THE UNIVERSITY
of EDINBURGH

On the Simulation and Design of Manycore CMPs

Christopher Callum Thompson

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2014

Abstract

The progression of Moore's Law has resulted in both embedded and performance computing systems which use an ever increasing number of processing cores integrated in a single chip. Commercial systems are now available which provide hundreds of cores, and academics have proposed architectures for up to 1024 cores. Embedded multicores are increasingly popular as it is easier to guarantee hard-realtime constraints using individual cores dedicated for tasks, than to use traditional time-multiplexed processing. However, finding the optimal hardware configuration to meet these requirements at minimum cost requires extensive trial and error approaches to investigate the design space.

This thesis tackles the problems encountered in the design of these large scale multicore systems by first addressing the problem of fast, detailed micro-architectural simulation. Initially addressing embedded systems, this work exploits the lack of hardware cache-coherence support in many deeply embedded systems to increase the available parallelism in the simulation. Then, through partitioning the NoC and using packet counting and cycle skipping reduces the amount of computation required to accurately model the NoC interconnect. In combination, this enables simulation speeds significantly higher than the state of the art, while maintaining less error, when compared to real hardware, than any similar simulator. Simulation speeds reach up to 370MIPS (Million (target) Instructions Per Second), or 110MHz, which is better than typical FPGA prototypes, and approaching final ASIC production speeds. This is achieved while maintaining an error of only 2.1%, significantly lower than other similar simulators.

The thesis continues by scaling the simulator past large embedded systems up to 64-1024 core processors, adding support for coherent architectures using the same packet counting techniques along with low overhead context switching to enable the simulation of such large systems with stricter synchronisation requirements. The new interconnect model was partitioned to enable parallel simulation to further improve simulation speeds in a manner which did not sacrifice any accuracy.

These innovations were leveraged to investigate significant novel energy saving optimisations to the coherency protocol, processor ISA, and processor micro-architecture. By introducing a new instruction, with the name wait-on-address, the energy spent during spin-wait style synchronisation events can be significantly reduced. This functions by putting the core into a low-power idle state while the cache line of the indicated

address is monitored for coherency action. Upon an update or invalidation (or traditional timer or external interrupts) the core will resume execution, but the active energy of running the core pipeline and repeatedly accessing the data and instruction caches is effectively reduced to static idle power. The thesis also shows that existing combined software-hardware schemes to track data regions which do not require coherency can adequately address the directory-associativity problem, and introduces a new coherency sharer encoding which reduces the energy consumed by sharer invalidations when sharers are grouped closely together, such as would be the case with a system running many tasks with a small degree of parallelism in each.

The research concludes by using the extremely fast simulation speeds developed to produce a large set of training data, collecting various runtime and energy statistics for a wide range of embedded applications on a huge diverse range of potential MPSoC designs. This data was used to train a series of machine learning based models which were then evaluated on their capacity to predict performance characteristics of unseen workload combinations across the explored MPSoC design space, using only two sample simulations, with promising results from some of the machine learning techniques. The models were then used to produce a ranking of predicted performance across the design space, and on average Random Forest was able to predict the best design within 89% of the runtime performance of the actual best tested design, and better than 93% of the alternative design space. When predicting for a weighted metric of energy, delay and area, Random Forest on average produced results within 93% of the optimum result.

In summary this thesis improves upon the state of the art for cycle accurate multicore simulation, introduces novel energy saving changes the the ISA and microarchitecture of future multicore processors, and demonstrates the viability of machine learning techniques to significantly accelerate the design space exploration required to bring a new manycore design to market.

Lay Summary

Many tasks performed by computers can be broken down into multiple smaller tasks can be processed independently. These can then be worked on in parallel, by multiple processing cores within the computer. So long as the work can be split into enough independent jobs it is possible to keep increasing the speed at which the whole task is computed by adding more processing cores to the system. The degree to which a task can be split into independent tasks is called the parallelism of the task. It is also possible to make each processing core more powerful, and in turn increase the overall processing rate. However each increase in performance of an individual core comes at ever increasing non-linear cost, in both space and energy, eventually reaching a limit to what is physically possible. Because of this, the best design of computer for a given application depends on the parallelism inherent to the application, and the energy and space constraints in which the application must execute. An extremely parallel application would suite a large number of processors, while one which cannot be parallelised needs the most powerful single-core system available (within the constraints).

This thesis addresses the design of such parallel computing systems from several inter-related directions. Firstly through innovations in the simulations of such systems it enables much greater simulation performance at extremely high levels of detail and accuracy, enabling greater exploration of the performance of an application across a number of design parameters. Secondly it leverages this simulation infrastructure to evaluate the effectiveness of some existing techniques in a new context to address scalability issues with the communication challenges that rise from splitting a problem between hundreds of processing cores. These focus mostly on reducing the exchange of information through tracking the sharing state of memory, and through increasing the energy efficiency of waiting periods during synchronisation. The thesis then presents and evaluates novel contributions to the problem of "cache coherency" at the extreme scale of hundreds to a thousand processing cores in a single processor. Cache coherency is an aspect of the inter-processor communication inherent to the most common way of parallelising programs, where all processing cores can access the data from any other processor working on the same task. The thesis finally evaluates a set of algorithms from the field of Artificial Intelligence called "machine learning" algorithms, to predict the performance of new applications across a large design space of potential embedded computer designs. It finds that existing algorithms can make very good predictions for identifying the best overall design choice,targeting a number of design goals, such as performance, or a balance between energy and design size.

Acknowledgements

Foremost I would like to thank my PhD supervisor, Prof. Nigel Topham, whose guidance, advice, and support has not only helped to inspire my research and articulate my ideas into writing, but has also prevented this PhD from descending into an endless number of only tangentially related, although interesting, side projects. I would also like to thank Dr Björn Franke for his continued support and advice; always interested to talk and quick to volunteer for proof-reading, which always came back with invaluable advice. Additional thanks go to my second supervisor Dr Vijay Nagarajan for the valuable discussions and encouragement.

The colleagues and friends from the CArD and PASTA groups to which I would like to extend my gratitude are too numerous to list, but I would like to give special thanks to those closer friendships I have made and those who's regular discussions were both fruitful and enjoyable. Stephen Kyle, Harry Wagstaff, Matthew Beilby, Volker Seeker, Tom Spink, Tobias Edler von Kock, Oscar Almer, Miles Goulde, Marco Eliver, Igor Böhm, and from outwith the department Jennifer Mankin, have all contributed greatly to this journey.

I would also like to say thank you to Dr Peter Boyle, in the School of Physics and Astronomy, who was the first person to encourage me towards undertaking a PhD.

Thank you to my parents for their endless support in my academic career, and for always pushing me to perform to my fullest, and to set my goals high.

Finally I would like to say thank you to my partner in life Rachael Barton, whose proof reading, tireless support, encouragement, and patience has seen me through the highs and lows of my PhD, and without whom both this thesis, and my life, would most certainly be incomplete.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Christopher Callum Thompson*)

Table of Contents

Chapter 1

Introduction

In recent years there have been several factors pushing chip manufacturers to pursue chip-multiprocessors (CMP) with an ever growing number of cores. Moore's Law continues [1], giving us ever more transistors each year, but silicon processes have hit a frequency wall, preventing performance increases by simply raising the operating frequency [2]. There is also very little instruction level parallelism (ILP) left to extract via techniques such as superscalar architectures, known as the ILP wall [3; 4; 2]. These factors are driving chip designers to exploit thread level parallelism (TLP) using both simultaneous multi-threading (SMT) and CMP techniques [5; 6; 7]. These can all be clearly seen from 2005 in the typical Moore's Law graph, Figure 1.1.

On top of this, the power consumed by transistors is no longer scaling down with their size, so while more transistors can still be packed onto a chip, doing so now increases the power consumed, where traditionally the power saved by shrinking transistor sizes would offset the additional transistors used. This means that while extra transistors are available, they cannot be used in power hungry features like those used to aggressively exploit ILP. Simply adding more of the same cores to exploit thread-level parallelism (TLP), without sacrificing per-core performance, will still result in a large increase in power. As a result of the frequency, ILP and power walls, chip designs for applications with significant TLP are looking towards using smaller, energy efficient cores to extract performance. Smaller cores are especially attractive as single core processors scale approximately quadratically in power and area, for linear performance, while an ideal multicore scales linearly in power and area for linear performance gains. This simplified model of course neglects the achievements of features such as clock and power gating to keep single core power consumption low, and the interconnect and synchronisation overheads of multicores. It also ignores the issue of

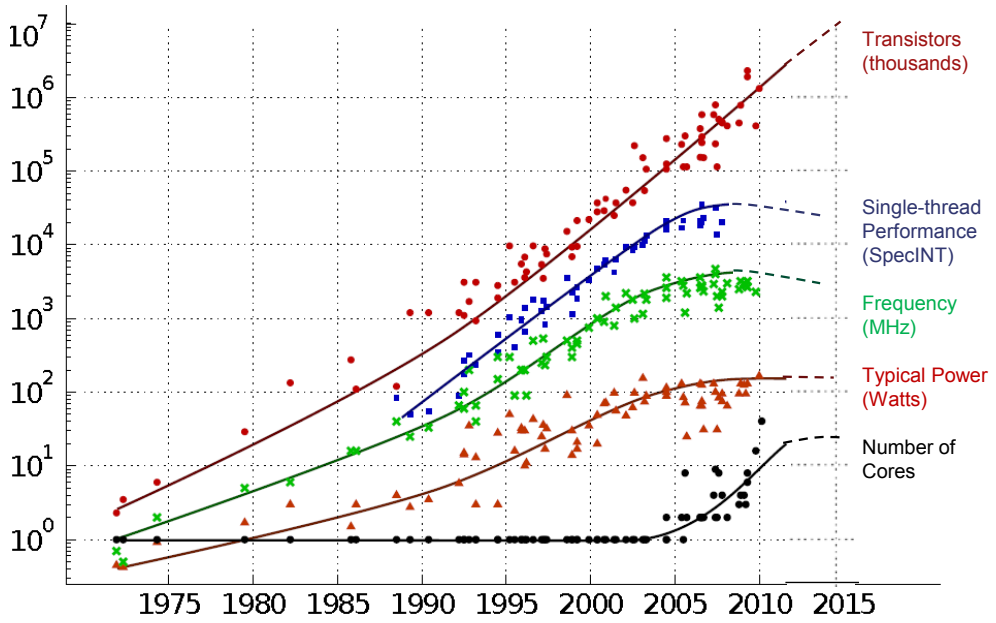


Figure 1.1: Microprocessor trends from 1970–2010, showing the exponential growth in transistors with Moore's Law. The abrupt end of frequency and single core performance scaling, and growth in power consumption, is clear from 2005 where CMP architectures become popular. Source of illustration: C. Moore (AMD) [8]

.

Amdahl's Law of parallel speedup [9], assuming that applications are perfectly parallel. As such, the ideal balance between per-core performance and number of cores is highly dependant on the application demands and the constraints of the operating environment. In many cases, from extreme scale super computing to deeply embedded biomedical sensors, there is a delicate balance between providing sufficient performance, while being as energy efficient as possible.

Unfortunately, CMP systems bring new problems which can severely limit the scalability of the system: thread synchronisation, interconnect complexity, and cache coherency.

For small scale CMPs like those in most consumer systems today, which contain between 2 and 8 cores, a simple shared bus or crossbar can connect all of the cores to the shared cache or memory system. This configuration allows for simple snooping coherence protocols, which perform well for such small systems, but are not efficient for more than 8 cores [10]. Manycore processors however (>16 core CMPs), require more scalable interconnects and coherence mechanisms, with the most common interconnects used being the 2D mesh in a tiled architecture, and bi-directional ring bus.

Just because these designs are currently in use, this does not mean they are the optimum choice. The best interconnect may be application dependent, and could borrow elements from multiple interconnect topologies, to provide a balance between the advantages and disadvantages of each. Architectures without a shared bus require more complicated coherence protocols, because memory traffic is not visible to all cores. Less homogeneous multi-core devices such as a multi-processor system-on-chip (MPSoC), often use less regular interconnects, using network-on-chip (NoC) architectures like Butterfly networks, and may not even support direct core-to-core communication. The best cache coherency protocol is still an open question, and depends very much on the application, the size and design of the underlying architecture, and the memory consistency and programming models to be provided.

When considering developing a new MPSoC or manycore processor the number of possible design combinations presents a massive design space, such as those in Figure 1.2, with subtle trade-offs and design interactions. To reason about what design is best for a given target application requires detailed simulation of many different possible solutions.

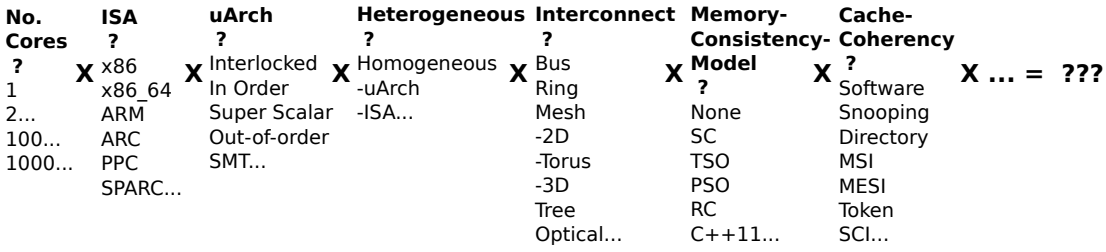


Figure 1.2: Design space state explosion for CMP to manycore processors.

Cycle accurate multicore full system simulation typically is either very slow, requires specialised hardware, or is non-deterministic and lacking in accuracy (i.e. not truly cycle accurate). This means that an architect cannot quickly iterate through different designs to find the optimal solution, and must either settle for a potentially sub-optimal solution, or spend many hours waiting on simulation results.

The effort of simulating manycore systems has also left a large number of design issues unsolved, with many issues relating to large scale interconnects and scalable cache coherency mechanisms remaining. Developing a simulator capable of exploring these problems enables the micro-architectural experimentation and analysis required to begin addressing these problems effectively.

1.1 Research Goals

The objective of this thesis was to address three fundamental and interrelated challenges facing the design of future manycore processors: fast and accurate simulation, scalability, and design-space-exploration. One of the most significant impediments to manycore research is the challenge of providing accurate, flexible, and yet high performance simulation models. Without such models, the design of future manycore systems will rapidly become an intractable problem. An initial goal of this research was to advance the state-of-the-art in simulation technologies to significantly increase the simulation speed of truly cycle-accurate models. Using these fast simulation models, the second goal was to investigate issues that constrain the scalability of future manycore processor designs. Key issues in scaling manycore processors include: the interconnect between cores, the coherency protocols used within the interconnect, and the energy efficiency of inter-processor synchronization primitives. One of the aims of this work was to demonstrate that fast models would enable new innovations in these areas to be explored, and to validate the specific novel solutions proposed in this thesis. The third goal of this thesis was to show that fast simulation models can be used to populate machine-learning models, and in turn those models can be used to rapidly search the design space for good manycore architectures.

1.2 Contributions

The primary contributions of this thesis are new techniques which enable much higher simulation rates than have previously been possible for large scale embedded MPSoCs, while maintaining cycle accurate correctness. Using using commodity hardware the results are competitive even against FPGA simulation and prototyping options. Extending this highly accurate simulation platform to explore cache-coherent CMPs, while still improving upon the state-of-the art, this thesis then presents novel techniques for reducing energy consumption of CMPs and demonstrates that scalability concerns of centralised directory architectures can be addressed by extending existing software-hardware coherency techniques. Finally this thesis demonstrates that by applying the high speed, accurate, simulation technologies developed in this thesis to existing machine learning techniques, close to optimal multicore and manycore architectures can be predicted quickly and efficiently for new applications.

Specifically the contributions of this thesis are: Improved simulation performance

of multicore, cycle-accurate, simulation through more efficient NoC simulation, by using compact cache friendly data structures, and by tracking packets through NoC regions to reduce the simulation work. Simulation performance has also been improved by increasing the available parallelism in MPSoC simulations, by decoupling the core and interconnect simulation, and exploiting cache-incoherency of embedded architectures to increase the timing slack between simulation threads without sacrificing accuracy. The thesis demonstrates that the combination of these techniques allows for software based cycle accurate simulation rates significantly better than the state of the art, competitive with FPGA based techniques. It also shows that many of these techniques can be used to construct an efficient cache-coherent simulator capable of simulating up to 1024 cores in a fully cycle accurate system, with detailed NoC models, with better accuracy, and higher simulation speeds, than the current state of-the-art in multicore software simulation. This thesis proposes a novel manycore architecture which offers different bandwidth/latency trade-offs, enabling an energy efficient multicast coherency mechanism to address scalability problems with traditional coherency protocols. It demonstrates that existing software-hardware techniques for coherency filtering are capable of addressing the directory-associativity problem without an extravagant hardware directory, reducing pressure on the directory or alternative coherence protocol, reducing the required die space, and reducing energy required to process coherency transactions. This work proposes and evaluates a new hint instruction which can eliminate the dynamic energy used in spin-wait synchronisation primitives, while requiring minimal hardware or program modification. It also proposes an optimisation to the existing atomic-exchange instruction found in many ISAs to reduce cache write-back traffic and cache-line "ping-ponging" for contended mutex's. The thesis then proposes and evaluates a new sharer encoding and multicast scheme which can be used to reduce the interconnect traffic and number of unnecessary cores involved in multicast coherency events, addressing the scalability problem of directory space from the sharer encoding space dimension, and addressing communication bandwidth and latency scalability challenges. Finally the thesis demonstrates that existing machine learning techniques can be applied to the design space of large scale MPSoCs, running mixed program workloads, to enable the rapid identification of near-optimal MPSoC configurations for new workloads.

1.3 Publications

The relevant publications in which the author was primary or co-author are:

- First Author: *High Speed Cycle Approximate Simulation for Cache-Incoherent MPSoCs [11] – 2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII)*
  In this work the author was responsible for all major components of the research, including adding cycle accurate multicore interconnect simulation to Arcsim, developing cache incoherent multi-threaded benchmarks (based on an existing embedded benchmark suite and runtime), and developing NoC models from existing Verilog designs. The author was also responsible for ensuring accuracy against RTL by analysing detailed cycle accurate Verilog based simulation and comparing with cycle by cycle traces from the developed simulator, as well as conducting parallel experiments on FPGA prototype platforms along side the simulator based experiments to verify accuracy of performance counters from full benchmark results. This paper forms the basis of Chaper 4.

- Co-author: *Scalable multi-core simulation using parallel dynamic binary translation[12] – 2011 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XI)*

- Co-author: *A Parallel Dynamic Binary Translator for Efficient Multi-Core Simulation[13] – 2013 International Journal of Parallel Programming*
  In the above work the author contributed to benchmark porting efforts and analysis of results – explaining the unusual performance behaviour as simulations are scaled up and an increasing proportion of the benchmark runtime is spent in synchronisation. The author was also responsible for proposing and implementing a direct-memory-access optimisation for the JIT compiled simulation code, which contributed approximately 30% , on average, to the performance achieved.

1.4 Structure of this Thesis

The remainder of this thesis is structured as follows:

- Chapter 2 presents background information on multicore systems, cache coherency, and system simulation of single and multicore targets.
- Chapter 3 discusses the important related work, relating to each of the sections in Chapter 2 and the work undertaken in this thesis.
- Chapter 4 investigates the simulation of embedded multiprocessor systems, presenting new techniques for increasing the simulation speed and accuracy of multicore simulation.
- Chapter 5 presents the extension of the incoherent simulation work to cache coherent manycore architectures.
- Chapter 6 presents the scalable coherent manycore architecture used in Chapters 5 through 7 of this thesis, along with analysing the scalability of the architecture and the potential for existing software based techniques to address the directory associativity problem often tackled with expensive hardware.
- Chapter 7 presents a novel hint instruction to drastically reduce the energy consumption wasted on synchronisation operations along with another energy saving microachitectural optimisation and novel coherence sharer state compression and multicast scheme, which reduces the energy required to send multicast invalidations, using no more storage overhead than the state of the art.
- Chapter 8 investigates the feasibility of extending previous machine learning based design space exploration techniques to larger manycore embedded systems.

Chapter 2

Background

2.1 Parallel Computing Systems

The concept of using multiple processors to increase performance is long-standing within the field. There are a huge variety of parallel computing architectures, with various degrees of coupling between processing elements. The first sections of this chapter look at how programmers can utilise these platforms. Then, because this thesis focuses primarily on integrated single-chip multiprocessors, this thesis looks in more detail at issues which affect shared memory parallelism and on-chip interconnects. The final section of this chapter looks at the challenges of simulating these parallel systems when designing and exploring new architecture and programming options.

2.2 Programming Models

Programming these parallel systems usually takes one of two forms, often depending on the underlying architecture of the system:

1. Message Passing
2. Shared Memory

Ranked in order of increasing coupling, message passing is the most disconnected form of parallel computing. Each processing thread can only communicate with other threads via explicit messages, no thread can inspect or modify another threads' state. In this model, processing elements need only be connected by a form of communication network. This can be a slow network such as TCP/IP, a dedicated on-ship interconnect optimised for the message passing API in use, or an abstraction on top of a

shared memory system, with operating system (OS) provided interprocess communication (IPC). Shared memory systems appear to the running process as if all threads are running in a single, shared, address space. So, if thread A modifies a global variable, all other threads will see this change. The semantics of how soon the new value propagates are dependent on the memory consistency model of the system, discussed in Section 2.3.1.

Both mechanisms have real world implementations scaling from large scale distributed systems, to tightly coupled single chip systems. For example, message passing architectures range from world wide distributed Grid Computing platforms, to exotic dataflow architectures, while shared memory systems can similarly range from large distributed non-uniform memory architecture (NUMA) systems (where different memory regions live on different nodes and must make remote memory requests) down to vector machines (where an array of parallel processing elements operate in a parallel indexed fashion on the same memory space, executing the same program in lockstep, like modern graphics processors (GPUs)). It should be noted that GPU architectures usually do not provide cache coherence for the L1 caches, so there is stronger data coupling in CMP systems, with weaker thread timing synchronisation. Modern GPU systems also relax the thread timing synchronisation, moving closer to a manycore general purpose architecture.

