have presented an application model at a DSP native level for a real-time medical video processing application involving multi-resolution decomposition, noise reduction and image enhancement. Similarly, we have developed a time specification model for the memory communication and bandwidth usage of a general-purpose multi-core processor architecture. This model allows us to compute the performance of the platform under the conditions of actual memory and bus-access parameters.

In a performance design where we employed the above models, we have focussed on bandwidth reduction and latency optimization. Because of the memory-rich processing algorithms, straightforward data partitioning is unable to exploit a full locality of data. This leads to substantial inefficiencies in performance and waste of memory bandwidth. Therefore, we have proposed a combination of data partitioning and functional partitioning. As a result, a substantial performance improvement is realized by a memory-communication model that incorporates the memory and compute requirements of the video processing in two ways.

1. An inter-function partitioning is chosen to satisfy the required compute resources across the processor cores and optimized for communication between functions.
2. An intra-function partitioning is chosen for each function separately, so that caching will be exploited for low latency.

We have validated the memory communication model on a multi-core processor platform processing 2K$\times$2K images in real time at 15 Hz. Results show a considerable memory-bandwidth reduction of 70% and a latency reduction of 40-70%, compared to straightforward data-parallel implementations. The difference between the performance prediction and measured performance is within 10-20% in all cases. The proposed techniques have proven to be valuable for regular signal processing functions, such as filtering and noise reduction. However, if processing becomes more irregular, such as in analysis, the proposed partitioning algorithm has to become more adaptive and flexible to preserve the efficiency gain.

References

1. McCalpin, J., Moore, C., Hester, P.: The role of multicore processors in the evolution of general-purpose computing. CTWatch Quarterly **3**(1) (Febr. 2007) 18–30
2. Beyls, K.: Software Methods to Improve Data Locality and Cache Behavior. PhD thesis, University of Gent, Belgium (Jun. 2004)
3. Jaspers, E.: Architecture design of video processing systems on a chip. PhD thesis, Eindhoven University of Technology, The Netherlands (Apr. 2003)
4. Albers, R., Suijs, E., de With, P.H.N.: Optimization model for memory bandwidth usage in x-ray image enhancement. In: Electronic Imaging, SPIE. Volume 6811. (2008)
5. Seinstra, F.: User Transparent Parallel Image Processing. PhD thesis, Universiteit van Amsterdam, The Netherlands (May 2003)
6. Farin, D., de With, P.H.N.: A generic software-framework for distributed, high-performance processing of multi-view video. In: Electronic Imaging, SPIE. Volume 6496. (2007)
7. Florent, R., Mequio, C.: US Patent Appl. 2003/0026505: software system for deploying image processing functions on a programmable platform of distributed processor environments

8. Steinsaltz, Y., Geaghan, S., Prelle, M.J., Bouzas, B.: Leveraging multicomputer frameworks for use in multi-core processors. In: HPEC Workshop. (2006)
9. Meenderinck, C., Azevedo, A., Juurlink, B., Alvarez, M., Ramirez, A.: Parallel scalability of video decoders. Journal of Signal Processing Systems (August 2008)
10. Chen, Y.K., Li, W., Li, J., Wang, T.: Novel parallel hough transform on multi-core processors. ICASSP, IEEE International Conference (April 2008) 1457–1460
11. Chen, Y.K., Li, W., Tong, X.: Parallelization of adaboost algorithm on multi-core processors. Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on (Oct. 2008) 275–280
12. Groth, A., Eck, K.: Real-time implementation of a multiresolution motion-compensating temporal filter on general-purpose hardware. In: Real-Time Imaging IX. SPIE. (2005)
13. Black-Schaffer, D.: Block Parallel Programming for Real-Time Applications on Multi-core Processors. PhD thesis (April 2008)
14. Thiele, L., Wandeler, E., Chakraborty, S.: Performance analysis of multiprocessor dsps. Signal Proc., IEEE **22**(3) (2005)
15. Gautama, H., van Gemund, A.: Performance prediction of data-dependent task parallel programs. In: Euro-Par. Volume 2150 of LNCS. (2001) 106–116
16. Muller, G.: An incremental execution architecture design approach (2007)
17. Erbas, C., *et al.*: A framework for system-level modeling and simulation of embedded systems architectures. EURASIP J. Embedded Syst. (1) (2007)
18. Albers, R., Boosten, M., de With, P.H.N.: Options for new real-time image-processing architectures in cardiovascular systems. In: Medical Imaging 2007, SPIE. Volume 6509. (2007)
19. Hristea, C., Lenoski, D., Keen, J.: Measuring memory hierarchy performance of cache-coherent multiprocessors using micro benchmarks. In: SC, Proc. of conf., ACM (1997)
20. Marsan, M.A., Balbo, G., Conte, G., Gregoretti, F.: Modeling bus contention and memory interference in a multiprocessor system. IEEE Trans. Comput. **32**(1) (1983) 60–72
21. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2003)
22. Vadlamani, S., Jenks, S.: Architectural considerations for efficient software execution on parallel microprocessors. IEEE IPDPS Symposium (2007) 1–10
23. Jin, R., Agrawal, G.: A methodology for detailed performance modeling of reduction computations on smp machines. Perform. Eval. **60**(1-4) (2005) 73–105
24. Noordergraaf, L., van der Pas, R.: Performance experiences on sun's wildfire prototype. In: Proc. of 1999 ACM/IEEE conf. on Supercomputing, New York, NY, USA, ACM (1999) 38
25. McVoy, L., Staelin, C.: lmbench: portable tools for performance analysis. In: ATEC'96: Proc. Annual Technical Conf. on USENIX 1996, USA (1996) 23–23
26. Radhakrishnan, S., Chinthamani, S., Cheng, K.: The blackford northbridge chipset for the intel 5000. Micro, IEEE **27**(2) (March-April 2007) 22–33
27. van der Tol, E.B., Jaspers, E.G., Gelderblom, R.H.: Mapping of h.264 decoding on a multiprocessor architecture. Volume 5022., SPIE (2003) 707–718

A Dual Mesh Tiled CMP

Preethi Sam, Matthew J. Horsnell, and Ian Watson

University of Manchester,
Department of Computer Science, Oxford Road, Manchester M13 9PL
{samp,horsnelm,iwatson}@cs.man.ac.uk
http://intranet.cs.man.ac.uk/apt/projects/jamaica/

Abstract. The era of a billion and more transistors on a single silicon chip has already begun and this has changed the direction of future computing towards building chip multiprocessors (CMP) systems. Nevertheless the challenges of maintaining cache coherency as well as providing scalability on CMPs is still in its initial stages of development. Previous studies have shown that single bus based cache coherent CMPs do not scale. Directory based CMP systems provide better scalability compared to snoop based protocols, but have overhead in terms of the space for a full map directory as well as high latency during broadcasting of writes to widely shared data. In this paper we explore the scalability of a cache coherent Tiled CMP with dual mesh networks, using a combination of snoop and limited directory based cache coherency protocol. A limited directory based scheme with low area overhead is used over one mesh network for handling all requests and non-broadcast based cache coherency responses. The second mesh network acts like a broadcast tree and is specifically used for supporting broadcast based invalidations to widely shared data. The cache coherency protocol is optimized by removing the need to generate acknowledge messages during writes to widely shared data. Scalability of the architecture is presented in terms of the speed up obtained by running multithreaded JavaGrande benchmarks on the system for up to 64 processors. The system is compared against a perfect memory model which quantifies the inherent parallelism within the benchmark. The worst case speed up for a 64 processor Tiled CMP, with inherently scalable benchmarks, is 62% of the ideal speed up of 64, and 78% of the speed up w.r.t the perfect memory model.

Key words: Tiled CMP, cache coherency

1 Introduction

The transition from uniprocessor to chip multiprocessor (CMP) systems has already been made for both high end server as well as desktop machines, as seen in Sun's Niagara, and Intel and AMD's quad and dual core systems. A major difference between uniprocessor and multiprocessor systems is that uniprocessors do not need to address the issue of cache coherency. However, CMP systems are in essence, architecturally, a smaller version of cache coherent multiprocessor

systems. Since CMPs are targeted towards applications that will most likely use the shared memory programming paradigm there is a need for a cache coherency protocol to be implemented over such systems. Apart from handling cache coherency, CMP systems should be scalable (in terms of number of processor cores on a single chip) in order to achieve faster execution speeds in parallel applications. It is a known fact that bus based CMP systems do not scale beyond a small number of processors [1]. Therefore, alternative switch based interconnects topologies are becoming popular with many CMP systems. One of the most popular switched interconnect is the mesh topology that is being used to integrate multiple processor cores (tiles) on a single chip [2][3]. Nowadays, the abundant on-chip wire and transistor resources [2] have resulted in CMP systems that implement processing logic in a tile fashion and interconnect them using some symmetric network for simplification of the layout on chip. These CMP systems are known as Tiled CMPs. However, the bigger challenge is to maintain cache coherency on such unordered networks and at the same time achieve application scalability. Therefore in this paper we explore the scalability of cache coherent tile based CMP system using multiple networks. The cache coherency uses a combination of both snoop and limited directory based protocols. The main idea is to use a limited directory based protocol without the overhead of storage space for tracking the list of sharers as well as reducing the latency during broadcasts to widely shared data.

2 A Dual Mesh Tiled CMP

We present a Tiled CMP architecture, with two mesh networks as shown in Figure 1. Tiles within the dual mesh CMP can be either processor or memory (L2) tiles. One of the mesh networks serves as an broadcast tree with the L2 tile as the root node. A mesh based invalidation network provides higher bandwidth and supports more processors as compared to a single bus based network [4]. Also, it does not suffer from the wire delay problems that are associated with a single bus [5]. The other mesh network implements a limited directory protocol. Processor tiles consist of the multithreaded JAMAICA processor core [6], which is a 5-stage in order pipeline with private L1 Instruction and Data caches. The memory tile is part of a Non-Uniform Cache Access (NUCA) L2 cache.

2.1 Tile-Network Interface

The tile interfaces to the mesh network using four ports - North, East, West and South (NEWS) [1]. Each port consists of a master and slave handler, referred to as port master and port slave, respectively. Port slaves receive and process packets while masters transmit packets. Processor tiles contain a single port master and slave pair at each port, while L2 tiles contain 8 port masters and slaves per port. The routing unit within a tile uses the dimension order routing protocol for determining the next tile that receives the packet enroute to its destination [4]. The packet switching scheme at each port is store and forward,

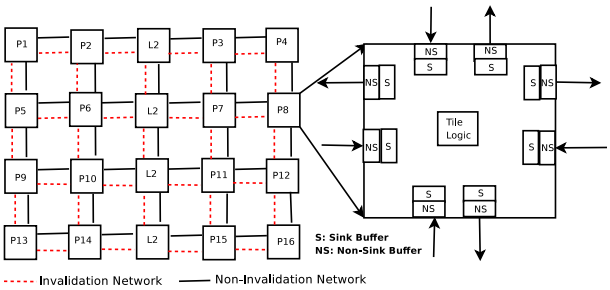


Fig. 1. Dual Mesh CMP

i.e. the complete message is buffered before it is sent out of the port master [4]. The buffer width at each of the masters and slaves is equal to one cache line width (256 bits) plus the header packet size (64 bits). The link width connecting two adjacent tiles is 160 bits. Request messages need a single cycle, whilst response messages take two cycles for transmission of messages between adjacent nodes on the mesh network. Each port slave is equipped with its own finite state machine (FSM) based control logic to *fetch*, *decode* and *forward* the packet. The FSM at the port master is used for buffering and transmitting the packet.

2.2 Cache Line States

The L1 cache line can be in the following states: Modified (M), Owned (O), Shared (S) and Invalid (I). The O state implies that the line is shared between the instruction and data caches on the same tile. Lines in L2 can be Exclusive (E), S, M, Pending (P) and I states. The P state implies that the L2 does not service any other request for that cache line address until it receives a data or acknowledge message for the same line. The S state in L2 can be further categorized as either Shared Single Owner S(SO) or Shared Multiple Owner S(MO). The S(SO) state implies that the line exists in the S state within a single L1 cache. On the contrary, the S(MO) state implies that the line is shared by multiple L1 caches. M state in L2 implies that the data at L2 is stale. Apart from state, the L2 cache line also contains the following fields: ownerid - field that stores the processor identifier (id) that requests the cache line first or has the line in M state; L bit to detect if a lock has been taken on that cache line; and F bit to indicate that the line is shared by two or more processors, i.e. this flag is used to decide if a broadcast needs to occur on a write to this line.

The L1 cache uses the Least Recently Used (LRU) cache eviction policy, while the L2 evicts a line based on the cache line state, L bit status and the LRU count. Lines with the L bits set are not preferred for eviction. Among the different cache states, lines with states such as E (first choice) or S are preferred for eviction compared to M lines. The LRU policy is used when there are multiple lines with E or S states. The reason for choosing E or S lines over M lines for

eviction is because the L2 need not initiate a writeback transfer from the L1 cache for a modified line.

2.3 Deadlock Avoidance

In order to avoid deadlock because of buffer space constraints at the memory and processor tiles, virtual channels are provided (sink channels) at each port master and slave [7][1]. The sink channels service cache coherent requests or responses that are deemed as critical by the cache coherency protocol. The sink channel is a non-blocking buffer that accepts critical responses and guarantees consumption of these messages. Both sink and non-sink channels share the physical link that is attached to the port. In this system, responses that cause the L2 to transit from the P state to any other state are considered critical. This is because, the L2 waits on such responses, stalling requests for the same address until the response arrives. Apart from these critical messages, responses generated by L2 are also considered critical. Again, this is because, requests (for the same address) that are forwarded by the L2 (e.g. read to a M line) after the response messages for the previous request (write caused the L2 cache to transit to M state) was sent out, will cause the L2 to transit to P state.

2.4 Cache Coherency Protocol

The cache coherency protocol uses a limited directory protocol (with buffer space provided to store the id of a single processor) at the L2 for read and write requests to non-widely shared data, while the snoop protocol is used solely for broadcast purposes. All L1 requests are sent to the appropriate L2 cache bank based on a static address mapping policy. The various requests that are generated by the L1/L2 caches can be classified in general as Read Misses, Write Misses, Writebacks and Evictions. The cache state transition diagrams for L1 and L2 caches is shown in Figures 2 & 3, respectively.

Read Miss. L1 read misses or requests (RD_SH) generate read responses (RD_SH_RESP), if the line is in E or S(SO) or S(MO) state in the L2. On read to E state, the processor id of the requestor is stored in the ownerid field of the L2 and the state transits to S(SO). On a read to S(SO) line, the L2 sets the F bit, indicating that the line is in S(MO) state. Lines accessed from main memory by the L2, for the first time, are always set to the E state. If the line is in the M state at the L2, the request is forwarded to the processor dictated by the ownerid field within the L2 cache line. The L2 transits to P state on such a request and waits for an acknowledgement (ACK) message from the requestor. L2 does not service any request for the same cache line address until the ACK is received. On receiving the ACK, the L2 transits to S(MO) state. L1 requestor always transits to the S state on receiving a read response.

Write Miss. Write requests from the L1 are of two types, write misses (RD_EX) and upgrades (UP). Write misses occur on a L1 cache miss, while upgrades occur when a write hits on a S line in L1. On a write miss to E lines in L2, a write miss response (RD_EX_RESP) is generated. If the line is in S(SO) or M state at the L2, the write miss is forwarded to the owner and the L2 enters the P state. The owner responds with a RD_EX_RESP to the requestor and invalidates its own cache line. Until the requestor sends an ACK to the L2, the L2 remains in P state. On a write miss to S(MO) lines, a broadcast invalidation (B_INV) message is sent over the invalidation mesh network (a tree based broadcast with the L2 as the root node) and the L2 generates a RD_EX_RESP to the requestor. On an UP, the L2 generates either an upgrade response (UP_RESP), if the line is in S(SO) or a B_INV message if the line is in S(MO) state. Note, on a broadcast invalidation, no ACK messages are generated by the receivers. On all writes the L2 transits to the M state and sets the ownerid field equal to the processor id of the requestor. L1 requestor always transits to the M state.

Writeback and Eviction. If the L1 evicts an M line, a writeback (WB) message is generated to the L2. On receiving a WB message the L2 transits to E state, clears the F flag and the ownerid field. L1 does not generate any traffic on discarding a S line. If the L2 evicts a S(SO) line, an invalidate message is generated to the processor id corresponding to the ownerid field. If the line is in S(MO) state, a B_INV message is generated on the invalidation network. If the L2 needs to evict a M line, it generates a WB_PENDING message to the L1 and waits in the P state until the L1 generates a WB message. The evicted M line from the L2 is written back to main memory.

2.5 Read Invalidations

It becomes necessary to track stale reads in the case of a write after read (WAR) and at the same time refrain from invalidating cache lines in case of a read after write (RAW). Figure 4 shows the difference between these cases. In the case of the RAW, P1 might see the B_INV because of the UP, even before its RD_SH reaches the L2. Therefore, P1 sets an invalidation flag indicating that it has detected an invalidation for the same cache address as the RD_SH request. If P1 receives a RD_SH_RESP_RAW from P2, it discards the invalidating flag and sets its L1 cache state to S. Also, P1 has to generate an ACK back to L2 to indicate the read has completed and prevent any other intervening write from invalidating the cache line in P1. In the WAR case, the RD_SH_RESP from the L2 is discarded by P1 because the invalidation flag was set when P1 sees the B_INV because of P2's UP. It must be noted that in the case of a Write-Read-Read scenario, the L2 generates a RD_SH_RESP response packet for the second READ request. But, this RD_SH_RESP causes the cache line to be invalidated because of the invalidate flag that was set by the second READ on seeing the original write. Therefore, in the case wherein the RD_SH_RESP packet finds the invalidate flag set, unnecessary invalidations *might* occur. Note, that a full directory protocol

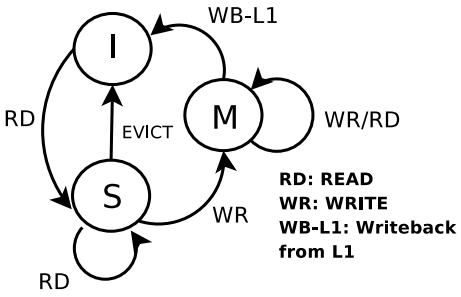


Fig. 2. L1 Cache State Transitions

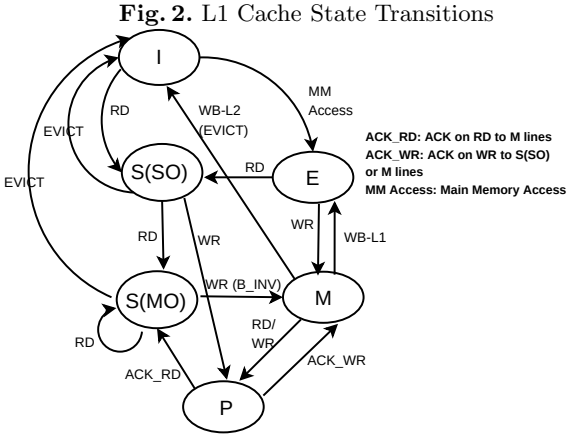
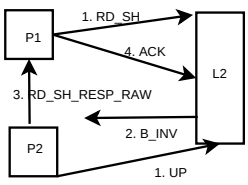


Fig. 3. L2 Cache State Transitions

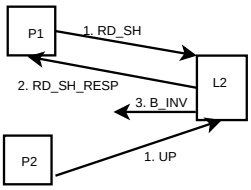


1. A read request from P1 and a write request from P2.

2. L2 services P2's write and generates a broadcast invalidate

3. L2 services P1's read and forwards the request to P2, which generates a RD_SH_RESP_RAW. P1 should commit in the S state even though P1 sees the B_INV before the RD_SH_RESP.

