A Communication Characterisation of Splash-2 and Parsec

Nick Barrow-Williams, Christian Fensch and Simon Moore
Computer Laboratory
University of Cambridge
npb28@cl.cam.ac.uk, cf309@cl.cam.ac.uk, swm11@cl.cam.ac.uk

Abstract

Recent benchmark suite releases such as Parsec specifically utilise the tightly coupled cores available in chip-multiprocessors to allow the use of newer, high performance, models of parallelisation. However, these techniques introduce additional irregularity and complexity to data sharing and are entirely dependent on efficient communication performance between processors. This paper thoroughly examines the crucial communication and sharing behaviour of these future applications.

The infrastructure used allows both accurate and comprehensive program analysis, employing a full Linux OS running on a simulated 32-core x86 machine. Experiments use full program runs, with communication classified at both core and thread granularities. Migratory, read-only and producer-consumer sharing patterns are observed and their behaviour characterised. The temporal and spatial characteristics of communication are presented for the full collection of Splash-2 and Parsec benchmarks. Our results aim to support the design of future communication systems for CMPs, encompassing coherence protocols, network-on-chip and thread mapping.

1 INTRODUCTION

To produce the most efficient hardware designs, it is vital to have a detailed understanding of application behaviour, as without this knowledge it is extremely difficult to correctly partition resources between communication and computation. This paper focuses on the communication behaviour of multi-threaded benchmarks – an increasingly important factor in overall program performance.

The communication patterns exhibited by a multi-threaded benchmark are determined by a number of factors. The programming, machine and parallelisation models as well as the application algorithm all play a significant role in defining the nature of thread to thread communication. By using an idealised architecture for many experiments, this work aims to abstract away many of these factors, exposing the true sharing present in the algorithms used.

We analyse a large number of applications running on a shared-memory, chip-multiprocessor (CMP) architecture. The applications are selected from the Splash-2 [12] and Parsec [3] benchmark suites. Of particular note is that the target machine model has evolved from a multi-node system (Splash-2) to a chip-multiprocessor (Parsec). As described by Bienia et al. [2], core-to-core communication is considerably faster on a CMP than in a multi-node system and this shift in machine model allows programs to be written using new parallelisation models previously untenable on a multi-node machine. New parallelisation models imply different communication patterns and this paper aims to thoroughly characterise this shift.

The characterisation falls into three sections. In section 4.2, the spatial and temporal characteristics of thread to thread communication are examined. Data is presented showing how much sharing occurs between threads and at what times the transactions occur. This information could be used for thread mapping and interconnect topology design. Section 2.2 analyses the sharing patterns that are present in each benchmark. Three patterns are described: read-only, producer-consumer, and migratory. The importance of these patterns influences both caching policy and coherency protocol design.

2 BACKGROUND

2.1 Benchmarks

For this study, we select two benchmark suites: Splash-2 [12], released in 1995 and Parsec [3], first released in 2008 and updated in early 2009.

Splash-2 is a mature benchmark suite containing a variety of high performance computing (HPC) and graphics applications. The dominant parallel platforms at the time of the suite's creation were multi-node systems, with CPUs often being housed in separate machines relying on board-to-board communication between nodes. The extremely high latency of these links requires the algorithms to minimise thread to thread communication wherever possible. The suite has remained extremely popular. However, for evaluating newer designs, recent publications [3] have suggested

SPLASH-2	
barnes	n-body Simulation
cholesky	Matrix Factorisation
fft	Complex 1-D FFT
fmm	Fast Multipole n-body
lu	Matrix Triangulation
ocean	Ocean Current Simulation
radiosity	Graphics
radix	Integer Sort
raytrace	3D Rendering
volrend	3D Rendering
water-nsquared	Molecular Dynamics
water-spatial	Molecular Dynamics
PARSEC	
blackscholes	Financial Analysis
bodytrack	Computer Vision
canneal	Engineering
dedup	Enterprise Storage
facesim	Animation
ferret	Similarity Search
fluidanimate	Animation
freqmine	Data Mining
streamcluster	Data Mining
swaptions	Financial Analysis
vips	Media Processing
x264	Media Processing

Table 1: The Splash-2 and Parsec workloads

that many of the algorithms are now out-dated, largely due to the increasing dominance of the CMP as a parallel computing platform and the new communication opportunities present in such systems.

Parsec is a more recent benchmark suite, offering a wider variety of applications rather than focusing on HPC. The large advances in silicon technology now allow the integration of many processing cores on a single die, each with access to sizeable shared caches, drastically reducing the latency cost of inter-core communication. This important change has been taken into account during the design of the algorithms used in Parsec. Furthermore, the suite includes a number of benchmarks that spawn more threads than the number of cores available, leaving the operating system to schedule work in an effective manner. No such benchmarks are present in the Splash-2 suite. The full selection of benchmarks from both suites is shown in table 1.

2.2 Sharing Patterns

Sharing in multithreaded benchmarks can be classified in a number of ways. Here we describe the terms used throughout this paper. First, a word is described as shared if it is written to or read from by more than one processor[1] during the execution of a benchmark. This separates the memory into shared and private regions, defining where communication could have taken place.

However, not all reads and writes to such a shared region are actually used to communicate data. The application might use a refinement strategy, rewriting results until they meet a certain quality before they are communicated to other processors. As such, we say that only the writes that produce the final value are *communicating writes*. A similar classification is possible for read operations. A read is a *communicating read*, if it reads a value that has been written by a different processor for the first time. Subsequent reads by the same processor do not communicate any new information and are an artefact of register pressure or instruction encoding (the latter is most certainly the case for x86 binaries). Figure 1a shows communicating and non-communicating memory access for one memory location. Communicating accesses are shown in black, while non-communicating accesses are shown in gray. In the following discussion, we will only consider communicating accesses.

The way in which shared words are accessed can be used to further categorise the memory locations. The number and ordering of reads and writes can indicate a certain sharing pattern. In this paper, we examine three such patterns: read-only, migratory, producer-consumer [1, 11].

Read-only A word is declared read-only if during the entire execution of a program it is written to either zero or one times, and is subsequently read by at least one processor that is not the writer. In addition, no read access is allowed before the single write access. An example of a read-only relationship is shown in figure 1b. Read-only sharing is most commonly observed when an input file is read into a program and the content is then consumed by several of the threads in the parallel phase of execution. In this pattern, each data word may be read several times by a variety of different processors but is never over-written once first read. Therefore any intermediate values used in further computation must be stored elsewhere. A consequence of such a pattern is that these words do not strictly require any coherence support.

Migratory Migratory sharing occurs when a shared data structure is accessed and modified inside an atomic region repeatedly by different processors during program execution. This pattern is identified by a read to a newly produced data value followed by a write, without an interrupting read or write from another processor.

Migratory sharing is common in shared memory benchmarks and predictability is also high, with regions exhibiting migratory behaviour often doing so for the rest of a benchmark's execution. Migratory sharing is of interest as it also behaves sub-optimally on MESI protocols [10]. Examining figure 1c, we see the first read from P1 will return with *shared* permissions, only to immediately require an up-

[1] Since we use for most benchmarks as many threads as there are processors in the system, the terms thread and processor can be interchanged in all sections, except for one paragraph in section 4.2.