2.2.1 Message Passing

Message passing can be conceptually simpler to program correctly. Data and control flow between the various processing nodes is explicit and clear in the program description, and there are no concerns about data races or mutual exclusion algorithms. The clear separation between local computation and communication makes the programming model much more scalable, and is the predominant means of programming large scale supercomputers (although each individual node may use shared memory parallelism and vector processing).

2.2.2 Shared Memory

Shared memory programming is how most consumer and small scale parallel systems communicate and share information. In the simplest case, all memory is visible to all cores, at the same memory location, so if one thread writes to a variable at address A, any other thread can read from address A and see the new value for that variable.

On modern CMP systems, and older SMP systems with off-chip memory controllers, this is simple because all cores and threads share the same main memory, with cache coherence either enforced by hardware, software, or a mixture of the two. Modern multi-socket machines, and multi-node system, have a less uniform view of the system memory: the address space is partitioned across the different nodes in the system. When a thread needs to access memory, it may need to make a remote memory request, to be served by another node or processor in the system. This is referred to as a non-uniform memory architecture, and the granularity of address space partitioning and allocation, and the protocol for migrating, caching and owning memory regions can vary greatly between different architectures. Because of this, optimisations on a shared memory program for one architecture often do not transfer directly to other shared memory architectures.

2.3 Cache Coherency & Memory Consistency

To improve performance, processing cores usually have a small local memory called a cache, which is used to temporarily hold copies of data from main memory, while it is being used by the core. For a single core system, this drastically reduces the time spent fetching data from main memory, but presents a problem when there are multiple processors or cores in a system. Since each core can cache a copy of any data element, it is possible for two or more cores to cache the same element simultaneously, and without any extra hardware or software enforcement, such cores could modify their local copies of the data. The problem here is that there is no way for each core to know if another has modified any data, and if both cores write back their cache lines, one of them will be over written in main memory. The data elements need not even be the same address, so long as they are cached in the same block. This problem is known as cache coherency, and an ideal system should maintain a view of the shared memory space where all cores can see stores from other cores, and accidental write-back masking cannot occur. The term coherency usually refers to the fact that reads and writes from different cores are visible to each other, and Culler *et al.* [14] provide a more formal definition which explains the property of cache-coherence well:

> More formally, we say that a multiprocessor memory system is coherent if the results of any execution of a program are such that, for each location, it is possible to construct a hypothetical serial order of all operations to the location (i.e., put all reads/writes issued by all processors into a total order) which is consistent with the results of the execution and in which:
> 1. operations issued by any particular processor occur in the above sequence in the order in which they were issued to the memory system by that processor, and
> 2. the value returned by each read operation is the value written by the last write to that location in the above sequence.

Fulfilling this definition results in "write-serialization", which means that accesses to a specific memory location (from any processor) are seen in the same order by all processors.

It should be noted that this definition of cache-coherency does not imply any strictness to the ordering of accesses to different memory locations with respect to each other – this is defined instead by the memory consistency model. The stricter the consistency model, the easier it is to reason about program behaviour and write programs. However, the more relaxed models allow for more aggressive hardware optimisations and less hardware coherency synchronisation, giving better performance. Some processors, like the Sun SPARC architecture, allow the user to switch between multiple models, allowing them to pick the balance they would like, while most consumer systems such as x86 and ARM have a fixed model.

Almost all consumer CMPs provide cache coherency, but deeply embedded MPSoCs forego cache coherency to reduce hardware complexity and power consumption. Because hardware-enforced cache coherency can be expensive, and lead to performance interference between different cores, putting the task of managing memory consistency in the programmer's hands can allow for better control of worst case execution time and a cheaper, more energy efficient end solution. This is a level of programming complexity which most programmers do not want to deal with however, and most software assumes relatively strict cache coherency and memory consistency guarantees. There are also coherence strategies which use only software [15; 16] and can be assisted by the compiler [17].

2.3.1 Memory Consistency Models

When multiple threads of execution on different cores perform a series of memory load and store operations, the memory consistency model defines exactly what orderings are allowed on a given system. These range from the strictest – sequential consistency (SC), to much more relaxed schemes, like release consistency (RC). This section describes a few of the more popular memory consistency models, although a more thorough introduction to these and more memory consistency models may be found in Adve and Gharachorloo's tutorial [18].

2.3.1.1 Sequential Consistency

Sequential consistency (SC) is the strictest of all consistency models. It requires that all threads and cores in the system see the same ordering of all memory operations. This ordering must be equivalent to a possible interleaving of all thread executions on a single time multiplexed in-order core. In other words, the memory accesses are all in program sequential order, with a single global ordering [19].

2.3.1.2 Total Store Order

Total store order (TSO) is the next most relaxed form of consistency, which relaxes SC and allows load operations to be executed in any order. There does not need to be a globally agreed ordering for load operations, however all store operations must complete in a globally agreed ordering – total store ordering [20]. This means that if two cores make a store operation, all cores see them happen in the same order. Memory fences are used on loads to ensure they do not overtake the global store order view, so when a core completes a store operation, a subsequent load cannot see a stale value for a memory location which had a store from another core that committed before its own in the global ordering.

TSO is the memory consistency model found on the x86 processors [21] used in most consumer systems, and is also supported by the Sun SPARC architecture [22].

2.3.1.3 Partial Store Order

Relaxing TSO further, partial store order (PSO) does not guarantee stores from a particular core will appear in memory any particular ordering, with memory barrier instructions required to enforce a particular ordering between two memory operations [22].

2.3.1.4 Release Consistency

Release consistency (RC) is one of the most relaxed consistency models available. Instead of enforcing an order on all load or store operations, it introduces new instructions, with acquire and release semantics. All standard memory operations can appear in any order to other cores, except that on acquire all external stores that were issued before a release are visible, and on a release, all local stores so far are committed and visible. This ensures that the shared memory regions are consistent across memory synchronisations primitives such as a mutex (when to acquire the lock one would involve an acquire instruction, and on release one would similarly involve a release instruction). However, if proper thread synchronisation primitives are not used, there are no guarantees about memory orderings.

2.3.1.5 C++11

The new C++ standard includes a memory consistency specification, allowing for relaxed and atomic memory operations. Programs that need the extra performance of a relaxed memory consistency model can explicitly annotate memory accesses to allow the compiler to avoid emitting memory fences. For memory with extremely relaxed requirements it may be possible to avoid hardware coherency at all (although it may be necessary to track dirty bytes or words in the cache, to prevent over-writing new data with stale, i.e. to provide memory coherency, without the memory consistency or in-cache coherency requirements). How future platforms' coherency and consistency models are adapted to closely model the C++ standard will be an interesting development to watch.

2.3.1.6 Transactional Memory

A completely different approach to parallel programming is encompassed by the concept of transactional memory. This model allows a thread to build up a "transaction" of memory operations, which only finally commit to be visible to the rest of the system if a final condition check is met. This is a popular method for speculative execution, where a thread speculatively takes a lock, operates on a data element, and then commits back the changes if and only if the lock was successful. This allows the internal processor architecture to operate efficiently on the task, while the speculative lock is in flight through the coherency system. An efficient speculative system will kill the thread as soon as the lock returns with a failure, but speedups can still be achieved letting the

thread run to completion if there are enough other cores to take on other speculative tasks. Performing this in software has typically been too expensive, but there have been experimental research platforms such as Atlas [23], and Intel has recently incorporated some hardware transactional memory support into its processors [24]. Transactional memory operations are one area where incoherent memory regions could be used: with a high enough compute-to-data ratio the thread may operate speculatively within an incoherent buffer, then commit back to coherent memory space upon completion. It could also be possible to use large scratchpads or the data-cache as a write-buffer (speculation would have to halt until it was no longer speculative if the cache required a dirty write-back or the scratchpad was full, however).

One form of this transactional memory style operation is a technique called Speculative Lock Elison (SLE) [25] which allows speculative parallel execution of critical sections provided by a mutex lock. SLE begins speculation at the lock acquire instruction sequence, but does not acquire the lock, and continues executing the code path for the critical section until the lock release instruction are reached. If at any point a coherency conflict is detected – a memory location read within the critical section is written to by another processor, or a memory location stored in the critical section is read by another processor – this causes the speculation to abort and either re-attempt speculation, or fall back to actually acquiring the lock and progressing with traditional execution. The processor may attempt to speculate a number of times before resorting to actually acquiring the lock. The speculation must also be aborted if there are insufficient resources to track all memory locations accessed during the speculation, or of the speculation mechanism cannot track any more register changes.

Transactional operations like this can be more efficient than traditional synchronisation because coarse grained locks may protect multiple sensitive memory locations, depending on the control flow within the critical section (such as a lock on a hash-map), and fine-grained locks can incur more significant overheads in terms of synchronisation impact of acquiring these locks, and memory space for storing them (along with the cache-affects of how and where these locks are stored relative to the data they guard and other locks in the code). By speculatively executing through the critical section coarse grained lock regions can be effectively executed in parallel, and only abort and serialise when a true dependency is encountered, speculating fine grained lock regions can remove much of the synchronisation overhead while still providing correctness. SLE is an effective transactional approach because it can be applied to existing code without any changes (and in some cases can be automatically be applied by

the hardware without re-compilation). The fact that critical sections are often short (a requirement to avoid lock contention) means that the resources needed in the processor to speculatively execute them are quite minimal over what already exists for coherency and branch prediction support.

2.4 Coherency Mechanisms

Cache coherency is usually maintained via Snooping or Directory based protocols [26; 27], although more alternative protocols such as Token coherency have been proposed. Each of these broad classifications encompasses a large variety of different protocols.

2.4.1 Snooping

One of the simplest means of ensuring hardware coherency [27], and one of the earliest implemented for microprocessors [28] is a technique called snooping. This relies on all processors using a shared bus to memory, where every core can see the memory requests of other cores by "snooping" the bus. Many of these snooping protocols can be implemented using an alternative broadcast scheme for coherency messages, separate from the memory interconnect.

2.4.1.1 Write-through Invalidate

The simplest protocol is to use a write through cache with a write-invalidate protocol [27]. This allows any core to load any value into cache at any time, because the main memory is always up to date. If they see a store on the bus they invalidate their local copy, and must fetch an updated copy from main memory or a shared cache level the next time they require the value.

2.4.1.2 Write-through Update

With these protocols, instead of invalidating cached copies upon witnessing a write, all the snooping caches take the new value off the bus to update their local copies. This requires the snooping coherency controller to have a port into the data store of the cache RAM however, while write-invalidate only requires the coherency controller to write into the tag RAM.

2.4.1.3 Write-back Invalidate

Obviously write-through caches have a significant performance impact for write-heavy workloads. Also it is often undesirable to broadcast writes because this requires every memory write to gain bus arbitration, and spend energy transmitting the data. As such a variety of write-back invalidate protocols have been developed, usually named after the coherency states involved, which have been shown to be significantly more energy efficient [10].

- MSI: The simplest write-back scheme, Modified-Shared-Invalid, assumes that any clean cache line is shared, and to perform a write it must send all of the sharers an invalidate signal, transitioning the local line the Modified state, and all others to Invalid. To access a cache line which another core has in the Modified state, the modified cache line must be written back to memory and either changed to Shared or Invalid, depending on if the new core wishes to load or store from the address.

- Write-once: This protocol [29] is a hybrid write-through and write-back protocol with states Invalid, Valid, Reserved and Dirty. When a processor first writes to a clean memory location, it performs a write-through operation which both informs all other cores to invalidate their copy, and updates the backing memory store, the cache line now changes to the Reserved state. A subsequent write transitions the cache line to Dirty. Upon a read from another core, the Reserved state would simply revert to Valid, and the second core can retrieve the value from memory, however from Dirty, the first core must block the main memory from providing the data, and provide the data directly. The data should also be written to backing store, and the state transitioned to Valid. If another core wishes to make a write, a similar series of events takes place except the original core will arrive at the Invalid state. This is very similar to MSI, except the first write performs a write-through to optimise the touch-once then share access pattern.

- MESI: Modified-Exclusive-Shared-Invalid [27] is an optimisation of MSI where a cache line currently in use by a single core is marked as Exclusive, rather than Shared. This allows the core to promote it to Modified without sending a message to the other cores in the system, and transitioning from Exclusive to Shared does not require a write-back.

- MOSI: Modified-Owner-Shared-Invalid is another extension of MSI in which a cache can take ownership of the most up to date value of a cache line. This means once a cache line has been modified and another core requests it, the core with the modified line can transition to Owner of this cache line, and serve all requests for the modified cache line, until it writes back to main memory sometime in the future. The owned state can also encapsulate the Forward state of MESIF, if Owned-clean states are allowed.

- MOESI: [26] This is an obvious combination of the MOSI and MESI states, to enable serving of modified cache lines from the owning core, while also incorporating the Exclusive state optimisation.

- MESIF: Proposed by Intel [30], this protocol substitutes the Owned state for a Forwarding state. Unlike the Owned state, the Forwarding state must always be clean. Introduced to optimise the case where a cache miss is fulfilled by another on-chip cache, this state allows exactly one cache to respond, rather than all Shared state caches [31].

2.4.1.4 Write-back Update

The final set of protocols allow the update of remote caches, while avoiding write-back to main memory for every store operation.

- Firefly: [32] This protocol contains three states, corresponding to the M, E, and S states of MESI. Some argue that because there is no invalidation in Firefly that it does not have an Invalid state, but when cache line has no contents because it is cold then it could be considered Invalid. If a cache line is held clean by a single core, it is Exclusive, this can transition to Modified without signalling other cores, or transition to Shared if another core makes a request for the cache line. If the cache line is Modified and another core requests it the cache line must be written back to main memory and transitioned to Shared. Once in the Shared state the cache line remains in this state until it falls out of all caches due to capacity or conflict misses over time. A write to the cache line while in the Shared state acts as a write-through update operation, where both the main memory and all snooping caches are updated at the same time. For this reason sometimes Firefly is classified as a write-through update protocol, even though the single sharer case is a write-back behaviour.

- Dragon: [33; 32] A very similar protocol to Firefly, Dragon differs by differentiating the Shared state into Shared-clean and Shared-dirty, equivalent of the Shared and Owned state of MOESI. A write to a shared cache line in Dragon requires updating all snooping sharers, but does not write back to main memory, making it more efficient than Firefly. The most recent writer to a cache line takes the Shared-dirty state, while all others are Shared-clean. It is the responsibility of the Shared-dirty cache to fulfil cache line requests and write-back to main memory on eviction (which will be due to self invalidation from capacity or conflict misses).

2.4.1.5 Snoop Filtering

To reduce the accesses to all caches when broadcasting invalidation or fetch requests, and traffic for more complex interconnects, it is possible to add special filtering controllers to various points on the bus. These reduce accesses to caches which don't contain the data by storing an approximate but conservative overview of the sharing state, which can be used to remove cores from the broadcast operation when it is known that it definitely has not cached this data. One mechanism is to create a form of directory-like cache which sits on the bus and is responsible for actually sending out broadcasts. When it has information about the cache line it uses this to reduce the broadcast to a multicast, but if the cache has over-flowed it will send a full broadcast. This can save on energy when a cache line is being rapidly passed between a subset of the cores in the system. Another snoop-filtering mechanism is to put Bloom-filter structures in front of each core, or clusters of cores, to track addresses that have been recently cached. The Bloom-filter with either guarantee that a cache line is not present, so the request will not propagate to the cache, or return that the cache line may be present, so the request must be forwarded. So long as the energy to maintain the Bloom-filter is less than the energy to access all of the cache it guards there is the potential to save energy. If the cache tag RAM has a limited number of ports it may also improve performance, since the core will have to arbitrate with bus snooping requests less often.

2.4.2 Directory

Directory protocols are a more scalable solution which do not require the cores to constantly snoop and broadcast to each other, instead a dedicated structure is used to track the state of cache lines in the system, and is used to coordinate the cache

line coherency state transitions. On older multi-processor systems, which use discrete processor chips, it would be more likely that a full directory is implemented in main memory, to store the sharing state for every cache line in the memory space. This has a very large memory overhead, but allows the simultaneous caching of any and all cache blocks in the system. For multiple processor chips the first common level in the memory hierarchy is the main memory, so this is an obvious level at which to perform the directory service. On-chip directories can either act as a directory-cache for an off-chip full directory, or maintain directory information just for memory blocks currently residing in cache. Both on- and off-chip directories need not be centralised, and can instead be distributed across last level cache blocks, or processor nodes in a large scale multiprocessor system. Distributing the directory enables higher throughput, at the expense of complicating the directory coherence protocol.

For inclusive cache hierarchies the sharer information can simply be kept in the last level cache (LLC) tags, maintaining the directory information without dedicating a whole new directory structure. However for large scale systems, a fully inclusive shared cache can become undesirable, and the directory structure must either become extremely large and associative to allow all on-chip caches to be fully utilised, or accept that it cannot contain the maximum number of unique cached memory blocks' sharing state.

2.4.2.1 Sharer-reconstruction

Instead of requiring the directory to hold all on-chip cache state, the directory can instead be used to cache a subset of the cache state, providing efficient coherency transitions when possible. When there is a miss in the directory, a broadcast probe must be sent out to reconstruct the sharer information from the processors in the system. This can provide good performance when the application benefits most from full utilisation of the on-chip caches, but suffers the high energy and performance cost of having to snoop the whole chip whenever there is a cold cache miss or the working set is significantly larger than the on-chip caches [34].

2.4.2.2 Forced Invalidations

An alternative to snooping to reconstruct sharer state is to instead invalidate cache entries when the directory becomes full and needs to replace an entry. This saves the expensive snoop on a directory miss, but can reduce performance by limiting the utility

of on-chip caches if the directory is not large or associative enough. Previous work has shown that this is more scalable than the snoop filtering approach [34].

2.4.2.3 Off-chip Backing Store

Instead of discarding directory overflows, it is possible to overflow the on-chip directory to a dedicated region of main memory. This can be handled either by hardware, or trap to a software handler. There are two aspects of the directory which can overflow, the number of cache line entries, or the sharer list for a given entry, examples of each are discussed in Section 3.1.

2.4.2.4 Highly Associative Directory Caches

Various work has been done in increasing the apparent associativity of the directory cache to cope with the high associativity of the on chip caches (an N core CMP with M way set associative caches requires an $M \times N$ way associative directory structure for full coverage). These have predominantly focused around using different hashing functions to index into different ways, to reduce aliasing between different cores' caches, allowing them to use larger sets, but fewer of them, to reduce parallel lookup energy and time. The main work is encompassed by the Z-Cache [35], Cuckoo Directory [36], and Scalable Coherence Directory [37] (SCD) papers discussed in Section 3.1.

2.4.3 Other Sharer-Tracking Mechanisms

2.4.3.1 Token Coherence

Rather than have a directory structure to track all sharers, Token Coherence [38] works using broadcasts in the same way as a Write-back Invalidate snooping protocol, but enabling correctness on unordered interconnects by using tokens. With Token Coherence each cache block has an associated number of tokens, T, the exact number is not important to the correctness of the protocol but limits the number of sharers which can simultaneously hold a read-only copy of the data. The tokens reside in caches, main memory, and in-flight transactions, but exactly T tokens must exist at all times for every cache line. This requires additional memory space to store the token count for cache blocks which are not live in the processor caches, and extra bits in each processor cache line. For a read operation the processor requires at least one token, while for write operations all tokens must be held. This token protocol ensures that all tokens

for a cache line contain the most recent cache-line data, which is only modified when all tokens are together at a single core. The token policy ensures correctness, but the mechanism by which tokens are acquired depends on the specific performance policy implemented, with TokenB (Token Broadcast) for example making a full chip broadcast for both read and write misses. If a read request is received at a core or cache, then a single token is sent back with a copy of the cache line data and the local count reduced by one, if a write request is made then all tokens must be returned. When a cache line is self-invalidated then the token count must be written back to the next level of cache or memory hierarchy. When races occur, such as two concurrent write-misses to the same cache line, a time-out on the transaction is used to ensure that cores which fail the transaction make a subsequent attempt after a fixed delay. To prevent starvation, after four consecutive failed requests a special Persistent request is made. Each processor has a small table to record any pending persistent requests, and the processor will always forward tokens for this cache line to the requesting processor to ensure that it is satisfied. It has been shown that for a 16 core processor, across a range of benchmarks, requests are only re-issued for 2-3% of transactions, and persistent requests only required for 0.1-0.3%.

Token coherence can require a lot of cross chip traffic, but its point-to-point style of communication is well suited to a highly connected architecture such as a mesh or torus, however the broadcast requirements can be challenge to the on-chip bandwidth without hardware broadcast or multicast support [39]. While other performance policies such as TokenD (Token Directory) and TokenM (Token using Multicast) reduce the bandwidth requirement, the number of bits required to track the token counter, along with the cross-chip traffic cost, both scale poorly when considering architectures with hundreds of cores and workloads which require whole-chip data sharing.

2.4.3.2 Sharer-Chaining

Alternatively an in-hardware linked list of sharers can be used in a similar fashion to Scalable Coherence Interface (SCI) [40]. This method allocates every cache block a home node (like a distributed directory), where only a pointer to the first sharer is stored. Each core that holds a cached copy then joins the linked list by holding a pointer to the next core in the list. To read the cache line, a core must send a request to the home node. Then it will be inserted into the start of the linked list, with the home node pointing to it, and it pointing to the core which was previously at the start of the list. When a core wants to write to the address, it must walk the linked list

sending invalidations, until it has received all responses, the home node entry is locked to prevent read requests progressing until the write is complete. A similar process must occur if the home node runs out of capacity and must evict the entry for this cache line.

2.4.4 Software Managed Coherency

Not all platforms provide transparent hardware coherence, but it is possible to use the operating system and traditional MMU memory protection features to provide a running application with the illusion of cache coherency, and even provide a distributed system with the appearance of a single shared memory space. These work by marking shared pages as read-only, then on a write fault communicate with other processors in the system to arbitrate write access and disable the other read-only pages (depending on the memory consistency model), before allowing the processor to promote its page to writeable. If another processor has a read-fault on this page, it will request a "diff" or full copy from the core which modified the page. There has been work on optimising this process [41] and relaxing the memory model to allow multiple writers to the same page. There are also hardware assisted techniques to automatically handle remote memory accesses, where pages which are often modified are pinned to a single core, somewhat like the Owned state in the MOSI protocol, except that writes may be stored directly to the remote cache, without a local duplicate being created [41].