RAW scenario - read should commit in S state in the cache



1. A read request from P1 and a write request from P2.

2. L2 services P1's read and generates a RD_SH_RESP packet.

3. L2 services P2's write and generates a broadcast invalidate. If P1's RD_SH_RESP reaches after the B_INV, P1 should invalidate its cache line.

WAR scenario - read cache line should be invalidated by the broadcaset due to the write.

Fig. 4. RAW and WAR operation

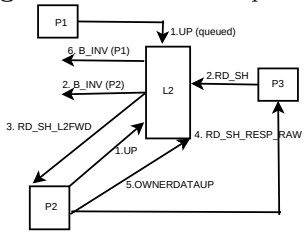


Fig. 5. Stale UP

that supports ACK messages to be generated during invalidations prevents this scenario, because the processor generating the read never receives an invalidation message.

2.6 Stale UP

Consider the following scenario shown in Figure 5. Processors P1 and P2 send UP messages and P3 sends a RD_SH message to the L2. Suppose that L2 services P2's UP before P1; it sends a B_INV message and transits to M state. P1 on seeing the B_INV, sets a flag within its request table indicating that the request has changed from an UP to a RD_EX. L2 then services P3's request and transits to S(MO) state. L2 now services P1's UP request, and sends a B_INV packet on the invalidation network. This is a stale UP case and the L2 never detected P1's request change from an UP to a RD_EX. In such a situation, P1 detects that the L2 has serviced a stale UP packet and sends an UP_INV packet to the L2 on the sink channel. On receipt of this UP_INV packet, L2 transits to E state, clears the owner id field and sends a RD_EX_FAIL packet back to P1 acknowledging that it has detected the stale UP. P1 now sends a RD_EX packet to the L2, instead of an UP.

3 Results

This section describes the evaluation of the system using some of the kernel Java Grande benchmarks [8] . Five multithreaded kernel benchmarks, namely LUFact, SOR, Crypt, Series and Sparse were run on the simulator.

3.1 Simulator Configuration

A cycle accurate version (models pipeline stalls, L1 cache contention and queuing at the memory controller) of the JAMSIM simulator [9] was used to implement the dual mesh architecture. The JAMSIM simulator (models perfect memory, single bus, hierarchical buses and crossbar based CMP architectures), written in Java models hardware components as software objects [9]. It uses a JAMAICA ported version [10] of the JIKES RVM for compiling and optimizing Java byte codes to JAMAICA machine instructions, and providing bootup and thread scheduling functionality. The simulator was modified to include dual mesh networks. Other changes involved creating communication ports and their associated logic, router logic in each tile and incorporating the new cache coherency protocol. All ports within a tile are clocked every cycle. State machine based port slaves and masters simulate the delays in accessing, processing and forwarding a packet at all tiles. In order to simulate network contention, both request and response packets are transmitted over the mesh network. Intra port delays are modelled by blocking one of the many simultaneous requests that arrive for a single master port. Inter port delays are modelled by having separate request and grant control signals between adjacent tiles. All port slaves within a tile

contend for a lock in order to access the cache (L1 or L2) or any resource within the processor and memory tiles at any point in time. This simulates contention for shared resources within a tile. The lock also implements the functionality of a single ported cache. All results gathered from the simulator are within the parallel execution phase of the benchmarks. Simulations were performed using dual context JAMAICA processor cores, i.e. each processor core supports two threads.

Five configurations of the architecture, each with varying processor and L2 tile count were simulated. Table 1 shows the various processor and L2 tile combinations.

Table 1. Processor and L2 Combinations

Number of Processors	Number of L2 Tiles
1	1
2, 4, 8	2
16	4
32	8
64	16

Table 2 shows the access times (in processor clock cycles) for different cache sizes and the main memory delay assumed [11].

Table 2. Multi Threaded Processor Configuration

Component	Size and Access Time
L1 Data Cache	64KB, 1 cycle
L1 Instruction Cache	64KB, 1 cycle
1 L2 tile	4MB, 26 cycles
2 L2 tiles	2MB, 17 cycles
4/8/16 L2 tiles	1MB, 9 cycles
Main Memory	200 cycles

3.2 Speed Up

The speed up of a benchmark is calculated using the formula:

$$SpeedUp = \frac{ExecutionTime(1Processor)}{ExecutionTime(Nprocessors)} \quad (1)$$

The speed up is measured w.r.t the performance of the benchmark using a single processor for the same architecture. Figure 6 shows the speed up of all benchmarks up to 64 processors assuming a Parallel Random Access Memory (PRAM)

model [12] with a special unit to handle locks [9], therefore presenting a true picture of the inherent parallelism within the benchmark. Speed up graphs on the dual mesh scheme for all benchmarks run up to 32 processors and for three of the five benchmarks, run up to 64 processors is presented in Figure 7. As seen from the perfect memory speed up graph, SOR and LUFact (for the data set chosen) are inherently not scalable. Therefore, they were not run on the 64 processor configuration.

All benchmarks, except for Crypt at 64 processors, show similar behaviour to that of the perfect memory speed up graphs. For the 64 processor configuration, the speed up for Crypt with the dual mesh scheme is 40 (Figure 7), while that with the perfect memory simulator is 51 (Figure 6). Therefore, the speed up for Crypt with the dual mesh scheme is 62% of the ideal linear speed up of 64, and 78% of the speed up obtained from the perfect memory simulator. The drop in speed up is primarily attributed to the delays in the dual mesh network. Figure 8 shows the average read and write latencies (average delay to satisfy a read miss and a write miss, respectively) for the crypt benchmark from 16 up to 64 processors, with the 64 processor configuration run for different data sets (50000 and 500000). It can be seen that as the number of processors increase the average read and write latencies also increase. Increasing the data set size (thereby increasing the number of requests to L2) has no effect on the read and write latencies. This proves that the read and write latencies are dependent on the network architecture rather than the contention induced by the benchmark.

Figure 9 measures the effect of stale UP, and unnecessary invalidations of cache lines on receiving a read response packet, both mentioned in Sections 2.5 and 2.6 that might occur in the dual mesh scheme because of not having an ACK message during writes to widely shared data. We see that the effect of unnecessary invalidations and stale UPs is minimal, with the maximum being 1.5% for LUFact.

4 Related Work

Snoop and directory based cache coherency are extensively researched in several papers. Combination of snoop and directory based protocol was explored in Multicast snooping [13]. In order to avoid acknowledgements during a broadcast invalidation, it uses an ordered address network. It requires a full directory to ensure that the prediction made to multicast the request contains the correct set of cache line owners. In constrast, the dual mesh scheme uses a limited directory, supporting non-ack tree based broadcast on a separate mesh network.

Tiled based CMPs are a popular subject of research and development in both academia as well as in industry. They can be broadly classified based on the programming paradigm adopted within the system - control flow and data flow model. Most tiled CMPs use the mesh topology with wormhole based packet switching, credit based flow control and dimension order static routing protocol. Some of the popular tiled based CMP in the industry are as follows: The Tilera

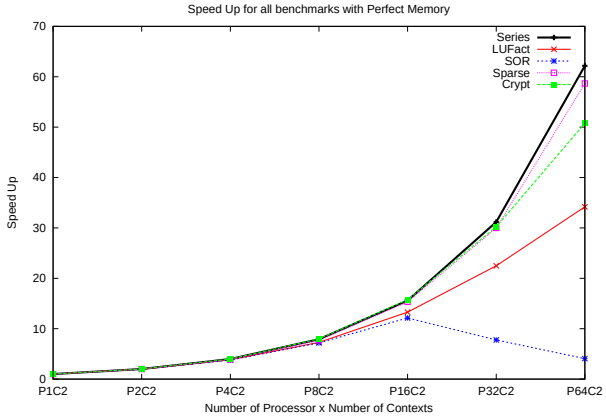


Fig. 6. Speed Up from Perfect Memory Architecture

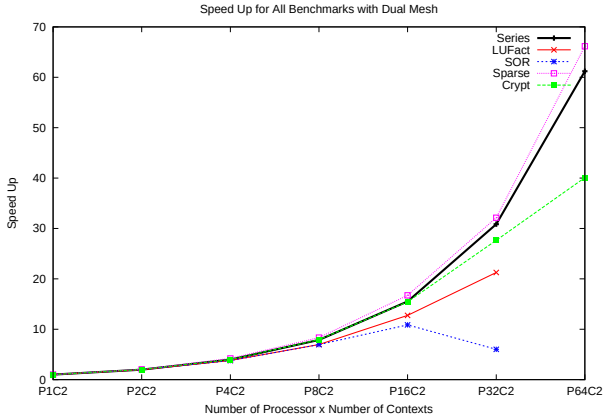


Fig. 7. Speed Up from Dual Mesh Architecture

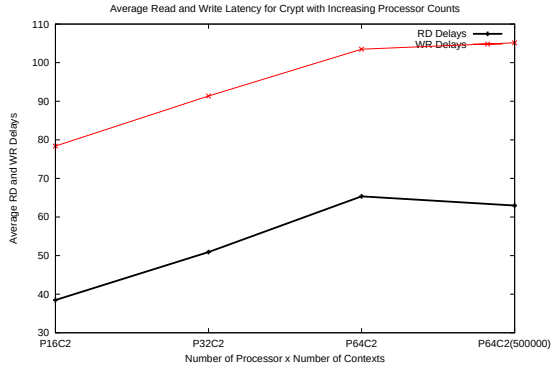


Fig. 8. Read and Write Latencies with Varying Processor and DataSet Configurations for Crypt: Dual Mesh

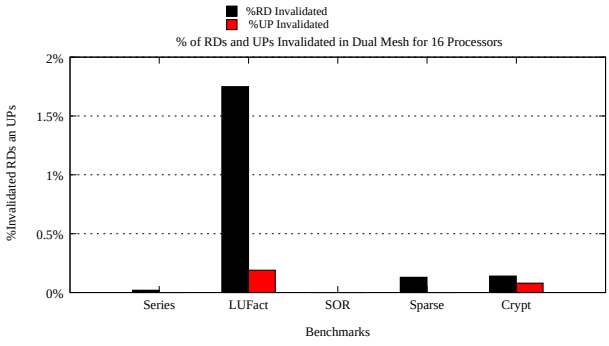


Fig. 9. % of Invalidated RD and UPs in Dual Mesh

system that uses five mesh networks on chip, each for a specific purpose to handle shared memory (control flow) as well as dataflow based computing. It does not support caching of shared data. However, in the dual mesh scheme we support cache coherency using multiple networks. Intel's Tera Scale [14] architecture uses 80 tiles interconnected using a mesh network. Tiles can be general purpose processor cores or special-purpose computing engines. It uses a MESI based full directory protocol to support shared memory communication. We use a limited directory based cache coherency protocol that does not have the space overhead of a full directory and optimizes on the write latency to widely shared data.

Academic based Tiled CMPs include, the TRIPs architecture [15] which supports a dataflow model of computation and also uses a two mesh networks, one for connecting memory tiles and the other for providing communication between processing elements; Priority NoC [16] that uses a mesh connected NUCA based L2 cache interconnecting 8 processor tiles. It relies on multiple optimizations to the network router architecture, packet switching technique to reduce the delays (queuing and network traversal) for request and responses messages by giving priority to short request or control based messages, such as ACKs. It uses a MESI based full directory protocol. The dual mesh scheme described uses a limited directory in comparison to the full directory scheme described. Similar to the Priority NoC scheme, it uses the network features to send read and write messages on different virtual channels, but does not provide for any priority. The distinguishing feature between the dual mesh network and the Priority NoC scheme is the use of two separate networks which serves the purpose of splitting the cache coherency traffic. Balfour et. al. present a dual mesh architecture in [17], exploring the delay, power and area constraints within the architecture rather than evaluating the performance of a cache coherent system.

5 Conclusion

It is well known that CMPs have already appeared, and will continue to dominate the design of future computing systems. The best possible use of CMP systems comes from extracting high degrees of parallelism from existing applications. Given the results obtained, we conclude that the dual mesh scheme is a promising approach for small and medium sized (up to 64 or even 128 cores) CMP systems. However, very large scale CMP systems (256 or more cores) would require a shift from the shared memory programming model and hence independence from cache coherency protocols to achieve high scalability.

Acknowledgments. The authors would like to thank Dr. Mikel Lujan for proof reading this paper and providing constructive comments.

References

1. Hennessy, J.L., Patterson, D.A.: Computer architecture: A Quantitative Approach. Morgan Kaufmann Publishers, SanFrancisco, California (2007)

2. Dally, W., Towles, B.: Route packets, not wires: on-chip interconnection networks. In: DAC: Proceedings of Design Automation Conference. (2001) 684–689
3. W., D., Griffin, P., Hoffmann, H., et. al.: On-chip interconnection architecture of the tile processor. IEEE Micro **27**(5) (2007) 15–31
4. Duato, J., Yalamanchilli, S., Li, L.: Interconnection Networks: An Engineering Approach. Morgan Kaufmann Publishers, SanFrancisco, California (2003)
5. Kumar, R., Zyuban, V., Tullsen, D.M.: Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In: Proceedings of the 32nd International Symposium on Computer Architecture, June 2005. (June 2005) 408–419
6. Wright, G.M.: A single-chip multiprocessor architecture with hardware thread support. PhD thesis, University of Manchester (January 2001)
7. Hansson, A., Goossens, K., Radulescu, A.: Avoiding message-dependent deadlock in network-based systems on chip. VLSI Design (February 2007)
8. Smith, L.A., Bull, J.M., Obdrzalek, J.: A parallel java grande benchmark suite. In: Supercomputing 01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing. (2001) 6
9. Horsnell, M.J.: A Chip Multi-Cluster architecture with locality aware task distribution. PhD thesis, University of Manchester (2007)
10. Dinn, A.: JaVM user guide (August 2006) http://intranet.cs.man.ac.uk/apt/intranet/csonly/jamaica/Documents/TechnicalNotes/JaVMUserGuide/JaVMUserGuide.html/ Last Accessed June 2008.
11. Muralimanohar, N., Balasubramonian, R.: Interconnect design considerations for large nuca caches. In: ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture. (2007) 369–380
12. Gibbons, A., Spirakis, P.G.: Lectures on Parallel Computation. Cambridge University Press (1993)
13. Bilir, E., Dickson, R.M., Hu, Y., Plakal, M., Sorin, D.J.: Multicast snooping: a new coherence method using a multicast address network. In: Proceedings of the 26th annual International Symposium on Computer Architecture. (May 1999) 294–304
14. Azimi, M., Cherukuri, N., Jayasimha, D., et. al.: Reevaluating amdahl's law. Integration Challenges and Tradeoffs for Tera-scale Architectures **11**(3) (2007) http://www.intel.com/technology/itj/2007/v11i3/1-integration/1-abstract.htm: Last Accessed September 2008.
15. Gratz, P., Kim, C., Sankaralingam, K., Hanson, H., Shivakumar, P., Keckler, S., Burger, D.: On-chip interconnection networks of the trips chip. IEEE Micro **27**(5) (2007) 41–50
16. Bolotin, E., Guz, Z., Cidon, I., Ginsoar, R., Kolodny, A.: The power of priority: Noc based distributed cache coherency. In: Proceedings of the First International Symposium on Network-on-Chips (NOCS '07). (May 2007) 117–126
17. Balfour, J., Dally, W.J.: Design tradeoffs for tiled cmp on-chip networks. In: ICS '06: Proceedings of the 20th annual international conference on Supercomputing. (2006) 187–198

Comparing Programmability and Scalability of Multicore Parallelization Paradigms with C++

Christian Terboven, Christopher Schleiden, and Dieter an Mey

JARA, RWTH Aachen University, Center for Computing and Communication
Seffenter Weg 23, 52074 Aachen, Germany
{terboven, schleiden, anmey}@rz.rwth-aachen.de

Abstract. The number of paradigms offering approaches to exploit parallelism is increasing, but programmers still consider Shared-Memory parallel programming to be a hard task. In this work we compare such paradigms - namely OpenMP and Threading Building Blocks - for their suitability to parallelize an object-oriented sparse linear algebra library. We evaluate the programmability and scalability of different parallelization concepts and different implementation strategies.

1 Introduction

Multicore processors provide more compute power for less energy and produce less heat, which has become the delimiting factor for building processors with higher clock rates. While the hardware industry is clearly pushing multicore architectures for the years to come, the software industry has not yet found an all-convincing paradigm for efficient parallel programming. But in order to profit from current and future microprocessor developments, programmers have to parallelize existing applications and take parallelism into account when writing codes from scratch [SL05].

Over the last few years, OpenMP [ARB08] has evolved as the de-facto standard for Shared-Memory parallel programming in the field of high performance technical computing, but so far it is not commonly used for general application development. OpenMP consists of a collection of compiler directives, library functions and environment variables to support directive-based fork-join parallelism. While it is comparably easy to learn and use and allows for incremental parallelization of existing applications, it has been found that some features are missing to achieve full performance on recent multicore architectures, most prominently thread binding [TaMS07] and better control for memory placement [TaMS+08]. In addition, the support for the C++ programming language is limited [TaM06].

In order to improve parallel programming support for C++ programmers, Intel has released the Intel Threading Building Blocks (TBB) [Rei07] template library. It builds upon the concept of generic programming and provides abstractions for parallel algorithms, containers and the like, also providing fork-join parallelism. While TBB allows for incremental parallelization as well, using

TBB may require significant program restructuring measures if the algorithms and data structures were not designed in a STL-like fashion, i.e. making use of the iterator concept. Thus, TBB have not yet achieved significant reputation in the traditional HPC world.

POSIX-Threads and WIN32-Threads are the native interfaces of Unix and Windows for multi-threaded programming and can be used for algorithmic parallelization as well, although with significantly less comfort than OpenMP or TBB.

In order to compare the programmability and scalability of these three parallelization paradigms with C++, we examined the design and parallelization process of a template-based sparse linear equation solver library, which is an excerpt of the DROPS Navier-Stokes solver [GPRR02]. We evaluated and optimized several design concepts of embedding the parallelization into an object-oriented programming style and examined how the multithreading paradigms fit into that. The library has been used in several parallelization projects already [TSaM+05], [JTS+07] and we use the abbreviation *laperf* for further reference.

The rest of this paper is organized as follows. In chapter 2 we describe the computational task we want to solve and give a brief overview of the parallelization paradigms. We present our implementation strategies and our findings concerning programmability in chapter 3. In chapter 4 we take a look at the performance and in chapter 5 we present a brief summary and an outlook on future work.

2 Computational Task and Parallelization Paradigms

The C++ programming language is gaining interest in the field of High Performance Computing, especially to implement numerical algorithms. The solver kernels of the two applications considered for benchmarking are examples of using object-oriented modeling and programming without performance degradation. Using an object-oriented approach and generic programming, numerical algorithm implementations can even look pretty similar to mathematical equations. Figure 1 shows an excerpt of a CG-type solver, such a kernel is heavily used in the compute intense part of many PDE solvers. From a researcher's point of view, this programming style significantly eases algorithmic modifications and code maintainability.

Our target is to create an efficient parallelization of such a code while maintaining the object-oriented and generic programming style. In order to parallelize the presented task, we created a library providing a data type for a dense vector and a sparse matrix in compressed row storage (CRS) format. As we also implemented all numerical functions defined on those types, we were able to evaluate different parallelization paradigms and implementation strategies. Our goal was to provide all building blocks for parallel sparse numerical algorithms, while hiding the parallelization details from the user. If possible, the parallelization should not hinder any algorithmic modification.

Code 1 Iteration loop of a CG-type solver in C++.

```
1  vectorT p(dim), r(dim), q(dim), ...; // vector
2  matrixT A(rows, cols, nonzeros); // sparse matrix in CRS
3  [...]
4  while (iter < this->max_iter && [...]) {
5            p = r + p * beta;
6            q = s + beta * q;
7            x = alpha * p + x;
8            r = r - q * alpha;
9            s = A * r;
10           [...]
11           ++iter;
12 }
```

2.1 OpenMP

OpenMP supports Fortran, C and C++. A so-called *parallel region* in OpenMP consists of a pragma followed by a structured block. At the entrance of a parallel region, additional worker threads are created. The number of threads to be used can be specified by an environment variable or API calls.