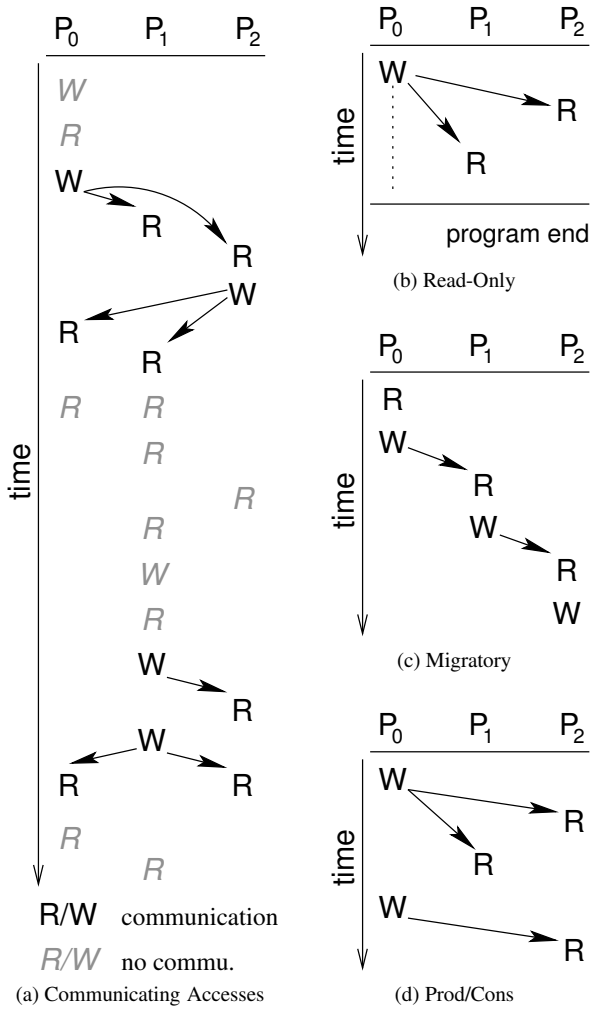


Figure 1: Communicating accesses and memory access ordering for sharing patterns on a single memory location.

grade for the write to *modified* state, requiring additional coherency traffic for each migration.

Producer-Consumer Producer-consumer sharing in a shared memory program can be defined in a number of ways. All require a persistent relationship between write and read sets for a given memory location. In the strictest definition, a location is only flagged as exhibiting producer-consumer behaviour if each write comes from a single processor, and is always followed, before the next write, by a load from the consuming processor. Our experiments showed that for our benchmark selection, this pattern of accesses was extremely unlikely to occur multiple times without interruption. Additionally, we found that the producer does not remain constant and is quite likely to change. For this reason the definition was relaxed to allow any number of writers. In this scheme, the strength of the relationship is reported as the probability that for each communicating write to a memory address, a communicating read will follow from a given processor. In this paper, words are reported as exhibiting producer/consumer sharing if there is a greater than 50% probability that a specific reader will consume each write to a given location.

In addition to analysing the producer/consumer pattern directly, we also focus on the stability of the read set of shared memory locations. The read set for a memory location is considered stable when for each processor it is known whether that processor will consume or not consume a produced value. The read set is considered unstable, if it is not known if a processor consumes or does not consume a produced value.

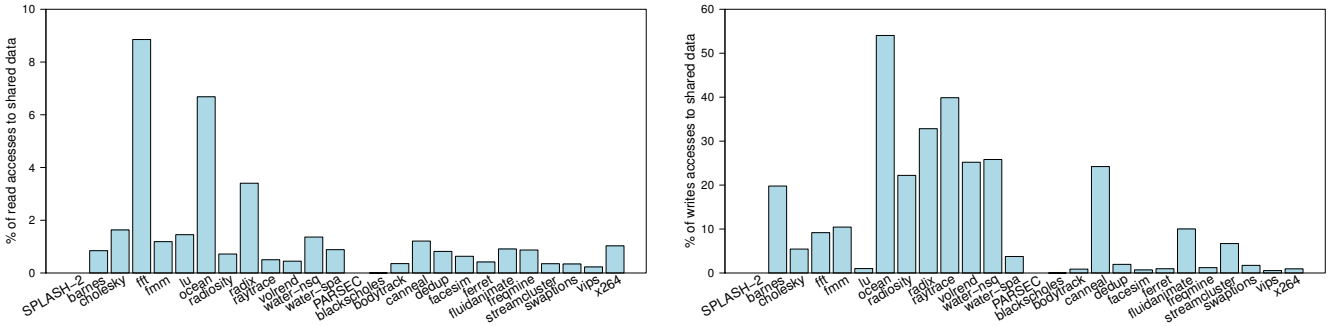
Figure 1d shows a memory location exhibiting producer-consumer characteristics. Processor P_0 acts as the producer, while P_1 and P_2 act as consumers. In this example, P_2 is a stable consumer (since it consumes every produced value) and P_1 is an unstable consumer (since it consumes 50% of the produced values). Thus, the stability of read set for this memory location is 50%, i.e. 1 in 2 processors.

This sharing pattern is important as it behaves sub-optimally under a widely used MESI cache coherency protocol [8]. The producing processor's permissions will oscillate between *modified* and *shared*, with the consumer switching from *shared* to *invalid*. This generates a large volume of messages both to and from the directory, which may be physically remote to the processing node.

3 EVALUATION SETUP

Simulated Architecture We use Virtutech's Simics simulator [7] to generate cycle accurate memory accesses traces for a 32 processor x86 system running Linux 2.6.15. Using a full Linux operating system allows us to run a wide variety of unmodified benchmarks with full library support. Each processor has a single in-order pipeline, similar to the cores found in Intel's Larrabee CMP [9]. However, to maintain high simulation speed, no further pipeline details are modelled, leaving each core with a fixed throughput of 1 instruction per cycle. To provide timing information in the traces we attached a cache hierarchy of private L1s and a large shared L2. The private caches are kept coherent using a MESI protocol across a zero cycle, infinite bandwidth crossbar. The details are summarised in table 2.

Trace Generation We generate memory access traces using a modified version of the tracer module provided by Virtutech. We have extended the module to determine which thread is currently executed by each core, providing additional information for benchmarks that spawn a large number of threads. To retrieve this data we read the `tr` register and follow the pointer it contains to the appropriate entry in the thread table of the Linux kernel, tagging each memory access with both the thread number and processor on which the operating system executed it. We optimise the output to



(a) Read Accesses

(b) Write Accesses

Figure 2: Fraction of read and write accesses to shared memory locations that communicate data. A read is considered communicating when it reads a value that has been produced by another processor and has not been read before. A write is considered communicating when it produces a value that is read by a different processor.

Core count	32
ISA	x86
Pipeline	In-order, fixed CPI = 1
L1 Cache	32kB, 64B lines, 4-way assoc., Hit latency 1 cycles
L2 Cache	8MB, 64B lines, 32-way assoc., Hit latency 10 cycles
Main memory	Latency 400 cycles
Interconnect	0 cycle, infinite bandwidth crossbar
OS	Linux 2.6.15

Table 2: Simulated system parameters

reduce the size of the traces generated, although the larger files even when compressed are still over 100GB.

In order to prevent thread migration we tied generated threads to a specific processor. This was done for all Splash-2 programs and the Parsec programs of *blackscholes*, *canneal*, *fluidanimate streamcluster* and *swaptions*. We were not able to do this for other programs in the Parsec benchmark suite, since the program either creates more threads than CPUs are available or threads are created in a non-trivial way.