It is also possible to use techniques like this to simply reduce the demands on the hardware coherency system, to isolate pages that do not require coherency (such as code pages and thread local memory) and only run hardware coherency on pages for which it is necessary.

On deeply embedded systems without any form of page protection, it is up to the programmer to manage cache behaviour and ensure there is no write-masking. Special cache-bypassing instructions can be used to synchronise cores and coordinate the flushing of caches as necessary. Systems like this can be easier to program using an abstracted message passing library, leaving the complicated cache management to the expert library writer.

Software based coherency can also benefit greatly from compiler support [17], although the results of many studies comparing software and hardware coherence has been inconclusive, due to the different circumstances under which each is preferable [42]. Like many things, a compromise where some hardware support is provided, but software only uses it when necessary, is probably the optimal solution.

2.5 Architecture Scalability

When designing a CMP or MPSoC architecture there are a lot of areas that are a scalability concern, the most dominant being the performance of the cache coherency mechanism, and the properties of the interconnect joining the cores, caches, and other on-chip components. This section will focus on the scalability of the interconnect options.

2.5.1 Buses and Crossbars

The shared bus, and the crossbar are two of the most poorly scaling interconnect options. The shared bus is a single communication link that is shared and arbitrated for all components, connecting all cores to the memory system. This allows the simple implementation of snooping coherency, but scales poorly as more devices are attached to the bus. Ignoring physical constraints at first, the bus bandwidth must be shared between all devices, this means that the per core bandwidth scales as $^1/_N$, where N is the number of cores in the system. When the electrical characteristics of the bus are considered, adding more devices requires the bus to grow approximately linearly with the number of devices. Unfortunately, the longer the bus, the greater its electrical capacitance, so the bus must operate at lower frequencies or be driven at a higher voltage. The wire lengths along with the number of devices will also affect the complexity of arbitration, and how quickly it can be resolved. This all means that the frequency and bandwidth of the bus itself may have to be reduced, making the per core bandwidth scaling worse than $^1/_N$.

Unlike a shared bus, a crossbar allows all connected devices to send a message to another device simultaneously, so long as there is not a conflict at the destination. This interconnect is good for core-to-core communication, or NUMA systems. One of the largest crossbar architectures actually produced is probably the 80-core Cyclops-64 [43] (discussed in Section 3.2.5), produced by Cray to be used as nodes in a supercomputer. Unfortunately crossbars cannot be used with passive snooping, so a snooping based coherence protocol would require dedicated request and acknowledge messages. They also scale extremely poorly in terms of area when adding more nodes, and when increasing the width of the connection.

2.5.2 Mesh and Tiled Architectures

One of the most popular manycore interconnects is the 2D mesh, used to build a tiled architecture. The 2D mesh has many traits which make it an obvious choice when designing a scalable architecture:

- bandwidth scales linearly with the number of nodes
- area overhead per node is constant, each new node requires the same router and connecting wire overhead
- switch radix is constant, all routers are the same, and adding more nodes does not increase the number of ports required on each
- supports core-to-core communication
- redundant routing for fault tolerance
- short wires, so the links can be run at high frequencies.

Unfortunately the downsides to mesh topologies are:

- average node to node latency is ${}^{2}/_{3}\sqrt{N}$
- every router is at least 5×5, which is quite large, and can take multiple cycles to route
- hard to verify that complicated routing rules and virtual channels are free of deadlock
- no simple multicast or broadcast scheme
- no single point of serialisation for coherency protocols, which requires more complex protocols with a greater number of messages.

2.5.3 Tree and other NoC architectures

There are other interconnects that trade off bandwidth in return for latency, and use much simpler switches. One of the simplest conceptually is a binary tree (or H-tree, named for the shape it takes when routed on a chip), where a central last level cache or coherency controller might reside at the root. The tree may also be used without a root node, in a point to point communication fashion, where messages get reflected back

up the tree when they have reached the first common node. This would resemble each half of the chip being an H-tree fanout, with the root of each tree linked to connect the two halves of the chip together. Picturing it like this makes the downside of the tree obvious, the bisection bandwidth is that of the single link between the two roots.

The benefits of tree based topologies are:

- switch area overhead per node is approximately constant, N nodes require $N-1$ switches
- node to root latency is $\log 2N$
- switch radix is constant, all routers are the same, and adding more nodes does not increase the number of ports required on each
- every router is 2×1 (or 2×2 if core-to-core routing is enabled at all levels), which is small and fast to route
- supports core-to-core communication
- centralised point for coherency transactions if a coherency controller such as a directory is placed at the root node.

Unfortunately the downsides are:

- bisection bandwidth is limited to a single link, like the shared bus
- no redundant routing for fault tolerance
- long wires at the root of the tree, so operating frequency may be limited.

The bandwidth concerns can be mitigated slightly by using Fat Trees [44], which use higher bandwidth links between the lower nodes and the root, but this does not help for small packets. Long wires can also be split with registers, but this introduces extra latency and can complicate arbitration policies.

Between the extremes of the mesh and tree architectures, are other multi-state logarithmic networks, such as the delta networks: Omega, Butterfly, Baseline, and Benes [45]. These all work by implementing a perfect shuffle network between the source and destination, taking $\log 2(N)$ layers to do so.

2.5.4 Design Space Exploration

One of the biggest reasons for doing cycle accurate simulation is to perform explore the parameter space of a hypothetical design. When trying to architect an MPSoC for a new application, or design new features for an upcoming CMP, it is important to be able to test new considerations quickly, and be able to prune out unlikely candidates early on in the process.

When it comes to selecting from existing hardware features to match a new application, recent work [46] has shown that machine learning techniques can go a long way to predicting the performance of a new application across a large design space, providing the design space itself has been profiled already with a large set of training applications, and the new application has been profiled on at least one prospective design. It is possible that this will extend to larger many-core architectures and this thesis tests that hypothesis in Chapter 8.

2.6 Simulation

One of the most powerful tools for an engineer is a good simulator. Simulators are invaluable for tasks such as providing performance estimates while iterating over design options, verifying design correctness, or developing and testing application code ahead of hardware availability.

2.6.1 Emulation and Instrumentation

When simulating processor architectures, the first hurdle is performing correct functional behavioural simulation, parsing and executing each instruction faithfully to the target ISA. When performing on-host ISA simulation, i.e. simulating the same target architecture as the host platform the simulation is being run on, an approach called instrumentation can be used, where code is inserted into the application binary being simulated around key events (such as control flow and memory instructions) through recompiling or dynamic binary translation (DBT) techniques. These extra code sections can be used to calculate changes to the timing model, or communicate with another thread executing the timing model to perform asynchronous simulation of micro-architectural state and simulated IO devices. Instrumentation based simulators often boast very close to native execution speeds, due to actually executing on-host, but suffer accuracy problems with multicore systems, and are limited to executing the host

ISA. This makes them difficult for investigating ISA modifications, or interconnect micro-architecture features.

Emulation based simulators are those which decode the target instructions and re-map them to instruction sequences on the host. Using JIT-compilation and DBT techniques they can provide simulation on-par with instrumentation based systems, but allow more flexible simulation of modifications to the target ISA.

It is possible to construct an accurate simulator using either technology, but truly cycle accurate multicore simulation is usually slow regardless of this choice.

2.6.2 Single-Core Simulation

The simplest simulations are those which only deal with a single core system, as they do not have to deal with memory contention modelling, synchronise accesses with other cores, nor handle any discrepancy between host and target coherency protocols. While still a powerful tool for core micro-architectural research, they are often not useful for looking at modern multithreaded programs. This is not to say there is nothing that can be taken from high speed single core simulation to help accelerate multicore simulation however. The next chapter covers a few notable single core simulators, while the concepts such as decoupled simulation to exploit simulation parallelism used in the fastest single core simulators are discussed in the rest of this section.

2.6.3 Serial Simulation

The simplest way to write a deterministic, correct simulator, is to write it as one thread of serial execution. For functional simulation this might involve interpreting a fixed number of instructions, before switching to an IO device or another processor thread, and for cycle accurate simulation will often result in a core simulation handler executing a number of attached models one cycle at a time, in a round robin fashion. While simple and correct, this is the slowest form of simulation.

2.6.4 Decoupled Simulation

Decoupled simulators work on multiple parts of the simulation in parallel, in separate threads. Some simulators decouple functional and micro-architectural simulation [47; 48; 49], while others use decoupled threads to provide JIT compilation services to accelerate the primary simulation thread [50]. Multicore simulators can simulate target

cores in separate host threads, but the degree of decoupling here is very dependant on the simulation model and accuracy requirements.

2.6.5 Parallel Relaxed System Simulation

Multi-processor simulators have been parallelized to take advantage of modern multi-core systems, with the simulation divided between host cores, usually at the granularity of target cores. However, these simulations still synchronize after every cycle to maintain correctness; this introduces significant overhead, and stalls the whole simulation when the operating system context-switches one of the threads. To reduce this overhead simulators have increased the synchronization quantum beyond a single cycle, but this introduces inaccuracies in simulation. These systems also only typically model a shared bus or crossbar, and may not model the effect of shared bandwidth correctly if the quantum is larger than a single cycle.

The most relaxed and decoupled simulations run target cores in parallel threads, and parts of the interconnect in other threads, using timestamped message passing between them to form a loosely timed model. Absolute accuracy suffers in these systems, but they can still be useful for debugging routing protocols and software running on the system.

2.6.6 Hardware Accelerated Simulation

Perhaps the only way to perform cycle accurate simulation fast, is to use dedicated hardware to accelerate the most complicated parts of the model. There are successful projects which use FPGAs to implement all or part of the simulation (usually micro-architecture and interconnect models at least), and industry makes extensive use of large FPGA based modelling and prototyping systems [51]. Another use of specialised hardware is the use of GPU acceleration of functional and RTL simulation, where the extremely parallel and well synchronised nature of the graphics processor can provide large speedups.

2.7 Machine Learning

Machine learning is the name given to the use of algorithms which can “learn” a model to correlate some given inputs to an output value, by processing representative sample data. Most machine learning algorithms either classify (as in the classic “spam” filter

example), or predict. In the case of design space exploration this work is most interested in algorithms which can learn to predict features of a design, such as performance or energy, from training data and only a small set of input data upon which to base its prediction. This thesis treats machine learning methods as black box tools, so no deep understanding of their mechanisms is required.

2.8 Energy Modeling

Often when comparing two or more items in computer architecture, be it a different router implementation, or a different cache-coherency policy, it is desirable to be able to evaluate the power or energy differences involved. In an ideal world one would always be able to produce an energy figure in SI units of Joules, or Watts, with which the value can be compared against any other energy figure, within the same piece of work, or from other studies.

Unfortunately in computer architecture one is often dealing with abstract concepts, and the real world energy values are very much dependent on implementation details. This makes energy comparisons between different features implemented with different techniques completely incomparable, as the difference of interest is lost in the other implementation differences. To obtain this real world energy value also requires significant computational effort for gate level power simulation, or better yet a manufactured silicon implementation which can be measured.

When comparing abstract concepts such as cache coherency policies it is not feasible to implement a different chip for each point of interest, and this would be a huge waste of resources. Instead higher level approximate models can be used.

When using abstract energy models it is important to remember that two values can only be compared if they represent the same concept, and the same assumptions are made about the energy model for each. For example if you assume that a 2-way set associative cache requires twice as much energy to access as a direct mapped cache, then you can compare the energy attributed to the cache accesses in two different processor models. This model does not let you compare the cache energy to the energy consumed by the branch predictor, for example, only between the two caches. However, so long as the initial assumption that the energy of the associative cache is twice that of the direct mapped cache is defensible, the results of comparing the two models are sound.

This thesis makes use of a few energy energy models at similar levels of abstraction.

For Chapter 8, which uses non-communicating multi-program workloads, it is assumed that dynamic core energy is the same regardless of platform, since the program executed does not change significantly regardless of how many cores it is run on (this neglects the difference that frequency and voltage would make). It also neglects static energy consumption, assuming that compared with network and dynamic core energy it will be negligible. Finally the network energy is modelled as a fixed energy cost for a packet traversing a link, weighted by the width of the link, i.e. a fixed cost per wire, per transfer. The only energy used from these models is the network energy measurements, because these are the only ones that change in the model used, and without synthesis and detailed simulation there is no reasoning with which to weigh them against either dynamic core energy or static energy. For Chapter 8's purposes the energy model is only required to provide an interesting feature, which behaves like an energy model, to the machine learning model. As such the accuracy of the model itself is not very important, since the work is evaluating whether the model can predict it. The important factor is that the energy model is complex enough to be as hard to predict as a more accurate model. This energy model is not very different from that used by Fensch *et al.* [52], which assumes that network energy is proportional to the quantity of data transferred.

For Chapter 6 a much more sophisticated model is used, which isolates multiple static, dynamic, and wire energy components for individual comparison. Here it is assumed that static leakage power scales linearly with the number of transistors, and as such, for the presented architecture, with the number of cores. Energy lost to static power over the runtime of a benchmark is therefore modelled by the number of cores, multiplied by the runtime. These values are directly comparable to other static energy values in the chapter. Because there are no synthesized models, the dynamic power of different components must be isolated and compared only to other components of the same type. For processor cores, the number of instructions executed, and the number of data cache accesses are presented as metrics for dynamic core energy, since for the simple core used, cache accesses may be a significant contribution to the overall energy consumption, while other instructions will not vary so greatly. Network switches are categorised by their radix and bus width, making the reasonable assumption that a wider bus takes more energy to route than a narrower bus, and a higher radix router requires more energy also. Without performing some level of synthesis a reliable weighting with which to compare them cannot be provided, so no comparison is performed. Finally, for wire energy a capacitive model is used, like that used in other

research [53; 54], where it was noticed that for a fixed technology implementation (i.e. same voltage, drive strength, silicon and metal characteristics, and wire width) the constants can be combined to give Equation 2.1. This reads as: total wire energy equals the sum of all wire transitions (from 0 to 1, and 1 to 0), multiplied by the length of the wire. This result is equivalent to the self energy term in the more advanced energy model presented by Sundaresan and Mahapatra [55]. This more advanced energy model takes into account the interaction between adjacent wires in the bus, but the model used in this thesis is too abstract to calculate appropriate coupling capacitances between adjacent wires. Enhancing the energy model would also be a matter of calibration and engineering effort, rather than providing any innovation over the current state-of-the-art; the more advanced energy model is only slightly more difficult to compute than the model in use already.

$$Energy = C \times \sum Transition \times Length \tag{2.1}$$

By using this wire model the wire energy of all buses in the NoC can be compared, unlike the routing energy which must be considered separately.

For Chapter 7 the same core energy model is used, considering instructions executed and cache accesses an instructive approximation for dynamic core energy. However for the interconnect only the routing events of the multicast network are considered as, this is the area of focus for this chapter.

Chapter 3

Related Work

This chapter follows a similar format to the background material, but focuses on specific items of related work that demonstrate the concepts discussed in the Chapter 2. The order of material is the same with the first section looking at coherency and directory schemes, the second looking at existing and proposed manycore architectures, the third looking at system simulation and the final chapter looking at higher level design space exploration and prediction.

3.1 Coherency Components

3.1.1 Efficient Directory Storage

Since the directory has moved to be an on-chip structure, and space is now a serious constraint, there have been several notable works to reduce the size of the directory, taking primarily one of two orthogonal approaches: reducing the required size for each entry by shrinking the representation of the sharer set, and reducing the number of entries required for acceptable performance.

3.1.1.1 Sharer list representation

Since the number of cores actively sharing a cache line is unlikely to be the set of all cores in the system, one way to reduce the storage requirements of the coherence system is to reduce the size of each entry, by reducing the number of cores that can be simultaneously tracked.

The easiest way to store all possible cores is to use one bit per core, requiring exactly n bits of sharer state, where n is the number of cores in the system, but to reduce

the number of identifiable sharers another encoding approach is needed. The simplest option is the "Limited Pointer" or Dir$_k$[N]B [56] representation, which stores k pointers to cores for exact messaging, requiring $k\log_2(n)$ bits of sharer state, and then either invalidates a sharer to make room (Dir$_k$NB), or falls back to broadcasting when there are more than k sharers (Dir$_k$B). The ATAC paper [57] extends this slightly to track the number of sharers once a broadcast is required, so that non-sharing cores do not have to respond to invalidation requests. This AckWise protocol requires the same storage space of $k\log_2(n)$ bits (if a sharer pointer is converted into a counter once there are more than k sharers), and can significantly reduce the energy required if the coherence protocol requires acknowledgements for broadcast or multicast messages. The later work in this thesis avoids the issue entirely however, by designing a coherency protocol which does not require these acknowledgements, and as such does not benefit from tracking the number of sharers.

An alternative is to let the sharer list overflow into another data structure, either hardware or software managed. This overflow mechanism has been implemented using a software fall-back in the LimitLESS protocol running on the Alewife machine [58], which although comprised of physically distributed processors, and a similarly distributed directory, does not allocate enough memory for every coherency block to be shared by all possible sharers simultaneously, as this would be a huge waste of memory. Instead the LimitLESS protocol traps to a software handler via an interrupt when the local node encounters a sharer overflow while handling a request, and handles the situation with software. From this point on, until the sharers are invalidated, any coherency action for this region would trap to software and let the runtime system or operating system manage the transactions. Because the processors in the Alewife system supported a very lightweight and fast interrupt mechanism, and each processor was closely coupled to its slice of physical memory, the overhead for this was acceptable. Unfortunately in a single chip manycore the general purpose cores running the software stack would be too far away from the directory and memory controller to perform this task efficiently. Thus to implement LimitLESS on a modern manycore would require either a dedicated processing core to handle these directory overflows, or the software part of limitless would have to be implemented in a dedicated hardware engine.

Taking things back on chip, Scalable Coherence Directory (SCD) [37] proposes using highly associative directories and allowing a single cache line to occupy multiple entries across the sets, enabling it to expand the sharer list dynamically as required,

at the expense of directory capacity and associativity for other cache entries. This approach requires every directory way to be evaluated if a cache line in the shared state is accessed however, such as to invalidate upon a write to the cache line, which can consume a large amount of energy, and if performed serially will take significant time for the transaction. Since most sharer patterns only have a small number of sharers, the average performance is acceptable however, but poor handling of pathological cases (such as all cores sharing to read a barrier state flag) can significantly reduce the performance. The actual encoding used by SCD is hierarchical bit-vectors, where $\sqrt{(n)}$ bits are used for the root vector, and for every bit which is set in this vector, a second entry is used with another $\sqrt{(n)}$ bits to fully specify the sharers in that subtree.For a small number of sharers ($\sqrt{(n)}/\log_2(n)$) a limited number of pointers are kept first, so small sharer sets fit within a single entry, then the state gradually fills to represent a full bit vector as the number of sharers increases to a full chip, and occupies $\sqrt{(n)}+1$ directory entries, requiring at least this degree of associativity in the directory. The final sharer list could be multicast in multiple bursts, for example each bit in the root vector could result in a multicast to its region, with the full bit-vector attached as a $\sqrt{(n)}$ bit payload. Unlike lossy encoding schemes, if acknowledgements are required from sharers an early self invalidation from a sharer can be handled as it happens, rather than the core still having to respond to an invalidation request as in the case with AckWise. The SCD paper claims that capacity invalidations are sufficiently rare that they do not need to define a replacement policy, but the two options when a second level bit-vector must be evicted are to invalidate all sharers in this bit-vector, or let the entry be evicted, but assume that if an entry is set in the root bit-vector, but the second level is absent, that is is a full broadcast to that subtree. Working with this second option could allow space to be reclaimed by de-allocating second level bit-vector entries which have occupancy greater than some threshold (such as 90%, or completely full).

In the directory comparison paper [56] where Dir_k[N]B is discussed, an alternative suggestion is made of combining together core pointers, where each pointer bit is represented a two bit symbol. For example walking through the core ID in binary a 0 would be encoded as 01 and a 1 as 10, cores are combined into the sharer representation with a simple bitwise or on the symbols, resulting in a 11 symbol where the core IDs diverge. This is an early suggestion for a lossy encoding of the sharer vector, which does not result in a full broadcast; sometimes referred to as DirX, or Tristate encoding [59; 60]. In Chapter 7 this thesis evaluates the Tristate option for 1024-core systems with synthetic tests and finds that it suffers many pathological cases. While

being better than a full broadcast, it appears to be a better use of space to simply store a coarse bit-vector, a conclusion shared by Gupa *et al.* [61], although Mukherjee and Hill arrived at a different conclusion when using benchmark data [59]. In this same paper, Mukherjee and Hill suggest using a Gray coding scheme to encode the tristate information, to reduce the number of different bits set to the same 11 combined state when sharers are adjacent. This means that adjacent pairs of cores sharing information will only have a difference in one symbol, and as a result will always be perfectly identifiable, where the naive tristate encoding has pathological cases such as core 7 and 8, which when added in a naive tristate encoding also aliases to cores 0-15. However, performing the Grayscale encoding in hardware adds complexity to the directory controller, and does not result in an encoding suitable for multicasting in the same manner that coarse vector and naive tristate are. The same work also proposes performing the Gray-code mapping at the software layer when allocating the processor threads to cores; when trying to map a core adjacent to another, it instead identifies a core which results in a Gray-code neighbour, i.e. it identifies a core which only differs by one bit in its location. This allows the naive tristate hardware to function efficiently, and enables higher dimensionality mappings, such as software which is logically mapping threads to a 3D mesh, can be mapped to the same naive tristate encoding effectively, where the hardware Gray-code based tristate encoding can only operate on a single dimension (having no information about the application). Unfortunately, using this software Gray-code approach results in sharers which are physically non-adjacent on the processor. This can result in poor use of the memory hierarchy and interconnect bandwidth if there are shared caches which could be utilised, or direct core-to-core data forwarding is supported. It also could prevent regions of the processor dropping to a lower power state as threads are not allocated contiguously, and could result in a energy inefficient multicast path when messaging all sharers, because the symbol that does differ, resulting in a path branching, could be in the more significant bits of the core location code.