Several work-sharing constructs are available to distribute work between multiple threads. If, for example, a loop inside a parallel region is preceded by an OpenMP work-sharing loop directive, the loop iterations are distributed across the threads of the current team. The way in which the loop iterations are distributed can be controlled elegantly via clauses of this directive. As worksharing is not well-suited e.g. for recursive algorithms, OpenMP 3.0 introduced the concept of Tasking in May 2008 to denote independent program parts to be executed by the threads of the current team.

2.2 Intel Threading Building Blocks

Threading Building Blocks are provided as a C++ template library. Applying them does not include using any language extension. In order to use the library, a scheduler object has to be initialized once, where the number of threads to be used can be specified explicitly or determined automatically.

TBB provides several skeletons and parallel replacements for STL-type algorithms to express parallelism. Either work is divided into smaller tasks for independent execution implicitly, or tasks are specified explicitly. It also provides STL-type containers (e.g. a vector) together with parallel operations. While the programmer can influence the work distribution, the default is to let the runtime schedule chunks of work to the threads. Execution of unfinished tasks is independent of creating additional tasks.

2.3 Native Threads

POSIX-Threads on Unix and WIN32-Threads on Windows offer an API to create and manage threads. Implemented as a library, they can be used from virtually any programming language. While offering full functionality for thread management and several synchronization capabilities, the user is solely responsible for implementing work distribution.

3 Programmability

To parallelize the code discussed in the previous chapter without breaking the object-oriented approach, we evaluated several concepts that are described in 3.1. Based on that, we developed several implementation strategies how to fit in the parallelization paradigms in 3.2. Finally we discuss how to take care of memory placement on cc-NUMA architectures in 3.3.

3.1 Parallelization Concepts

Taking the given requirements into account, the design of our sparse matrix type `matrix_crs` implementing a compressed row storage format looks as shown in figure 2. We are using two template parameters:

- `T`: data type of matrix elements, usually `double`.
- `Alloc` (optional): STL-type allocator for memory management on cc-NUMA machines.

The constructor takes the matrix dimensions as arguments as well as an optional argument for specification of the parallel loop schedule. The latter is only applicable if OpenMP parallelization is selected.

Code 2 Type signature of our sparse matrix data type.

```
1 template <class T = double,
2       template <typename> class Alloc = std::allocator>
3 class matrix_crs   {
4  public:
5    matrix_crs(size_t rows, size_t cols, size_t nonzeros,
6          const enum OpenMPScheduleType _eScheduleType)
7    { [...] }
8 }
```

As will be described below, our approach is efficient. It is also very easy to use in the considered application codes, as the existing matrix and vector classes can be hidden by our types using an appropriate `typedef` declaration.

In order to evaluate our approach of parallelization under the hood of the object data types, we started off with a rather simple concept. We designed the template class type as described above and used plain operator overloading to provide all necessary functions. We used preprocessor macros and (large) switch constructs to differentiate between different parallelization paradigms and implementation strategies.

Given the six implementation strategies presented in the next chapter, this made maintaining and extending the library code very hard. This approach also introduced expensive temporary copies in complex expressions. As a consequence, it did not deliver good performance, although all operations could be parallelized. We concluded that this approach is not well-suited for productive code development.

The second approach relied on inheritance: We defined an abstract base class for the vector and sparse matrix data types and provided a specialization for each implementation strategy. This solved most code manageability issues, but had some performance impacts as well. Complex expressions like $r = b - A \times x$ had to be transformed by the compiler by introducing additional temporaries (here: $t1 = A \times x; t2 = b - t1; r = t2$). In addition to that, we found that the compilers were not able to eliminate the expensive virtual function calls during optimization.

The third and so far most successful approach was to use template expressions [Vel95], in order to avoid the generation of temporaries. Thus, we only have one class for each data type, but added a compile time constant template parameter to indicate the implementation strategy. Specialized implementations are differentiated by this parameter and a switch construct.

Using a self-implemented expression template mechanism, our library can profit from complex expressions, as they are "unrolled" into a single `for`-loop which can easily be parallelized. Thus we increased the amount of work per parallel region and decreased the total number of parallel regions invoked, which results in a higher efficiency. The only downside is that in the debugging process the complicated template mechanism becomes visible to the user.

3.2 Implementation strategies

We examined several implementation strategies for our library design. In the following description we differentiate between the view from a user of our library, and the view from the actual library implementation.

1. *OpenMP with Internal Parallelization (OMPIN)*: In this version, each operator or class member function contains a distinct parallel region. The parallelization is completely invisible to the user of the class library, and using the data types is safe both from serial and parallel code. Thus, this approach is well-suited to be offered by a library. The downside is that entering and exiting a parallel region for each operator call involves some overhead, which can become significant depending on the amount of work to be parallelized.

Also, it is not possible to avoid the barriers between several function calls, e.g. between lines 6 and 7 in figure 1.

2. *OpenMP with External Parallelization (OMPEX)*: In this version we exploit OpenMP's orphaning capability. The operator or class member functions only contain work-sharing constructs. The parallelization becomes visible to the user, as he has to insert the parallel region directive into the code. Thus we employ OpenMP's orphaning concept to combine multiple worksharing regions into one parallel region to save the fork-join overhead. But the parallelization is not completely safe any more, because if any function with orphaned work-sharing constructs would be called from within another work-sharing construct, the result would be undefined according to the OpenMP specification. Such a situation cannot reliably be detected by the compiler nor the library.
3. *OpenMP with External Parallelization and nowait (OMPEX_NW)*: This version is based on the previous approach, but in addition we eliminated the implicit barrier at the end of each worksharing construct by adding `nowait` clauses to the worksharing directives. Obviously the user now has to manually insert barriers at all points where they are required to prevent data races, e.g. between lines 8 and 9 in figure 1. Such an implementation is virtually unusable for general purpose and should not be offered to the user, but can be quite efficient in library implementations. Our library provides several solvers using exactly this implementation strategy, which have been used successfully in [TSaM+05], [JTS+07].
4. *OpenMP with Tasks (OMPTASK)*: In this version we employed the brand-new tasking model of OpenMP 3.0. The parallelization is invisible to the user, but the library routines have to be called from within a parallel region in order to make use of the thread team. Thus, this approach is safe to use. By the time of this writing we were unable to evaluate this approach for performance, as the Intel 11.0 compilers providing an implementation of OpenMP 3.0 were not able to translate this code correctly.
5. *TBB with Algorithms (TBB_ALG)*: In this version we used TBB's algorithmic skeletons to create implicit tasks. The parallelization is completely invisible to the user and this approach is always safe to use. The only requirement is to initialize the TBB library once during the program's runtime. From an implementation's point of view, this version is very similar to the OMPIN version, only using TBB as the parallelization paradigm.
6. *TBB with Tasks (TBB)*: In this version we used TBB's explicit tasks. The parallelization is completely invisible to the user. The only requirement is to initialize the TBB library once during the program's runtime. The goal is to avoid implicit synchronization introduced by OpenMP's worksharing constructs or TBB's algorithmic skeletons (e.g. between lines 6 and 7.). In contrast to the OMPEX version this approach is always safe to use, but requires a lot of care from the implementer as all parallelization patterns have to be implemented manually.

Only the OMPIN, OMPTASK and both TBB versions can be recommended without reservation. With the other strategies the parallelization either becomes visible to the user or the user would have to take additional measures to ensure correctness, which is error-prone. But the safety costs additional synchronization overhead, which is not necessary from an algorithmic point of view.

If performance is important enough to make some concessions in the usability of a class library, the OMPEX version could be recommended as well. We assume that the user can be burdened with the requirement to provide a parallel region to make use of worksharing inside the library. This strategy works well as long as the user exactly knows about the libraries expectations on the environment and no other code like output operations has to be called. Nevertheless, already for the GMRES solver this strategy turned out to be hard to use correctly.

The performance impact of the different implementation strategies is discussed in chapter 4.

3.3 Memory Allocation on cc-NUMA Architectures

In order to overcome the memory bandwidth limitations of bus-based SMP systems, shared memory computers frequently are cc-NUMA (cache-coherent nonuniform memory architectures) machines. Neither native threads nor OpenMP nor TBB have any notion of the hardware a program is running on, thus they do not explicitly support distributed memory allocation on cc-NUMA nodes. The user is forced to use operating system-specific calls or compiler-specific environment variables to pin threads to processors and to control page allocation. Disrespecting a cc-NUMA architecture can lead to severe performance impacts and might even inhibit achieving any speedup at all, as shown in table 1.

One additional goal of our library was to make the cc-NUMA support transparent to the user in the same extent as the parallelization. Modern operating systems typically use the first-touch allocation strategy, that is putting a page close to the processor accessing it first.

We decided that the cc-NUMA support strategy would be a good template parameter argument of our data type. The C++ STL library already provides allocators as an abstraction of the memory management. An allocator provides an interface between the application's view of the memory system and operating system calls. While there are several specialized allocators available for various different needs, to the best of our knowledge there is no allocator available taking care of memory placement optimization on cc-NUMA systems, thus we designed some. By specifying one of our specialized allocators the user can provide hints on how to lay out data structures on cc-NUMA architectures.

So far we only provide support for distributed page allocation during initialization, because features like page migration are not yet provided by all operating systems nor are they available as a standard, but could be encapsulated in our design as well. Here we describe two different distribution strategies that have proved to be applicable to a range of application scenarios, both making use of appropriate thread binding to prevent any influence of operating system scheduling effects:

- *dist_allocator*: This allocator expects two arguments, namely an OpenMP loop schedule type and optionally a chunksize specification. Internally it calls `std::malloc()` to allocate a block of uninitialized memory and employs an OpenMP-parallelized loop to initialize that block with zeros using the specified scheduling and chunksize. This allocator should be used when during the computation the data is accessed in the same pattern, for example when the same OpenMP schedule is used.
- *chunked_allocator*: This allocator expects an argument of type `std::vector`. Internally it calls `std::malloc()` to allocate a block of uninitialized memory and initializes that block with zeros. The initialization loop is parallelized as well, the vector elements are interpreted as indications of where the chunks for individual threads start and end. Using this allocator, the user can provide knowledge of how the data should be laid out on a cc-NUMA architecture.
  We successfully applied this allocator in the application scenarios where the sparse matrix had a known structure. By precomputing the optimal distribution of matrix rows onto threads we achieved a perfect load balance.

For many use cases a cc-NUMA-aware allocator allows for comfortable handling of memory placement. The performance impact of using our allocators on a cc-NUMA architecture is discussed in chapter 4.

4 Performance and Scalability

We compared the performance of all implementation strategies on two recent multicore architectures, which exhibit quite different properties:

1. FSC RX200: A 2-socket quad-core Intel Xeon 5450 (3.0 GHz) machine. On this architecture, 4 pairs consisting of two cores share a 6 MB L2 cache each. It offers a flat bus-based UMA memory architecture with severe memory bandwidth limitations.
2. SUN Fire V40z: A 4-socket dual-core AMD Opteron 875 (2.2 GHz) machine. On this architecture, each core has 2 MB L2 cache. It offers a cc-NUMA memory architecture.

For all performance experiments, we used the Intel C++ compiler version 10.1.015 and Scientific Linux 5.1 64-bit. In chapter 4.1 we evaluate the scalability of a sparse Matrix-Vector-Multiplication (SMXV) kernel as it is invoked in line 9 of figure 1, in chapter 4.2 we discuss the performance of an GMRES solver which is part of our library.

4.1 SMXV kernel

A sparse Matrix-Vector-Multiplication operation is typically the most compute-intensive part in CG-type iterative linear solvers. For this comparably simple routine, neither the selected parallelization paradigm nor the applied worksharing-based implementation strategy has any significant effect on the performance, as

long as the load is equally distributed on the threads and the threads are bound to separate cores. Table 1 shows the performance of our library using three different allocators on both the UMA and cc-NUMA architecture. We compare these results to the implementation provided by the Intel MKL 10.1.

Each entry displays the runtime performance in seconds and we always used two threads. The medium dataset has a memory footprint of some 60 MB, the large dataset has a memory footprint of some 300 MB and about 19,600,000 nonzero values. As both datasets are larger than the caches, we used a scattered thread binding in order to maximize the memory bandwidth on the cc-NUMA machine and the cache capacity on the UMA machine [OKWT07].

Machine	**Dataset**	**laperf (OMPIN)**			**MKL**
		std	dist	chunked	
FSC RX200 (UMA)	medium	406	—	—	397
FSC RX200 (UMA)	large	398	—	—	383
SUN Fire V40z (cc-NUMA)	medium	—	310	389	177
SUN Fire V40z (cc-NUMA)	large	—	329	428	182

Table 1. Sparse Matrix-Vector-Multiplication performance [MFLOP/s] with two threads.

On the UMA machine, where the allocator strategy has no influence on the performance, our implementation slightly outperforms the MKL. This proves the efficiency of our library implementation.

On the cc-NUMA machine our implementation is up to 2.4 times faster than the MKL, which does not provide any means of specifying a data distribution scheme. Using the *chunked* allocator with knowledge of the matrix and data layout clearly increases the performance over the static distribution of the *dist* allocator. This highlights the importance of respecting the cc-NUMA architecture and will also be discussed in the following chapter.

4.2 GMRES: Implementation strategies

Table 2 compares the performance of the TBB and OMPIN versions of our GMRES solver on the cc-NUMA architecture.

Both OpenMP and TBB have only little support for thread affinity, and no explicit support for data placement. While the techniques to pin OpenMP threads to processor cores are well-exploited, these solutions are not yet applicable to tasks. Thus, the Tasking paradigm has a major disadvantage over OpenMP's worksharing: Data affinity is lost, hurting the scalability badly. Increasing the number of threads from two to four delivers an improvement of only 1.12, using more than four threads is not profitable at all.

While the OMPIN version with just one thread is slower than the TBB version, it allows for the specification of cc-NUMA aware allocators and respects

the data distribution in its parallelization. Given our data types, the user simply has to use a specialized allocator to significantly increase the performance of the whole solver. Considering the OMPIN version, our allocators introduce a small performance overhead over the standard allocator, but the speedup can be increased from 3.03 to 5.77 using the *dist* allocator and to 6.45 using the *chunked* allocator with eight threads.

Parallelization	**Allocator**	**#Threads**	**Runtime**
TBB	std	1	389.1
		2	211.8
		4	188.3
		8	183.9
OMPIN	std	1	421.6
		2	231.4
		4	197.6
		8	139.3
OMPIN	distributed	1	432.4
		2	229.9
		4	143.5
		8	74.9
OMPIN	chunked	1	446.3
		2	229.5
		4	121.3
		8	69.2

Table 2. Performance comparison of TBB and OMPIN version and different allocator strategies for a GMRES solver on the cc-NUMA machine.

Table 3 shows the performance of selected implementation strategies of a GMRES solver on the UMA architecture, leaving out any cc-NUMA effects. Each entry displays the runtime performance in seconds. As the allocator strategy has no effect on the flat memory architecture, we always used the *std* allocator. Using only one thread, the TBB and OMPIN versions perform about the same speed, but the OMPEX version is clearly faster because of less internal overhead.

Using more than two threads with the TBB version is not profitable. We are still investigating this, but we suspect that as tasks are not bound to a specific thread, there is not data affinity and the cache content is lost whenever a task switches, which might significantly decrease performance. Using the affinity-based scheduling introduced in the recent version of TBB did neither improve the performance, nor had it significant influence on the thread scheduling according to the experiments we did so far. This needs further investigation.

On the one hand the OMPEX version outperforms the OMPIN version in all cases. But on the other hand the OMPEX version is very dangerous to use for an incautious programmer. Given our goal to enable flexible algorithm design by

Parallelization	Allocator	#Threads	Runtime
TBB	std	1	220.7
		2	116.2
		4	112.3
		8	104.9
OMPIN	std	1	218.3
		2	117.2
		4	86.4
		8	62.1
OMPEX	std	1	203.5
		2	108.9
		4	82.9
		8	60.5

Table 3. Comparison of different implementation strategies for a GMRES solver on the UMA machine.

providing efficient and easy to use parallel data types, the current choice is to use the OMPIN parallelization strategy. According to our findings, the performance improvements of the OMPEX version are not in balance with the disadvantages in terms of use. Nevertheless it can be used within a library internally.

We did not present results for the native threading APIs. As all API calls are inside operator or class member functions, the parallelization is completely invisible and using the data types is safe both from serial and parallel code. But thread creation and termination for each operator call involves significant overhead and it is not possible to span a thread's lifetime over multiple consecutive function calls (e.g. line 6 and 7 in figure 1), unless non-standardized extensions offering thread pooling functionality are used. Because of that and the lack of sophisticated load balancing facilities, the performance is worse than the OMPIN version and using native threading is associated with a significantly higher programming effort.

5 Conclusion and Future Work

Combining the object-oriented and generic capabilities of C++, we designed a library which enables the user to write efficient and highly abstract parallel code. The only differences between the serial and the corresponding parallel version are additional template parameters in the type definitions. Achieving a high efficiency while hiding the parallelization has been made possible by employing a self-implemented template expression mechanism.

We discussed several implementation strategies for such a library and performance measurements show that avoiding fork-join overhead (OMPEX version) can improve performance, but this approach should not be recommended for data type implementations. We found using C++ classes with internal OpenMP

parallelization (OMPIN version) to be the best compromise between programmability and performance today and recommend this strategy for any similar software project. The version OMPEX and OMPEX_NW offer a path to increased scalability, but hand the responsibility for defining the parallel region and the synchronization over to the knowledgeable programmer. Both OpenMP and TBB tasking models offer promising properties in terms of programmability as well, but using them it currently is not possible to respect the cc-NUMA characteristics of an architecture, which is crucial for performance and scalability.

Future work will include investigation on how compilers and language constructs (especially of C++0X) can further optimize the performance and scalability by saving synchronization overhead. We are also investigating on how tasking models could provide support for data affinity.

References

[ARB08] ARB. OpenMP Application Program Interface, version 3.0, May 2008.

[GPRR02] S. Gross, J. Peters, V. Reichelt, and A. Reusken. The DROPS package for Numerical Simulations of Incompressible Flows using Parallel Adaptive Multigrid Techniques. *IGPM-Report*, 211, 2002.

[JTS+07] Lenka Jerabkova, Christian Terboven, Samuel Sarholz, Thorsten Kuhlen, and Christian Bischof. Exploiting Multicore Architectures for Physically Based Simulation of Deformable Objects in Virtual Environments. In *Virtuelle und Erweiterte Realität, 4. Workshop der GI-Fachgruppe VR/AR*, Weimar, Germany, 2007.

[OKWT07] Michael Ott, Tobias Klug, Josef Weidendorfer, and Carsten Trinitis. Autopin - Automated Optimization of Thread-to-Core Pinning on Multicore Systems. In *Workshop Programmability Issues for Multi-Core Computers, International Conference on High-Performance Embedded Architectures and Compilers*, Goteborg, Sweden, January 2007.

[Rei07] James Reinders. *Intel Threading Building Blocks*. July 2007.

[SL05] Herb Sutter and James R. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[TaM06] Christian Terboven and Dieter an Mey. OpenMP and C++. In *Second International Workshop on OpenMP (IWOMP 2006)*, pages 300–311, Reims, France, June 2006.

[TaMS07] Christian Terboven, Dieter an Mey, and Samuel Sarholz. OpenMP on Multicore Architectures. In *International Workshop on OpenMP (IWOMP 2007)*, Beijing, China, June 2007.

[TaMS+08] Christian Terboven, Dieter an Mey, Dirk Schmidl, Henry Jin, and Thomas Reichstein. Data and Thread Affinity in OpenMP Programs. In *Workshop Memory Access on future Processors: A solved problem?, ACM International Conference on Computing Frontiers*, Ischia, Italy, May 2008.