For Splash-2, we use the recommended input size for all benchmarks. For Parsec, we use the *simmedium* input size, to keep simulation time and resulting traces manageable while still accurately reflecting benchmark behaviour.

Communication Characterisation When identifying thread to thread communication we track the consumers of a value written to memory at word-level granularity. This analysis is done purely on an address level, and does not take into consideration any write-back or coherency effects. On the consuming side, an infinite cache is assumed. Thus, a value that has been consumed once will always be directly accessible by the consuming CPU. Additionally, we do not record consumptions by the producing CPU. Furthermore, we ignore all communication that resulted from values produced during the initialisation phase showing the communication during the parallel phase of the execution only. Carrying out the analysis in such a way provides a lower bound on the amount of communication that must take place, regardless of interconnect or coherency protocol design. Results from such experiments provide a useful specification for the development of on-chip communication systems.

4 EXPERIMENTAL RESULTS

In this section we present the results of our communication analysis. In section 4.1, we establish some general properties about memory accesses to shared memory locations. Section 4.2 investigates communication patterns, analysing which processor communicates with which other processor. Section 4.3 classifies the observed communication into three sharing patterns: read-only, migratory and producer-consumer. Finally, section 4.4 looks into how stable (and as such predictable) the read set of communicating write instructions is.

4.1 Communicating Accesses

Not all accesses to shared memory locations are actually used to communicate: values might be reread from memory due to capacity in the register file or values are being refined for several iterations before being communicated. As we focus on communication, we first identify the number of accesses to shared address space that actually communicate data as discussed in section 2.2. Figure 2 shows the percentage of reads and writes to shared memory locations that communicate data. One thing to notice is that on average only 1.5% of reads actually communicate data. However, this might be partially an artefact of using an x86 machine for these experiments. Due to the instruction encoding and lack of programmer visible registers on x86, it is quite common that almost every instruction reads from memory. As for communicating writes, one notices that Parsec benchmarks have significantly less communicating writes (4.2%

on average) than Splash-2 applications (20.8% on average), suggesting a refinement of values before they are communicated. We will use the number of communicating accesses as the basis for many normalisations in the following sections.

Figure 3 shows the communication to computation ratio. We find that expressing this ratio using communicating read (figure 3a) or writes (figure 3b), does change the absolute figures but not the general trend. An exception to this is *water-spatial*, which looks like an average communication bound benchmark based on the number of instructions per communicating read, but computation bound based on the number of instructions per communicating write.

4.2 Communication Patterns

Figures 4 and 5 show the observed spatial communication patterns for the evaluated applications. Figure 6 shows this behaviour over time, for four representative benchmarks. All plots are normalised to the maximum CPU to CPU communication relationship observed in that program. No columns or rows in the graphs have been swapped. The CPUs or threads appear in the order as numbered in the operating system.

Spatial Behaviour The first thing to notice is the relative diversity of communication patterns for the Splash-2 programs. *Cholesky*, *lu*, *radix*, *ocean* and *water-spatial* have very distinct communication patterns that are not observed elsewhere in the benchmark selection. Secondly, we notice that many programs exhibit very strong communication between neighbouring CPUs. For example, *Barnes* and *fmm* show increased neighbour communication with *blackscholes* and *streamcluster* also showing similar patterns. *Fluidanimate* exhibits a comparable trend, though each CPU does not communicate its nearest neighbours but rather with its 4th neighbour to either side. Both benchmark suites include a program that shows strong all to all communication, *fft* for Splash-2 and *canneal* for Parsec. Parsec contains many applications that show less uniform, but still random traffic (*dedup*, *swaptions*, *vips* and *x264*). We only find two programs in Splash-2 that shows this kind of behaviour (*radiosity* and *raytrace*). A further category of programs show no recognizable pattern, but show strong communication between a few CPUs with almost no communication between the rest (*water-nsquared*, *bodytrack*, *facesim*, *ferret* and *freqmine*).

From a communication pattern perspective, Splash-2 shows more variation than Parsec. In addition, the structured patterns in Splash-2 often involve a high radix communication with one CPU communicating with 10 to 16 other CPUs. Parsec on the other hand is dominated by either low radix or unstructured communications. All of these patterns present interesting challenges for communication system design.

Thread-Level Analysis Unlike Splash-2, some Parsec benchmarks generate threads dynamically during the parallel execution phase. As such, certain communication patterns between threads can be hidden due to thread creation, mapping and migration. To eliminate this interference and expose true sharing patterns, we tracked the communication based on the thread ID for programs that showed unstructured communication patterns. Figure 7 shows the results for *dedup*, *ferret* and *x264*. In all cases distinct communication patterns become visible that were previously hidden.

Dedup generates 3 classes of threads that exhibit different kinds of behaviour: the first group (threads 33 to 64) produces data, which is consumed by the second group (threads 1 to 32). However, only 8 threads in this group produce any significant amount of data that is consumed. The threads in the second group collaborate in groups of 4 threads to produce data for the third group (65 to 96). The threads in the last group show random communication among themselves.

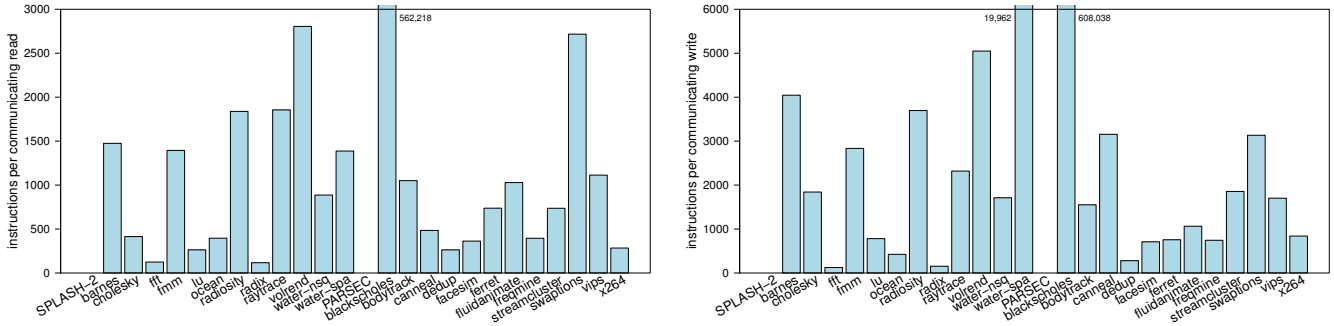
Ferret spawns the largest number of threads of all Parsec programs (133 threads). The first 32 threads show very strong neighbour communication, while the remaining threads show very limited communication. This suggests that the mapping of the first threads is of much greater importance than the higher indexed threads.

X264's thread based communication pattern shows that half of the spawned threads exhibit little communication. For the other half, a strong communication with 5 other threads can be identified, likely due to the sharing of frames in the compression algorithm.

The strength and regularity of the sharing exposed by performing thread based analysis has implications for thread mapping in Parsec benchmarks. A more intelligent spawning and mapping may well lead to clearer locality in the processor level results.

Temporal Behaviour The results presented so far focus on the spatial behaviour of the benchmarks. However, the temporal behaviour of the communication is also of upmost importance when considering interconnect design.