The aforementioned paper by Gupal *et al.* introduces another other method of shrinking the sharer state: instead of one bit per core, use one bit to represent a fixed number of cores – a coarse vector representation of the sharers. This is another lossy compression scheme, which must send messages to a minimum of n/k sharers, where n is the number of cores, and k the number of bits used. If more than $\log_2(n)$ bits are used the memory can be used to represent a limited number of fully qualified core pointers until there is no more space, and the system reverts back to a coarse vector

representation, helping address the inefficiency of coarse vector for a very small number of sharers. However, these core pointers would require unicast messages on most networks, while coarse vector is simpler to multicast. This is a very simple representation to compute, unlike the Gray-coded tristate encoding, and is significantly more robust than any of the tristate options when given random sharer patterns. Like AckWise, coarse vector can be used as an obvious enhancement to Dir*k*B, replacing the broadcast with a multicast. If more than $\log_2(n)$ bits of state are used, coarse vector can be used as an enhancement to AckWise, giving some multicast (or broadcast filtering) support with the bits not required by the sharer counter in AckWise.

Acacio *et al.* propose another very compact encoding for binary tree based systems using distributed directories where each core is responsible for a slice of the directory [62]. The paper proposes a Binary Tree encoding, for binary trees which have directories at the leaves, not the root, which stores the highest level of the tree that is shared in common between all sharers and the home node for the cache line, performing well for data located in a home node close to the cores using it, and only using $\lceil \log_2(\log_2(n)+1) \rceil$ bits per directory entry. However for data located far away from the home node, this performs very badly, so Acacio *et al.* suggests using a small full map cache, in front of a larger directory storage which uses the compact binary tree encoding. To further improve on the effectiveness of the binary tree encoding they propose using an extra two bits of storage to reference what they call a Symmetric Node of the home node; this involves allowing all 4 possible combinations of the two most significant bits of the home node address, enabling three alternative reference locations to be used for the tree. To compute the new sharer state, the smallest tree covering all sharers from the four possible reference nodes is used, giving much better performance when sharers are clustered together, but are not in the same sub-tree as the home node; this requires $\lceil \log_2(\log_2(n)+1)+2 \rceil$ bits of state. Neither of these compact encodings allow for the unique referencing of a single sharer however, so even invalidations to exclusive lines will require multiple excess messages, and increasing the storage to $\log_2(n)$ bits does not benefit the compressed sharer representation. To deal with this Acacio *et al.* propose a final more elaborate encoding, which uses $\log_2(n)$ bits to represent the single sharer state, and switches to a binary tree representation they call Binary Tree with Sub-Trees. This binary tree representation uses two binary trees to encode the sharers, a tree relative to the home node, plus an extra tree relative to one of the symmetric nodes; when the sharer state is computed the combination of home node relative tree, plus symmetric relative tree, which gives the least false sharers is chosen,

although the paper does not give an example algorithm to compute this efficiently.

The simple binary tree encoding is very simple to multicast from the home node, requiring only a very small number of bits to be transmitted to determine when the correct tree depth has been reached for the subtree broadcast to be sent; the other schemes however require $\log_2(n) + \log_2(\log_2(n))$ and $\log_2(n) + 2\log_2(\log_2(n))$ bits respectively, because the symmetric node must be fully specified.

For a single directory, at the root of the tree, this binary tree style encoding could be adapted to point to the lowest subtree which contains all sharers, avoiding the issue of the home node not being in the same subtree as the sharers. However, it requires $\log_2(n)$ bits of sharer state to specify an arbitrary node in the binary tree from the root (or $\log_2(n) + \log_2(\log_2(n))$ for the simpler encoding which uses a full leaf destination and a short code to specify the depth to truncate at, like would be required for the symmetric node multicast), rather than the more compact $\log_2(\log_2(n) + 1)$ storage for the distributed sharer case.

Another method, presented as SPACE [63] by Zhao *et al.*, is to store a separate cache of sharer vectors, and for each directory line store a pointer into the sharer cache entry which matches the sharer vector. This only works well if there are a limited number of sharing patterns in use, but similar patterns can be merged at the sacrifice of a small amount of aliasing. The biggest downside to this method is the expense of finding the sharer vector in the table, which can be highly associative, but also the required number of entries for good performance on 1024-core architectures may grow to the point where a large sharer vector is once again needed to point into the sharer table.

One option for keeping individual directory entries light is the linked-list, or chained directory, option used in the IEEE standard Scalable Coherent Interface [40] (SCI). In this protocol the home node keeps a pointer to only the first sharer, and then each sharer keeps a pointer to the next. In theory this has no limit to the number of uniquely tracked sharers, but requires traversing the whole linked list, sending an invalidation to each sharer and awaiting their response, in order to perform an exclusive operation on the cache line. SCI requires each core to be associated with its own slice of directory, and when taking into account the overhead of memory control logic is likely to be a waste of die area for a processor with hundreds of tiny cores. A reasonable alternative would be to operate SCI across a group of clusters, and rely on another encoding, such as full or coarse bit vector, for the local tile, and use SCI to deal with inter-tile coherency.

Recently a proposal was made for a completely different sort of directory struc-

ture, the Tagless Coherence Directory [64], which uses a table of Bloom filters to conservatively store information about which cores might hold copies of a given cache line. When a cache line must be accessed in an exclusive fashion, the Bloom filters are queried to return a full map bit vector which identifies the cores which definitely do not contain a copy of the cache line, requiring a message to be sent to all those remaining which may contain a cached copy of it. This message may then be sent unicast, multicast using the full bit vector (requiring a problematic number of wires if there are a large number of cores) or be fed into one of the lossy multicast encodings such as coarse vector or tristate, or the lossless multi-phase multicast for a hierarchical bitvector.

3.1.2 Invalidation Multicast Energy

Something not discussed in the early papers is the network energy used to multicast these schemes. In fact, many of the papers still expect multiple unicast messages to be used, under the assumption the interconnect was a simple mesh. In fact many of the encoding schemes discussed so far are amenable to multicasting on a binary or higher order tree, but there is a trade-off to be made between the accuracy of the encoding, and the number of bits that must be transmitted along the network to carry the encoding. The unicast based invalidations of DirNB and DirB, while perhaps reasonable for a mesh architecture, require duplicate messages being sent often up very similar paths in a tree network. Unfortunately, the closer to the root the longer the wire length, and the greater the probability that two messages will share the link. Since the longer links require more energy to transmit over, the most common links are the ones that require the most energy . In this model, the downside for Dir[N]B is that there can be a huge amount of wasted energy duplicating messages over these links, while multicast messages can share the same link, at the expense of more target address bits being transmitted, or more cores finally receiving the message than necessary.

However, this model makes the false assumption that each transfer will take the same energy to change the wire state of the link (which might be true if the packets were very different, or the link was a serial interface such as transmission line), but for a regular parallel wire link the Dir[N]B messages will change only a few of the target address bits, leaving the message payload identical. This means that unicast messages in fact do not waste significantly more wire energy, as might be naively expected, they simply require more routing energy near the root, and compared to lossy multicast

messages they require less routing energy near the leaves of the tree. Where multiple unicast messages perform particularly badly is the time taken to send the messages. A tree based design is not suited for unicasting to a large number of sharers because only one message can be sent from the root per cycle, where a multicast requires only a single cycle of time at the root to sent the message to all targets. Also, from an energy perspective it may be preferable to incur greater power usage near the leaves in routing unnecessary packets, than route extra packets near the root, as this would spread the thermal load more evenly throughout the chip.

From this realisation that the message payload to all sharers will require the same energy, it is clear that the energy difference between protocols comes from the number of bits needed to make routing decisions, and the number of false paths taken because of a lossy encoding scheme. Dir[N]B results in no false paths, but requires $\log_2(n)$ bits of routing address at the root, reducing by one bit every level up to a single bit for the final switch, and effectively an extra bit worth of energy for every routing bit which differs between subsequent sharers. The naive tristate option is similar to the unicast energy, except requiring twice the number of routing bits, and reducing by two bits at a time ascending the tree; however the energy is increased further by involving false paths where sharer information has been lost in the encoding. The Conservative Tree Encoding proposed in Chapter 7 has an identical routing bit behaviour to the tristate routing, but can be configured to trade-off extra routing bits, for less false paths, and can truncate the routing bits arbitrarily at different levels depending on the desired behaviour (over provisioned for the lower levels to support very random sharer distributions for example).

The binary tree based encodings which use symmetric nodes require $\log_2(n) + \log_2(\log_2(n))$ bits of information to be sent to the first common node between the symmetric node and the home node; essentially through the whole height of the network. A root based binary tree scheme would send $\log_2(n) + \log_2(\log_2(n))$ bits from the root, reducing by one bit per level until it reaches the root of the broadcast subtree, at which point only a single bit or two must be sent to indicate the presence of a valid broadcast packet.

The full and coarse vector are very different in that they require either n or k bits at the root, but the number of bits halves at every level until there are no more bits remaining. For full vector this is the leaf of the tree, the core, but for coarse vector this is some point lower in the tree, from which a broadcast is performed with no routing information. Which of these options requires the least energy requires detailed energy

simulations running benchmark workloads which are representative of those going to be run on the processor, because different sharer patterns can drastically affect the number of false sharers produced by a lossy encoding, and as such the trade-off point between the different multicasting encodings.

3.1.3 Efficient Coherence Messaging Support

Optical interconnects are becoming popular for proposed interconnect optimisations, for instance ATAC [57] and C^3 [65], and there are other architectural proposals to reduce the latency of some operations such as transmission-line rings [66]. However, none of these make any contribution to the efficient multicast of invalidations beyond that already possible with a full sharer vector or coarse vector, which has been shown to be a significant problem for larger scale multicore processors by Jerger *et al.* [39] in their work to introduce multicast support to mesh routers. While these are very suitable options for smaller multicores, there is room for improvement in the manycore scale era, especially when exotic optical interconnects are not an option.

The ATAC [57] design, discussed in more detail in Section 3.2 uses a similar mesh and tree design to that proposed later in Chapter 6, but with a full mesh connecting all cores, and a broadcast tree from the centre of each of the 16 tiles (up to 64 cores per tile). Tiles themselves are connected with an optical broadcast ring bus. This design requires a sophisticated optical interconnect, and with the directory partitioned into N slices (one slice per core), the directories themselves are not very area efficient (RAM control logic overhead is relatively large with small RAMS) and sharer invalidations require acknowledgements. This is the traditional distributed directory approach, which trades off the increased bandwidth of a distributed directory, with the increased bandwidth and energy requirements, along with greater latency on S→I/E transitions. Jungju Oh *et al.* [66] demonstrate that an electrical transmission-line based ring can provide a similar low latency network around the chip which could enable an ATAC like architecture without the optical interconnect. These architectures as proposed however only support unicast or full broadcast operations, while using a few extra transmission bits to filter the broadcast into a multicast could potentially save much energy in the traditional bus based components of the architectures, and in the coherency controllers in the L1 caches which must handle the broadcast operations.

Another important consideration is physical proximity between cores which share data. One of the advantages of snooping protocols is that it is very simple for cores

to forward data, without involving a third device, such as a shared cache or directory. Directory architectures can still support data forwarding, especially in architectures with good core-to-core communication such as ring bus or mesh based topologies, but this still requires a rather indirect series of communications with the directory, which may not be near to the cores with the data, even if the data does not have to travel very far. It is still worth performing this data forwarding, as data packets are much larger and require more bandwidth and energy to transfer than coherency messages, but it would be better if the coherence protocol could operate locally.

One method of reducing the latency and energy of accessing the directory, is to use a topology which allows effective directory caching, usually a hierarchical topology such as a tree. In these cases the coherence requests can often be resolved locally, and the data forwarded if possible from a nearby cache. The coherency protocol presented later in this thesis is very amenable to this hierarchical directory cache optimisation, and only neglects it in order to reduce development time.

An alternative approach which can reduce traffic to the directory by keeping coherency requests local, is the Proximity Coherence protocol [52]. This allows cores to snoop neighbouring cores' caches to gain access to cache lines which they posses. This works by allowing each cache line in a L1 cache to track a number of neighbours, which can then receive a copy of the line in their cache without sending the request to the directory. Exclusive and modified lines can be forwarded with these raised privileges, but shared cache lines may only be forwarded in the shared state. When a invalidation arrives for a forwarded cache line it must forward the invalidation, and wait for response from all sharers before responding to the coherency request form the directory. If the owner core wishes to evict the cache line due to self-invalidation it must inform the directory of the change in sharers, and await the directory's response before completing the cache line invalidation. Proximity Coherence is in some ways a hybrid of SCD and a MESI based directory protocol, with a directory maintaining the root coherency state, but SCD style sharer chaining allowed between adjacent cores. Unfortunately, due to the potential chain of sharers, Proximity Coherence based coherency requires an acknowledgement from the cores upon invalidation from the shared state. This means that inexact sharer state encodings must track the number of sharers, like AckWise, and receive unicast responses; it also makes it initially incompatible with the coherency protocol proposed in Chapter 6 of this thesis.

3.1.4 Tackling the associativity problem

Sharer state is not the only way to shrink directory storage requirements, there are other orthogonal directions to save space and energy. Reducing the number of entries in the on-chip directory will reduce the size and energy of the RAM required to implement it, while less associative structures will have less control logic overhead and result in less time or energy taken to locate an entry. Unfortunately simply reducing the capacity or associativity alone results in a huge drop off in performance.

3.1.4.1 WayPoint

One solution is that proposed by Kelm *et al.*, called WayPoint [34], which reduces the associativity of the directory to a fraction of that required to fully represent all of the cores in the system, and instead overflows the set into a hardware managed in-memory structure. The WayPoint paper makes an in-depth analysis of the scalability of other coherence mechanisms, like probe-filtering – which instead of evicting when directory capacity is exceeded, sends a broadcast to reconstruct sharer state when a new or previously evicted line is requested, and determines that even with the aid of assisting hardware, this does not scale up to 1024 cores.

WayPoint operates by using a in-memory linked list structure, which stores any directory cache entries which will not fit due to associativity or capacity constraints. This structure uses the existing processor cache hierarchy to access the data, which allows it to keep good performance, without adding the extra complexity of another cache or memory controller. WayPoint is assessed using Dir_4B encoding for each node, but the WayPint mechanism is compatible with any directory based sharer representation. When using WayPoint to provide the extra directory capacity and associativity to enable a full map coverage, the performance with smaller directories is comparable to that of directories with $4\times$ to $16\times$ the capacity. When comparing associativity sensitivity WayPoint can enable performance close to a full map directory with as few as 4-way associativity in the directory, often out-performing even 64-way associativity for a directory without WayPoint, which must evict sharers on capacity and conflict misses.

WayPoint and other compressed sharer encoding schemes are not mutually exclusive, but complement each other. For example an ideal system could use WayPoint to enable sufficient associativity and effective capacity of the directory, while a compressed encoding reduces the space required per-line, and reduces the energy and traf-

fic required on multicast. However, for many workloads it is possible to reduce associativity requirements through filtering of the memory regions which actually require coherence, as shown is Chapter 6.

3.1.4.2 ZCache and Cuckoo Caches

An alternative approach to tackling the associativity problem is to increase the effective associativity by reducing aliasing, through the use of different hash functions to index different sets. Two examples of these structures are the Cuckoo directory [36] and ZCaches [35], which can help prevent evictions by increasing effective associativity without the overhead of a traditional high-associativity cache structure. These are again orthogonal solutions akin to WayPoint, and could be used as the foundation for the directory storage. Sanchez *et al.* make a good case for ZCaches in their Scalable Coherence Directory paper [37]. Both the Cuckoo Directory and SCD papers suggest that the best way to save energy is to reduce the associativity of the directory cache, but to over provision the number of entries by about 25%, allowing the effective associativity introduced by mixed hashing schemes to function effectively. This helps compact the directory because fewer larger sets are more silicon efficient than multiple smaller sets.

The SCD paper also mentions hierarchical directories, suggesting that they can overly complicate the coherence protocol, but under constrained architectures such as if implemented in a physically ordered tree, a hierarchical directory protocol need not add significant complexity. It is possible that a hierarchical directory cache could take on the roll of caching the second level bit-vector for lines which require multiple entries in the SCD hierarchical-bit-vector representation, with an eviction taking the roles previously discussed of eviction or resorting to broadcast on that sub-tree. This would also enable a single cycle multicast form the root, as the second level bit-vector is held at the root of the sub-tree and does not need sending from the root itself.

3.1.5 Software assisted solutions

Of course using software assisted approaches to reduce the coherency demands on the system is likely to reduce not just the bandwidth requirements, but also the storage requirements for limited directory systems. Cuesta *et al.* [67] use a page based coherence state tracking mechanism implemented in the TLB to reduce the demand on their directory based coherency protocol, and in doing so show that directory size can be

reduced as much as 16 times while still providing the same performance, for systems up to 64 cores. Their work fails to analyse the effect on the associativity requirements however, which this thesis addresses by performing a similar exercise on 32 to 1024-core systems with an without a very similar coherency filtering scheme, the analysis of which can be found in Chapter 6.

Taking this idea to its limit results in a system with no hardware coherency, such as that proposed by Fensch and Cintra [41]. This architecture operates by prohibiting a memory page to exist in a modified state in any two caches simultaneously, but requires the programs be written using release consistency, a much more relaxed memory model than that which many programs are written for.

The best solution is probably a balance of all of these techniques. By using page based coherence filtering with a multi-hash directory and an overflow scheme such as WayPoint, the directory could be significantly under provisioned, while still maintaining the same performance as SCD or Cuckoo Directories alone.

3.2 Multicore Architectures

There have been a a number of commercially produced, and even more academically proposed, manycore architectures recently. This section discusses the various features of those most relevant.

3.2.1 Intel Research and Products

Intel has explored three main experiments into the manycore architecture space with their experimental Teraflops chip, Single-chip Cloud Computer (SCC) research platform, and then their commercial Xeon Phi architectures. Although both SCC and Xeon Phi execute the x86 ISA, the architectural design and programming model for each is very different.

3.2.1.1 Teraflops Chip

The original Teraflops experiment was implemented in a 65nm process, and used 80 specialised VLIW floating point cores in a 10×8 mesh running at 5GHz [68; 7], with 3D stacked memory. Using 100,000,000 transistors and a 275mm^2 die, each of these specialised cores occupied 3mm^2 while providing two single precision floating point

engines. Part of the sacrifice made to provide this density was to use 2KB data memories and 3KB instruction memories for each of the cores, relying on the 2.56Tb/s mesh bisection bandwidth and stacked memory interface to provide the data, along with careful programming; the processor also lacked cache coherency. One of the main discoveries when testing the processor was that adding more cores to some problems did not improve performance, and instead hampered it, with the extra cores interfering with memory traffic more than they accelerated the computation [69]; this problem was realised again in the work of Chapter 6. The chip realises its design goal of 1 TeraFLOPS single precision performance, at only 62 Watts running at 3.16GHz, giving credence to the idea that multiple simple cores can be more energy efficient than fewer larger cores. Scaling to 5.7GHz, for 1.81 TeraFLOPS of performance, required increasing the voltage from 0.9V to 1.35V, and increased the power consumption from 62W to 265W; a reduction in energy efficiency of 58%. This experiment confirmed that more cores are better than increasing voltage and frequency, from a power efficiency perspective, so long as the program itself scales well.

3.2.1.2 Single-chip Cloud Computer (SCC)

Based around the simple P54C Pentium architecture in Intel's 45nm process, the SCC provided 48 simple cores each operating within its own discrete memory space and communicating via message passing. The SCC architecture was based around tiles of two cores each, arranged to form a 4×6 2D mesh, providing a core-to-core message passing interface and a path to the memory controllers, but no coherence mechanism between the cores. Intel's SCC appeared to the programmer as a cluster of 48 isolated systems, each capable of running a lightweight operating system or runtime, rather than a traditional CMP system. This configuration was familiar to developers used to writing MPI programs for larger scale compute clusters, but very different to those targeting accelerator architectures such as those based on GPUs. The SSC was developed as a research platform, and is not a commercial design that was taken further and productised.

3.2.1.3 Xeon Phi

Intel took a very different approach to designing their Xeon Phi accelerators (developed from the failed Larabee graphics accelerator), using a design much closer to their traditional CMP architecture. Returning to a single unified coherent memory space

the first generation Knights Ferry (KF) architecture was still based around the P54C architecture in 45nm, but with 64-bit, four-way SMT, and 512-bit wide vector support for high throughput floating point processing. This KF architecture was provided with only 32 cores and is estimated to require a die of approximately 700mm^2. The Xeon Phi architecture is based around a bi-directional ring, not unlike the Cell Broad Band Engine, with each direction providing a high bandwidth data ring, and a narrower set of address and coherence rings. Original designs for the Xeon Phi provided only a single data, address and coherence ring per direction, but simulations suggested that this would not scale beyond 32 cores as the address and coherence networks were saturated; to alleviate this the production Xeon Phi silicon provides two sets of address and coherence rings per direction, with a single 64-byte wide data ring. Coherency is provided through a distributed directory along with each core's L2 cache, and provides consistency conforming to the standard x86 memory model (TSO), making the Xeon Phi as simple to program for as any standard x86 desktop processor. Even the required performance optimisations are similar to those required to write scalable applications on large scale multi-socket x86 systems.

Moving to 22nm the first production generation of Xeon Phi, Knights Corner (KC), uses the same modified P54C core, but increases the number of cores, enabling to up to 61-core designs for 244 threads and up to 1.2 TeraFLOPS of double precision compute performance. This design has been used as part of the Tianhe-2 supercomputer to achieve 33.86 PetaFLOPS, taking the number one spot in the Top 500 supercomputer rankings.

The current generation Xeon Phi, Knights Landing (KL) moves away from the P54C architecture, moving to a more modern Airmont Atom core design. This new core provides out-of-order execution, as opposed to the P54C's in-order pipeline, while maintaining 4-way SMT. Despite the more sophisticated microarchitecture the 14nm process allows KL to pack up to 72 cores into each processor. One of the benefits of the new microarchitecture is that Xeon Phi now supports the full modern x86-64 ISA, while the modified P54C only supported some of the 64-bit extensions and did not support the vector instructions used in Intel's traditional x86 line.

Intel's provision of a full 64-byte wide data network between not only the 72 cores, but also the memory controllers, demonstrates clearly that a number of extremely wide wire links are possible on such large architectures, motivating the use of 32-byte wide mesh links in Chapter 6 of this thesis.