[TSaM+05] Christian Terboven, Alexander Spiegel, Dieter an Mey, Sven Gross, and Volker Reichelt. Parallelization of the C++ Navier-Stokes solver DROPS with OpenMP. In G. R. Joubert, W. E. Nagel, F. J. Peters, O. Plata, P. Tirado, and E. Zapata, editors, *Parallel Computing (ParCo 2005): Current & Future Issues of High-End Computing*, volume 33 of *NIC Series*, pages 431 – 438, Malaga, Spain, September 2005.

[Vel95] Todd Veldhuizen. Expression templates. *C++ Report*, 7:26–31, 1995.

Towards Automatic Profile-Driven Parallelization of Embedded Multimedia Applications

Georgios Tournavitis and Björn Franke

Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh
United Kingdom

Abstract. Despite the availability of ample parallelism in multimedia applications parallelizing compilers are largely unable to extract this application parallelism and map it onto existing embedded multi-core platforms. This is mainly due to the limitations of traditional auto-parallelization on static analysis and loop-level parallelism. In this paper we propose a dynamic, profile-driven approach to auto-parallelization targeting coarse-grain parallelism. We present our methodology and tools for the extraction of task graphs from sequential codes and demonstrate the various stages involved in this process based on the JPEG-2000 still image compression application. In addition, we show how the joint detection of multiple levels of parallelism and exploitation of application scenarios can lead to performance levels close to those obtained by manual parallelization. Finally, we demonstrate the applicability of our methodology to a broader set of embedded multimedia codes.

1 Introduction

In recent years multi-core computing systems have become widely available and their efficiency in multimedia, signal processing, wireless or graphic applications has been already proved [1, 2]. However, automatic mapping of applications onto these platforms is still a major research challenge. The predominant manual approach to application parallelization is clearly not scalable enough to cope with the growing complexity of embedded codes and is bound to fail without strong programming tools support.

While auto-parallelization has a long research history, success in the embedded systems domain is limited. This is especially true for embedded multimedia applications which exhibit properties and constraints significantly different from the traditional scientific high-performance computing domain targeted by auto-parallelizers such as e.g. SUIF [3], Polaris [4]. In order to guarantee correctness parallelizing compilers heavily depend on static analysis for data dependence testing.

In this paper we propose to incorporate additional dynamic profiling information in the extraction of coarse-grain task graphs. We address the large body

of legacy codes written in sequential programming languages like C and present a pragmatic approach to parallelization combining static and profile-driven analysis as well as multiple levels of parallelism. Based on the JPEG-2000 still image compression application we demonstrate how to automatically derive a parallel OpenMP implementation and compare its performance with both a manually parallelized implementation and one generated by a static state-of-the-art auto-parallelizing compiler.

Contributions The recent advent of multi-core architectures has sparked interest in automatic extraction of coarse-grain parallelism from sequential programs, e.g. [2, 5, 6]. We extend it in numerous ways and among the specific contributions of our paper are:

- the introduction of an *"executable IR"* that simplifies the back-annotation of high-level code and data structures in the compiler with information collected during profiling,
- the joint extraction of task graphs and the detection of application scenarios [7] in order to efficiently exploit the semi-dynamic nature of many multimedia applications,
- the automatic detection of scalar and array reduction statements, using a *hybrid* statical and profile-driven analysis.

Overview This paper is structured as follows. In section 2 we present our parallelization methodology and tools. This is followed by a case study based on the JPEG-2000 application in section 3 before we evaluate our parallelization approach on a wider range of embedded multimedia applications in 4. We discuss related work in section 5 and summarize and conclude in section 6.

2 Profiling toolchain

At the heart of the proposed parallelization methodology is our profile-driven dependence analyzer. In the following paragraphs we give a brief technical description of the main components of our current prototype. An overview of the compilation, profiling and analysis process is depicted in Figure 1.

2.1 Source Code Instrumentation

Our primary objective is to enhance and disambiguate the static structure of a given program using precise, dynamic information. The main obstacle to this process is correlating the binary program execution back to the source code. Unfortunately, traditional compiler flows discard most of the high-level information and source-level constructs during the code generation process. Thus, binary instrumentation approaches require the use of debugging information which are imprecise since only symbol table information and the corresponding line-of-code of an instruction is maintained.

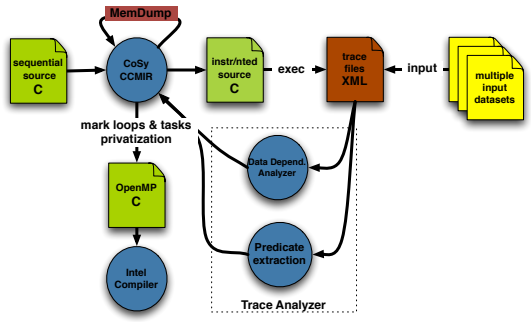


Fig. 1. Overview of the compilation, profiling and parallelization flow.

In contrast, our instrumentation framework (*MemDump*) operates on a medium-level Intermediate Representation (IR) of the CoSy compiler and can emit a plain C representation of its IR, enabling fast, native execution. Effectively we create an *"executable IR"* that allows for an efficient and precise *back-annotation* of profiling information to the compiler internal data structures such as data dependence graphs. This differs e.g. from [6] where an instrumented microarchitectural simulator is used for profiling and difficulties arising from the particular idiosyncrasies of the underlying instruction set architecture have to be addressed explicitly.

2.2 Profile-Driven Dependence Analyzer

The profile analyzer operates on the previously generated traces and evaluates memory and control-flow operations to dynamically construct data and control flow representations of the program under inspection. While a large number of executed IR nodes need to be processed this process is fast as no low-level machine details need to be considered.

Trace parsing Each "trace item" is processed using the procedure described in Algorithm 1. During this process, the control-flow handler reconstructs a global Control Flow Graph (CFG) of the application and maintains context information (e.g. call and loop stack, normalized loop-iteration vector). The corresponding data-flow handler keeps a mapping from the memory address space of the application to the nodes of the CFG. For each memory location it records the node which modified (def) it last and the normalized iteration vector at that time. In the current prototype the memory granularity is at a byte-level. Data-dependence information is registered in the form of *data-edges* which are inserted in an overlay of the CFG. Each data-edge is annotated with information about the address regions that are communicated from the source to the target node, creating a Control Data Flow Graph (CDFG) of the application. Finally, each data edge contains a bit-vector which records on which levels of the loop-nest this particular edge was carrying a loop-carried dependence.

Algorithm 1: The main algorithm executed by the Profile-Driven Dependence Analyzer to extract the CDFG.

```
Data
· CDFG(V, E_C, E_D): graph with control (E_C) and data-flow (E_D) edges
· bit_e[]: bitfield in each e ∈ E_D
· set_e: address set in each e ∈ E_D
· it_a[]: iteration vector of address a
· M[A, {V, it}]: hastable maps memory addresses → {V, it_a} tuple
· it_0[]: current normalized iteration vector
· u ∈ V: current node
Procedure instruction_handler
I ← next instruction from trace
if I is a memory instruction then
    a ← address accessed by instruction
    if I is a DEF then
        update last writer in M
    else if USE then
        find matching def from M
        if def→use edge e ∉ CDFG then
            add e in E_D
        set_e ← set_e ∪ {a}
        foreach i : it_a[i] ≠ it_0[i] do bit_e[i] ← true
        it_a ← it_0
else if I is a control instruction then
    v ← node referenced by instruction
    if edge (u, v) ∉ E_C then
        add (u, v) in CDFG
    u ← v
```

Privatization We maintain a complete list of true-, anti- and output-dependencies as these are required for parallelization. Rather than recording all the readers of each memory location we keep a map of the normalized iteration index of each memory location that is read/written at each level of a loop-nest. This allows us to efficiently track all memory locations that cause a loop-carried anti- or output-dependence. A scalar x is privatizable within a loop if and only if every path from the beginning to the loop body to a use of x passes from a definition of x before the use. Hence, we can determine the privatizable variables by inspecting the incoming and outgoing data-dependence edges of the loop.

Parallelism detection After the trace processing has been completed, the analyzer performs an additional post-analysis based on the augmented CFG to determine parallelizable loops and control-independent basic-block regions. In addition, for each loop the analyzer produces a detailed summary of variables that have to be privatized to eliminate any false-dependencies. Finally, whenever a loop is not parallelizable we inform the user about the offending data-dependencies, providing information about the (i) source/destination node, (ii) the data which was communicated and (iii) the context it occurred.

Reduction detection In the presence of pointer or indirect array accesses (e.g. sparse matrices) static analysis makes conservative assumptions and, thus, limits the amount of exploitable parallelism. We enhance the existing static analysis and propose a *hybrid approach* for these complex cases. As a first step we use a simple static analysis pass in our compiler to select reduction statement *candi-*

dates. Then, we instrument our source code to explicitly tag these operations. Finally, the dependence analyzer determines which candidates are *valid* reductions. The reduction detection step is based on the following necessary properties of a reduction:

1. The reduction operator is both commutative and associative.
2. There is a true self-dependence which denotes the accumulation to the partial result of the reduction.
3. Only the final result of the reduction is used later i.e there is no outgoing dependency from the reduction.
4. No true dependency is fed into the reduction.

It is important to stress that this methodology not only tracks scalar and array reductions but also *coupled* reductions, where more than one statements of the loop "accumulate" on the same targets.

2.3 Parallelization Methodology

The main stages of our parallelization approach are:

1. Source code instrumentation using *MemDump*
2. Compilation of the instrumented source using a standard ANSI-C compiler.
3. Program execution and generation of multiple trace files for a set of representative input datasets.
4. Processing of the trace files with the profile-driven dependence analyzer.
5. Identification of application hot spots.
6. Parallelization of the hotspots based on dynamic dependence information, starting with the outer-most loops.
7. Parallel code generation for the selected regions using one of the following OpenMP directives.
   - *Parallel for* for well structured parallelizable *for*-loops.
   - *Parallel taskq* for any uncountable or not well structured parallelizable loops[1].
   - *Parallel sections* for independent coarse-grain functions or control-independent code-regions.
8. Guarding of loop-carried dependencies either using *atomic*, *ordered* or *reduction* OpenMP clauses.
9. Privatization of variables that according to the dependence analyzer are causing a false-dependence.

[1] Task queues are an extension to the OpenMP standard already available in the Intel compilers, but also part of the recently-released OpenMP 3.0 standard.

2.4 Scenarios and Predicate Extraction

Application scenarios [7] correspond to "modes of operation" of an application and can be used to switch between individually optimized code versions, or prefetch data from memory or optimize communication if a suitable predictor can be constructed.

For sufficiently large sub-regions of the CFG we analyse the conditional statements at their roots and trace the variables involved in the condition expressions back to their original definitions. At this point we construct a new predicate expression that can be used as an early indicator about the application's future control flow. For many multimedia applications, for example, we have found that the statically undecidable control flow can be determined as soon as the relevant header information has been read from the input stream, i.e. long before the actual branch to a particular code region (e.g. a specific mode of a multimedia codec) is taken.

2.5 Complexity and applicability of the approach

As we process data dependence information at byte-level granularity we may need to maintain data structures growing potentially as large as the entire address space of the target platform. In practice, however, we have not observed any cases where more than 1GB of heap memory was needed to maintain the dynamic data dependence structures, even for applications taken from the SPEC 2000 integer and floating-point benchmark suites.

While the dynamic traces can reach several GB's in total size, they can be processed online. Our tools are designed to build their data structures incrementally as the input trace is streamed into the analyser, thus eliminating the need for large traces to be stored.

As our tools operate on the same level as the medium-level compiler IR we do not need to consider detailed architecture state, hence our tools can operate very efficiently. In fact, we only need to keep track of memory and control flow operations and make incremental updates to hash tables and graph structures. In practice, we have observed processing times ranging from a few seconds for small kernel benchmarks up to about 60mins for large multimedia codecs operating on long video streams.

2.6 Safety

Despite the fact that our method cannot guarantee correctness and still presumes user verification, in all the benchmarks that we considered so far there was not a single case of a loop/region falsely identified as parallel (*false positive*). In addition, the high potential of thread-level parallelism and the immense performance gap between automatic and manual parallelization, urge for a more pragmatic approach. We consider the tools and the synergistic techniques proposed in this paper to be a step towards this direction rather than a replacement of static dependence analysis.

3 Case Study: JPEG-2000 Still Image Compression

In this section we present a case study based on the state-of-the-art JPEG-2000 image coding standard. We explain our parallelization approach for an open-source implementation of the standard, OpenJPEG, and report performance figures for the resulting OpenMP implementation.

3.1 JPEG-2000 Still Image Compression

JPEG-2000 is the most recent standard produced by the JPEG committee for still image compression and is widely used in digital photography, medical imaging and the digital film industry. The main stages of the wavelet-based coding process as defined in the JPEG-2000 standard are shown in the diagram in figure 2.

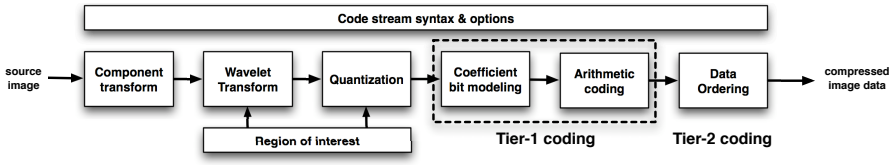


Fig. 2. Overview of the JPEG-2000 coding process.

3.2 Detection of Parallelism

JPEG-2000 exhibits multiple levels of data-parallelism (e.g. tiles, components, code-blocks). Even without tiling most of the coding stages can be performed independently on each component. Furthermore, during entropy bit coding, an additional level of parallelism can be exploited using a code-block decomposition. In the following paragraphs we go over the individual steps involved in the parallelization according to the methodology presented in section 2.2.

Static Analysis of DOALL Loops Initially, we evaluated the amount and type of parallelism a production auto-parallelizing compiler (Intel *icc* v10.1) can extract from our application. Enabling all optimizations available in the Intel compiler (e.g. multi-file interprocedural analysis, auto-parallelization and auto-vectorization) had no impact on the overall performance of the application for both compression and decompression. The compiler vectorized 17 and 11 loops, respectively, but could not parallelize any loops. Setting the profitability threshold to its minimum value resulted in 31 and 20 vectorized loops for compression and decompression, respectively, and 54 and 27 parallelized loops. This more aggressive parallelization scheme, however, resulted in a slow-down of 8 over the default setting.

Profile-Driven Analysis Following hot spot detection we use our analysis tool described in section 2 to extract parallelism at any level of the call hierarchy under the detected hot spots *dwt_encode* and *t1_encode_cblks* of the JPEG-2000 application.

While the main loop of the discrete wavelet transform can be easily expressed as a parallel DOALL loop, the situation for *t1_encode_cblks* is more complicated. Two loop-carried dependencies at the outermost loop level prevent parallelization at this level. Using the automatic reduction detection of the Trace Analyzer, we can automatically detect that this update can be substituted with an appropriate parallel reduction operation. Indeed, manual code inspection revealed that this is due to the update of an accumulation variable which is not used before later stages of the encoding process.

Another issue arises from the particular dynamic memory allocation and buffer management scheme which was used in the function *t1_encode_cblks* to minimize the cost of repetitive calls to *malloc()*. The particular buffer allocation function causes another loop-carried dependence detected by our tools. After privatization of the temporary buffer each thread manages its own buffer, eliminating the previously detected dependence. However, at this point our tool still requires manual intervention as it cannot be inferred that calls to the library function *malloc()* can be reordered. In an improved version of our tools we will include this information for this and other system functions.

3.3 Deriving a Parallel Implementation

After we have identified the available parallelism, the next step is to actually derive a parallel OpenMP implementation.

While OpenMP eliminates much the burden of explicit thread synchronization, we still need to address data-sharing. Basically, there are two kinds of memory access, *shared* and *private*. Based on the true data dependencies we classify the variables accordingly and for each private variable a separate copy is created for each thread.

Dynamic data structures need special attention to preserve correctness. In the case of *dwt_encode* it is straightforward for our tool to privatize the dynamically allocated data structure, since these are allocated within the dynamic context of the function. On the other hand, for *t1_encode_cblks* the situation more complicate since the allocation is just before the call to the function. Therefore, we need to ensure that the allocation is performed independently for each thread. We achieve this by propagating the parallel region out of the scope of *t1_encode_cblks* up to the point where the allocation takes place. The work-sharing directive will still result in the same parallel loop execution, but accesses to shared data before the work-sharing construct are executed only by the original thread in a region protected by an OpenMP *master* directive.

3.4 Performance Evaluation

We evaluated the achievable speedup of the parallelized benchmark on an SMP workstation ($2 \times$ Intel Dual-Core Xeon $5160/3GHz$), running Fedora Core 6 Linux. Both the sequential and OpenMP versions of the code were compiled using Intel® Compiler (v10.1) and optimization level $-O3$. The reported speedups are over the unmodified sequential code.

Performance results of both encoding and decoding using either parallelization at component or block-level of *t1_encode_cblks* are shown in figure 3. Given that we targeted approximately 70% of the original, sequential execution time in our parallelization and the theoretical limit given by Amdahl's Law, obtained speedups of $\approx 2\times$ are clearly promising. Comparing the performance of the component-level and the block-level decomposition it is clear that the latter is more scalable since the maximum number of components in an image is three. This also explains why the performance on four cores in Figure 3 is almost identical to the one on three cores.

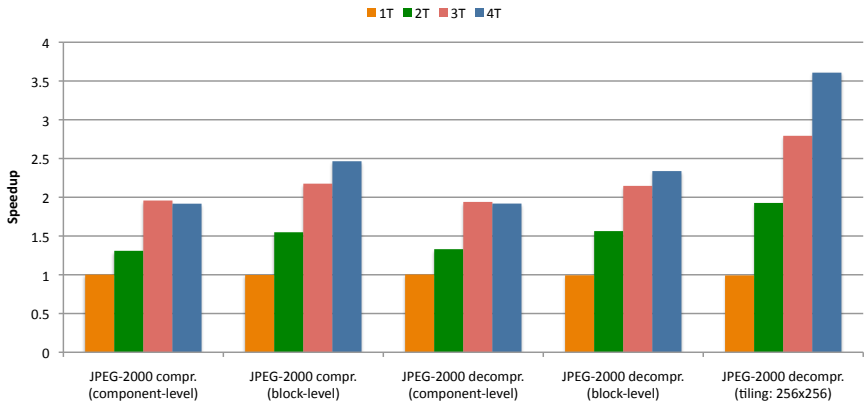


Fig. 3. Speedups achieved for various functional modes of the JPEG-2000 codec.

Tiling Parallelizing JPEG-2000 decompression at a tile-level achieved far better scalability than the one-tile parallelization schemes, reaching a speedup $3.6\times$ on four cores and 256×256 tiles. In fact, these results are in line with those obtained by manual parallelization by expert programmers [9, 10]

Scenarios For the JPEG-2000 application we have extracted execution scenarios and computed predicate expressions that can be used to determine later tasks at the earliest possible point in the program execution. The main observation is that this input-dependent behavior is in fact observable at the control-flow level. Typically, the cause for variability in multimedia applications is restricted to a small set of input parameters (e.g. file header information or command line arguments). The parameters that we experiment with were the following (A) turning lossy compression true/false, (ii) using one or multiple tiles of various

sizes qnd (*iii*) using source images of one or three components. In all cases the predicates generated by our tools accurately describe the precise comparisons that distinguish the four possible states. After finding these predicates, the next step is to locate the earliest point of the program where these predicates can be safely evaluated. As expected, this is the moment the relevant fields are read from file header.

4 Broader Evaluation

In this section we present additional performance results for selected applications taken from the Mediabench, UTDSP and MiBench benchmark suites. We have selected these applications based on their suitability for parallelization and compatibility with our tools.