In figure 6, we show the temporal communication behaviour of one CPU for four programs. With the exception of *canneal*, it possible to identify patterns in the communication behaviour over time. As such, even if a CPU communicates with every other CPU during the program execution, it is not necessarily the case that every CPU receives all communications. For example, CPU 2 in *barnes* only communicates with all other CPUs during very short phases in the program's execution. For the first quarter, there is some light traffic towards CPU 16 to 31. After a short period in a synchronisation phase (which results in communication to all other CPUs), the focus of communication shifts to CPU 0 to 15. During this period, we also observe a period of heavy communication with CPU 1, for approximately 10% of the total execution time. A similar behaviour can be seen in *bodytrack*: for the majority of the parallel phase there is little



(a) Instructions per communicating read access.

(b) Instructions per communicating write access.

Figure 3: Instructions per communicating read and write accesses.

communication between CPUs. During 2 phases (which add up to approximately 30% of the execution time), we notice an all to all communication. This communication is mostly light, but at times shows some heavy bursty communication targeted at CPU 4, 6, 8, 13, 22 and 23 during the first phase and targeted at CPU 0, 3, 6, 7, 10 – 14, 22 and 24 during the second phase. Another interesting communication pattern can be observed in *streamcluster*. While there is some random, light communication to all other CPUs, it shows that for about 15% of the execution time there is heavy communication present towards CPU 18 and 20. For the remaining time, light traffic is observed. While it can also be seen in figure 5 that most traffic from CPU 19 is directed towards CPU 18 and 20, it is surprising that it is only present for a relatively short period of time.

4.3 Sharing Patterns

Figures 8a, 9a and 10a show the proportion of the shared memory space exhibiting each of the three described sharing patterns (see section 2.2). Please note that while a location can only be read-only shared, it can under certain conditions participate in both a producer/consumer and migratory behaviour. We show how many different CPUs access that memory location. For producer/consumer and read-only sharing, this indicates the number of different CPUs that consume the value. For migratory sharing, it shows the number of different CPUs that participate in the migratory pattern over the entire parallel phase.

First, we find that for 9 out of 24 programs our sharing characterisation scheme covers almost all shared memory locations. For another 7 programs, we can characterise 50% or more of shared memory locations. The remaining programs do not exhibit any recognised sharing pattern. This is best described as a "multiple producer/multiple consumer". Finally, we observe that, with the exception of *water-spatial*, *water-nsquared* and *canneal*, few memory locations are involved in a communication involving more than 8 CPUs.

Read-Only Sharing Figure 8a shows the percentage of the shared memory space that is used for read-only sharing. It is further divided by the number of different CPUs that read a word from this space. We see that *raytrace*, *volrend*, *canneal*, *streamcluster* and *x264* use almost all of the shared address space in a read-only manner and to a lesser extent *radix* and *ferret*. We also notice that while there is some data that is being read by 16 or more CPUs, most sharing is performed between up to 7 CPUs.

While figure 8a shows a spatial analysis of sharing pattern, figure 8b presents a quantative analysis of read accesses to shared data. We find that most applications that use their shared address space in a predominantly read-only manner, also direct most shared reads to these regions. The exceptions are *ferret* and *x264*, which use 61% and 71% of its shared memory space in a read-only way, but only 7% and 19% of its read accesses read this data. Several benchmarks (*fmm*, *ocean* and *fluidanimate*), which do not use a significant portion of their address space for read-only data, direct 40% to 50% of their shared reads to these regions.

Migratory Sharing Figure 9a shows the percentage of shared memory locations that participate in a migratory pattern. It is further divided by the number of different CPUs that participate in this pattern. We find that only 5 Splash-2 benchmarks (*barnes*, *fmm*, *lu* and *water-nsquared*) use a noticeable fraction of their shared memory space for migratory data. For Parsec, we find that all benchmarks, apart from *canneal*, *streamcluster* and *x264*, use a significant amount of the shared memory space for migratory communication. Analysing how many CPUs use a particular memory location for a migratory sharing pattern, we see that most migratory locations are only being used by 2 CPUs. A few locations are used by up to 7 CPUs. The only exception to this is *water-nsquared* and *swaptions*. In *water-nsquared*, almost all migratory locations are shared between all processors. In *swaptions*, about two third of the migratory address space is used by more than 7 CPUs.

Figure 9b shows the percentage of communicating writes that participate in a migratory sharing pattern. It can be seen that all applications exhibit migratory behaviour to some extent. We can identify three kinds of clusters: applications that have less than 10% of migratory patterns, applications

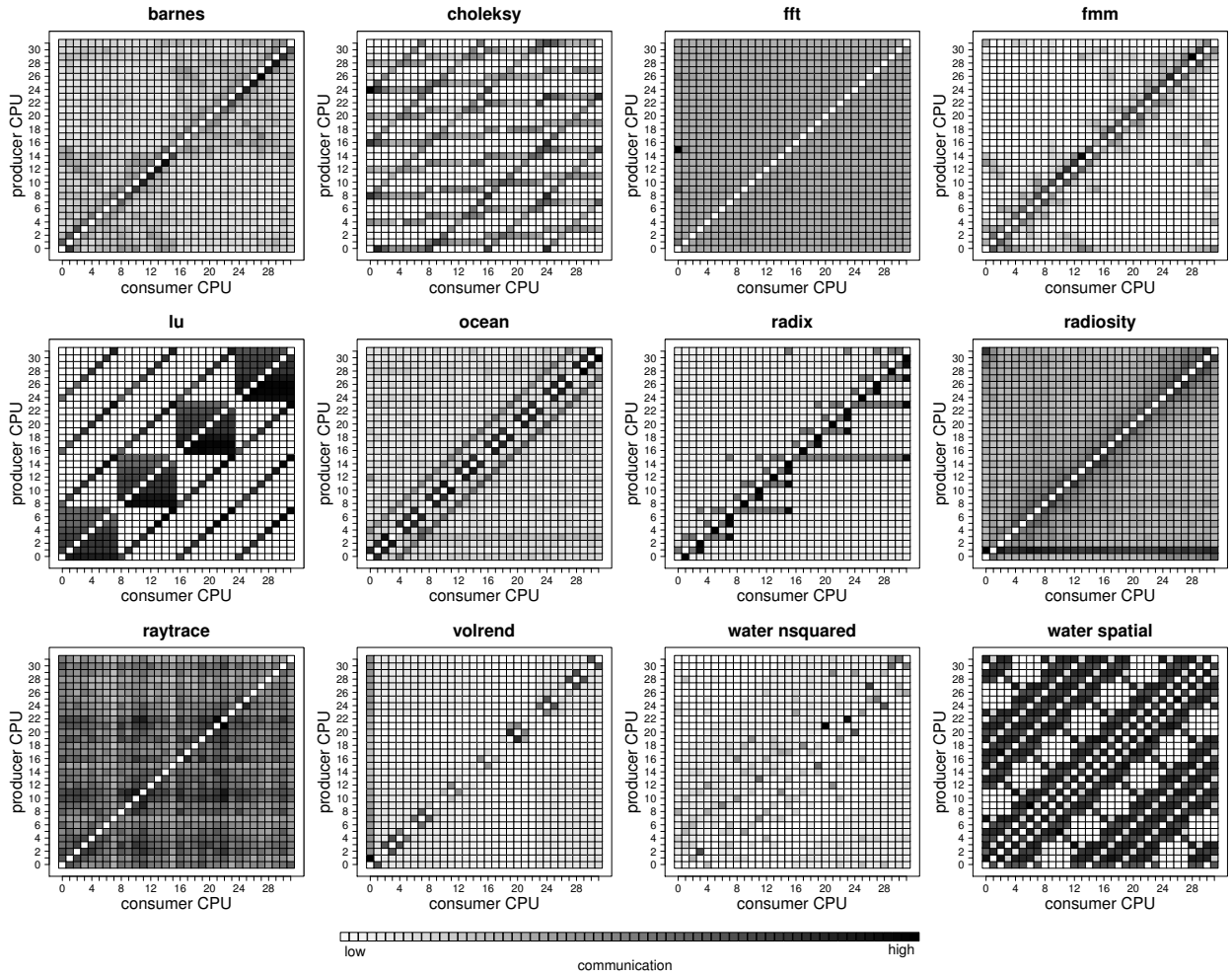


Figure 4: Normalised communication between different CPUs during the entire parallel phase of the program for the Splash-2 benchmark suite.