3.2.2 Tilera TilePro/TileGX

Tilera is one of the few companies producing industrial manycore processors, used primarily for high bandwidth network applications or multimedia transcoding. Their original 64-core Tile64 processor is based on 32-bit, RISC, 3-way VLIW core in a 8×8 2d mesh, with hardware managed caches providing 8 KB instruction and 8 KB data L1 cache, and 64 KB unified L2 caches local to each tile [70; 71]. The Tile64 also provides "neighborhood caching", which allows any core to access the L2 on other tiles, acting as a large shared L3 cache. Each core has full virtual memory support, providing a 32-bit virtual memory space for applications on a 64-bit (36-bits implemented) physical address space, and the Tile64 can run SMP Linux. Tilera's largest chip provides 72 64-bit processor cores, operating at up to 1.2GHz in a 2D mesh architecture. The TileGX processors uses a non-blocking single-cycle "Terabit" mesh router architecture to provide their very high bandwidth. This mesh is split into 5 dedicated channels and provides full cache coherency (although the memory consistency model is unspecified) across the processor [72]. The Tilera processors embody the SoC design philosophy, providing not just four memory controllers and 26 PCI-express lanes, but eight 10G network ports, for 80GB/s of networking bandwidth. Tilera's success demonstrates that there are market niches where highly parallel systems, using full general purpose processing cores, provide the best balance of energy, compute, latency, and bandwidth.

3.2.3 Rigel

The Rigel [73] architecture was originally conceived as a 1024-core accelerator architecture, designed to run programs written and compiled specifically with their programming interface, in a similar fashion to programming for graphics cards as accelerators. As such the architecture initially provided no cache coherency, instead using the lower global cache level as the only cache for accesses to memory regions which required coherency. Unlike graphics cards however, Rigel provides full Multiple Instruction Multiple Data parallelism, like a traditional CMP architecture, with each core being a full 2-issue in-order RISC pipeline with a single precision floating point unit. The hybrid hierarchical-tiled architecture of the Rigel processor shows that some consideration has been put into the design, which uses clusters of eight cores with private instruction caches and a single shared data cache which are in turn connected by a bi-directional binary tree with another 15 clusters to form a tile of 16 clusters. Eight of these tiles of 16 clusters are then connected to the global cache banks and memory

controllers through a multi-stage crossbar for a total of 1024 cores.

The Rigel accelerator is designed to be programmed with their Low-Level Programming Interface (LPI) which exposes a task based parallel programming model utilising software based relaxed consistency cache coherency for some memory regions, and enforcing stronger coherency for communication structures by preventing them from being locally cached. This model works well for many benchmarks which have already been designed to run on accelerator architectures, leveraging the compute and memory bandwidth without the impact of managing strict memory consistency between the 1024 cores.

Later work on the project adds cache coherency to the Rigel architecture, investigating the possibilities of probe filtering and directory caching as scalable solutions. Probe filtering works much like a directory cache, but capacity evictions are not removed from the core caches, and instead a directory cache miss requires a full chip broadcast probe to reconstruct the sharer state. Despite adding hardware support to accelerate the probe broadcast and collection phases this option does not provide sufficient scalability in the Rigel architecture. Because capacity misses for eviction based directory caches also presents a performance problem (due to the required associativity for such a large manycore system) they developed the WayPoint [34] coherency scheme discussed in more detail in Section 3.1.4.1.

3.2.4 ATAC 1000

Proposed by Kurian *et al.*, the ATAC [57] architecture is a scalable tile based architecture designed to scale from 64 to 1024 cores, based largely on the traditional distributed directory structure on a 2D mesh. What ATAC contributes to the field is an on-chip optical broadcast ring, which is used in ATAC to send broadcast coherence invalidations and long distance point to point messages. The ATAC architecture is divided into 64 tiles, in a 8×8 mesh, with each tile's hub containing an optical transceiver to inject messages into the ring, and read messages out of it. The ring is finely tuned so that each receiver coupling removes a fraction of the optical signal which is small enough to allow the other 63 hubs to receive it, but large enough such that after the 63rd the signal is below the detection threshold. Due to the nature of light it is possible to transmit multiple wavelengths of light simultaneously down the same wave-guide, which ATAC uses to eliminate contention on the optical broadcast ring by tuning the wavelengths of the light sources for each hub to be different. The wavelengths must

still be very close together however, because transmission through wave-guides is sensitive to the wavelength of the electromagnetic wave, and different wavelengths will propagate at slightly different efficiencies, and speeds, known as dispersion. The selected wavelengths must also maintain sufficient frequency separation that the modulation frequency does not cause the modulated signals to overlap in the frequency domain, resulting in interference. By using multiple wave-guides each transmission on the optical bus can contain a number of bits, just like an electrical bus; ATAC uses a 64-wave-guide ring to provide 64-bits per message on the optical network. By using the same frequency on all wave-guides, a message sent from a hub can avoid parallel bus related skew that could otherwise be introduced by dispersion, assuming that each of the 64 wave-guides is manufactured within suitable tolerances, and is length matched throughout the chip. This low contention network requires that each receiver be capable of filtering out 63×64 frequency–wave-guide combinations for a 64-hub, 64-bit wide optical network, with each filter loop in every receiver requiring precision manufacturing. One feature of the optical network which is not described is how flow control and rate limiting is handled. Because the bus itself is not arbitrated, each receiver may receive up to 64 messages per cycle to process, and while there are FIFO buffers to cover momentary bursts of traffic, eventually the FIFOs will become full and no more traffic may be sent. Of course it is posible to utilize the optical bus itself to send messages indicating the full status of each FIFO to the associated sharer, with each transmitter keeping 63 bits of state to store the corresponding input FIFO state for each destination, but the authors do not explain if this is the scheme they use, or how much impact this has on the bus bandwidth and latency.

Another potential shortcoming of the ATAC architecture is the lack of support for packet combining of broadcast responses, as in the Rigel architecture, so responses must be processed serially. Although very few broadcasts require a full response, for a 1024-core architecture those few instances, such as when a barrier variable is cleared, require a single directory node to process 1023 unicast point to point response messages, with a mixture of electrical mesh and optical network paths.

One of the largest shortcomings of the ATAC architecture is the distributed directory model that ATAC uses, which is highly inefficient and likely would not scale to the 1024 cores they simulate. This would be due to the high associativity required for good performance, and the small size of each of the directory slices. By partitioning the directory into 1024 slices using address partitioning, each slice must have sufficient associativity to achieve the full performance potential of the processor, which for

simple schemes is equal to the associativity of the L1 cache level – at least 1024 ways for the ATAC 1000. With each of the 1024 directory slices being capable of caching 1024 cache lines, this totals 1M cache lines, or 32MB of cache with 32 byte cache lines, which matches the 32KB of private L2 the authors provide for each core. To summarize, the ATAC 1000 is provided with 1024, 1024-way fully associative directory caches, which is a huge waste of area and will consume an unreasonable cost in energy. By using serial walk-through of the directory tags the physical associativity can be reduced in trade-off for latency, while maintaining the same logical associativity, with the additional benefit that on average the energy of a lookup will be reduced by about half, since the average linear search will need to scan through half of the tags; the fewer physical ways the closer the energy saving approaches 50% and the closer the average latency comes to 500 cycles. Even in the optimal case, reading on average 512 tags requires far too much energy for a power efficient design, so this is clearly an area where the ATAC design is not fully thought out.

Finally, Kurian *et al.* do not evaluate the scalability of the ATAC architecture, presenting only benchmark results which are compared relatively within either the 64-core or 1024-core design configurations, and not comparable between the two. The fact that when scaling from 64 to 1024 cores the electrical mesh bus width was increased from 64 to 256 bits wide suggests that architecture did not scale well with the simple linear increase in on-chip bandwidth, and a quadrupling of bandwidth and quartering of large packet latency was required to combat the huge increase in point to point latency and traffic requirements of the larger design.

3.2.5 Cyclops-64

Part of the IBM Blue-Gene project, one of the earlier research manycore processors was the the Cyclops-64 [43].This processor is comprised of either 80 or 160 cores depending on one's perspective, with each of the 80 cores providing two thread execution engines based on a subset of the IBM Power ISA, but sharing a single 64-bit floating point engine. Each of the 80 cores also contained two 32KB SRAM scratchpad memories, which could be partitioned to provide a contiguous address space of NUMA global memory, although the private scratchpad partition could still be accessed by all cores through a non-uniform address scheme. The architecture provides no data caches, and as such no coherency, instead relying on the programmer to manage memory allocation and duplication between the private scratchpads/NUMA global

memory. Instruction caches were provided, but only 16 32KB I-caches were provided for the whole processor, with five cores sharing each.

The most unusual component of the Cyclops-64, at least compared to other many-core processors, was the use of a massive 96×96 7-stage, non-blocking, crossbar to connect the cores to each other (to access the distributed SRAMs) and the memory controllers and other off chip connections. Surprisingly the crossbar only occupies 6% of the 462mm^2 chip, while providing more than sufficient bandwidth between the cores and memory elements.

3.2.6 Godson-T

The Godson-T is a homogeneous 64-core processor, arranged in an 8×8 grid, to form a mesh network where each core is a 1GHz RISC core based on the MIPS ISA. In their paper [74] Wang *et al.* investigate the memory hierarchy of the processor. The Godson-T has 16 L2 cache banks (four on each edge) with four memory controllers (the four L2 cache banks from each edge are connected to a single memory controller via a crossbar), and conclude that on chip L2 cache can satisfy the memory requirements of the processor, and that almost all of the on chip traffic is generated by memory traffic. This means that the processor designer has to be careful to allocate enough bandwidth so that critical links, such as those near L2 cache banks, do not become a bottleneck, in order to fully realise the bandwidth of the off chip connections.

Strangely Tan *et al.* give a contradictory account of the architecture of the Godson-T [75], to that of Fan *et al.* [76] and Wang *et al.* [74], describing the architecture as a hierarchy consisting of a 5×5 mesh of tiles, with all but the centre tile consisting of a 4-core processing node. This gives a total of 24 tiles of 4 cores per tile, or 96 cores. This hierarchy is completely different to the 8×8 mesh described previously. Within each tile the 4 cores share a write through L1 cache and are connected by a full 7×7 crossbar. The L2 cache is global to the whole chip and fronts the memory controllers.

3.2.7 Cicso Metro, a.k.a. Silicon Packet Processor

Cisco ships managed switches which use 188-core Tensilica processors as network processors [77; 70; 4]. At 130nm the processor die is 18x18mm, with the cores being 0.5mm^2 and 30% of the chip is taken up by DRAM and extensions. Four spare processors allow for defects, so this is technically a 192 core processor with 4 disabled for yields. The total area per core is a little under 1.7mm^2 and draws 35W at 250MHz.

3.3 Simulators

This section discusses some of the more relevant simulators and how they relate to the work on cycle accurate multicore simulation in this thesis.

3.3.1 Single Threaded Simulators

Many of the simulators used commercially and academically for cycle accurate or approximate simulation are designs around single threaded discrete event simulation, with SystemC based simulation, Simics, and Gem5 based simulations being the most popular.

3.3.1.1 SystemC

One of the biggest industry tools for device, processor, and full platform simulation is the SystemC modelling language, based around heavily templated C++ with a main simulation kernel handling the event loop, and extensive signalling and transaction support. SystemC can be used to write cycle by cycle models, through loosely timed transactional models, to purely functional simulation. The most relaxed timed model is a version of Transaction Level Modelling which supports timed transactions (TLM 2.0), and uses a feature called the Quantum Keeper to track timing offsets between simulation elements, using timing annotations in the transaction objects. This simulation is still all run in a single thread running the SystemC scheduler however. SystemC is a powerful simulation tool because of its flexibility and the large library of models already available, but its performance in cycle accurate and even functional simulation is poor compared to other dedicated simulators.

3.3.1.2 Simics

Simics [78], is an emulation based functional full system simulator supporting multiple ISAs, originally developed in academia but spun out commercially under Virtuatec and now owned by Intel. Simics has been used to provide the core functional simulation to a number of cycle accurate simulators, and provides some powerful features for application development and debugging on virtual prototype hardware, competing with TLM based SystemC models in industry. The lack of cycle accurate support allows it to be fast, but makes it inappropriate for use in design space exploration or performance profiling, and more suitable to application development ahead of hardware availability.

3.3.1.3 Gems

Gems [79] provides cycle accurate full system simulation by leveraging Simics as a functional emulator for the target SPARC ISA, and driving it from its own timing models. Gems can use the Opal out of order processor timing model and Ruby interconnect models, but development has ceased in favour of its derivative Gem5, based on the M5 simulator in place of Simics, allowing the tool to be fully open source. Because Gems drives Simics a single instruction at a time, and runs in a single thread, performance is relatively poor.

3.3.1.4 SimFlex

Like Gems, SimFlex [80] uses Simics as its functional emulation core, but uses statistical sampling of the application to avoid the overhead of simulating each instruction one by one with a timing model. By using statistical sampling SimFlex can achieve good performance, but accuracy suffers because it does not observe the entire application behaviour.

3.3.1.5 Gem5

GEM5 [81] is currently one of the most popular tools for SoC and MPSoC simulation and design-space exploration. Based originally on the Gems and M5 simulators it uses emulation based simulation to provide simulation models for multiple ISAs, while also supporting a variety of micro-architectural models for each core type, scaling from a simple atomic model of one cycle per instruction, to full out of order superscalar models. The interconnect is also modelled with a similar degree of configurability, with the most detailed interconnect simulations provided by the Ruby Garnet models, while simpler and faster models are supported with the Ruby Simple interconnect models and GEM5 Classic shared bus based models. GEM5's accuracy has been evaluated against a dual-core reference platform [82] demonstrating up to 35% error, but across Splash2 and ALPBench it achieved an RMS error of 8.8%. Unfortunately GEM5 suffers from relatively poor simulation speed; running on the same University of Edinburgh compute cluster as used in later chapters of this thesis, the ECDF [83], it provides simulation speeds of 0.06 MIPS to 0.275 MIPS with an average speed of 0.157 MIPS when simulating 4- to 16-core shared memory systems with the simple Atomic core model and Ruby interconnect model. While the Garnet interconnect models may provide detailed router microarchitecture models, the topology options are limited to a set of

regular structures such as mesh and torus, inhibiting experimentation with more exotic or novel NoC topologies. Additionally its timing/control-flow only based modelling means that effective wire level power modelling cannot be performed accurately. It also means that relaxed memory consistency models cannot be accurately modelled or verified because the absence of data caching means that host memory consistency and coherency is applied, regardless of that specified in the model's coherency protocol. Much of the Garnet model infrastructure is written in Python, which also puts the models at a performance disadvantage to those written carefully and compiled from pure C/C++.

3.3.1.6 ARMn

ARMn is a simulator designed to simulate multicore ARM systems [84], which connects multiple cycle accurate processor elements based on Simit-ARM together with a SystemC interconnect model. Out of the box it provides a simulation infrastructure with a configurable interconnect model enabling bus, mesh, and star based topologies with full cycle accurate simulation. Unfortunately it does so for a message passing API only, and does not model a shared memory system, requiring programs to be written against a special message passing library. This limits its usefulness in general purpose multicore simulation and design space exploration. The simulations speeds achieved are also not particularly fast, with reported speeds of under 7K cycles/s for a 16 node torus coupled to a synthetic traffic generator, although a simple four node bus can be simulated at a little over 400K cycles/s. As a result of the slow speed and inflexibility of simulation target of ARMn, the simulator is of little use outside of performing experiments on the specific architecture which it models, and is not useful for more general design space exploration.

SimpleScalar

SimpleScalar [85] is another historically popular simulator, capable of functional down to detailed cycle accurate simulation. SimpleScalar was parallelised by Zhong *et al.* [86], but SimpleScalar must run its own ISA, requiring its own compiler, and does not offer the simulation performance of more modern simulators.

3.3.2 Tightly Coupled Parallel Simulators

Multi-processor simulators have been parallelized to take advantage of modern multi-core systems, with the simulation divided between host cores, usually at the granularity of target cores. However, these simulations still synchronize after every cycle to maintain correctness; this introduces significant overhead, and stalls the whole simulation when the operating system context-switches one of the threads. To reduce this overhead simulators have increased the synchronization quantum beyond a single cycle, but this introduces inaccuracies in simulation. These systems also only typically model a shared bus or crossbar, and may not model the effect of shared bandwidth correctly if the quantum is larger than a single cycle.

3.3.2.1 GPGPU Simulation

One of the most interesting parallel simulators, is the CUDA based GPGPU simulation of ARM multicore processors by Pinto *et al.* [87]. This simulator uses a Nvidia GPU, running a Cuda kernel which implements the instruction set simulator, cache model, and simple interconnect model. Despite executing the simulation in lockstep, it provides simulation rates up to 1,800 MIPS for a 8192-core simulation. When adding a simple switch arbitration model the simulation rate droops to approximately 1 MIPS for 32-core simulation up to 50 MIPS for a 4096-core simulation, with single instruction synchronisation.

Although it currently only supports a subset of the ARM ISA, and functional simulation, a large NoC model is highly amenable to GPGPU acceleration. Communication bottlenecks are too high to currently run the core simulation on the CPU and interconnect on the GPU, communication latencies outweighing the time saved by executing it on the GPU, but moving the whole simulation onto GPU architectures could be a promising way forward for cycle accurate manycore simulators in the future. Alternatively future architectures with closer coupling between the CPU and GPU such as the Heterogeneous System Architecture (HSA) [88] could reduce the communication latency to the point where it becomes profitable.

3.3.3 Parallel Relaxed System Simulators

3.3.3.1 BigSim

BigSim [89] is a modern multiprocessor simulator designed to deliver scalability and performance estimates on current and future large scale super-computers. BigSim works by running message passing based applications developed with MPI or Charm++ on a system much smaller than the simulation target, but running as many threads as would be run on the real target, and capturing the messages passed through the API with timing information. The timing estimation for code sequences can be based on either user annotated timings for straight line code segments, scaled wall clock time of the time taken to execute the code segment on the simulation host, or a weighted heuristic based on hardware performance counters such as number of floating point instructions, branch instructions and memory instructions; BigSim does not currently support a cycle accurate simulation of the target platform processors. Timing estimates for the interconnect are based on the latency of a message through the network topology of the simulation target, under infinite bandwidth/zero congestion circumstances. The authors argue that this is accurate enough for applications with high compute to communication ratios, and for their presented benchmarks this does seem to be the case. However, a slight increase in network traffic can have significant impacts on the latency of communications as the network approaches saturation; in Chapter 6 Figure 6.13 shows how drastically the communication latency over a network can change under increasing traffic demands. While BigSim is obviously useful for supercomputer scale performance estimates of message passing applications, the lack of microarchitecture simulation, overly simple network model, and lack of shared memory program support make it unsuitable for manycore CMP simulation.

3.3.3.2 FastMP

FastMP [90] attempts to address the issue of simulation scalability for multicore platforms. Their platform combines checkpoint based sampled execution to minimise the simulation work required, and analyses the discrepancy between the CPI of each core and the average CPI across all simulated cores to try and address errors in simulation accuracy at runtime. Unfortunately FastMP can only simulate applications which do not share data between threads, largely missing the point of parallel system simulation.

3.3.3.3 Wisconsin Wind Tunnel

Wisconsin Wind Tunnel (WWT) [91] is one of the earliest parallel simulators, but unfortunately requires applications to use an explicit interface for shared memory. Its direct execution simulation method also limits it to running on CM-5 machines, making it impractical for modern usage. WWT II is the evolution of the first generation WWT, and Mukherjee *et al.* [92] transition the WWT II methodology to other host architectures. Unfortunately it still does not model anything other than the original target memory system, and requires applications to be modified to explicitly allocate shared memory blocks.

3.3.3.4 HORNET

HORNET [93] is a scalable cycle accurate simulator for on chip interconnects. The cores can be fed from simulation (such as the MIPS simulator that is integrated into HORNET), from previously generated traces, or from instrumented natively executing code, such as with PIN. HORNET is also integrated with a power model based on ORION 2.0 [94] and thermal model using HotSpot 5.0 [95]. This simulator is demonstrated to show good scalability up to 24 host cores, for 64- and 1024-core mesh architectures, but unfortunately does not give any absolute performance figures for comparison. The paper confirms that for accurate results the simulation feeding the interconnect model must be run with the interconnect simulation providing feedback, otherwise the interconnect injection rates could be much higher (since the cores do not wait the correct delay before the next request, as they assume an ideal interconnect), resulting in lower execution time. The paper also indicates that congestion modelling is only significant in bandwidth heavy applications, with benchmarks that do not stress the interconnect showing minimal error when congestion is not modelled. Benchmarks that do produce a lot of interconnect traffic exhibit significant error if the congestion is not modelled however.

HORNET does support a cycle-by-cycle execution mode, but has not been verified against a target platform. This means it is only useful for reasoning about abstract design-space exploration rather than evaluating the best MPSoC configuration for a given application.

3.3.3.5 Marss

Marss [96] is a "cycle-accurate" multicore x86 simulator based on QEMU [97] and PTLsim [96], designed to model modern superscalar x86 architectures. It supports relatively complex interconnect architectures, but is limited to x86 simulation, and is also non deterministic. Being limited to x86 means that it is not suitable for experimentation with modifications to the ISA, such as adding new instructions. It is also not one of the ISAs used by low power embedded cores likely to be found in a many-core processor design, although it is the ISA of choice for the Xeon-Phi accelerator. By being non-deterministic Marss demonstrates that it is not performing a true cycle-accurate simulation, and some event timings and relative orderings are being decided by host execution order, rather than strict timing model order. This makes performing experiments on features such as relaxed memory consistency models difficult, if not impossible, and also makes it extremely hard to debug subtle race conditions either in simulated hardware protocols, or in the application being simulated.

3.3.3.6 SlackSim

SlackSim [98] is one of the early simulators exploring the relationship between synchronisation granularity and accuracy. It is a parallelised cycle by cycle simulator which simulates cores with caches in different threads, and allows for configurable synchronisation slack between the components. The simplest form is to synchronise with a barrier every N cycles, with synchronisation every cycle providing full cycle accuracy, and increasing error as the synchronisation period, or quantum, is increased. SlackSim also introduces a different form of relaxation, where the difference in cycle time between the slowest thread and fastest thread is maintained within a defined slack. Similarly to relaxing the simulation period, relaxing the slack allows one to trade off performance for accuracy. Like many of the simulators here, the accuracy of SlackSim has never been verified against existing hardware. The performance of SlackSim is typically around 100 KIPS, while simulating a 4-way out-of-order processor, although this does not involve a detailed NoC simulation.