In Figure 4 we present the speedups for eleven parallelized benchmarks. The average speedup using 4 threads is 2.66. Applications with ample coarse-grain DOALL parallelism in outer loops like *susan*, *stringsearch* and *compress* achieve relatively higher speedups and present good scalability. On the other hand applications, like *histogram*, and *lpc* exhibit either large sequential parts, or are only parallelizable in lower-levels of nested-loops. As a consequence, these applications are not scaling so well, attaining relatively lower speedups. In addition, application like *FFT* and *epic* manage to get significant performance gains despite the considerable slowdowns (57% and 32% respectively) of the single-threaded execution of the parallel code.

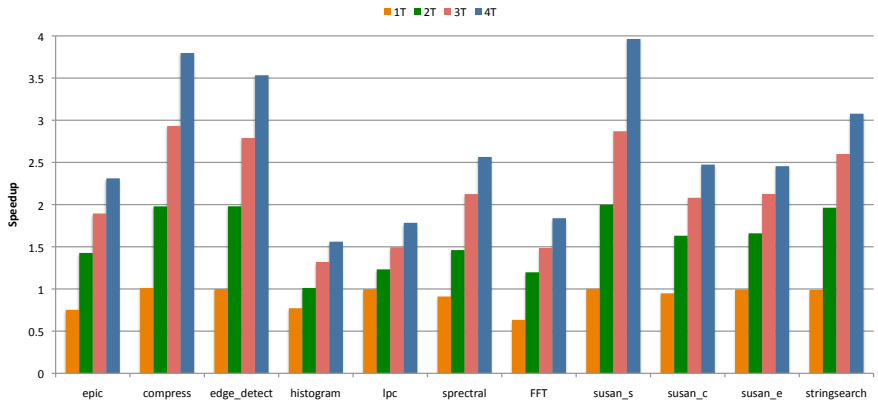


Fig. 4. Speedups achieved using the parallelized implementations of the benchmarks on a 4-core machine.

5 Related Work

Dynamic Dependence Analysis (DDA) [11] is a method to extract dependence information using the actual data accesses recorded during the runtime of a

program execution and aims at improving loop-level parallelization of scientific applications. Hybrid dependence analysis [12] unifies static and run-time analysis and has been successfully applied to automatic parallelization, where it helped extracting more loop-level parallelism in scientific codes.

Similarly, in [13] DDA is investigated as an aggressive, i.e. unsafe, dependence checking method to uncover loop-level parallelism. They present a study of the parallelization potential previously unexploited by the SUIF parallelizing compiler for a set of Digital Signal Processing (DSP) kernels and multimedia applications. The same authors show in [14] and [15] that dependence vectors extracted by means of DDA can be exploited to combine loop parallelization and functional pipelining. This work, however, is focused on the general feasibility of using dynamic dependence information, but does not present a specific methodology for deriving a parallel implementation.

The techniques presented in[6] are most relevant to our work. Load and stores memory operations are instrumented in Dynamic SimpleScalar to extract information about the data producer and consumer functions of the program. Based on this profile information an interprocedural data flow graph is constructed. This graph and additional profiling information summarizing the execution time of each function are used to cluster functions to form a functional pipeline. Then, this information is provided to the user who decides and implements the parallel code using multithreading. In a case study this method is evaluated against the *bzip2* benchmark, but it lacks features like code generation, automatic detection of higher-level parallelization constructs (e.g. reductions) and incorporation of the statically available information of our approach.

The problem of pipeline-parallelism exploitation in C programs is addressed in [16]. Based on user directed source code annotations a task pipeline is constructed and communication across task boundaries is tracked by means of dynamic analysis. While this approach is practical, it presumes programmer's insight to the underlying algorithm and it requires extensive user guidance and manual intervention.

MAPS [5] uses profiling data to annotate CDFGs for the extraction of task graphs from sequential codes. This technique is heavily dependent on a code partitioning framework (TCT) which targets a specific distributed memory MPSoC and assumes accurate static data dependence analysis. In addition, this approach has only been evaluated against two applications, JPEG and ADPCM.

Dependence-profiling approaches are also used in the context of Thread-Level Speculation (TLS) [17–19]. Unlike our proposal which uncovers *always-true* parallelism, TLS approaches use profiling information to mitigate the cost due to mispeculation on *may-dependencies* while preserving the sequential semantics.

6 Conclusions

We have demonstrated that profile-driven parallelization is a powerful methodology to uncover parallelism from complex applications such as JPEG-2000 and other media processing codes. Our automated approach to parallelism extrac-

tion and OpenMP code generation is capable of delivering performance figures competitive to those achieved by expert programmers during manual parallelization. We have shown that we can derive alternative parallelization schemes corresponding to different parallelization granularities, handle dynamically allocated data structures and compute predicate expressions for observed application scenarios.

References

1. Stolberg, H.J., Berekovic, M., et al.: HiBRID-SoC: a multi-core system-on-chip architecture for multimedia signal processing applications. DATE (2003)
2. Takamichi, M., Saori, A., et al.: Automatic parallelization for multimedia applications on multicore processors. IPSJ SIG Technical Reports **4** (2007)
3. Hall, M.W., Anderson, J.M., et al.: Maximizing multiprocessor performance with the SUIF compiler. Computer **29**(12) (1996)
4. Blume, W., Eigenmann, R., et al.: Effective automatic parallelization with POLARIS. IJPP (Jan 1995)
5. Ceng, J., Castrillon, J., et al.: MAPS: an integrated framework for mpsoc application parallelization. DAC (2008)
6. Rul, S., Vandierendonck, H., et al.: Function level parallelism driven by data dependencies. ACM SIGARCH Computer Architecture News **35**(1) (3 2007)
7. Gheorghita, S., Palkovic, et al. A system scenario based approach to dynamic embedded systems. ToDAES to appear.
8. Marcellin, M.W., Gormish, M.J.: The JPEG-2000 Standard. Kluwer Academic Publishers, Norwell, MA (1995)
9. Meerwald, P., Norcen, R., et al.: Parallel JPEG2000 image coding on multiprocessors. IPDPS (2002)
10. Norcen, R., Uhl, A.: High performance JPEG 2000 and MPEG-4 VTC on SMPs using OpenMP. Parallel Computing (2005)
11. Peterson, P., Padua, D.P.: Dynamic dependence analysis: A novel method for data dependence evaluation. LCPC (1992)
12. Rus, S., Rauchwerger, L.: Hybrid dependence analysis for automatic parallelization. Technical report (2005)
13. Karkowski, I., Corporaal, H.: Overcoming the limitations of the traditional loop parallelization. HPCN (1997)
14. Karkowski, I., Corporaal, H.: Design of heterogenous multi-processor embedded systems: Applying functional pipelining. IEEE PACT (1997)
15. Karkowski, I., Corporaal, H.: Exploiting fine- and coarse-grain parallelism in embedded programs. IEEE PACT (1998)
16. Thies, W., Chandrasekhar, V., et al.: A practical approach to exploiting coarse-grained pipeline parallelism in C programs. MICRO (2007)
17. Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., Torrellas, J.: Posh: a tls compiler that exploits program structure. PPoPP (2006)
18. Carlos García Qui n., Madriles, C., et al.: Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. PLDI (2005)
19. Wu, P., Kejariwal, A., Cascaval, C.: Compiler-driven dependence profiling to guide program parallelization. LCPC (2008)

Investigating Contention Management for Complex Transactional Memory Benchmarks

Mohammad Ansari, Christos Kotselidis, Mikel Luján,
Chris Kirkham, and Ian Watson

The University of Manchester
{ansari,kotselidis,mikel,chris,watson}@cs.manchester.ac.uk

Abstract. In Transactional Memory (TM), contention management is the process of selecting which transaction should be aborted when a data access conflict arises. In this paper, the performance of published CMs (contention managers) is re-investigated using *complex benchmarks* recently published in the literature.
Our results redefine the CM performance hierarchy. Greedy and Priority are found to give the best performance overall. Polka is still competitive, but by no means best performing as previously published, and in some cases degrading performance by orders of magnitude. In the worst example, execution of a benchmark completes in 6.5 seconds with Priority, yet fails to complete even after 20 minutes with Polka. Analysis of the benchmark found it aborted only 22% of all transactions, spread consistently over the duration of its execution.
More generally, all *delay-based* CMs, which pause a transaction for some finite duration upon conflict, are found to be unsuitable for the evaluated benchmarks with even moderate amounts of contention. This has significant implications, given that TM is primarily aimed at easing concurrent programming for mainstream software development, where applications are unlikely to be highly optimised to reduce aborts.

1 Introduction

Transactional Memory (TM) [1, 2] promises to ease concurrent programming effort in comparison to fine-grain locking, yet still provide similar scalability and performance. TM has seen a rise in research activity as it became clear that scalable software would be essential to take advantage of future multi-core technology.

In TM, code blocks that access shared data are defined as *transactions*, similar to how they are guarded by locks in traditional explicit concurrent programming. However, in contrast to locks, a TM runtime manages conflicting data accesses between the code blocks, and the developer is freed from the responsibility of orchestrating lock acquisition and release. The TM runtime logs all read and write accesses for each transaction, and compares them to detect conflicts. A *CM* (CM) is invoked when two transactions have conflicting accesses, and *aborts* one of the transactions. A transaction *commits* if it completes executing

its code block and does not aborted due to conflicts, making its writes to shared data globally visible.

Several CMs (or contention management policies) have been published [3–6] that offer a variety of algorithms for selecting the victim transaction to abort. Since their publication in 2004-5, the CMs have not been re-evaluated in the light of new, complex, TM benchmarks. In this paper, we investigate the performance of eight well-known CMs using Lee's routing algorithm [7] and a port of the STAMP benchmark suite [8]. Our investigation reveals several interesting results.

The most important result is that Polka, the published best-performing CM, suffers severe performance degradation when even a moderate (22%) proportion of executed transactions abort. This trend extends to other delay-based CMs investigated. Overall, Greedy and Priority provide the best performance, although Greedy offers stronger progress guarantees.

The paper is organised as follows: Section 2 describes the CMs, and Section 3 describes the complex benchmarks used in the evaluation. Section 4 presents the evaluation, and Section 5 further investigates the effect of changing Polka's parameters on its performance. Section 7 summarises the paper.

2 CMs

A CM is invoked by a transaction (the *calling* transaction) when it finds itself in conflict with another transaction (the *opponent* transaction). The CM decides which transaction should be aborted, although delay-based CMs wait for a finite amount of time to give the opponent transaction a chance to commit. The CMs investigated are: Aggressive, Polite (called Backoff in this paper), Eruption, Karma, and Kindergarten from [3], Polka from [4], Greedy from [5], and Priority, a new variant on Timestamp [3]. Of these, the following are delay-based CMs: Backoff, Eruption, Karma, and Polka. Brief descriptions of each CM follow.

Backoff gives the opponent transaction exponentially increasing amounts of time (delay) to commit, for a fixed number of iterations, before aborting it. Default parameters [3]: minimum delay of 2^4ns, maximum delay of 2^{26}ns, and 22 iterations.

Aggressive always aborts the opponent transaction immediately.

Karma assigns a transaction dynamic priority equal to the number of reads performed by it. Karma gives the opponent transaction, for a dynamic number of iterations, a fixed amount of time delay per iteration to commit. If the opponent transaction has not completed after all the iterations of delay, it is aborted. The delay given is 1000ns per iteration, and the number of iterations is equal to the opponent's priority minus the caller's priority.

Eruption, like Karma, assigns dynamic priorities to transactions based on the number of reads. Conflicting transactions with lower priorities add their priority to their opponent, increasing the opponent's priority, to allow the opponent to 'erupt' through any conflicts it has, or may have, to completion.

Kindergarten makes transactions abort themselves when they conflict with a transaction the first time, but abort the opponent if it is encountered in a conflict a second time, and so on.
Polka combines Karma and Backoff by extending Karma to give the opponent transaction exponentially increasing amounts of time delay to commit, before aborting the opponent transaction. The delay parameters used are identical to Backoff's. Additionally, if a conflicting object is being read by several transactions, Polka will immediately abort all of them if the calling transaction wishes to write to the conflicting object.
Greedy aborts an opponent transaction if it is younger or sleeping, else waits for it indefinitely (i.e., if the opponent is older, and not sleeping). A waiting, or suspended (e.g. during I/O) transaction is marked as 'sleeping'.
Priority aborts the younger of the conflicting transactions immediately. Priority can lead to a transaction never completing if it conflicts with an older transaction that has a fault that prevents it from completing. Greedy provides stronger progress guarantees than Priority by not allowing such a situation if the faulty transaction is suspended.

3 Platform & Benchmarks

Results are obtained on a 4x dual-core (8 core) Opteron 2.4GHz system with 16GB RAM, openSUSE 10.1, and Sun Java 1.6 64-bit with the parameters `-Xms1024m -Xmx14000m`. DSTM2 [9], a software TM implementation, is used to evaluate the CMs. Past research in contention management has also used DSTM2, its variants, or predecessors. In this paper, DSTM2 is set to its default configuration of eager validation, visible reads, and visible writes.

The benchmarks used are Lee's routing algorithm [7], and KMeans and Vacation from the STAMP benchmark suite (version 0.9.5) [8]. All the benchmarks have been ported to DSTM2. STAMP's Genome benchmark has been investigated, but is not presented as it generates very few conflicts on the hardware used in the experiments, and its results give no greater insight than the results from Vacation. As shown in Table 1, eight benchmark configurations are used[1], with a range of transactional conflict rates (contention) are used in this evaluation. The parameters used for each benchmark are those suggested by their respective providers, except KMeansHS and KMeansLS, which we created for quick experiments. Below, the benchmarks are briefly described, and in particular their concurrency characteristics are mentioned with respect to the inputs detailed in Table 1.

Lee's routing algorithm is a circuit routing algorithm that automatically connects pairs of points in parallel, without overlapping any existing connections.

[1] Note to reviewers: all eight configurations are presented for completeness, but the four high-contention configurations suffice, and only using them would allow for larger graphs in Figures 1 and 3. We intend to present only those four configurations in the final paper, unless requested otherwise.

Configuration Name	Application	Configuration
KMeansL	KMeans low contention	clusters:40, threshold:0.00001, input_file:random50000_12
KMeansH	KMeans high contention	clusters:20, threshold:0.00001, input_file:random50000_12
KMeansLS	KMeans low contention with small data set	clusters:40, threshold:0.0001, input_file:random10000_12
KMeansHS	KMeans high contention with small data set	clusters:20, threshold:0.0001, input_file:random10000_12
VacL	Vacation low contention	relations:65536, percent_of_relations_queried:90, queries_per_transaction:4, number_of_transactions:4194304
VacH	Vacation high contention	relations:65536, percent_of_relations_queried:10, queries_per_transaction:8, number_of_transactions:4194304
Lee-TM-ter	Lee low contention	early_release:true, file:mainboard
Lee-TM-t	Lee high contention	early_release:false, file:mainboard

Table 1. Parameters used for each benchmark configuration used in the evaluation.

The application loads pairs of points from an input file (measured execution time excludes parsing of the input file). Threads attempt to find a route between a pair of points by performing a breadth-first search of the grid from the first point, avoiding any grid cells occupied by previous connections. If a route is found, 'backtracking' writes the route onto the grid. Transaction-based routing requires backtracking to be performed transactionally. An early release [10] variant (Lee-TM-ter) removes data from the transaction's read set during the breadth-first search, which reduces false-positive conflicts. Execution is completely parallel, and the amount of parallelism is controlled by the amount of overlap in the connections attempted. The input file used has 1506 connections, i.e. transactions to commit, many of which are quite long, which increases contention and transaction execution time.

KMeans groups a large pool of objects into a specified number of clusters in two alternating phases. A parallel phase transactionally assigns objects to their nearest cluster, and a serial phase re-calculates cluster centres based on the mean of the objects in each cluster (initial cluster centres are random). Execution continues until two consecutive iterations generate similar cluster assignments within a specified threshold. The input files supply a large number of objects to cluster, and thus transactions to execute, but parallelism is controlled by the distribution of objects to the randomised cluster centres. Furthermore, randomised cluster centres result in considerable execution time variance, as observed in Section 4, Figure 1. Transactions are extremely short since they only read cluster centres and assign objects to the closest one.

Vacation is a travel booking database simulation that has operations to book or cancel cars, hotels, and flights on behalf of customers transactionally, and must update the customer's linked list of reservations as necessary. Threads can also modify the availability of cars, hotels, and flights transactionally. The input parameters lead to low contention for the hardware used, and transactions are short since they update the simulated database and customers' linked lists.

4 Initial Evaluation

Each benchmark configuration is executed using each CM, and using 1, 2, 4, and 8 threads. Each unique combination of benchmark configuration, CM, and threads is called an experiment. Each experiment is automatically terminated after 20 minutes, and when this occurs the associated CM is deemed 'too poor' for the given experiment, and we say the CM has *failed* the experiment. Results are averaged over eight runs of each experiment, except the failed experiments, which are run only three times to reduce doubt.

Figure 1 and Table 2 show the execution time results. Single thread results are presented to give an idea of execution time variance for the benchmarks, as obviously there is no contention or non-determinism when using a single thread, and thus execution times should be almost identical. This is true for all benchmarks except KMeans, where the randomised initial cluster centres have a significant impact on execution time. For performance comparisons, only the multi-threaded results are of interest, and less so with KMeans due to the large variances observed in it, although KMeans is still important due to the failures seen.

The results are mixed, with different CMs showing competitive performance with different benchmarks, reflecting the varying contention and execution characteristics of the benchmarks. For instance, Polka shows good performance in VacH and VacL, but Kindergarten does not, and the opposite is true in Lee-TM-t and Lee-TM-ter, especially as the number of threads increase. In general, a high consistency of good performance is only seen with Greedy and Priority. Averaged either over all threads, or only over 2-8 threads, the performance difference between them is less than 0.6%.

Polka is the published best CM [4], in the past producing best or near-best execution times for all benchmarks it has been executed with in comparison to other CMs. For VacH and VacL this is certainly the case, but this benchmark results in low contention on the hardware used. Strikingly, Polka is one of the worst in Lee-TM-t, and consistently joint worst in KMeansH and KMeansL. KMeans experiments, and Lee-TM-t exhibit large amounts of contention that increase with the number of threads, as shown in Figure 2. Worryingly, KMeansL at 4 threads has at least a 78% commit rate (using Priority CM, not theoretical best commit rate), but Polka fails to complete. Polka manages to complete execution with Lee-TM-t as there are only 1506 routes to connect transactionally, and thus the number of transactions executed is much fewer than KMeans, which typically executes millions of transactions. In KMeansHS and KMeansLS, which

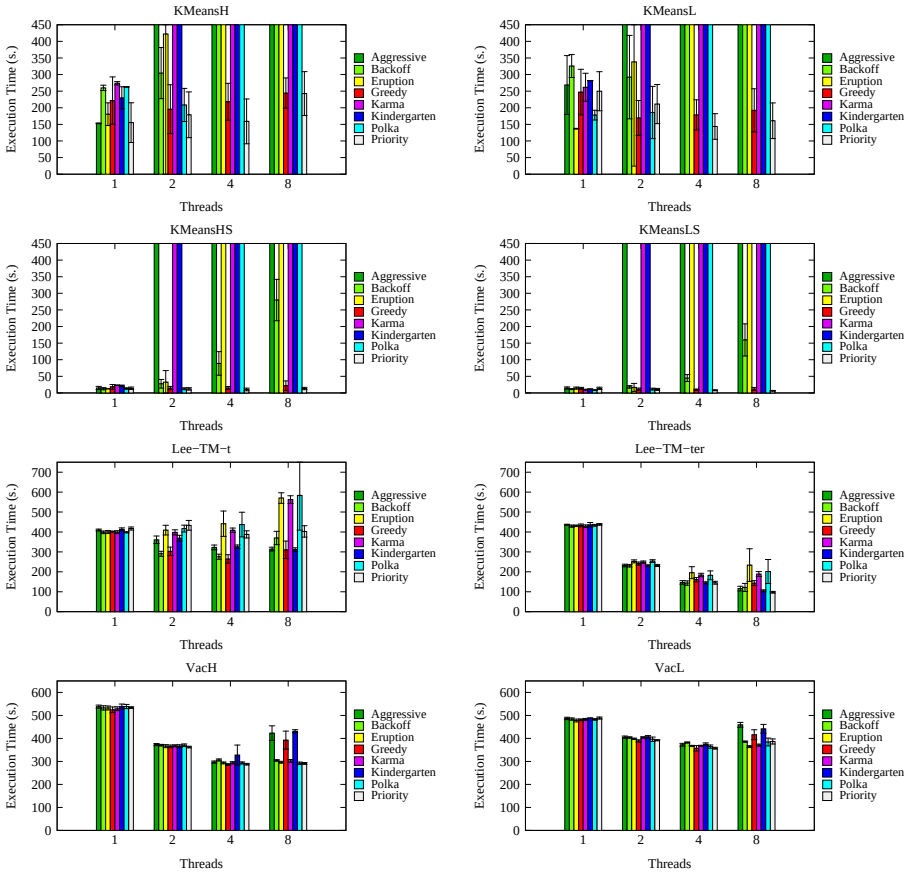


Fig. 1. Execution times with ±1 standard deviation. Times that go beyond the y-axis range are experiments in which the CM failed (to finish within 20 minutes). Less is better.