with around 30% migratory patterns and applications that show 50% or more migratory patterns. We observe a full range of results, suggesting optimisation of migratory patterns is important, but will never yield universal improvements.

The Parsec benchmark suite exhibits more migratory sharing pattern than Splash-2. Migratory patterns are much easier to support in a CMP environment than in a multi-node system and as such, it is no surprise to find them more heavily used in Parsec, which has been developed with a CMP platform in mind.

Producer-Consumer Sharing Figure 10a shows the percentage of shared memory locations that participate in a stable producer-consumer relationship (as defined in section 2.2). It is further divided by the number of different CPUs that consume the word that has been produced. The first striking observation is the almost complete absence of stable producer/consumer memory locations in Parsec (with the exception of *fluidanimate*). Second, in Splash-2 we find only 5 applications that use a significant amount of their shared memory space for producer-consumer pattern: *barnes*, *fmm*, *ocean*, *water-nsquared* and *water-spatial*. Third, there is a large variance in the number of CPUs that are involved in producer consumer patterns. For *water-nsquared* and *water-spatial*, we find that all CPUs are involved in the pattern. For the other four applications, we find that most produced data is consumed by a single CPU. As such, using broadcast techniques in an on-chip interconnect or coherence protocol is likely to benefit *water-nsquared* and *water-spatial*, but it will be of limited use for almost all other applications.

Finally, notice that *water-nsquared* and *water-spatial* are the only programs that exhibit a significant amount of sharing of data between more than 15 CPUs. The only program in the Parsec benchmark suite, which shows such a high degree of sharing is *canneal* and only for read-only data.

Figure 10b shows the percentage of communicating writes that access a location with a stable producer-consumer relationship. The main observation is that ap-

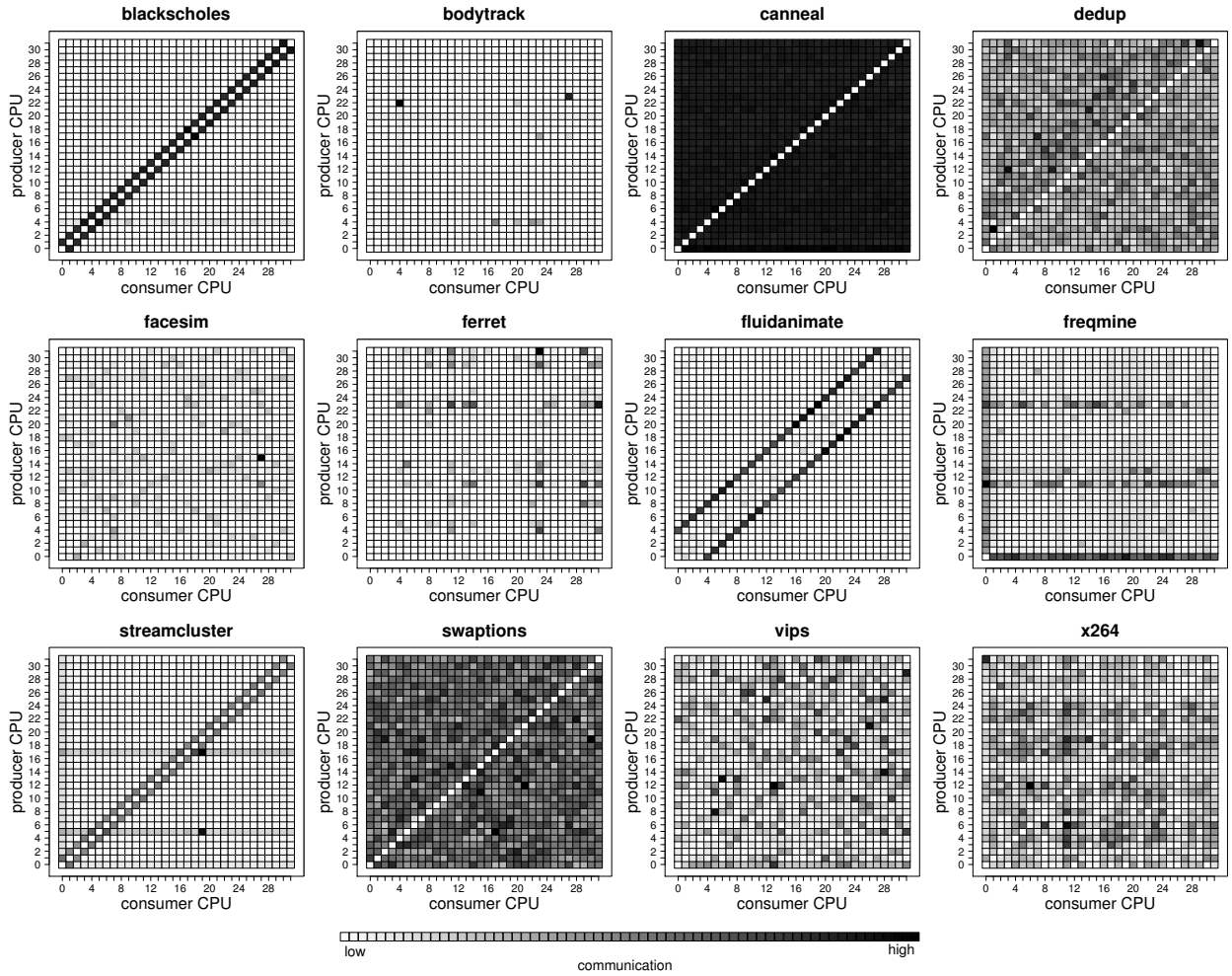


Figure 5: Normalised communication between different CPUs during the entire parallel phase of the program for the Parsec benchmark suite.

plications that use a significant fraction of the shared address space for producer-consumer communication, also use a signification fraction of communicating writes in this way. The two exceptions to this observation are *volrend* and *water-nsquared*. *Volrend* only uses around 10% of the shared address space for producer-consumer communication, but more than 55% of its communicating writes. *Water-nsquared* uses around 35% of its shared address space for producer consumer communication, but only 7% of its communicating writes.

4.4 Read-Set Stability

The read set is considered stable when it is known that a produced value will be consumed or not be consumed by a given processor. As such, a processor that always consumes a produced value contributes to a stable read set. Similarly, a processor that never consumes a produced value also contributed to a stable read set. A processor that consumes only half of the produced values contributes to an unstable read set. As such, a migratory sharing pattern will be classified as a mostly stable read set. A produced value is consumed by only one processor (and not consumed by all other processors). As such, a migratory location is considered highly predictable. In order to classify a location as stable, it is necessary that at least two communicating write accesses are performed on that location.