3.3.3.7 FaCSim

FacSim [48] is a single core simulator which decouples functional simulation from micro-architectural and memory hierarchy simulation, to enable high simulation speeds. Because there is only a single simulated processor there is no need to simulate mem-

ory contention or coherence traffic, and no potential for violation of memory orderings. This allows FaCSim to run an unbounded, fast, functional only, simulation to generate the program instruction stream and then feed it asynchronously to an optimised timing model of the processor and memory hierarchy. The models are connected through a shared memory circular buffer, making efficient use of a dual core host, and stall the functional simulation when the buffer becomes full. This simulation setup would also allow for the instruction stream to be stored to disk and run with multiple timing models offline, or streamed online to multiple timing models running in parallel. These features are only possible because the single core target system does not have any behavioural dependence on the timing of events in the model. For a multicore system the functional behaviour can be dependent on the timing because of the interaction between the simulated cores. It also means that the interconnect timing model can be extremely simple to calculate, as there is no opportunity for contention between multiple memory requests, which requires a more detailed simulation that a simple latency calculation based upon link distance and bandwidth. FacSim provides simulation rates up to 4 MIPS and has been shown to produce only a 6.8% root-mean-squared (RMS) timing error relative to reference hardware. This is not as fast as modern just in time compiled (JIT) simulators which compile the core timing model into the translated functional simulation, but is a good demonstration of the strengths of decoupling simulation components to improve performance through increased parallelism.

3.3.3.8 Graphite

Graphite [99][99] is a distributed multicore simulator that uses Pin [100] to instrument a natively executable application. The interconnect and distributed shared caches are simulated in parallel and can also be distributed across multiple nodes along with the functional execution. By instrumenting memory accesses with Pin, Graphite supports running a wide range of shared memory applications across multiple nodes, allowing the different timing and functional components of the simulation to run asynchronously within a configurable bounded slack. The three supplied methods of synchronisation are a lax barrier, which keeps all simulations components running close to lock-step, synchronising every N cycles (1,000 in their presented results), lax peer-to-peer synchronisation, which puts cores which are too far ahead of the slowest core to sleep briefly (presented results used 100,000 cycles) and lax synchronisation. The last method uses timestamped messages from the different components to estimate the global clock locally, and never suspends simulation components for the sake of timing

synchronisation. In all methods events are processed in the order they are received, and not reordered into correct timing order. As a result, although Graphite provides a scalable reasonably performant multicore simulation infrastructure, the accuracy is not sufficient for some more subtle micro-architecture experiments. For example an interconnect which arbitrates on a per-flit basis rather than a per-packet basis will result in interleaved flits from multiple packets, depending on the design of the memory controller this could be significantly worse, or better, for performance. By processing events in arrival order rather than timing order, this detail would be lost to a Graphite simulation.

3.3.3.9 Sniper

Sniper [101] is effectively Graphite, with the Pin based functional simulation replaced by an interval simulation technique. The authors claim that this should provide greater accuracy for complex architectures than the in-order model which Graphite models. Unfortunately their accuracy evaluation against a real world Intel Core-2 based system resulted in, on average, 25% error. This poor accuracy provides strong evidence that the cycle approximate techniques used by many of these multi-processor simulators is insufficiently detailed for micro-architectural experimentation, or for performance profiling of systems which are extremely timing sensitive, such as hard-realtime systems.

Sniper claims up to 2 MIPS performance when simulating a 16-core target on an 8-core host, approximately twice that of Graphite, although some of this may be due to differences in interconnect complexity modelled, and performance of the host machine.

3.3.3.10 ZSim

Another of the instrumentation based x86 multicore simulators, ZSim at first glance appears to be the holy grail of large scale manycore system simulation [102]. The paper presents a simulator apparently capable of simulating up to 1024 cores, with performance up to 1123 MIPS using a simple core model with interconnect contention, while demonstrating that relative to a 6-core system it can on average provide performance accuracy of 10% error. It achieves this by running a period simulation between 1K and 10K cycles where core models are processed for all simulated cores, generating memory events into a shared memory data structure, then running a model of the memory hierarchy to compute the timing effects, before returning for another period of core simulation. If your end goal is simply to measure final figure performance

on a given benchmark and simulated system, then this might well be sufficient, and certainly outperforms the other x86 instrumentation based simulators such as Sniper; however unlike full cycle accurate simulators it does not accurately simulate functional memory ordering within the 1K-10K cycle simulation phases, and it assumes that core-to-core interference events (such as coherence evictions) are rare. This assumption breaks down with low associativity shared resources, such as lower cache levels or directories, meaning the simulation will not be accurate to a designer trying to test the limits of their directory associativity or sizing options. Non cycle-accurate memory orderings can also severely distort spin-wait style timing statistics, such as the average time spent waiting on a spin-lock, or at a barrier; for example core *A* may be scheduled early in a simulation batch, and at the end of the period may acquire a lock variable. Core *B* is scheduled in a later batch, and immediately tries to acquire the lock, but fails and spends its entire scheduling period spinning (possibly up to 10K instructions), only to be scheduled before Core *A* on the next period, resulting in another 10K instructions of waiting. Core *A* finally releases the lock near the start of its simulation period, effectively holding the lock for only a few cycles, but resulting in core *B* spending almost 20K instructions more that the real system would have when waiting to acquire the lock. This can make efforts to optimise the memory hierarchy and interconnects difficult to measure, because the results are masked by simulator inaccuracies; it is even possible for a badly designed memory system to be masked by the host providing functional memory coherency, so protocol errors cannot be discovered.

3.3.3.11 HP COTSon

HP's COTSon simulator [103] uses AMD's SimNow^{TM} for functional modelling and suffers from some of the same problems as SimFlex [80] and Gems [79].

Monchiero *et al.* have presented a methodology to simulate shared-memory multiprocessors composed of hundreds of cores [49]. The basic idea is to use thread-level parallelism in the software system and translate it into core-level parallelism in the simulated world. The existing COTSon simulator is first augmented to identify and separate the instruction streams belonging to the different software threads. Then, the simulator dynamically maps each instruction flow to the corresponding core of the target multi-core architecture, taking into account the inherent thread synchronisation of the running applications. This approach treats the functional simulator as a monolithic block, thus requiring an intermediate step for de-interleaving instructions belonging to different application threads. ArcSim, as used in the remainder of this thesis, does

not require this costly preprocessing step. Its functional simulator explicitly maintains parallel threads for the CPUs of the target system. Monchiero reports this simulator performs at 1 MIPS for a single core simulation, scaling down to 0.7 MIPS for 1024 cores.

3.3.3.12 ArcSim

The simulator used in the rest of this thesis is based on the ArcSim simulator, developed at The University of Edinburgh. ArcSim is a high speed simulator designed to simulate the ARCompact RISC ISA for both functional simulation and cycle accurate simulation. ArcSim is compatible with the ARC600, ARC700 and ARCv2 ISAs, providing cycle accurate models for both 3, 5 and 7 stage interlocked in-order pipeline micro-architectures, with the 3 and 5 stage models developed around the synthesizable EnCore microprocessor, with both versions existing in silicon implementation.

ArcSim is a functional-first simulator, meaning that each instruction has its functional behaviour emulated in its entirety before the micro-architectural timing is calculated. This is relatively straight forward for an interlocked pipeline, and works as follows.

Each stage of the pipeline is represented by a 64-bit cycle counter, this counter represents the cycle at which the previous instruction left the pipeline stage (i.e. the first cycle at which a new instruction may enter the pipeline stage). When an instruction has been executed, first the branch predictor and instruction cache are queried to determine the earliest time at which the instruction could enter the pipeline, if this is greater than the current value stored in the first pipeline stage then it is updated to this new time, plus one cycle. The simulator is programmed with pipeline latencies for each stage, for each major class of operation, allowing it to now walk through the pipeline in this same manner; starting with each pipeline time being the greater of either the time the instruction left the previous stage, plus the pipeline delay of the current stage, or the existing recorded time in the pipeline stage plus the instruction's associated latency.

When coupling the pipeline model to the interconnect the cache-incoherent simulation in the next chapter takes the time for instruction-cache misses from the first pipeline stage, while data operations are taken from the memory stage. The coherent simulation in subsequent chapters simplifies the model by using the memory stage timing for both, because the small impact on modelling accuracy is less important due to the more abstract system being modelled, and it greatly simplifies development and testing of the simulator.

To achieve high simulation performance ArcSim uses an asynchronous dynamic binary translator (DBT), or just in time compiler (JIT), to accelerate simulation by translating target instruction sequences into native code, while also supporting the inclusion of code to model the core micro-architectural state changes for the instruction sequence. This can result in simulation speeds of up to 1000 MIPS, or 100 MIPS in cycle accurate mode. ArcSim supports full system simulation, along with syscall emulation to enable user-land application simulation, or a hybrid simulation mode where a bare metal environment can run in full system simulation mode with IO devices and full control of the MMU, but system calls are still trapped to the host to allow file access and console IO support.

Multicore simulation has only recently been developed in ArcSim, with the work presented in SAMOS 2011 [12] demonstrating functional simulation support for over 1024 cores at speeds in excess of 10,000MIPS, which this thesis extends through Chapters 4 and 5 to provide support for extremely accurate high speed cycle accurate simulation of multi-core and manycore processors and MPSoCs.

3.3.3.13 Parallel Embra

Like ArcSim, Parallel Embra [104] is a fast functional simulator for shared-memory multiprocessors which is part of the Parallel SimOS complete machine simulator [105]. It takes an aggressive approach to parallel simulation; while it runs at user level and does not make use of the MMU hardware, it combines binary translation with loose timing constraints and relies on the underlying shared memory system for event ordering, time synchronisation, and memory synchronisation. While Parallel Embra shares its use of binary translation with ArcSim it lacks its scalability and parallel JIT translation facility. Parallel Embra also provides no timing synchronisation or performance modelling, so is unsuitable for micro-architectural research.

3.3.3.14 Mambo and MalSim

Another effort to parallelise a complete machine software simulator was undertaken with Mambo [106]. It aims to produce a fast functional simulator by extending a binary translation based emulation mode; published results include a speedup of up to 3.8 for a 4-way parallel simulation.

Similarly, the MalSim [107] parallel functional simulator has only been evaluated for workloads comprising up to 16 threads. Despite some conceptual similarities with

these works, the work of this thesis aims at larger multi-core configurations with cycle accurate performance models.

3.3.4 Hardware Accelerated Simulation

The two main limitations to hardware accelerated platforms such as FAST, RAMP Gold, and ProtoFlex, are the cost of an FPGA platform to perform the simulation, and the limit to the number of simulatable cores imposed by the limited resources of the FPGA. This is unlike software simulation, where adding more cores may reduce performance, but the amount of RAM required to simulate even 1024 cores is well within the constraints of typical consumer hardware.

3.3.4.1 ProtoFlex

ProtoFlex [108] achieves reasonably high speed functional multicore simulation performing most of the simulation in FPGAs. Targeting the SPARC architecture ProtoFlex uses a pipelined core simulation engine called BlueSPARC in FPGA to simulate up to 16 instances of a SPARC processor model,with the aim to scale up in future by adding more engines. ProtoFlex supports full system simulation by falling back to a Simics based simulation when the FPGA model cannot simulate some part of the simulation, which can lead to a drop in performance below the theoretical 100 MIPS throughput of the BlueSPARC engine. This reduction in simulation efficiency can be reduced by using a on-FPGA embedded or soft processor to run the software fall-back model rather than require the high-latency communication to the host computer. The 16-core simulation achieves simulation rates up to 62 MIPS, which is now significantly slower than modern JIT compiled software simulators are capable of (typically several hundred MIPS per core, with results up to 1323 MIPS per core achieved by the current state of the art [109]).

Being already in-FPGA allows for the memory trace to be easily fed into FACS FPGA cache model, which is capable of modelling the CMP cache model at full speed, except for very memory active periods of simulation, and this allows for high speed "functional warming" of the cache state for a statistical sampling based cycle accurate model such as SimFlex, where ProtoFlex can run with functional cache model, then switch to cycle accurate software for brief phases to collect sample statistics which can be used to estimate full runtime performance.

3.3.4.2 RAMP Gold

There are several RAMP projects which use FPGAs in various ways for simulation, but the most recent and relevant is the RAMP Gold [110] project. This uses an FPGA to perform cycle accurate simulation of multicore and manycore CMP architectures, and is able to simulate up to 64 cores on a relatively cheap Xilinx Virtex-5 FPGA. RAMP Gold comprises two main in-FPGA components, a functional model of the cores, which time multiplexes the multiple instances of the simulated core sharing memory and cache resources between the models, and a separate timing model. The timing model performs the micro-architectural simulation for each of the simulated cores, and the timing model of the memory hierarchy (i.e. tags only, no data). The timing model drives the functional model's scheduler so that threads are scheduled in the correct order, but because data is only actually cached in a single cache shared by all models, RAMP Gold cannot simulate memory constancy models which violate sequential consistency, by virtue of being inherently a sequentially operating simulation sharing a single cache. As such, while an improvement over ProtoFlex in actually providing cycle accurate simulation, the circumstances in which it can be accurate is limited to a subset of the interesting memory consistency models and cache coherency protocols that are of interest to manycore chip designers.

In functional-only mode, RAMP Gold achieves a throughput of up to 100 MIPS, but only when the number of target cores can cover the functional pipeline depth. For fewer target cores (and non-synthetic workloads), the fraction of peak performance achieved is proportionally lower. In comparison to the peak performance of software-only simulation approaches (based on ISAs of comparable complexity and similar functional-only simulation) the performance of the FPGA architecture simulation is disappointing. However the close to 50 MIPS [110] performance for a 64-core, cache coherent, cycle accurate simulation is currently beyond the limits of software only simulation.

Other approaches to FPGA simulation such as RAMP [111] and ProtoFlex [108] suffer from the same performance issues and for none of the mentioned systems has scalability beyond 64 cores been demonstrated.

3.3.4.3 FAST

Another FPGA based simulation system, the FAST project [47; 112; 113], is one of the fastest cycle accurate simulators that can really claim to be cycle accurate. It does

this by performing functional simulation of each core in a high speed functional only simulator (QEMU [97]), and feeding the resulting instruction trace into a high speed timing model, implemented in a FPGA. The FPGA model simulates the pipeline for each core, along with the memory hierarchy, and contains the true memory state of the system, or Oracle Memory as the authors refer to it, from which the software models use cached regions to simulate ahead.

To enable full decoupling of the functional simulations and the timing model, checkpointing snapshots are used, with speculative run-ahead and roll-back, to ensure timing-accurate functional correctness from the functional simulation. These roll-back operations are triggered by events and checks in the timing model, which detect memory ordering violations by comparing the memory values in the instruction streams against the Oracle Memory when the timing simulation is performed..

Unfortunately, like the other FPGA based simulators, FAST has not been demonstrated above 64 cores, and total system simulation size is limited by the size of the available FPGA.

3.4 Machine Learning based Design Space Exploration

It was realised some time ago that new approaches to designing MPSoC systems were required, due to their increasing size and complexity. The paper by Flake [114] briefly discusses these issues.

The usual method of addressing these challenges has been to use high-level simulation to coarsely, but exhaustively, explore the MPSoC design space for a given application. The papers by Angiolini *et al.* [115] and Oliveira *et al.* [116] are examples of this approach. In these, the design space for a MPSoC application is simulated at a high and relatively inaccurate level, where simulation is fast. The information from such high-level explorations is then used to select large-scale system parameters, with detailed design and implementation left for later. In contrast, the approach used in Chapter 8 relies on slower, detailed, simulation of randomly distributed points in the design space, from which the performance at other points can be predicted. The paper by Gries [117] discusses and contrasts then-current methods for evaluating the MPSoC design space.

A machine learning approach has also been previously used to predict the runtime of programs, and then perform scheduling based on such information [118; 119]. These efforts have been focused more on online predictions for resolving the schedul-

ing problems, than on the system exploration it is used for in this thesis, and therefore does not address architectural differences in a quantifiable way.

Ipek *et al.* [120] have previously used Artificial Neural Networks (ANN) to predict the performance of simulated architectural systems. They use ANN methods due to their relative ease of use and the well-understood operations of these predictors rather than an evaluative approach comparing different predictors. They achieved prediction performance within 1-2% of the real values, but the evaluation used less complex system models than the approach taken in this thesis, leading to simpler design spaces.

The most relevant work in the area is the paper by Almer *et al.* [121] and his doctoral thesis [46], which uses machine learning to predict designs with optimal energy consumption, runtime, and EDP. The authors also try to predict whether a design will synthesise to their FPGA target, to eliminate areas of the design space which are not viable. As Chapter 8 was a collaborative work with Almer, a more detailed comparison with this work, and a break-down of the contributions, can be found there.

Chapter 4

Exploiting Cache Incoherence for Fast Parallel MPSoC Simulation

4.1 Introduction

When designing a system on chip (SoC) for any system it is important to evaluate performance characteristics, but when designing for a high volume deeply embedded system it can be especially important to minimise the area (and as such cost) of the silicon needed for the chip, along with the power requirements. This usually means tuning the performance to only just meet the worst case performance requirements, and no more.

In order to find this optimal configuration many iterations of software development and system configurations are required, typically using slow cycle by cycle simulators, perhaps with the use of faster functional only simulators to aid software development. Unfortunately evaluating the performance of new software, and finding the most cost effective system, requires a slow cycle accurate simulation of every likely SoC configuration, and using a typical single threaded cycle accurate simulation does not allow fast turnaround for a programmer doing performance optimisations.

Hardware prototypes implemented in field programmable gate arrays (FPGAs) are a great platform for running and evaluating performance of software on a prospective hardware design, but require significant upfront synthesis time to generate a new design configuration. This means they are more useful for evaluating different software options, on a fixed hardware platform, rather than rapidly testing multiple hardware options.

The size constraints of FPGAs also mean that larger multi-processor SoC (MPSoC)

designs are unlikely to be feasible, and must be tested in simulation. Expanding on these problems, FPGA development boards themselves are very expensive, so it is preferable for software developers to work in simulation, with the ubiquitous general-purpose computing power of compute clusters and server farms.

Many embedded systems use multiple processor cores without hardware cache-coherence, such as the SH-4 family SH7750 processor used in the SEGA Dreamcast [122]. In fact some of the popular embedded interconnect such as AMBA AXI do not support coherency, relying on the provision of atomic bus operations to synchronise communications between bus masters. Providing hardware coherence adds an unnecessary hardware development expense, in implementation and verification, and more significantly in silicon area and system power consumption; it also adds complications when reasoning about worst-case execution time and performance for hard real-time systems. As transistor counts increase multicore embedded systems are becoming more common, and so cache-incoherent MPSoCs are becoming an increasingly important simulation target; yet recent advances in simulation technology have largely been applied to cache-coherent targets.

Traditional cycle-accurate simulators take a cycle-by-cycle approach to the process, modelling the pipeline of each core and interconnecting buses in a single thread. This is the easiest way to ensure timing-accurate functional behaviour, deterministic simulation, and correct evaluation of memory interleavings between cores. Functional-first simulators perform a high-speed functional evaluation of behaviour and then reconstruct timing data using a timing model. This approach yields greater simulation speed and can be easily parallelized, but it is difficult to extend accurately to the case of cache-coherent MPSoCs, in which timing influences the behaviour of cores. The problem has been tackled before [113] but this implementation requires an FPGA to process the timing model, and significant communication between the timing model and functional simulation. However, in the cache-incoherent case the timing interactions between cores are limited to cache misses, cache flushes and cache-bypassing memory operations, making accurate parallel functional-first simulation possible while maintaining high simulation speeds.

This chapter presents a novel approach to simulating these embedded systems, where decoupling the simulation of the cores and interconnect is exploited while still maintaining cycle-accurate timing behaviour. This results in accurate performance modelling at simulation speeds up to 378 MIPS, or 117 MHz, with an average speed of 5.7 MIPS and 10.1 MHz across a large design-space of over 20204 design points.

The key contributions presented are:

- Leveraging incoherence to increase parallelism in the simulation.
- Efficient NoC simulation through packet tracking and cache friendly data structures.
- Faster than state-of-the-art simulation without sacrificing accuracy.

4.2 Target Platform

The target platform for this multicore simulator is a custom NoC based MPSoC flow, using the 3-stage variant of the EnCore RISC microprocessor. The tool flow takes a high level MPSoC description and generates Verilog for FPGA and silicon implementation, and equivalent configuration files for the simulator. The platform comprises clusters of 1-8 processor cores, connected through a per-cluster arbiter to an ARM AMBA AXI [123] based packet switched network, which connects cores to memory banks and devices, such as the UART, real-time clock, and display controller. The design options are summarised in Table 4.1 with an overview diagram in Figure 4.1. The complexity parameter influences the number of switches and number of layers in the switching network before adding cores and peripherals, extra switches are added if the configured complexity does not provide enough connectivity.

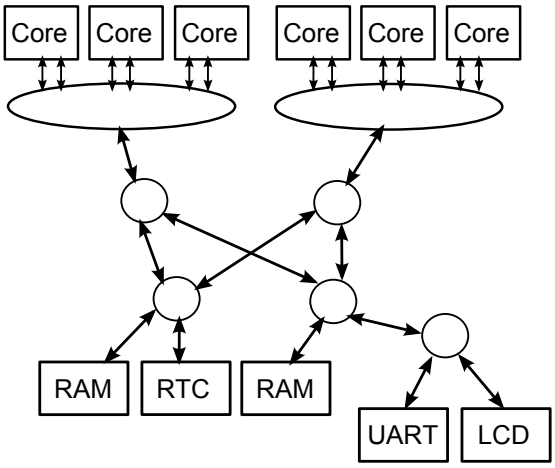


Figure 4.1: Overview of the target architecture.

The cores are a modern ultra-low-power, silicon optimised three-stage RISC architecture, implementing the ARCompact instruction set. Implementation in a Xilinx Virtex6 can be achieved at 75 MHz, with 65nm LP implementation exceeding 250 MHz.