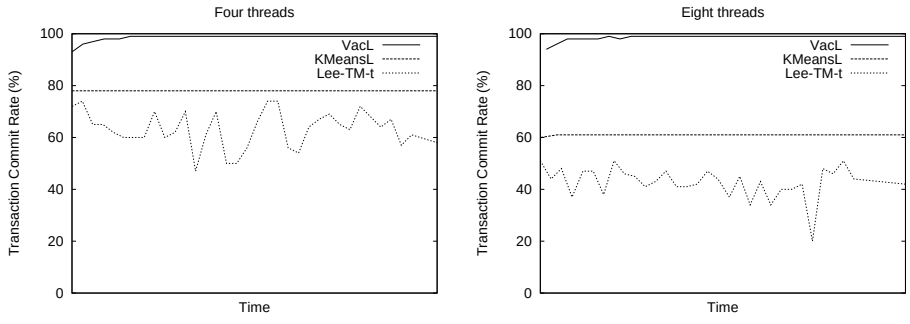


Fig. 2. Sample observed transaction commit percentages (using the Priority CM). VacL has far more commits (i.e., less contention) than Lee-TM-t or KMeansL

<= 110% of best execution time
<= 125% of best execution time
> 125% of best execution time, and < 20 minutes
> 20 minutes (DNF)

Benchmark	Thread	CM							
		Aggressive	Backoff	Eruption	Greedy	Karma	Kindergarten	Polka	Priority
KmeansH	2		304	203	196			208	**179**
	4				218				**159**
	8				244				**243**
KmeansL	2		292	187	**169**			186	211
	4				179				**144**
	8				192				**161**
KmeansHS	2		27.7	14.7	14.6			12.5	**12.1**
	4		88.9		14.8				**10.6**
	8		279.8		21.8				**13.4**
KmeansLS	2		18.3	10.7	11.3			11.6	**10.6**
	4		44.8		9.7				**8.0**
	8		159.7		12.0				**6.5**
Lee-TM-t	2	361	**290**	409	303	398	369	417	433
	4	322	276	448	**264**	409	326	437	388
	8	314	370	571	**310**	563	311	584	403
Lee-TM-ter	2	232	**230**	252	241	249	231	254	232
	4	147	**143**	198	161	184	145	182	145
	8	116	122	242	143	189	104	202	**98**
VacH	2	373	371	366	366	368	366	371	**363**
	4	297	306	294	**286**	294	328	293	288
	8	423	304	296	393	302	431	292	**291**
VacL	2	405	404	398	**389**	404	407	396	392
	4	373	382	367	**357**	368	375	364	358
	8	459	386	**364**	416	371	442	384	387

Table 2. Execution times (in seconds). Bold indicates best time for an experiment (i.e., a row).

typically take less than 20 seconds to complete with the well-performing CMs, Polka fails to complete in 20 minutes.

Critically, wherever Polka fails, Karma also always fails, but Backoff does not also always fail. As mentioned earlier, Polka combines Karma and Backoff. The difference between the first two and the latter is the number of delay intervals: Backoff has a fixed number of 22, whereas the other two calculate it dynamically. By deduction, the average number of iterations in Polka and Karma must have been larger than 22. We also note that Eruption similarly fails in KMeans experiments, and performs poorly in Lee-TM-t. This suggests that delay-based CMs in general are not suitable for applications that exhibit non-negligible amounts of contention.

Finally, KMeans experiments with 2 threads deserve further attention because in these Polka completes execution with a competitive execution time, and Karma does not. The difference between Polka and Karma are a) that Polka aborts a set of reading transactions immediately if the calling transaction wishes to write, and b) the amounts of time delay per iteration. Although the first point may explain Polka's higher performance, the second point calls into question the choice of parameters used for Polka, and, more generally, whether they were to blame for the poor performance observed in other experiments above. We investigate this further in the next section.

5 Investigation of Polka's Parameters

Polka has two tuning parameters: `LOG_MIN_BACKOFF` and `LOG_MAX_BACKOFF`, which bound the exponential delay. Polka calculates delay for an iteration as 2^n nanoseconds where n starts at `LOG_MIN_BACKOFF`, and increments by one every iteration up to `LOG_MAX_BACKOFF`. A spin loop that calls Java's `System.nanoTime()` is used to determine if the required delay time has passed. In this section, we investigate the performance effect of altering Polka's parameters. Scherer *et al.* [4] do not suggest a method by which the parameters should be calculated so we use the scheme explained below.

Through empirical evaluation we determined the minimum timing accuracy of our system to be 3600±100 nanoseconds. This is significantly higher than the published minimum value of 2^4ns (by using a `LOG_MIN_BACKOFF` of 4), but testing on other x86/Linux platforms similarly gave us minimum accuracies much higher than 2^4ns, and never less than 2500ns. Thus we set `LOG_MIN_BACKOFF` to 11, to give a calculated minimum delay of 2^{11} nanoseconds = 2048 nanoseconds $\approx$ 2 microseconds, but which of course rounds up to the minimum system accuracy. Since the original values were based on SPARC/Solaris, Polka potentially needs parameter re-tuning for every new hardware platform used.

For `LOG_MAX_BACKOFF` we select a range of values based on approximately *half* the average committed transaction execution time for each benchmark. The observed values are shown in Table 3 . We select `LOG_MAX_BACKOFF` values of 13 (≈8 microseconds), 16 (≈65 microseconds), 19 (≈528 microseconds), and 28 (≈134 milliseconds). The results of the executions are shown in Figure 3, again

averaged over eight runs for all experiments, except failed experiments, which are again only run three times.

	KMeans*	Lee-TM*	Vac*
1 thread	12	264288	126
2 threads	19	330592	167
4 threads	210	380275	265
8 threads	422	524702	537

Table 3. Average committed transaction execution time for each benchmark, in microseconds. Both high and low contentions not shown as execution times in the same order of magnitude for exponential delay calculation.

There is minimal effect of changing the parameters in VacL and VacH, as these have low contention, which leads to the CM being invoked rarely. For the remaining experiments, different parameters give the best performance improvement over the default parameters. Although the improvements are slight, this suggests per-application parameter tuning may be necessary. However, the important results have not changed, and Polka continues to give extremely poor performance in all KMeans experiments with 4 or more threads, irrespective of the wide range of tuning parameters used. This strengthens our original hypothesis: delay-based contention management may be unsuitable for applications with appreciable amounts of aborting transactions.

6 Related Work

Guerraoui et al. [5] developed the Greedy CM, which has provable progress properties, and their evaluation showed Greedy performed on par with Polka. Our results confirm their findings. Scherer and Scott [4] evaluated Polka using six benchmarks in nine benchmark configurations. Three benchmarks added, removed, and queried elements in a set, the fourth implemented a concurrent stack, the fifth a 'torture test' that updated all values in an array per transaction, and the sixth an LFU cache simulator. Clearly their benchmarks exhibited contention to provide variation in execution time between CMs, and they found Polka to be a consistent top performer, often by large margins over other CMs. Their investigation differs from ours in one critical way. All the CMs they investigated are delay-based, except Kindergarten, and they did not include Greedy or Priority, as neither had been published. Our investigation found all delay-based CMs and Kindergarten performed poorly compared to Greedy and Priority in benchmarks with appreciable amounts of aborts.

Our previous work in adaptive concurrency control [11], which dynamically changes the number of transactions executing simultaneously with respect to the measured transaction commit rate, resulted in applications' performance at any number of initial threads being similar to best-case statically-assigned number

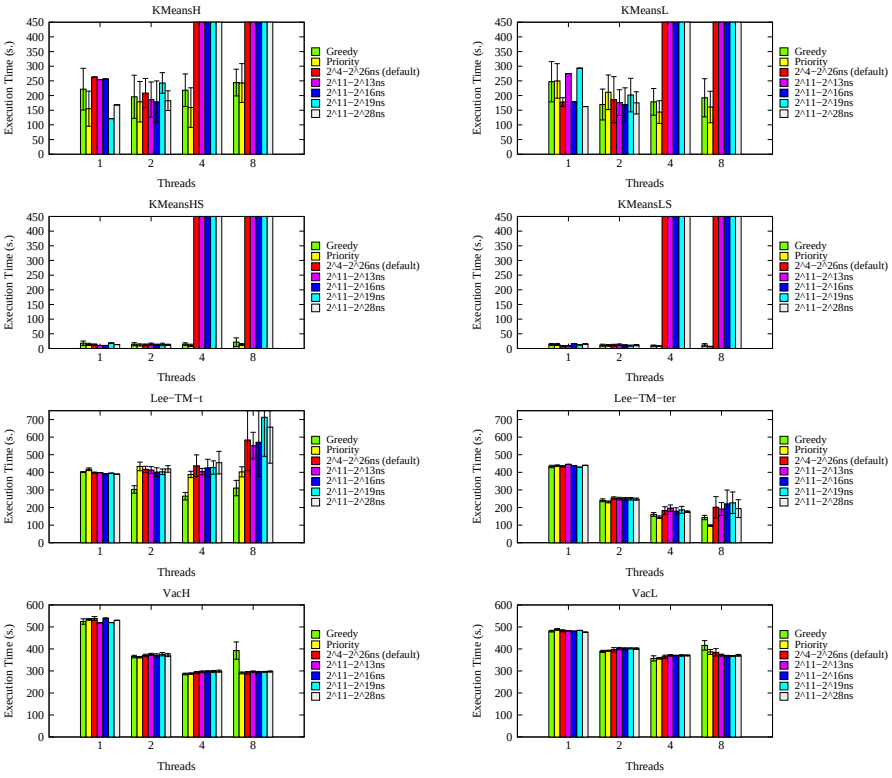


Fig. 3. Execution times with ±1 standard deviation for Greedy, Priority, and Polka with several minimum and maximum delay parameters. Lower is better.

of threads, for a given CM. Our adaptive mechanism would have had a dramatic positive effect on the performance of Polka in its failed experiments. Additionally, our work in reducing repeat conflicts [12] showed Polka's performance could be improved in applications that exhibit repeat conflicts.

7 Summary

This paper re-evaluates well-known CMs (CMs) in the light of newly published complex benchmarks. A number of important findings result from this investigation. In general, we found Priority and Greedy to be joint best-performing CMs, although Greedy provides stronger progress guarantees than Priority.

Although Polka still provides competitive performance in benchmarks with very low contention, the most important finding of our investigation suggests Polka, the established best-performing CM, and in general all delay-based CMs, are unsuitable for the evaluated benchmarks that exhibit even moderate amounts of aborting transactions. Although we do not quantify what is meant by 'moderate', in one benchmark Priority executed in 6.5 seconds with an average of 78% of transactions committing (i.e., 22% aborting), whilst Polka failed to complete executing the benchmark in 20 minutes (after which time the execution was terminated). This result has wider implications given that TM is strongly aimed at easing concurrent programming for mainstream software development, where execution is unlikely to be highly optimised to reduce aborts in the general case.

Polka has two tuning parameters, and investigating a range of values concluded there was no benefit in tuning them to improve the extremely poor results seen in KMeans experiments, although tuning led to a degree of performance improvement in the remaining results. However, different parameters provided better performance for different applications, suggesting the need for application-specific tuning. Furthermore, the need to re-evaluate the parameters for every hardware platform used was also highlighted. Conversely, Greedy and Priority have no parameters.

References

1. Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.
2. Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM Press, August 1995.
3. William Scherer III and Michael L. Scott. Contention management in dynamic software transactional memory. In *CSJP '04: Workshop on Concurrency and Synchronization in Java Programs*, July 2004.
4. William Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the 24th Annual Symposium on Principles of Distributed Computing*, pages 240–248. ACM Press, July 2005.

5. Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *PODC '05: Proceedings of the 24th Annual Symposium on Principles of Distributed Computing*, pages 258–264. ACM Press, July 2005.
6. Rachid Guerraoui, Maurice Herlihy, Michal Kapalka, and Bastian Pochon. Robust contention management in software transactional memory. In *SCOOL '05: Workshop on Synchronization and Concurrency in Object-Oriented Languages*, October 2005.
7. Ian Watson, Chris Kirkham, and Mikel Luján. A study of a transactional parallel routing algorithm. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pages 388–400. IEEE Computer Society Press, September 2007.
8. Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 69–80. ACM Press, June 2007.
9. Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 253–262. ACM Press, October 2006.
10. Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 92–101. ACM Press, July 2003.
11. Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Advanced concurrency control for transactional memory using transaction commit rate. In *EUROPAR '08: Fourteenth European Conference on Parallel Processing*, August 2008.
12. Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Steal-on-abort: Dynamic transaction reordering to reduce conflicts in transactional memory. In *HIPEAC '09: Fourth International Conference on High Performance and Embedded Architectures and Compilers*, January 2009.

Profiling Transactional Memory applications on an atomic block basis: A Haskell case study

Nehir Sonmez[1,2] , Adrian Cristal[1], Tim Harris[3], Osman S. Unsal[1], and Mateo Valero[1,2]

[1] Barcelona Supercomputing Center
[2] Universitat Politècnica de Catalunya
[3] Microsoft Research Cambridge
{nehir.sonmez, adrian.cristal, osman.unsal, mateo.valero}@bsc.es, tharris@microsoft.com

Abstract. In many recent works in Transactional Memory (TM), researchers have profiled TM applications based on execution data using lumped per-transaction averages. This approach both omits meaningful profiling information that can be extracted from the transactional program and hides potentially useful clues that can be used to discover the bottlenecks of the TM system. In this study, we propose partitioning transactional programs into executions of their atomic blocks (AB), observing both the individual properties of these ABs, and their effects on the overall execution on a Software Transactional Memory (STM) benchmark in Haskell. Profiling on the AB-level and focusing on the intra-AB relationships helps to (i) characterize transactional programs with per-AB statistics, (ii) examine the conflict relationships that are caused between the ABs, and thus (iii) to identify and classify the shared data that often cause conflicts. Through experimentation, we show that the AB behavior in most of the Haskell STM benchmark applications is quite heterogeneous, proving the need for such fine-grained, per-AB profiling framework.

1 Introduction

Some 30 years ago, Lomet proposed an idea to support atomic operations in programming languages, similar to what had already existed in database systems [17]. More than a decade has passed since the first description of a Transactional Memory system in hardware (HTM) [13], and that of a software-only implementation [22]. Nowadays, TM is being seen as one of the most promising ways of the programming revolution which is almost late: many- and multi-cores are already dominant in the market. While the upcoming Rock processor [26] is proof that TM hardware is near, one thing is for sure: "Multicore architectures are an inflection point in mainstream software development because they force developers to write parallel programs" [1].

Software Transactional Memory (STM) promises to ease the difficulty of programming using conventional mechanisms such as locks in a threaded, concurrent environment. Locking has many problems: simple coarse-grained locking does not scale well, while more sophisticated fine-grained locking risks introducing deadlocks and data races. Many scalable libraries written using fine-grained locks cannot be easily composed in a way that retains scalability and avoids deadlock and data races. Although STM-based algorithms can be expected to run slower than ad-hoc non-blocking algorithms or fine-grained lock based code, they are as easy as using coarse-grained locks: one simply brackets the code that needs to be atomic.

Although the number of STM benchmarks in the literature keep increasing [3,9,18,19,21], very little experience exists on the characterization of TM programs.

Some relevant transactional attributes discussed in these benchmarks as well as in [5,6,29] include the total transactional time, the number of, and the time spent committing/aborting transactions, readset/writeset sizes and their corresponding reads/writes. However, while using lumped sum averages for characterization might be acceptable for some arbitrary execution, this is not the most useful way to profile TM programs. For instance, a rough estimate might be obtained by dividing the total number of transactional reads made by the program by the total number of transactions, to conclude that the program makes so many transactional reads per transaction, but we argue that such a statistic is mostly useless. Firstly, a program can be composed of various long-short (time), large-small (readset and writeset sizes) transactions and different abort rates, and an average value will omit many useful profiling data. Secondly, a typical program includes execution patterns, loops, function calls; deserving and perhaps requiring a better, finer-grained characterization framework. Furthermore, omitting such profiling information can result in ignoring simple pathological problems of the TM system.

Although an STM program runs transactions, these are actually executions of a specific atomic block, the piece of code inside the *atomically{}* , marked to run atomically to the rest of the system in an all-or-nothing fashion. In runtime, atomic blocks execute as transactions. To better reason about transactional attributes, we propose partitioning all benchmarks by their atomic blocks (AB), their bare transactional source code. In particular, the contributions of this paper are as follows:

- A Haskell benchmark suite is presented where each program is partitioned into its atomic blocks, and all TM attributes such as: the number of times committed/aborted, time spent doing work and time spent committing and aborting transactions, the number of reads and writes in committing and aborting transactions and readset and writeset sizes are grouped as executions of the particular atomic blocks. The AB behavior in most of the Haskell STM benchmark applications turns out to be quite heterogeneous, proving the need for finer-grained, per-AB profiling framework.
- Critical sections can be seen as AB conflict matrices of conflicting shared variables, which help to reason about the transactional aborts. This is achieved by collecting during runtime, per each AB, (i) the set of the conflicting shared data (in Haskell's terms, transactional variables, or *TVars*) (ii) other ABs that these *TVars* conflict on, and the number of times the exact scenario occurs.
- Due to the merits of profiling conflicting ABs, we show that some ABs tend to get scheduled to run concurrently and abort one another often, while others do not. Such findings point towards future per-AB runtime optimizations.

Up to date, to the best of our knowledge, a brief summary of per-AB statistics that include the number of commits, abort and readset sizes, has been featured only by the Intel STM Compiler [2]. The work in [27] can examine the AB conflicts in TM applications; however it does not feature any intra-AB dependence relationships or the identification of the shared data that cause transactions to conflict. One work that utilizes conflicting variables to make runtime decisions does not identify ABs [28]. Furthermore, [4, 8] are two recent works that recognize and optimize on some of the issues that can also be detected by our profiling mechanism.

On the next section, we give an introduction to the Haskell STM runtime environment, followed by a description of our modifications described on Section 2. Section 3 explains our scope in depth on a singly linked-list example. Section 4 presents the Haskell STM benchmark used and the corresponding results. Section 5 wraps up the conclusions.