Figures 11a and 11b show the results for the stability of the read set. We find that both in the spatial and quantative analysis a significant number of locations and write accesses have a very stable read set (80% to 100%). In many cases these results roughly overlap with the migratory sharing results from figure 9. Minor differences in these results (for example more locations are classified migratory than there are locations with a read set stability) are due to slight differences in measuring these locations. For example, the last write in a migratory pattern does not have to be a communicating write. As such, if a migratory pattern consists of only 2 writes then it is possible that it will not be considered for the read set stability analysis.

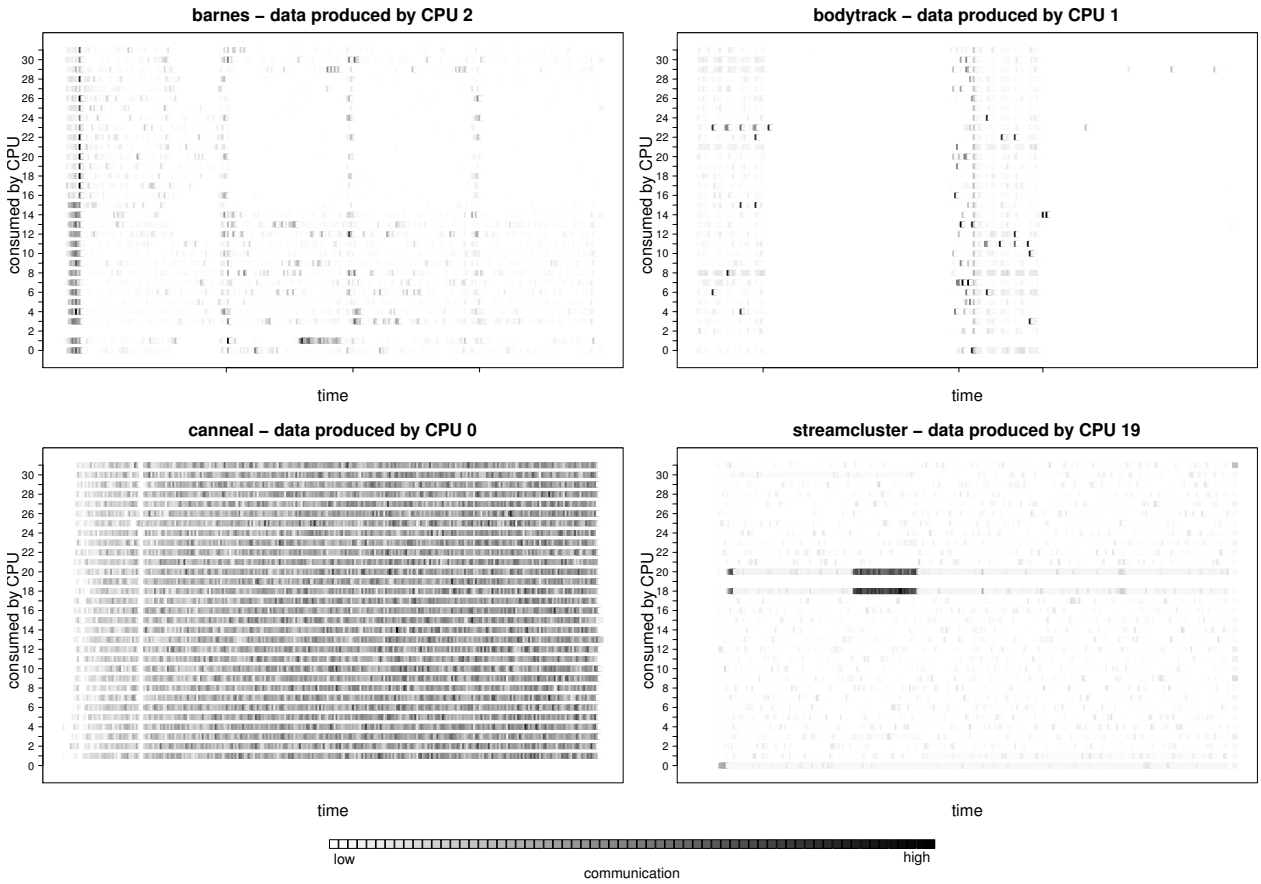


Figure 6: Normalised communication changes over time for a selection of CPUs and applications.

Exceptions to this overlap are *ocean*, *radix*, *volrend*, *water-spatial*, *bodytrack*, *dedup* and *ferret*. These benchmarks show a highly stable read set, which is not the result of a migratory sharing pattern. In general, we find that the stability in the read set is due to knowing that processors are not going to read a produced value. This behaviour is already exploited by current cache coherency protocols, which assume a value is not being consumed and hence do not do anything. In order to establish the stability of the read set due to knowing that the value will be consumed, we increased the threshold for detecting a stable producer-consumer relation to 70% and 90%. Figure 12 shows the results of the quantative analysis. We find that *barnes*, *canneal*, *fluidanimate*, *fmm*, *ocean*, *radix*, *volrend* and *water-spatial* have a significant fraction of read set stability due to knowing which CPU will consume a value.

Since a location can exhibit a stable read set with just two communicating writes, we further investigated the number of communicating writes for locations that were included in this characteristic. Table 11c shows these results. We see only in *barnes*, *fmm* and *volrend* memory locations with less than 5 communicating write accesses on average. All benchmarks show a significant number of communicating writes per memory location, suggesting that it is worthwhile to exploit read set stability in communication optimisation.

5 RELATED WORK

The works by Woo et al. [12] and by Bienia et al. [3], which present the Splash-2 and Parsec suites, contain a large amount of information on the benchmarks used here. These characterisations focus on synchronisation overhead, size of working sets, false and true sharing, and scalability. Unlike our study, they do not evaluate temporal and spatial communication patterns, nor try to classify shared data access patterns.

Bienia et al. [2] also compare the Splash-2 and Parsec benchmark suites. However, while they examine the sharing behaviour for both suites, this data is evaluated with a particular system in mind (i.e. data sharing is only observed if the data is shared through caches). Our study focuses on sharing patterns at a logical level. As such, we present insight on what kind of communication is present in the applications, regardless of execution platform.