Design Parameter	Possible Configurations
Core Architecture	ARC700 32-bit RISC
Pipeline	3-stage in-order
D-Cache Size	4 KB
D-Cache Associativity	Direct Mapped, 2-Way
I-Cache Size	4 KB
I-Cache Associativity	Direct Mapped, 2-Way
Cache line size	32 Bytes
Interconnect Protocol	AMBA AXI
Interconnect Topology	32-bit wide binary-routing network
Coherency Protocol	None – Cache-Incoherent
Cores per cluster	1 – 8
Clusters	1 – 8
Block RAMs	1, 2, 4, 8
Total Block RAM Size	512 KB for comparison with FPGA 2 MB for sim-only experiments
Complexity	1, 2, 4, 8, 16
Fifo Depth	2, 16
Core Freq(MHz)	12.5, 25, 50
NoC Freq(MHz)	12.5, 25, 50, 100

Table 4.1: MPSoC design configurations.

While it is possible to synthesize an FPGA design with 12 cores, as used for accuracy comparisons in Section 4.6, the University of Edinburgh has recently fabricated at chip at 65nm with 32 cores in a similar configuration, for use in the transceiver of a new wireless network technology [124].

The processor core is highly configurable, with a limited set of configurations used in this thesis. The selected configurations provide 32 32-bit registers, with 4 KB each of instruction and data caches which are configured as direct mapped or 2-way set associative for different experiments in this chapter. The pipeline is a latency and energy optimised 3-stage pipeline which approximately breaks down into fetch, execute, and memory & writeback. Branches have a fixed penalty, but this can be offset through its support for delay-slot instructions. Because of the short pipeline, many branch types can be evaluated in a single cycle of delay, which can be fully masked by scheduling an instruction into this delay slot (performed automatically by the compiler in many cases), although branches involving computation rather than status bits require a further cycle to evaluate, and under certain micro-architectural configurations an extra cycle further, which enables higher operating frequencies. ArcSim fully supports more complex branch prediction schemes in cycle accurate mode, but the short 3-stage pipeline does not warrant the area or energy expense of prediction logic, especially given its deeply embedded target domain. Further reducing the impact of branch overhead is the architecture support for "zero-overhead loops", which are take the form of a programmable hardware loop counter. The start, end, and count registers are programmed before entering the loop body, after which the loop is executed exactly the programmed number of times (unless the count register is modified inside the loop), with no branch delay penalties nor the overhead executing a branch instruction of during the execution. These are automatically generated for many loops, especially 'for' loops, by the compiler. Each of the two caches has its own bus master interface to perform fetch and write-back operations, but they are statically arbitrated to give the instruction cache priority. The bus protocol used by the bus masters is a single unified interface, so read request, write request and write data must be time multiplexed out, before they are converted onto the AXI interface. The write request will issue before the associated data, but read requests are statically arbitrated with lower priority than write requests and above data, so are inserted in a cycle between write requests and the trailing write data. Since the instruction cache has its own output port, it is statically arbitrated as higher priority than the data cache port, before being separated onto the discrete AXI channels.

The interconnect comprises five independent channels, implemented as in the AXI standard: three from master to slave for sending read requests, write requests, and write data, and two return channels for read data and write-complete acknowledgement. By using dedicated channels for each category it is simple to avoid deadlock and avoids issues of bandwidth sharing between the different channels.

Cores are connected to the interconnect through a cluster arbiter, which aggregates up to eight cores onto the interconnect fabric. The 8-way arbitration is implemented on a per channel basis (for the three outgoing channels) consisting of a three layer deep tree of two-way arbiters. Each arbiter contains a synchronous flip-flop to perform arbitration, which is toggled when the Ready line is asserted from the next arbiter, and a packet is ready to be sent from the previous. There are no other registers in the arbiters, just one final output register for the cluster, so the whole three level tree functions as a single cycle eight-way arbiter, with the correct arbiters being toggled due to the backwards propagation of the ready signal through the arbiter network. The arbiters can also be configured to toggle arbitration only on "Last" packets, for multicycle data write-backs. With this enabled, data will arrive at the memory controller in a contiguous series of packets, rather than interleaved with data from other cores. Depending on the implementation of the memory controller, the performance difference can be significant, and a simple memory controller may not support interleaved data packets. The cluster arbiter is fair for power-of-two cluster sizes, and fairness is still maintained under most other circumstances due to the time taken for responses from the memory controllers through the network. Because each core can only have two outstanding requests (one for each cache), and each requests takes several cycles, it's is unlikely that any processor core in a cluster would achieve a greater share of the available bandwidth. The binary-tree round-robin arbitration policy in the cluster prevents starvation and will provide all cores with at least the bandwidth share of $1/N$, where N is the smallest power of two equal to or greater than the number of cores attached to that cluster. Similarly the network arbitration policy will guarantee at least a $1/M$ share of the bandwidth to each cluster, where M is the number of 'master' ports on the same level of the interconnect as the clusters.

The rest of the switches in the system are two input two output, statically routed switches, implementing a binary routing network as proposed by Hopper and Wheeler [125] and described in the context of other networks and routers by Newman [45]. The routes for different memory regions are stored in a small lookup table at the core/cluster level, and encoded into the AXI slave address signals, while the slave to master routing

is performed by evaluating the route bit-string in reverse. Each output from the switch is essentially another two input arbiter, this time with the address bit check combined with the valid packet signal, and uses the same arbiter module as the cluster arbiter. Each switch can either be configured as buffered or combinatorial, so portions of the network propagate in a single cycle. The switches are generated in layers with the same number of switches, with the width of the layer determining the initial maximum number of slave or master ports, and the depth great enough that a packet can be routed between any master and any slave. If a generated interconnect cannot attach all masters or slaves, then extra switches are automatically added onto the master or slave port of the interconnect to grow the network. Devices on these extra branches will incur an extra hop of latency through the network.

Since there are five channels in the AXI standard implemented, even a simple two by two network (supporting up to four slaves, and four masters/clusters) requires 20 switches.

The aforementioned memory controllers are a multi-cycle pipelined bus slave serving access to a RAM comprised of FPGA block RAMs. The total available memory is divided equally between the number of controllers, and the RAM backed portion of the address space is mapped to them in a linear fashion, with each controller mapping a contiguous region. For designs where were evaluated against FGPA implementation the RAM was limited to 512 KB by the available bock RAMs, but for the larger design space experiment 2 MB of RAM was allocated to allow systems of up to 64-cores to operate while running 64 independent benchmarks, each with their own static data, stack and heap.

4.3 Motivation and Innovation

Simulation of this MPSoC was one of the driving factors for developing the simulator, to help with performance estimations and software development before the chip development completed. It is likely that there are many similar cases where a simulator such as this would be extremely beneficial.

It was realised when designing the simulator that the out of core traffic could be modelled much more efficiently than traditional coherent simulators. A timing accurate simulator for a coherent system must ensure that every single memory access happens in the correct order. This means that every memory access simulated must result in the simulation thread synchronising with a global clock reference, to ensure that all

memory operations that should happen before it have taken place. This tight synchronisation causes problems with simulation performance, as each simulation thread can not do much work without synchronising; the Slacksim paper [98] discusses this well. With a cache incoherent system a correct program must already assume that cached memory operations will not be immediately available to another processor core, and cannot guarantee visibility until a cache flush operation. A well behaved program should also not assume that it is running on an incoherent system, as events may cause cache lines to be flushed before the programmer is expecting it, such as interrupts. This means that all cached memory accesses should be free of data races between cores, and these memory operations can be performed as soon as possible in the independent core threads. Now a significant synchronisation problem has been removed, and only modelling the timing side effects remains. To do this requires the communication of cache miss events to a separate thread to model the NoC interconnect, which takes care of the timing impact on the out of core traffic. For memory accesses which will cause data races or require synchronisation (i.e. cache bypassing operations and cache line flushes) the core thread is synchronised to the interconnect thread, and the timing model completes its operation before continuing core simulation. In doing so, timing correct behaviour is maintained, with significantly less synchronisation than traditional coherent simulators.

4.4 Simulator Implementation

This simulator advances the previous work on Arcsim [126; 12], the high speed functional-first cycle accurate simulator, which uses modern JIT compiler techniques to accelerate both the functional execution and core micro-architectural model of the target ARC microprocessor. Through runtime configuration it supports many different microarchitectural options, including 3, 5 and 7 stage pipeline models, making it already a useful tool for single-core design space exploration. Previous work on this simulator is extended by implementing a decoupled interconnect model for multicore architectures.

When operating in cycle-accurate mode Arcsim reconstructs updates to the pipeline model after executing each target instruction (both in fully interpretive and JIT-compiled modes). It has previously been parallelised for functional simulation [12], and the core model simulation is explained in detail in Chapter 3 and in the paper by Böhm *et al.* [126].

To parallelise the multicore simulation while providing cycle-accurate modelling of the shared NoC interconnect a similar approach to the FaCSim [48] and FAST [47; 112; 113] simulators is used, making use of lock-free asynchronous producer-consumer circular buffers to decouple simulation of the cores from the interconnect.

These relatively large buffers provide significant slack for the core simulation threads to execute ahead of the interconnect model, which means the interconnect model is rarely waiting for work to process, and helps to cover the time when the thread simulating a particular core is scheduled off by the host operating system. This increases the efficiency of the simulation when the host system has fewer physical cores than the target being simulated, and is enabled by exploiting the reduced synchronisation required for incoherent programs.

Unlike either FacSim or FAST, which decouple the functional simulation and the core timing model, Arcsim [126] models the core micro-architecture interleaved in the same thread as the functional core simulation. Each core is simulated in its own thread, which means that the core timing model is as parallelised as the functional simulation, helping maintain performance when larger multicore designs are simulated. Because the NoC architecture is cache incoherent, the core model can safely include the caches, meaning only cache misses and explicit cache-bypassing instructions must be communicated to the interconnect model. This reduction in communication allows the use of smaller communication buffers when compared to other decoupled simulators such as FAST, COTSon [49] and FaCSim, allowing larger target systems to be modelled on smaller hosts systems. Figure 4.2 shows a simplified view of the memory components of the core simulation kernel, highlighting where asynchronous and synchronous communication is required, and the presence of both instruction and data caches in the core simulation. Embedded cores, such as the ones used here, are often simple in-order interlocked pipelines, which stall on memory events such as cache misses. This allows the time passed for a processor core to be accurately described as the the sum of cycles spent internally, plus the cycles spent waiting for IO requests. This is the second property of the embedded cores leveraged to decouple the simulation to this extent, and is in some ways analogous to the QuantumKeeper in SystemC TLM2.0 [127]. Features such as victim caches and store buffers can still be accurately modelled so long as the interconnect model is aware of them. For example, a store buffer of depth N would mean that the interconnect can process up to N cache write-miss events in parallel before adding time to the core's IO-time offset, while a write-back buffer would be modelled in the interconnect itself.

```
simulate_instruction(){
  update_icache();
  ...
  if(memory_op)
    perform_memory_operation();
}

update_icache(){
  if(!i_cache->is_hit(pc)){
    interconnect_i_queue->push_fetch(pc);
  }
}

perform_memory_operation(){
  if(is_bypass){
    if(is_read){
      interconnect_d_queue.push_read(addr);
      while(!interconnect_d_queue.empty()){ os_yeild(); }
      r_data = interconnect->get_read_value(core_id);
    }else{
      interconnect_d_queue.push_write(addr, w_data);
      while(!interconnect_d_queue.empty()){ os_yeild(); }
    }
  }else{
    if(is_read){
      r_data = mem->read(addr);
    }else{
      m->write(addr, w_data);
    }
    if(!d_cache->is_hit(addr){
      if(d_cache->is_dirty(addr){
        interconnect_d_queue.push_writeback(addr);
      }
      interconnect_d_queue.push_fetch(addr);
    }
  }
}
```

Figure 4.2: Simplified processor simulation instruction implementation, demonstrating communication to the interconnect thread.

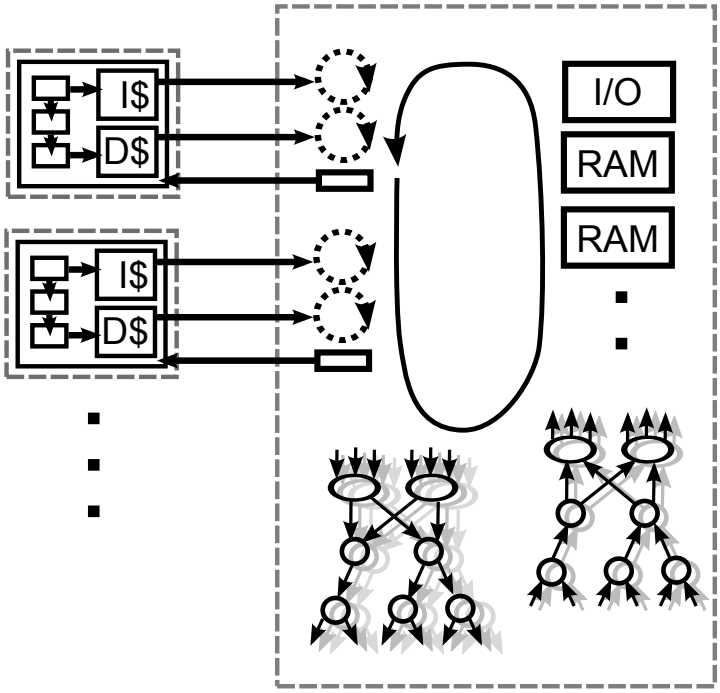


Figure 4.3: Overview of the incoherent simulation, threads are identified by dashed boundary boxes.

The simulator is designed to make full use of modern multicore processors, similarly to FAST running parallel functional simulations. One host thread is used per simulated core, another thread for the interconnect, and additional threads for the functional simulation of IO devices such as display and sound devices. The simulator also supports parallel JIT worker threads to improve the performance of the core simulation. When running the interconnect thread, the simulator will use at least $N+1$ host cores where N is the number of target cores being simulated. The core and interconnect simulation threads are shown in Figure 4.3, with dashed box boundaries representing operating system threads and dashed circles representing asynchronous lock-free queues. Other threads such as virtual displays and JIT compilation threads are omitted for clarity.

4.4.1 Details of the NoC Interconnect

Based on the AXI protocol the NoC is implemented with five identically implemented channels, as described in Section 4.2. The design flow takes a list of 'master' nodes, 'slave' nodes, a design metric termed the 'complexity' of the network, and other configuration parameters. The complexity controls the width and depth of the binary-

routing network generated; then extra switches are added if nodes cannot be connected, and redundant switches are pruned. Binary-routing networks use a multi-layer switch topology like the butterfly, comprising two input and two output switches. Unlike a traditional butterfly network, the presented network's source and destination nodes are not the same: there are explicit 'master' ports for cores and DMA-capable devices, and 'slave' ports for memory-mapped device and RAM interfaces. The tools then generate for the simulator an ordered list of these switches, with outputs and inputs explicitly connected with named 'wires', along with details of the master and slave connections and routing information based on memory addresses. In hardware the routing is performed by a binary routing string encoded in the AXI address field, which is rotated one bit each time it passes through a router, and rotated in reverse for the return routing.

Additional complexity is introduced because not all switches are registered. They can optionally be configured with FIFO buffers, or they can act entirely combinatorially from one registered switch or input to the next registered switch or output in a single cycle. In this mode they behave somewhat like an $N \times N$ crossbar, but with internal collisions. The simulator correctly handles these combinatorial switches while still modelling them in a modular per-switch fashion.

4.4.2 Modeling the NoC Interconnect

The interconnect model has two key roles in the simulation: not only must it model traffic and keep track of time spent in IO for each core, it must ensure timing-correct ordering of memory operations. Because the target platform is cache-incoherent, only cache-bypassing instructions must be committed in timing order to ensure timing-accurate behaviour. Because of this, cache transfers are modelled without actual data, and memory operations are committed by the core simulation thread. This is also how FaCSim operates, but because it only simulates a single core it does not need to worry about memory orderings. Cache miss operations are logged in the circular buffers by the core simulation and the core can continue simulation, but for cache-bypassing operations the core must wait until the buffer is flushed empty signalling the operation has been completed by the interconnect thread. This impacts performance somewhat; but it is required unless an alternative mechanism of ensuring correctness is employed in the functional simulation, such as checkpointing and roll-back as implemented in FAST [47].

Rather than compute routing tables at the cores to generate the routing bit-strings

like the hardware, the simulator uses the routing tables from the NoC description to perform master to slave routing at each switch, and constructs the return routing bit-string dynamically as it passes through the network. Routing tables are shared between the switches on each channel for the same logical switch node.

Since ArcSim already provides extremely fast cycle-accurate simulation of the core micro-architecture, it was a challenge to model the interconnect and memory controllers at sufficient speed to avoid bottlenecking the whole system. To achieve this numerous optimisations were used, such as extensive packet counting and fast forwarding, to ensure as little work as necessary is performed while processing the interconnect.

One of the most important steps to enabling the interconnect simulation to keep up with core simulation was the "fast-forward" mechanism. Each cycle the interconnect model checks to see if each core has an active transaction, and if not when the next transaction is due (by checking the pending transactions in the buffer). If there are no active transactions then the interconnect can directly skip ahead to the cycle of the next pending event – it fast-forwards the interconnect simulation up to the point in time where the next event may occur. This enables the significantly higher than 1 MHz simulation speeds for cache-efficient workloads.

To provide performance for those times when the interconnect is busy there are a few important optimisations, foremost packet-counting. Packet counting works by dividing the network into independent sub-networks, in this case each of the five AXI channels is as separate network, and tracking the number of active flits in each network. By counting flits into the network, and flits out of the network, the simulation can completely avoid processing a network with zero flits in transit.

The final non-standard optimisation is to expose an ‘active’ flag for each network switch. Although the implementation in this thesis did not use inheritance (each switch was a self-contained class), the description here will describe it as if C++ class hierarchies were used to implement switch behaviours, because this is where the optimisation is most important. Despite packet counting, even when a network portion must be processed only a fraction of the switches are likely to require processing. For switches without any active flits, either on inputs or in buffers, the processing should be terminated as quickly as possible to return to the interconnect loop and process the next switch. However, performing this early-out within the switch model code means that every idle switch still incurs a function call and return overhead (unless the code has been inlined)). To avoid this a public boolean value is exported from the base class –

this is important, as the base class member can be read directly from the object pointer without looking up the virtual table, calling into a virtual function, or resolving the switch to a more specific type. Because the active flag can be read directly, without any function call overheads, it is the most efficient way to avoid processing the switch and move on to the next.

Even without class hierarchies the optimisation is important, because unless switch processing methods are very carefully written and compiled as part of the whole interconnect source unit, the methods will not be inlined without modern link-time optimisation techniques. Given the complexity of the switch processing function for a non-trivial microarchitecture it is unlikely that the method would be selected for inlining, and the function call overhead would not be avoidable.

Switches are connected via pointers to a common 'message' type, with each switch output being a instance of this 'message', and each switch input being a pointer to the output message of whichever switch output it connects to. The 'ready' state of the destination is implicitly indicated by whether the message was cleared in the previous cycle. This communication mechanism allows switches to be completely agnostic of what they are connected to, and transfer information very efficiently, but does not allow switches to add each other to a list or other data structure to track active switches.

Good cache behaviour was achieved by first allocating all of the interconnect switches in a single array, and as the network was constructed the switches were selected from this array in network-dependence order. Because of this, as the switches are processed they are simply processed in array order, allowing the cache prefetcher to work efficiently, and many of the pointer references between switches will still be in cache, from when the source switch was recently processed.

The constructs used to track the state of each core, such as timing offsets and state machines, were not grouped into structs or classes for each processor, and then allocated as an array, but instead maintained as individual arrays of each attribute. This is because clustering them resulted in a dramatic performance impact, up to a third reduction in performance. Because of the usage pattern of these arrays it is more common to operate on a select few of these in a loop over the processors, where cache benefits are felt and packing them into a struct would cause the extra data to pollutes the cache. Following this reasoning it may be better to keep the 'active' state tracking in a separate array for the switches, than contain it within the switch struct, because this would avoid loading the whole switch into cache when it was not going to be processed.

A summary diagram of the main interconnect loop is provided in Figure 4.4 demonstrating where various optimisations are performed, and how phases of computation are interleaved.

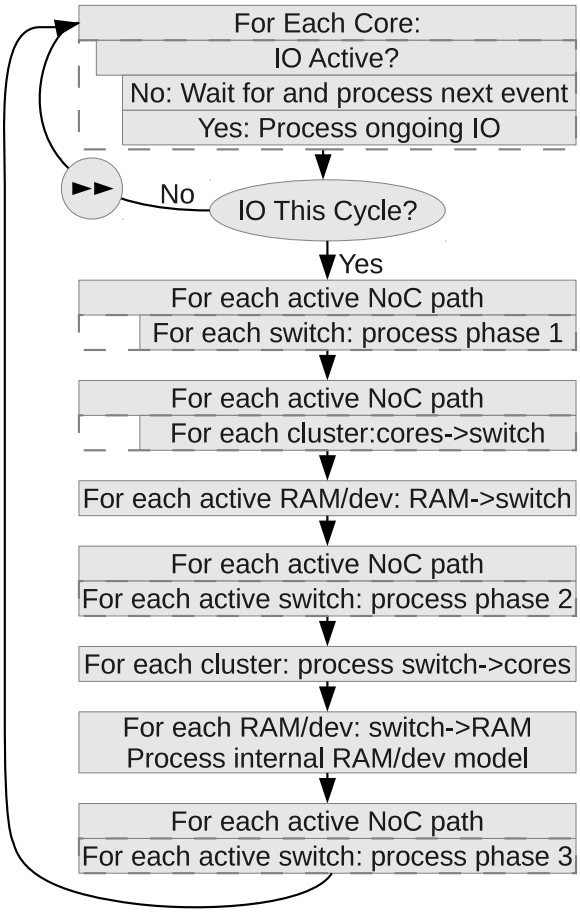


Figure 4.4: Overview of the interconnect simulation loop identifying work-saving optimisations.