TABLE I STM Operations in Haskell

STM operations	TVar operations
`atomically :: STM a -> IO a`	`newTVar :: a -> STM (TVar a)`
`retry :: STM a`	`readTVar :: TVar a -> STM a`
`orElse:: STM a -> STM a -> STM a`	`writeTVar :: TVar a -> a -> STM()`

2 Glasgow Haskell Compiler and Proposed Approach

2.1 The Glasgow Haskell Compiler and Runtime System

The Glasgow Haskell Compiler (GHC) [12] is a compilation and runtime environment for Haskell 98 [7], a pure, lazy, functional programming language. Since a few years now, the GHC has natively contained STM functions; abstractions for communicating between explicitly-forked threads using transactional variables, built into the Concurrent Haskell library [14]. STM can be expressed elegantly in such a declarative language, where Haskell's type system, particularly the monadic mechanism allows threads to access shared variables only when they are inside a transaction [10]. This restriction that guarantees strong atomicity can be violated under other programming paradigms, for example, as a result of access to memory locations through the use of pointers.

Haskell STM provides a safe way of accessing shared variables among concurrently running threads through the use of monads, allowing only I/O actions to be performed in the IO monad, and STM actions in the STM monad. This ensures that only STM actions and pure computation can be performed within a memory transaction, which also makes it possible to re-execute transactions.

Although the core of the language is very different from other languages such as C# or C++, the actual STM operations are used in a simple imperative style and the implementation uses the same techniques used in mainstream languages. Haskell has a small runtime system written in C, making it easy to experiment modifications. The STM support that has been present for some time led both researchers and functional programmers to write various applications, some of which were profiled in this work.

Threads in Haskell STM communicate by reading and writing transactional variables, or TVars, using a set of transactional operations, including allocating (*newTVar*), reading (*readTVar*), and writing (*writeTVar*) transactional variables (Table 1).

2.2 The transactional execution in GHC

In concurrent applications written in Haskell STM, an atomic block is constructed inside the *atomically{}* function. *Atomically{}* takes a memory transaction, of type *STM a*, and delivers an *I/O* action that, when performed, runs the transaction atomically with respect to all other memory transactions. This provides the sequence of code to be executed in an all-or-none fashion to the transactional management system. Following the transaction's start with *atomically{}*, transactional variables are created, read from and/or written to in program order, with the system maintaining a per-thread transaction log for all tentative accesses, called the transactional record (*TRec*). All the variables that are written to are called the "writeset" and all that are read from are called the "readset" of the transaction. These two sets usually overlap. This speculative execution, where the actual computational work takes place is called the work phase (WP).

Later comes the commit phase (CP), where the transaction might commit its changes into the memory. In this phase, where Haskell STM performs lazy conflict detection, the runtime system validates that the transaction was executed on a consistent system

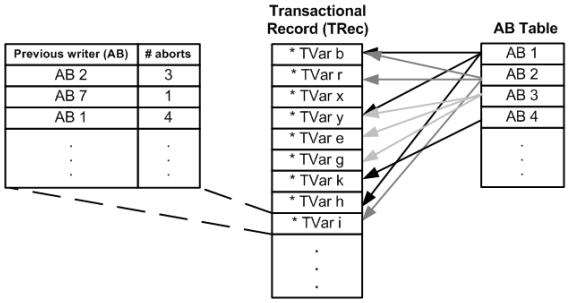


Fig. 1. Each AB keeps information on each conflicting TVar, its previous writer that caused the conflict, and the number of times each previously writing AB aborts the transaction on the specific TVar.

state, and that no other finished transaction may have modified relevant parts of this state in the meantime. First, a global lock is acquired (in the coarse-grained locking scheme of GHC). Next, validation is performed on the first part of the two-phase commit, going through the readset and the writeset, comparing each variable with its local original value that was obtained at the time of access. If all *TVars* accessed by the transaction turn out to be consistent, then all new values are actually committed into the memory in the second part of the two-phase commit. On encountering an inconsistent *TVar*, which means that some concurrent transaction committed conflicting updates, the system immediately aborts the transaction, discards the tentative modifications, and restarts the transaction. At the end of the CP, the global lock is released, which causes the course-grained locking technique to allow only one transaction to be in commit phase at a time. GHC also features a per-*TVar*, fine-grained locking version of the commit phase, however its behavior was not studied for this work, although the proposed profiling infrastructure would work with different locking implementations as well.

2.3 Conflicts and Aborts

A conflict occurs if two or more transactions access the same memory address and at least one of the accesses is a write. Transactions in Haskell STM that discover a conflict during validation immediately abort, discard all non-committed data, and restart. It might be useful to see all transactions in two groups: transactions that do not abort, and those that do. Inversely, transactions that make others abort and transactions that do not make abort exist, however these two categorizations are distinct. By grouping transactions as executions of ABs, it is possible to identify aborts in terms of specific conflicting *TVars* (*confTVar*) and see between which aborting and aborted (victim) ABs they occur. In the next section, we show how some *TVars* can be more involved in aborts than others. Having this information at runtime could make it possible to fashion optimization techniques that involve the aborting *TVars*.

The biggest overheads in STM come from keeping track of transactional variables, and causing wasted work by aborting the transactions in case of a conflict. Previous work in [19] shows wasted work for up to 50% on the Haskell benchmark. Some of those programs are also used on this work. Aborts can be costly depending on their frequency, and the amount of work done inside the transaction. Repetitive aborts degrade the performance of the program. One motivation for this work is to find out more about how aborts happen and if similar or repetitive aborts exist.

2.4 GHC Hooks

Precisely what makes AB-based profiling possible is that each atomic block execution, being a specific function call, is pointed to in GHC by a variable called the *atomically_frame_pointer*. This provides a unique identifier in runtime to group

atomic blocks encountered in the code, pointing to the beginning of the code inside of *atomically{}*. Although it would require modifications to the program code, another way of uniquely identifying ABs is by passing an explicit additional parameter to the atomically function, such as an integer constant, and then partitioning the profiling information accordingly. Such an approach of finding the appropriate AB identifier and accordingly grouping transactional profiling data should be feasible under many other STM systems.

ABs are added to a global AB Table (Fig. 1) when *atomically{}* is invoked. During CP, at the time of validation where the contents of the *TVars* are checked for consistency with their prior-to-start transactional values, once a *TVar* fails, the AB Table is updated with this new occurrence of a conflict along with (1) the AB that is currently being aborted, (2) the specific conflicting *TVar*, (3) the AB that last wrote to that *TVar*, causing the abort, and (4) the number of times this exact scenario occurred.

3 An Example: Profiling a Linked-List

To demonstrate the proposed approach, a transactional linked-list (LL) was implemented in Haskell performing 10% deletes, 10% inserts and 80% lookups. The links between the nodes (Fig. 2) are represented as *TVars*, therefore during list traversal, *TVars* are accumulated into the transaction's readset. Although this is not the most suitable concurrent implementation of a linked structure [24], it's a simple example that should serve to demonstrate our approach. Throughout the program, the atomic blocks are commented with "ABx" denotations.

The *main* function in line 36 in Fig. 3 takes the necessary arguments from the user (the number of operations, the list capacity, and the number of threads to fork) and calls *main1*. Although Haskell is a language with lazy evaluation, the *do{}* "syntactic sugar" enables to construct of a sequence of actions. Line 22 in *main1* is the first atomic block, AB1, where the initial half-list is created atomically, and following that, AB2 outputs the number of elements that it contains. Later, the Haskell synchronization variables (*MVars*) are created. Then, a desired number of threads are forked, each performing a lookup/insertion/deletion of a random integer, executing AB3, AB4 or AB5 respectively; and (ideally) performing deletes a tenth of the time, inserts another tenth, and on the rest doing lookups. When the total number of operations complete running on the threads, the *MVars* are taken: this barrier will make sure that all threads finish executing before the main thread ends. By invoking AB6 and AB7 on lines 29 and 30, the program will print the ultimate number of elements that the list contains. Finally, before the end, *ReadTStats*, a function that enables printing out the gathered statistics is invoked. In this program, AB1-AB2, and AB6-AB7 pairs can serve to illustrate the similarities, and AB3-AB4-AB5 the variation between atomic blocks. Due to space limitations, we cannot provide all the used functions, however a similar insert function can be found in [24]. The insert and delete functions (AB4 and AB5 respectively) are similar, except that delete performs one more *readTVar* because it needs to read two nodes ahead.

```
data LinkedList =
     Start  {nextNode :: TVar LinkedList}
     | Node {val :: Int, nextN :: TVar LinkedList}
     | Nil
```

Fig. 2. The data declaration of the transactional linked-list.

```
createThread :: Int -> TVar ListNode -> Int -> MVar Int -> IO ThreadId
createThread numOps tList maxNumber mvar =
     forkIO ( do { callNTimes numOps
                   (do
                   { rnd1 <- randomRIO (1::Int, 10)
                   ; rnd2 <- randomRIO (1::Int, maxNumber)
                   ; case rnd1 of
                   ; 1 -> do { atomically (deleteListNode tList rnd2) -- AB5
                             ; return ()}
                   ; 2 -> do { atomically (insertListNode tList rnd2) -- AB4
                             ; return ()}
                   ; otherwise -> do { atomically (lookupListNode tList rnd2) -- AB3
                             ; return ()}
                   })
                   ; putMVar mvar 1
                 }  )

createThreads :: Int -> Int -> TVar ListNode -> Int -> [MVar Int] -> IO()
createThreads n numOps tList maxNumber mvars
            = mapM_ (createThread numOps tList maxNumber) mvars

main1 :: Int -> Int -> Int -> IO ()
main1 numops listLength numThreads = do
     { ourList <- atomically (createSampleList (reverse[x|x<-[1..listLength],((mod x 2)==0)])) --AB1
     ; ourListAsString <- atomically (toString ourList) --AB2
     ; putStrLn (show (length ourListAsString))
     ; ourTList <- newTVarIO ourList
     ; mvars <- replicateM numThreads newEmptyMVar
     ; threads <- createThreads numThreads numops ourTList listLength mvars
     ; mapM_ takeMVar mvars
     ; ourList2 <- atomically(readTVar ourTList) --AB6
     ; listAsString <- atomically (toString ourList2) --AB7
     ; putStrLn (show (length listAsString))
     ; stats <- readTStats
     ; putStrLn (show stats)
     ; return ()
     }

main :: IO()
main = do { args <- getArgs
     ; let numops = read (args!!0)
     ; let listlength = read (args!!1)
     ; let numthreads = read (args!!2)
     ; main1 numops listlength numthreads}
```

Fig. 3. Major functions in the linked list code

3.1 Observed Statistics

All executions in this work, including the LL example were run on a 128-core SGI Altix 4700 system with 64 dual-core Montecito (IA-64) CPUs running at 1,6 GHz. The command "*MainLL 16000 100 8 +RTS –N8*" was executed to run the program on Fig. 3: 16000 is the total number of operations, 100, the maximum list size, and 8 the number of Haskell threads to use. The RTS option *–Nx* lets the user choose the number of OS threads to utilize, and since there are enough cores in the system, there is one thread per core.

The seven atomic blocks of the linked-list program and their individual statistics can

TABLE II AB Properties of the Linked List program

AB	# A	# C	write set	Read-only set	Total writes (A)	Total reads (A)	Total writes (C)	Total reads (C)	CP (A) ‡	CP+WP (A) ‡	Val (C) ‡	CP (C) ‡	CP+WP (C) ‡	Total Conf TVar
AB1	0	1	1	1	0	0	50	100	0	0	4,732	10,290	967,093	0
AB2	0	1	0	51	0	0	0	51	0	0	4,105	8,393	168,660	0
AB3	2,735	12,745	0	348,072	0	207,283	0	670,654	1,116,122	2,558,596	1,097,159	1,211,440	5,206,129	542
AB4	187	1,690	807	45,280	81	13,799	807	88,794	61,534	107,894	146,314	153,328	379,105	129
AB5	213	1,565	800	42,638	107	23,226	800	120,950	67,174	157,486	132,227	139,859	401,216	147
AB6	0	1	0	1	0	0	0	1	0	0	3,135	8,974	48,316	0
AB7	0	1	0	58	0	0	0	58	0	0	4,296	8,425	292,619	0
total	**3,135**	**16,004**	**1,608**	**436,101**	**188**	**244,308**	**1,657**	**880,608**	**1,244,830**	**2,823,976**	**1,391,968**	**1,540,709**	**7,463,138**	**560**

‡: in 1000s of cycles, A: Aborted transactions, C: Committed transactions, Val: Validation, CP: Commit Phase, WP: Work Phase

be seen on Table 2. There are three ABs that get aborted, and four that do not—basically in this example, the heavyweight atomic blocks AB3, AB4 and AB5 tend to get scheduled concurrently and cause conflicts.

The total transactional execution distribution can be seen in Figure 4. The wasted work caused by aborting transactions constitutes 14% of the overall transactional runtime. 78% of this is due to aborts of the atomic block AB3, which in total takes up 82% of the transactional execution, either committing or aborting. AB3 (lookup) commit counts are an expected eight times to those of the atomic blocks AB4 and AB5. Its writeset, commit and abort writes are zero, since it only performs lookup. Comparing AB1 and AB6, two atomic blocks with single element readsets, AB1 actually creates a list of 50 elements (writes:50), traversing all the possible nodes to create all the even numbers from 1 to 100 (reads: 100). AB6 does less work, only reading the ultimate set, which turns out to contain fifty-seven (plus one for the start node) elements in the end. This can be seen as well on the transactional reads and the readset statistics of AB7, which is very similar again to those of AB2.

For committed transactions, we profile executions in three parts: The work phase, the validation inside the commit phase, and the rest of the CP, writeback, where the writes are actually committed to memory. GHC will still check all *TVars* to see if any need to be written back, this is the reason why the atomic block that corresponds to the lookups, AB3, also spends time in the commit phase in aborted transactions. Read-only atomic blocks like AB3 cannot be aborters of other ABs.

3.2 Conflicting TVars and Aborts

The final column on Table 2, *confTVar*, indicates on how many distinct TVars the aborted transactions are due to. Our profiling mechanism collects information about all aborts during runtime where these conflicting TVars as well as the corresponding aborting/victim transactions can be observed. Theoretically, each run could have a different set of *TVars*, depending for example, on the scheduling order of the atomic blocks, the loops in the code, the randomized variables etc, so ABs do not have to exhibit the same behavior on each execution. However, for our benchmark, we generally see quite stable and meaningful behavior.

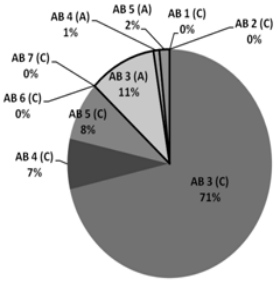


Fig. 4. Distribution of the total transactional execution (C: Committed transactions, A: Aborted transactions)

TABLE III
THE DISTRIBUTION OF CONFLICTING TVARS

Aborter/ victim	AB3	AB4	AB5	*total*
AB3	0	0	0	
AB4	362	82	77	52
AB5	356	69	87	51
total	**718**	**151**	**164**	**103**
confTVars	542	129	147	81
difference	**176**	**22**	**17**	**21**

TABLE IV
THE DISTRIBUTION OF ABORTS VS CONFLICTING TVARS

Aborter/ victim	AB3	AB4	AB5	*total*	AB3	AB4	AB5
AB3	0%	0%	0%	0%	0	0	0
AB4	44%	3%	3%	50%	65%	15%	14%
AB5	43%	3%	4%	50%	64%	12%	16%
total	87%	6%	7%				

TABLE V
THE DIFFERENCE: AB ABORT DUO

Aborter/ victim	AB3	AB4	AB
AB3	0	0	
AB4	122	14	1
AB5	54	8	
total	176	22	1

Table 3 shows the distribution of the *confTVars* that were collected during the execution in terms of the aborter and the victim ABs. *ConfTVars* is the number of distinct *TVars* inside the AB that caused an abort at least once. However, they can appear in more than one AB, therefore the total number of conflicting *TVars* on Table 2 is not the sum of all *confTVars* in all ABs: this sum instead, is the total number of unique *TVars* that conflicted. On the left side in Table 4, all 3,135 aborts that occurred are classified into aborter and victim AB groups, and on the right side, the corresponding shared conflicting *TVars* are depicted. For example, the case where AB5 caused an abort on AB3 constitutes 43% of the total aborts, but involves 64% of the total critical section (the conflicting *TVars*). Inversely, the case of two concurrent AB5 executions aborting one another occurs in 4% of the total aborts but involves 16% of the conflicting *TVars*. Overall, AB3 was a victim (was aborted) in 87% of the total aborts. Please note that the aborter/victim matrix can be highly asymmetric; this heterogeneous behavior is one of the key reasons why we argue for profiling on an AB granularity.

The shared *confTVars* (the row "difference" on Table 3) gives way to Table 5. For example, the conflicting *TVars* that caused an execution of AB4 to abort another concurrent AB4 execution, also caused aborts with AB5 in 36% of the total aborts of AB4 (8/(14+8)). Table 5 highlights those "hot" *TVars* that cause aborts in many ABs. Such information might be useful for resolving conflicts: Firstly, it might be a good idea to omit the scheduling of an AB concurrently with itself, if it is known to abort with itself. Furthermore, ABs forming such "conflict duos" (and triples, etc) could be scheduled with more care. Such an approach could diminish aborts that harm performance. The studies in [4,8,23,28], as well as work on contention management [20] act upon similar observations.

With a simple program such as the LL, we can see how all statistics are partitioned in different ABs. In the next section, we look at some other characteristics of ABs using a Haskell STM benchmark.

3.3 The Overhead

The atomic blocks give us a possibility that didn't exist before: to group, compare and view the interactions of transactional executions. However, in this endeavor, our approach needs to introduce new fields to the *TVar* structures such as the ones depicted on Fig. 1, as well as others that are used to accumulate the necessary information for transactional statistics, since GHC recycles the *TRec* structures that it uses for each transaction. New fields for all statistics were added to the *TRec* to be used for keeping related information on the number of aborted/committed transactions, the readset/writesets, number of transactional reads/writes and Validation/CP/WP runtimes in cycles.

These additions do add some overhead, but it is the most straight-forward way to achieve our objective. The slowdown introduced by the profiling mechanism depends on the number of ABs and the size of the *TVar* set that is saved for profiling. Although looking into a range of 5 ABs and hundred *confTVars* works faster than 20 ABs with a thousand conflicting *TVars* each, it still causes on average 7% more aborts on the system running the linked list application. Although the impact on aborts is small, more work is being carried on to reduce this overhead.

4 The Extended Haskell STM Benchmark

The benchmark consists of several programs from the Haskell benchmark in [19], alongside a Hash Table implementation [15] and a Parallel Sudoku Solver [25]. Some of the programs in this benchmark spend as much time aborting transactions as

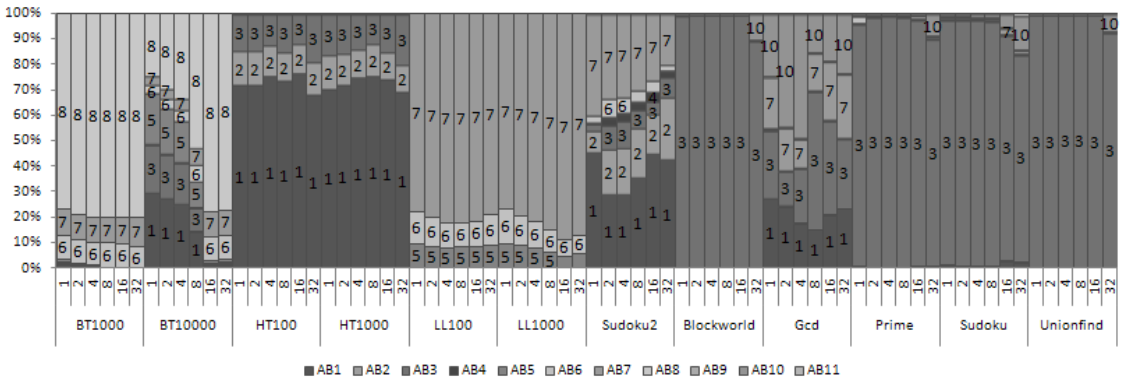


Fig. 5. Overall distribution of transactional time of committed transactions per atomic block on all benchmarks.

committing them. All of the reported results are based on the average of five executions on the Altix 4700 system. Up to 32 cores were used for the clarity of graphs. The outputs were sorted by the atomic block identifier, and then named AB1 through ABx, however they do not represent any initial order of execution. Please note that the atomic blocks with the same numbers are consistent in different runs of the same program but distinct among the benchmarks.