Chodnekar et al. [5] provide a communication characterisation methodology for parallel applications. Their work focuses on temporal and spatial traffic characterisation for a multi-node CC-NUMA system. However, their evaluation is tied to a particular physical implementation of a CC-NUMA machine. For example, the communication analysis assumes a communication network with a mesh topology. Our study

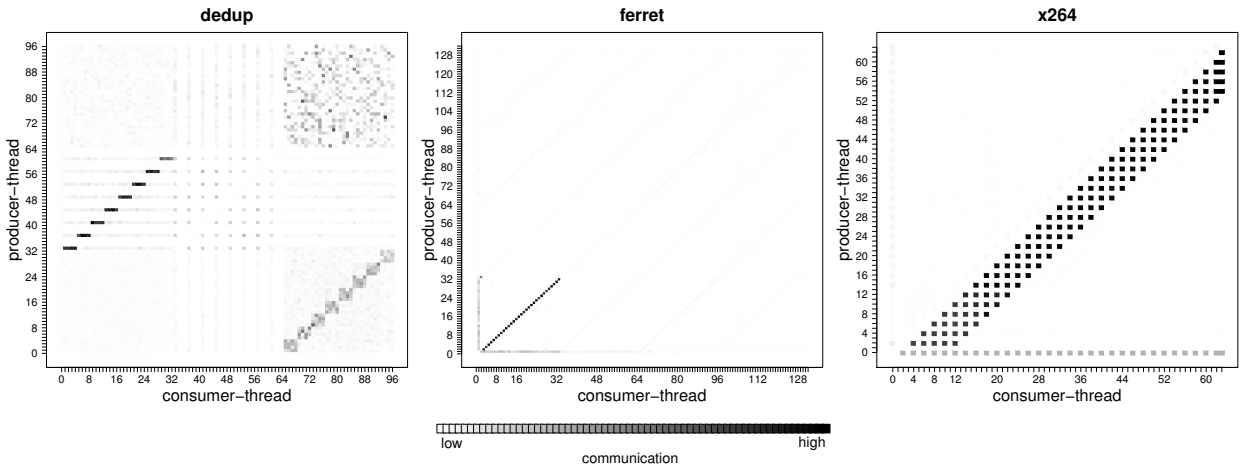
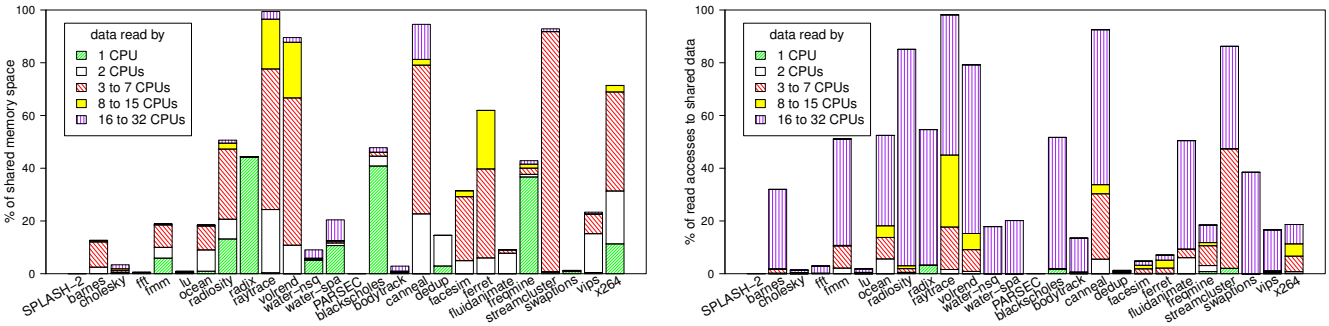


Figure 7: Normalised communication between different threads during the entire parallel phase of the benchmark.



(a) Spatial

(b) Quantative

Figure 8: Analysis of the read-only sharing pattern. The spatial analysis shows the percentage of the shared address space that is used according to the read-only sharing pattern. The quantative analysis shows the percentage of reads to shared address space that access a location that had been classified as read-only. For both analyses, we determine by how many processors the line is read and classify read accesses accordingly (Read-only locations with only one reading processor, are written by a different processor).

looks at communication with no particular topology in mind, providing generic results for use in future work.

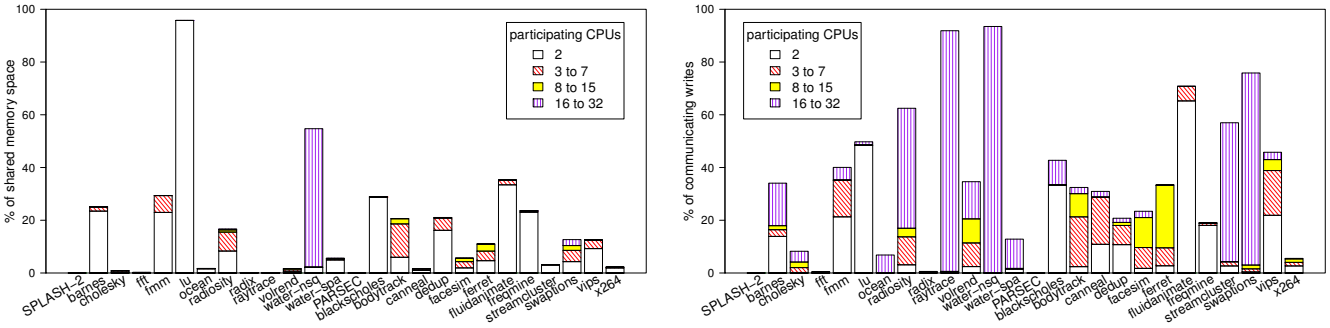
Hossain et al. [6] present an augmented cache coherency protocol for CMPs that tries to take advantage of producer/consumer and migratory sharing. It uses heuristics and additional status bits in each cache line to identify these patterns dynamically with local information available at each L1. All traffic observed in the system is then characterised using these heuristics. In our work, we use global knowledge about the application and do not miss patterns masked due to conflict misses. Additionally, their evaluation only includes a selection of programs from the Splash-1/2 benchmark suites.

There are many other publications that augment the cache coherence protocol to take advantage of specific sharing pattern such as [4, 10]. Many such works target multi-node systems. Similar to Hossain's work, they use a heuristic and only present results that show the improvement in performance of their scheme. To our knowledge, none of these studies investigates how much traffic falls into a particular category. It is beyond the scope of this paper to list them all.

6 CONCLUSIONS

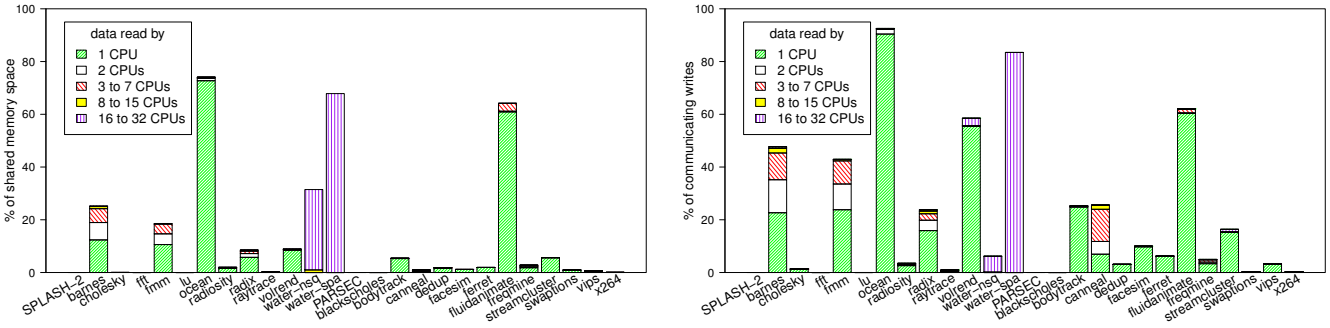
In this paper, we have presented a detailed analysis of the communication exhibited by the Splash-2 and Parsec benchmark suites. We have shown that using cycle accurate simulations at the thread level allows the characterisation of communication relationships otherwise masked by OS mapping and scheduling policies. The infrastructure provides sufficient speed to analyse the full duration of each benchmark, giving an insight into the temporal behaviour of the communication patterns. These results have an impact on a number of areas of processor design.

Thread Mapping By analyzing communication at a thread level, we are able to see that existing thread mapping policies do not optimise for physical locality of shared data. On current platforms, this is unlikely to cause problems but in an architecture with less uniform communication costs, this may be of increasing concern. However, many benchmarks exhibit good locality purely based on the thread ID



(a) Spatial (b) Quantative

Figure 9: Analysis of the migratory sharing pattern. The spatial analysis shows the percentage of the shared address space that is used according to the migratory sharing pattern. The quantative analysis shows the percentage of communicating writes that access a location that had been classified as migratory. For both analyses, we determine by how many processors participate in the migratory pattern and classify the write access accordingly.



(a) Spatial (b) Quantative

Figure 10: Analysis of the producer-consumer sharing pattern. The spatial analysis shows the percentage of the shared address space that is used according to the producer-consumer sharing pattern. The quantative analysis shows the percentage of communicating writes that access a location that had been classified as producer consumer. For both analyses, we determine by how many processors consume the produced values and classify the write access accordingly.

or CPU number. Further experiments could characterise the performance benefit of using this information in future CMP platforms.

Coherence Protocols By classifying shared memory locations and accesses into read-only, migratory and producer/consumer, we provide researchers with the benchmarks that will benefit most from communication aware optimisations. Protocol modifications targeting migratory sharing should see good improvements on the emerging workloads in the Parsec suite. Producer/consumer sharing however is harder to find, and schemes aiming to optimise for this behaviour may need to do so at a finer temporal granularity than used here. Finally, the large amount of read-only sharing present in many of the benchmarks reminds researchers to maintain good support for this basic pattern.

On-Chip Interconnect Many of the spatial and temporal results have an impact on interconnect design for CMPs. It is evident that there is no common case communication behaviour and also that the traffic is rarely constant. This places high demands on any interconnect architecture. The

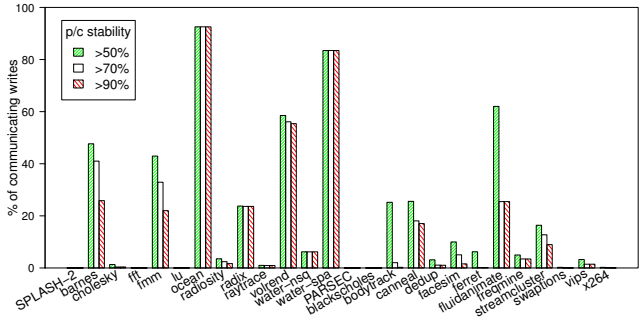
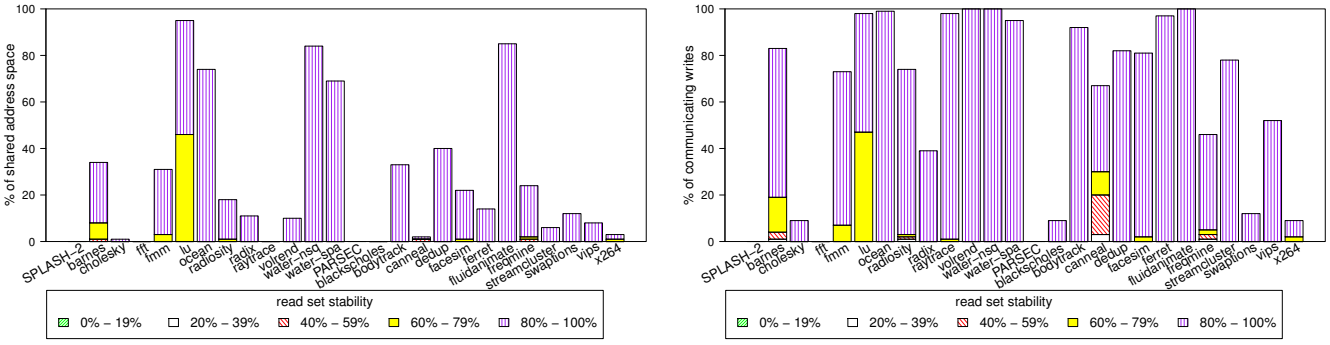


Figure 12: Quantative analysis of the stability of producer-consumer relation ship.

locality of the spatial communication has implications for the network topology choices a designer makes, however the temporal properties must also be considered. Clustering compute nodes to aggregate traffic may lead to congestion in the higher traffic phases of program execution. Finally, a number of the characteristics presented here could be combined to provide synthetic traffic patterns for router design and evaluation.



(a) Spatial

(b) Quantative

Program	Min	Max	Avg
barnes	2	5,519	2
cholesky	2	1,128	289
fft	2	446	20
fmm	2	2,141	4
lu	2	4,282	115
ocean	2	53,230	12
radiosity	2	229,744	61
radix	2	574	12

Program	Min	Max	Avg
raytrace	2	130,899	28,052
volrend	2	2,335	2
water-nsq	2	954	18
water-spa	2	955	10
blackscholes	32	64	32
bodytrack	2	10,101	251
canneal	2	4,095	152
dedup	2	4,451	451

Program	Min	Max	Avg
facesim	2	27,834	22
ferret	2	857	30
fluidani	5	2,558	11
freqmine	2	1,633	38
streamcl	2	826,793	4,132
swaptions	2	12,914	1,684
vips	2	4,289	83
x264	2	1,085	17

(c) Communicating writes per word

Figure 11: Stability analysis of the read set of produced values. In order to characterise the stability of a location, it is necessary that at least two communicating writes are performed. The spatial analysis shows the percentage of shared address space with two or more communicating writes. The quantative analysis shows the percentage of communicating writes that access a location with two or more communicating writes. For both analyses, we determine the stability of the read set for a location and classify the write accesses accordingly. The table shows the minimum, maximum and average number of communicating writes per line, which are included in the graphs.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers, Arnab Banerjee and Robert Mullins for their valuable feedback. This research was supported in part by the EPSRC grant EP/F018649/1.

REFERENCES

[1] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proceedings of ISCA 17*, pages 125–134, June 1990.

[2] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors. In *Proceedings of IISWC 2008*, pages 47–56, Sept. 2008.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of PACT 2008*, pages 72–81, Oct. 2008.

[4] L. Cheng, J. B. Carter, and D. Dai. An Adaptive Cache Coherence Protocol Optimized for Producer-Consumer Sharing. In *Proceedings of HPCA 13*, pages 328–339, Feb. 2007.

[5] S. Chodnekar, V. Srinivasan, A. S. Vaidya, A. Sivasubramaniam, and C. R. Das. Towards a Communication Characterization Methodology for Parallel Applications. In *Proceedings of HPCA 3*, pages 310–319, Feb. 1997.

[6] H. Hossain, S. Dwarkadas, and M. C. Huang. Improving Support for Locality and Fine-Grain Sharing in Chip Multiprocessors. In *Proceedings of PACT 17*, pages 155–165, Oct. 2008.

[7] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[8] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of ISCA 11*, pages 348–354, June 1984.

[9] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. *ACM Trans. Graph.*, 27(3):1–15, Aug. 2008.

[10] P. Stenström, M. Brorsson, and L. Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of ISCA 20*, pages 109–118, May 1993.

[11] W.-D. Weber and A. Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of ASPLOS 3*, pages 243–256, Apr. 1989.

[12] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of ISCA 22*, pages 24–36, June 1995.