4.1 Transactional occupation per AB on committed transactions

Figure 5 shows the distribution of ABs on the whole benchmark based on the total transactional time spent on committed transactions, which represents the total amount of useful work that each AB had to do for each program to complete. Clearly, we can see both heterogeneous (a variation among the atomic blocks of BT10000, Sudoku2, GCD), and homogeneous behavior (the Hash Table and the Linked List are always performing similarly in all core counts). The heterogeneity exists inside the programs that make use of the CCHR compiler as well: GCD runs differently from the other applications of the CCHR suite.

The predominant AB in BT/LL/HT benchmarks is always the lookup function, however, the initial list creation that creates a tree of 5,000 elements takes up a significant amount of time in the BT 10000 implementation, especially in 1-8 core runs. AB8 of the BT is the lookup function, which expectedly runs eight times as long as AB6 and AB7, (which insert and delete) however, the atomic blocks that do list creation and print also spend a significant amount of time in the WP, independent of the number of cores.

4.2 Abort Analysis per AB

Figure 6 shows the ratio of aborts to committed transactions, only for the atomic blocks that get aborted. Although the BT, HT and the LL are similar programs with each having heavyweight ABs for insert, delete and lookup functions, they have different abort rates. The HT and the LL are heavy aborters especially with high core counts. However, the BT, being a much more concurrent application aborts very little, especially with larger trees.

Most of the ABs demonstrated in CCHR applications that abort also abort more in larger core counts with varying rates per program. However, AB5 (pictured separately) aborts fourfold to commits. In reality, AB5 is a small atomic block with a readset that is equal to the number of forked threads, and a writeset of zero, that only executes as a transaction once, but causes up to four aborts running on 32 cores, or commits on the first try on lower core counts. Sudoku2 also has two aborting ABs

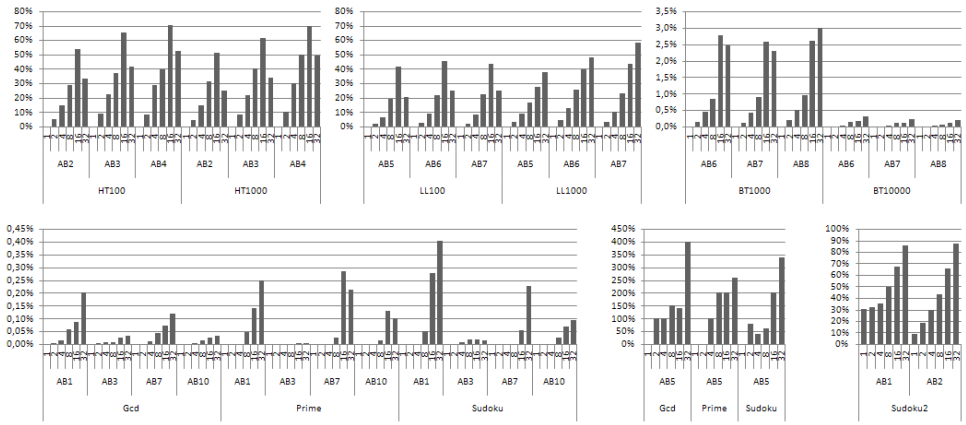


Fig. 6. Percentage of aborts per commits, on aborting ABs of all programs.

that abort quite frequently with more available concurrency, where the rest of the atomic blocks always succeed in committing.

Pushing concurrency on a not-so-parallel application, serialized commits (coarse-grained locking) and the serial, stop-all-the-world garbage collection all hurt the scalability of the program [19]. In other cases, the cache performance and other variables can easily affect the overall system performance.

4.3 Per-AB commit phase and validation

Figure 7 (last page) shows how each AB spends time during executions that end up in commits, in work phase and in commit phase (validation and writeback). Although they all total a 100%, the graphs had to be pruned for clarity, and only selected executions are depicted. The Sudoku2 application contains quite heterogeneous ABs in terms of validation and overall commit phase behavior. Almost all ABs scale poorly: although some writeback phases at first seem to scale better with more cores, the time spent on validation is quite overwhelming.

The LL spends less time in CP using 1000 elements, since the WP is now larger, operating on a bigger list. Although for the CP more work has to be done as well, the

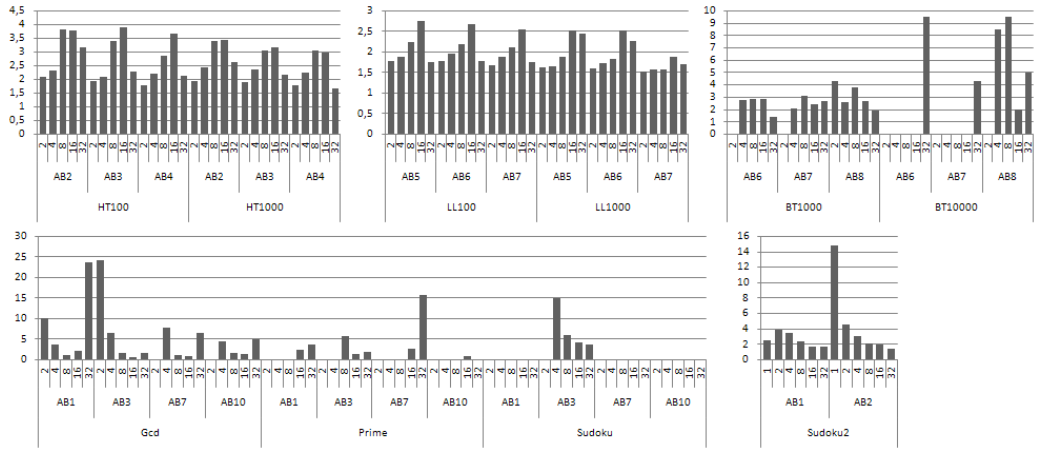


Fig. 8. The ratio of the per-transaction average runtime of aborted transactions to per transaction average runtime of committed transactions. It can be seen that transactions that end up aborting in the end take up a lot more runtime that those that manage to commit.

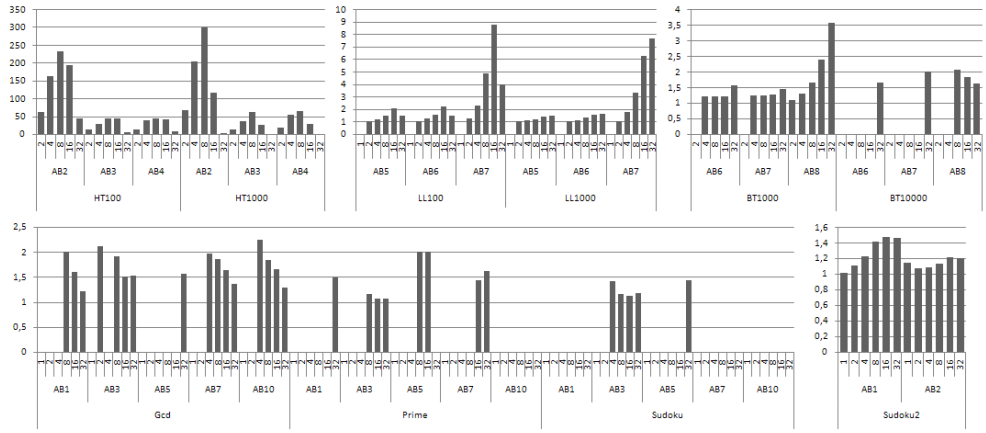


Fig. 9. Average number of aborts per each conflicting TVar.

insert/delete/lookup operations themselves take up more execution time. The same is also true for AB7 which is the lookup atomic block, where the CP seems to be less, while it is the WP that is getting huge. On the HT, the lookup function is AB2, where the CP again dominates an important part of the transactional execution.

The last subfigure in Figure 7 belongs to the programs that utilize the CCHR compiler [16], where same ABs can function differently solving different programs. Regardless of knowing the functionality of ABs inside such a compiler or a library, one can profile the behavior and attempt to optimize it accordingly. For example, AB5 has a CP that takes less percentage of time with more cores, which might be desirable, but a WP that always increases. AB9 does the opposite and AB4 seems to have an almost fixed distribution among all cores. To identify TM performance bottlenecks, besides knowing the per-AB runtimes, it might be useful to see on which phase each AB is the busiest inside the transactions.

4.4 Per-transaction Runtime Averages of Aborted and Committed Transactions

As mentioned on Section 2.B, a transaction in Haskell aborts as soon as an inconsistent *TVar* is encountered. Normally one might be tempted to think that aborted transactions would spend less time than committed transactions: The same amount of time in the work phase, and less time in validation (since abort's validation fails), meanwhile writeback is not performed for aborted transactions. However, Figure 8 shows that this is not true at all, and since commits are serialized, transactions wait on the coarse-grained lock. Those that manage to commit could also wait on locks, but it can be seen here that aborted transactions take a longer time waiting and those transactions that wait too long tend to abort in the end with a higher probability.

4.5 Aborts per Conflicting TVars

Aborts per *confTVars* (Fig. 9) is a "congestion metric" that shows how many aborts each conflicting *TVar* causes on average inside a specific AB. This is due to the frequency of repetitive aborts, because conflicted *TVars* are shared with concurrently scheduled instances of the some AB, or with other ABs. The larger the ratio of these two values (notice that by definition it has to be larger than 1), the more congested the AB is. For example, for a highly non-concurrent *singleInt* benchmark [19], this ratio would be huge, with one conflicting *TVar* but a very high number of aborts due to that *TVar*.

The Hash Table has a small and a very congested critical region. The Linked List is also similar, especially since the nodes close to the beginning of the list are very frequently accessed. The BT is somewhat more concurrently accessible than the LL.

Sudoku2 has two ABs out of eight that abort, but on average the lowest aborts per conflicting TVar ratio in the benchmark. Although its abort ratio (Fig. 6) is one of the highest, it can be concluded that its critical region is also large and it contains many *TVars* that can cause conflicts. This metric is important since such conflicting *TVars* can introduce many aborts, causing performance degradation and scalability problems in the system, and it might be useful to treat them accordingly, as done in [23] and [28].

5 Conclusions

Per-atomic block profiling can be a useful tool for many reasons. For simple micro benchmarks, it helps to see under the hood. For others, when it is not possible or doesn't make sense to know the behavior of each AB, this approach helps to recognize the transactional execution better and to identify certain AB behaviors such as identifying read-only and write-only, or heavyweight ABs, and isolating AB-related problems such as self-conflicting ABs or groups of ABs that abort each other and to construct conflict matrices. Clustering aborting TVars inside atomic blocks also provides an opportunity to identify conflicting shared data sets and to reason about the aborts that take place. In the future it might also be interesting to identify ABs that include nesting or IO in order to try to make appropriate runtime decisions.
We argue for and show through profiling TM applications, why a finer-granularity AB-based approach has to replace lumped averages. Our approach is general enough to be used with different atomic block implementations, eager conflict detection/resolution, or with other STMs.

Acknowledgements: This work is supported by the cooperation agreement between the Barcelona Supercomputing Center - National Supercomputer Facility and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625, by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC) and by the European Commission FP7 project VELOX (216852).

References

[1] A. Adl-Tabatabai, C. Kozyrakis and B. Saha, Unlocking concurrency, ACM Queue, vol. 4/10, pp 24-33, NY, USA, 2007.

[2] A. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha and T. Shpeisman, "Compiler and runtime support for efficient software transactional memory" in PLDI 2006, pp. 26-37, Ottawa, Canada.

[3] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, and I. Watson, "Lee-TM: A Non-trivial Benchmark for Transactional Memory", in Proc. ICA3PP 2008, Cyprus, June 2008.

[4] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham and I. Watson, "Steal-on-abort: Dynamic Transaction Reordering to Reduce Conflicts in Transactional Memory", in SHCMP'08, June 2008.

[5] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. "Bulk Disambiguation of Speculative Threads in Multiprocessors", in International Symposium on Computer Architecture (ISCA 2006), June 2006.

[6] J. Chung, H. Chafi, C. Cao Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. The common case transactional behavior of multithreaded programs. In Proc. 12th HPCA. Feb 2006.
[7] H. Daume III. Yet another Haskell tutorial. In www.cs.utah.edu/ hal/docs/daume02yaht.pdf. 2002-2006.
[8] S. Dolev, D. Hendler, and A. Suissa, "CAR-STM: Scheduling-based collision avoidance and resolution for Software Transactional Memory", in PODC '08: Proceedings of the twenty-seventh ACM symposium on Principles of Distributed Computing, pp. 125–134, New York, NY, USA, 2008.
[9] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: A benchmark for software transactional memory. In Proc. of EuroSys2007, 315–324. ACM, Mar 2007.
[10] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In PPoPP'05, Chicago, USA, June 2005.
[11] T. Harris and S. Peyton-Jones. Transactional memory with data invariants. In First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing, 2006.
[12] Haskell official site: http://www.haskell.org.
[13] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In Procs. of the Twentieth Annual ISCA, 1993.
[14] S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp 295–308, Florida, 1996.
[15] E. Kmett, An STM-based hash, http://comonad.com/haskell/thash/.
[16] E. S. L. Lam and M. Sulzmann. A concurrent constraint handling rules implementation in Haskell with software transactional memory. In DAMP'07, Jan 2007.
[17] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In Proceedings of an ACM conference on Language design for reliable software, pp 128-137, 1977.
[18] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing", in proceedings of IISWC 2008, September 2008.
[19] C. Perfumo, N. Sonmez, S. Stipic, O. Unsal, A. Cristal, T. Harris and M. Valero, "The limits of Software Transactional Memory (STM): Dissecting Haskell STM applications on a many-core environment", in Proceedings of the 2008 conference on Computing Frontiers, pp. 67–78, New York, NY, USA, 2008.
[20] W. N. Scherer and M. L. Scott, Advanced contention management for dynamic software transactional memory, in Proceedings of PODC '05, pp. 240-248, Las Vegas, USA, 2005.
[21] L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. "Delaunay triangulation with transactions and barriers", in IISWC 2007, pp. 107-113, 2007.
[22] N. Shavit and D. Touitou. Software transactional memory. In Symposium on Principles of Distributed Computing, pages 204–213, 1995.
[23] N. Sonmez, A. Cristal, T. Harris, O. S. Unsal, M. Valero, "Taking the heat off transactions: dynamic selection of pessimistic concurrency control", to appear in IPDPS 2009, Rome, Italy, May 2009.
[24] N. Sonmez, C. Perfumo, S. Stipic, A. Cristal, O. S. Unsal, and M. Valero, unreadTVar: Extending Haskell software transactional memory for performance. In Trends in Functional Programming Volume 8, Chapter 6, pp. 89-114, NY, USA, Apr 2007.
[25] W. Swierstra, A parallel version of Richard Bird's function pearl solver, http://www.haskell.org/haskellwiki/Sudoku#A_parallel_solver.
[26] M. Tremblay and S. Chaudhry. A third-generation 65nm 16-core 32-thread plus 32-scout-thread CMT SPARC processor. In IEEE International Solid-State Circuits Conference, Feb. 2008.
[27] C. von Praun, R. Bordawekar, and C. Cascaval, "Modeling Optimistic Concurrency using Quantitative Dependence Analysis", Symposium on Principles and Practice of Parallel Programming (PPoPP 2008), pp.185-196, UT, USA, February 2008.
[28] M. M. Waliullah and P. Stenstrom, "Intermediate Checkpointing with Conflicting Access Prediction in Transactional Memory Systems", In Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS), Miami, Florida USA, Apr 2008.

[29] L. Yen, J. Bobba, M. M. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "LogTM-SE: Decoupling hardware transactional memory from caches".In Proc. HPCA, pp. 261-272, Feb 2007.

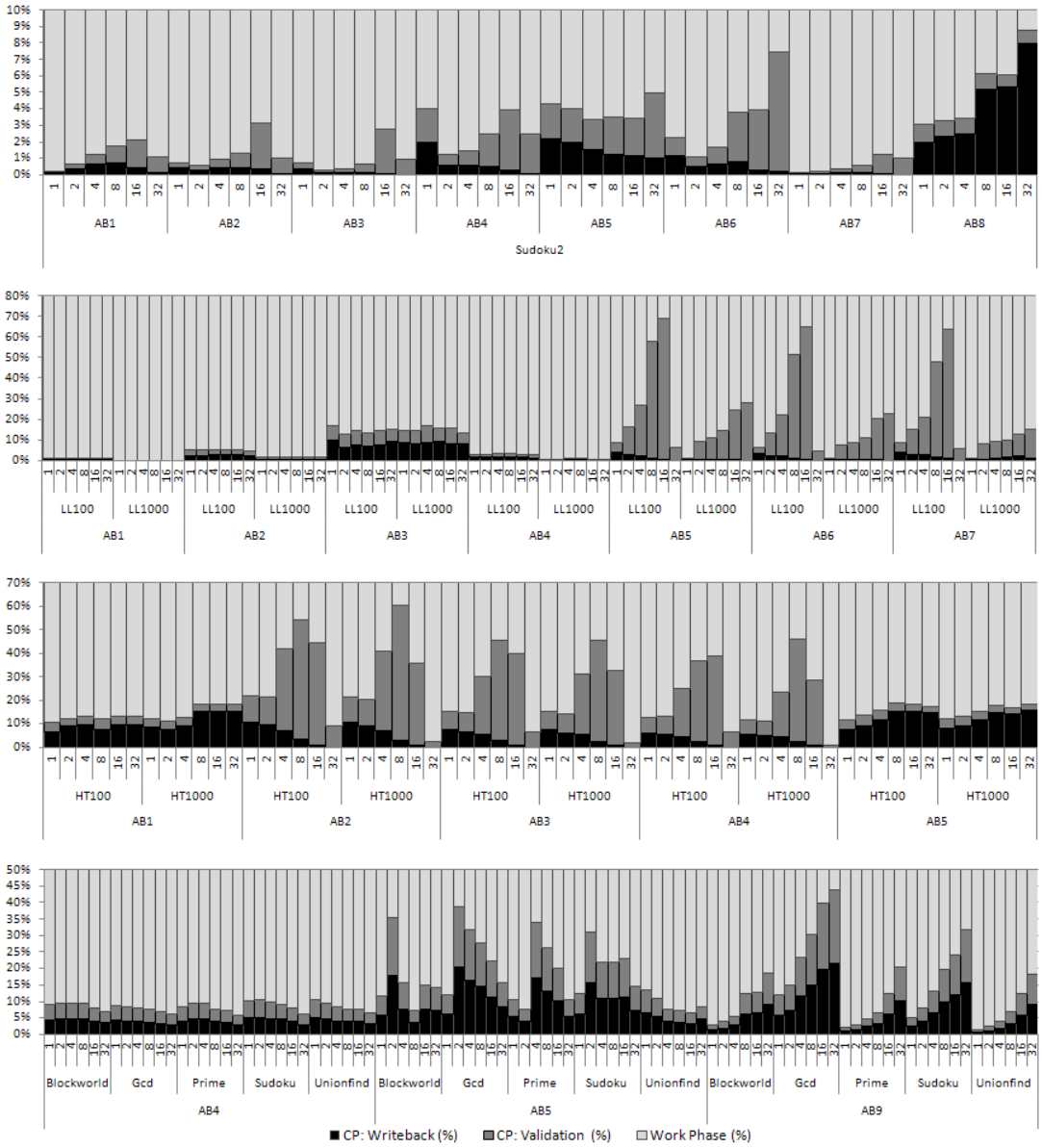


Fig. 7. Percentage of validation, writeback and the work phase on committed transactions.