To compute a single cycle of the network the simulator iterates over all switches (in active channels) three times, marked as phases 1, 2 and 3 in the diagram. First all switches check their inputs for an incoming packet, flagging themselves as 'active' if they take in data or still have data in a FIFO. Because the switches are correctly ordered in source→destination dependency order the combinatorial switches can simply propagate the packets here to their output: this is the same memory location as their destination's input, so they can be computed in a modular fashion rather than as a monolithic $N \times N$ switch. In the case where all switches are registered, packets can be consumed at this point; but when combinatorial switches are present they are left to be cleared up in phase three. The second phase is for registered switches to output their data packet onto the wire, depending on their internal state machine which is modelled

as part of the switch micro-architecture. Finally the third phase is run in reverse direction, from destination to source, to track back and clear up the packets left when combinatorial switches are enabled. In this phase packet collisions are detected and the internal switch state machines and arbiters updated. The simulation also updates traffic counters on switches which successfully transferred a packet. The overhead of simulating combinatorial switches is a reduction in simulator performance of approximately 30% for a benchmark with moderate traffic, because of the extra book-keeping involved. It was disabled for the work in this thesis, since all the designs tested used registered switches with FIFOs of depth 2 or 16 depending on the design. The output phase of connected devices is run between phase one and two, and the input phase between the second and third phases of the network.

Unlike FAST, this simulator does not have to model the data for all memory operations, only the information relevant to timing: the address, time, and size of cache misses. The exception is single transfer un-cached accesses which are processed in the micro-architecturally-accurate model of the memory controller or device model, ensuring the correct memory operation interleaving and behaviour on locked memory regions when atomic accesses are used.

4.4.3 Simulation Challenges

A potentially problematic feature of the functional-first simulation technique used in the simulator is that instruction-cache miss events are only realised after the completed execution of the previous instruction. This means that the event may only be discovered after the simulation of another memory event in a previous instruction, which it should have preceded. To solve this problem two buffers are implemented between the core and interconnect simulation components, one for each of the two pipeline stages which can generate memory events. When the interconnect reads off events from a core's buffers it waits until one of the following conditions is met before proceeding:

- There is an I-cache event, but no data events. It can continue because the data memory port cannot produce events which happen before this.
- There are entries in both buffers. It executes them according to their timestamps – potentially simultaneously in the same fashion as the core supports.
- There is a data entry and the core model has advanced beyond the point where an instruction cache event can over-take it.

- There are no events but the core model has advanced beyond the point where any event could be generated for this cycle.

- There is a data event and the data buffer is flagged as containing a cache bypassing operation, which is preventing the core simulation from continuing. This is the only condition under which the core is allowed to violate the ordering of instruction and data memory events, because the simulation cannot continue without running the memory operation through the interconnect or potentially violating the ordering of data memory operations from different cores.

The violation in the final clause is an extremely rare event. The I-cache event is still modelled and its latency accounted for, just at slightly the wrong time. It is also possible to write code such that this never happens, if it is extremely important that the platform is simulated accurately.

Because the simulator's correctness relies on a correctly written target application, which does not have accidental cached read/write data sharing, a second mode of simulation is provided which moves the data cache model to the interconnect. This mode fully models the data in the caches, and the correct interleaving of data cache line reads and write backs in the memory controllers. It is almost completely functionally true to the target system, while still maintaining significant parallelism, but loses the performance benefits of the mode primarily discussed in this paper, falling back to performance of 1-2 MIPS for small scale MPSoCs. This mode is used for debugging target applications which exhibit signs of accidental data sharing, and for coherent simulation modes.

4.5 Performance Evaluation

Unfortunately there is no standard parallel benchmark suite for cache incoherent architectures, so various multiprogrammed workloads were composed using Coremark and a subset of the EEMBC [128] suite. The benchmarks used were restricted by the target platform simulated, due to limited on-board memory and the capabilities of the current runtime only the following benchmarks could be run: Coremark, AutoCor, Conven, Fbital, FFT and Viterb. In addition to these standard benchmarks we also ran two memory bandwidth heavy in-house benchmarks: a panning image display benchmark 'imgdisp', and a synthetic cache-thrashing benchmark. The image display benchmark features both bad cache performance, due to its working set and access

patterns, and extensive uncached IO to the display controller, which is treated in the simulation as a synchronising event, making it a good indicator of simulator performance for workloads which feature communication and are IO heavy. Between the EEMBC and Coremark benchmarks there are a wide range of runtime behaviours, and with in-house memory intensive benchmarks cover the spectrum of compute and memory intensive workloads, while still exposing enough code complexity to challenge the core micro-architectural model.

Statically scheduled parallel workloads for 1 to 64 tasks were composed by randomly selecting benchmarks from this set. The benchmarks used in the 1 to 6 task workloads also used for accuracy comparisons can be found in Table 4.2.

To give a clearer measure of how performance varies with the benchmark behaviour 1, 3, 6 and 12 thread single benchmark workloads were also composed for the EEMBC suite and Coremark benchmarks. These benchmarks were run on appropriately configured 1-, 3-, 6- and 12-core MPSoC designs and simulated on a dual socket, 6-core Intel Xeon X5650 workstation, providing 12 cores at 2.6 GHz. Performance results are shown in Figure 4.5, with solid bars representing 4KB direct-mapped caches, and striped bars representing 4KB 2-way set associative caches.

The JIT worker threads were enabled for these experiments highlighting the best case for simulation performance. The 2-way set associative results for Conven and FBital demonstrate that extremely high simulation rates are possible for cache friendly benchmarks, with simulation rates over 377 MIPS, but the graphs also highlight that the simulator has two distinct interacting performance profiles. The core simulations run at approximately 14 MIPS per core in interpretive simulation, or 50-100 MIPS with the JIT enabled, and while the interconnect can keep up easily when there is minimal work to do, it can only simulate a saturated medium sized interconnect at around 1 MHz. Testing with a 64 thread cache thrashing workload on a 64-core target, resulting in 98.8% of each core's time being spent waiting for IO and only 0.006% of cycles able to be fast forwarded, gives a simulation rate of only 0.285 MHz on a 1.8 GHz 32-core server. However this still represents an aggregate simulation speed of 18.25 core-MHz, and despite the effective CPI of 170 provided a still competitive simulation rate of 0.107 MIPS.

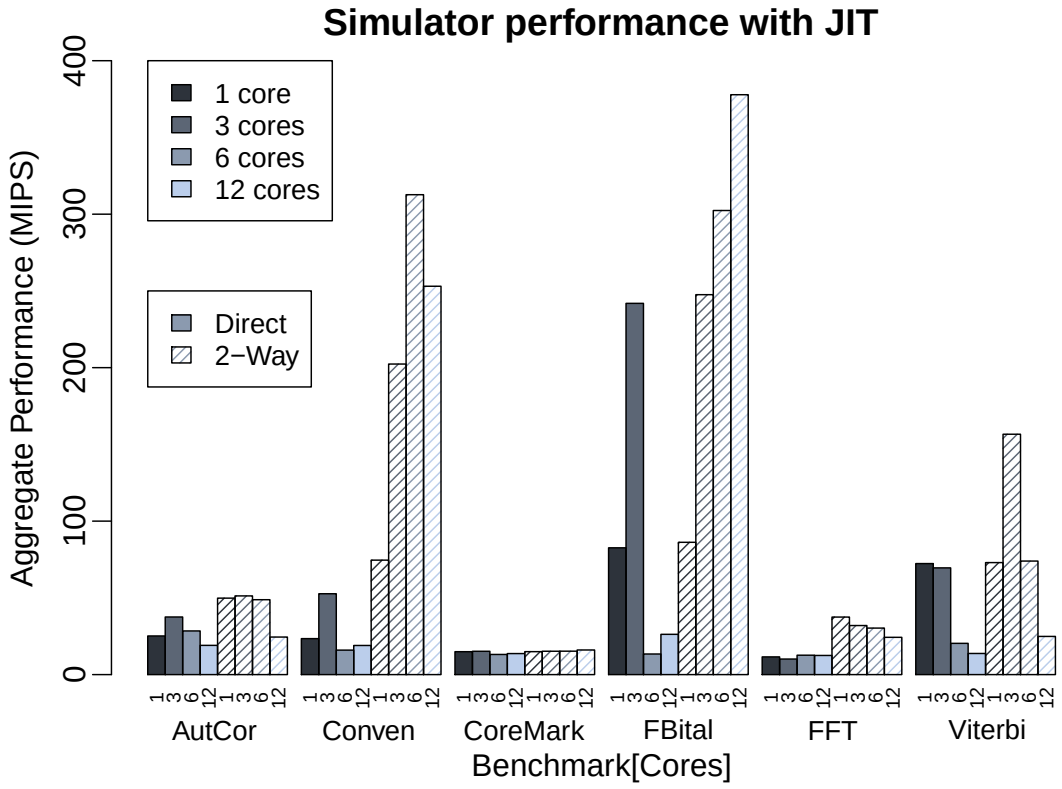


Figure 4.5: Simulator performance across the EEMBC and Coremark benchmarks for 1-, 3-, 6- and 12-core designs. Solid bars indicate 4KB direct-mapped simulated I and D caches, striped bars indicate 2-way set associative variants of the adjacent design.

Workload	Contained Benchmarks
1_0	imgdisp
1_1	fbital
1_2	conven
1_3	autcor
2_0	coremark, imgdisp
2_1	autcor, conven
2_2	fft, vinterb
2_3	viterb, imgdisp
3_0	conven(2), imgdisp
3_1	conven, coremark, cache_thrash
3_2	coremark, fbital, imgdisp
3_3	coremark, fbital(2)
4_0	conven(4)
4_1	autcor, conven, coremark(2)
4_2	autcor, fbital(2), imgdisp
4_3	autcor, fbital, fft, imgdisp
6_0	coremark(5), fft
6_1	autcor, fbital, viterb(2), cache_thrash(2)
6_2	conven, coremark(2), fbital, fft, viterb
6_3	autcor, conven, coremark, fft, imgdisp, cache_thrash

Table 4.2: Composition of multi-benchmark workloads.

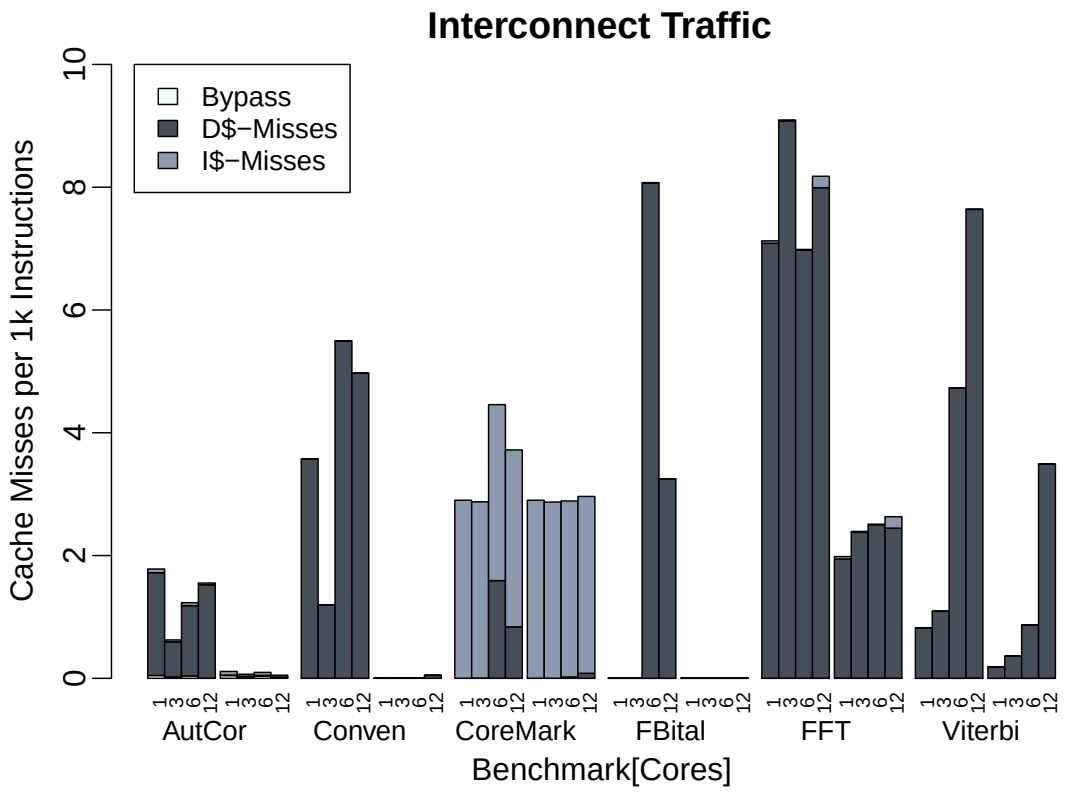


Figure 4.6: Cache behaviour across the EEMBC and Coremark benchmarks for 1-, 3-, 6- and 12-core designs. Bars are arranged in the same order as Figure 4.5, grouped into 4KB direct-mapped and 2-way set associative caches.

By analysing the cache behaviour of the benchmarks in the simulations which produced Figure 4.5 we can see the relationship between interconnect traffic and simulation performance. Figure 4.6 shows the breakdown of interconnect traffic across the simulations in terms of cache miss events per thousand instructions executed, isolating synchronising cache-bypassing traffic, data cache misses, and instruction cache misses. It can be seen comparing these two figures that good cache performance leads to exceptional simulation performance, although even when interconnect traffic causes simulation performance to drop, it is still significantly faster than the state of the art.

Simulator performance statistics were also collected from a large scale design-space exploration on a shared cluster computing facility, comprising a mix of 8- and 12-core nodes. The simulations involved the previously discussed generated workloads, of which 64 different multiprogrammed workloads were produced comprising between 1 and 64 independent tasks. The designs were also randomly selected from a design space of 12000 designs, the possible combinations of the configuration param-

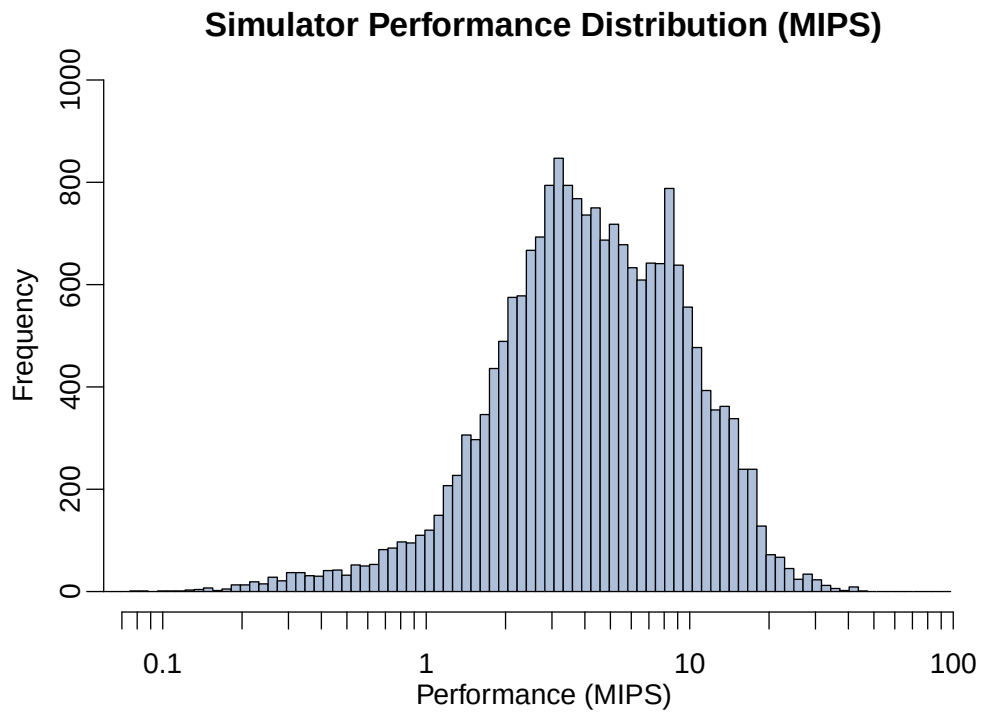
eters listed in Table 4.1, varying the number of cores from 1 to 64, divided between 1 to 8 clusters. In total 20204 design-benchmark combinations were simulated, which used 1008 different MPSoC configurations from the total design space. For time-allocation on the compute cluster, the number of host cores requested for a specific design/benchmark configuration was the minimum of the number of cores in the design, and the number of tasks in the workload, plus one for the interconnect. This was chosen because cores will shut down once there are no tasks remaining to execute. The bare-metal runtime has an IO heavy start-up phase which must be executed on all cores of the design, so simulation speed can appear artificially limited by having to simulate many cores and an interconnect on only a few allocated physical cores during this time. Unfortunately the cluster workload scheduler has greater throughput as scheduling both shorter running and less resource demanding tasks, in comparison to those with longer estimated run-times or greater physical core allocation. Because of this the results of the simulations are heavily weighted towards those with fewer active cores, since more of these were successfully scheduled and completed in the fixed timespan available for experimentation. The average simulation required 5.3 active cores, or approximately 6 host threads.

For the large scale cluster experiments the results are presented in two parts. Firstly, Figure 4.7(a) presents the performance in instruction throughput expressed in MIPS, as is typical for instruction set simulators. Here you can see the instruction throughput rarely drops below 1 MIPS (<5% of simulations), and on average achieves 5.7 MIPS aggregate across the simulated cores. In the worst case there are almost no simulations which executed at under 0.3 MIPS (<0.7% of simulations), and none below 0.08 MIPS, while the best-case simulations ran at an aggregate 45 MIPS.

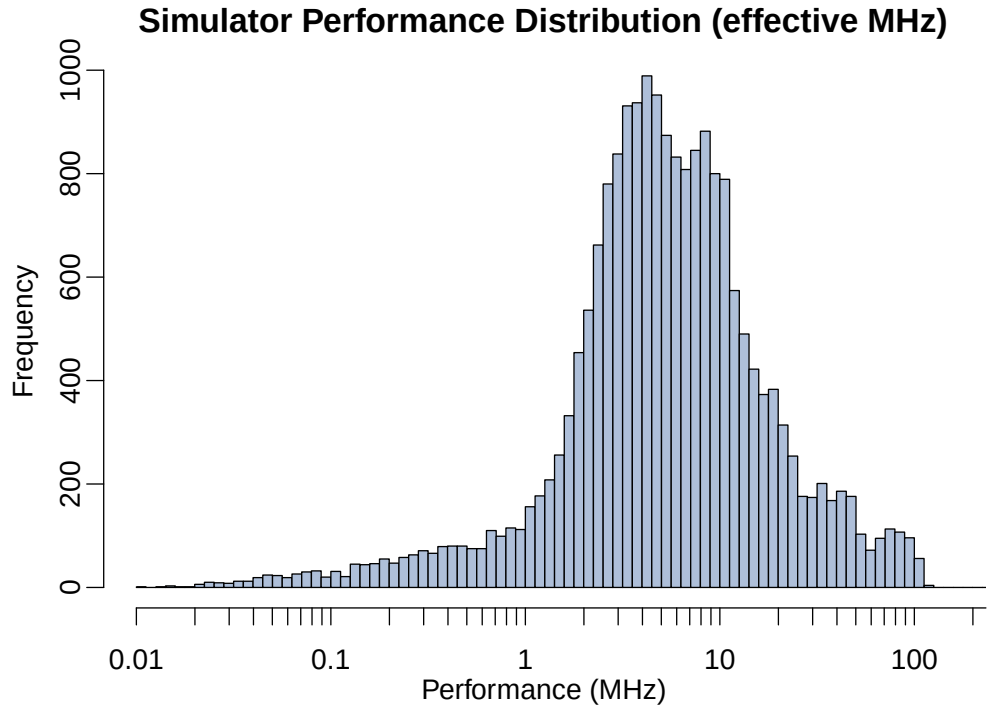
Secondly, Figure 4.7(b) presents the results as effective simulation rate of the whole MPSoC in MHz, not aggregated per core. For example a design with a 2:1 core to interconnect clock ratio reported as 10 MHz, means that each core simulated at 10 MHz, and the interconnect at 5 MHz (since it only executes 1 cycle for every two the cores execute). Similarly if the core to interconnect ratio was 1:2, 10 MHz means that the cores simulated at 5 MHz each, while the interconnect ran at an effective 10 MHz. On average simulation rates of 10.1 MHz were achieved, only 7.5 times slower than the fastest design which can be synthesized to a Virtex6 FPGA. The maximum performance was 117 MHz: significantly faster than the FPGA implementation, and approaching the 250 MHz projected speed for 65nm silicon implementation. Unfortunately there is a very shallow tail below 1 MHz covering 8% of the simulations, which

extends down to 0.01 MHz, indicating either pathological simulation conditions, or the simulation host in the cluster was being shared with an application causing a high degree of performance interference. The simulations were primarily run as a large design space simulation experiment, rather than for performance metrics, so time was not spent investigating the cause of the poor simulation performance of this tiny fraction of experiments. Looking at Figures 4.8 and 4.9 however helps to explain some of the behaviour. With a maximum number of required host threads of 12, no simulations except a few with a single active core execute at less than 0.5 MIPS. All 1.8% of the designs which execute below 0.5 MIPS are running on over-loaded hosts, while on average simulation throughput was increasing with the number of target cores up to this point. This, along with the marked drop in average simulation performance when more threads than the available 12 host cores are required, indicates that the performance was probably lost to the overhead of the operating system scheduler, and later work in this thesis on a coherent simulation, employing a user-space scheduler, does not suffer from this problem. As such the worst results most likely could be redeemed through such a user-space scheduling system.

These figures for extremely stressed interconnect traffic do not present a realistic view of the simulator performance, since most workloads make effective use of the core's private caches to reduce IO traffic; the simulations used to generate Figure 4.7 report an average IO/core cycles ratio of 7.4%, with a maximum of 93%. This is the reason for the extremely large performance distribution seen in Figure 4.7(a), which is unusual for a cycle-accurate simulator. The very lowest tails of the performance distribution are likely caused by simulations with more cores than host threads, which execute multiple instances of the panning image benchmark. Because this generates uncached IO to the screen, the core-interconnect synchronisation is triggered frequently. While not a problem in isolation, when the operating system cannot schedule all the threads to run simultaneously, it will often send the thread simulating the core performing IO to sleep, to let another core run. The interconnect thread will quickly process the single event and then hang waiting for the core to continue, because it cannot proceed beyond the current cycle time of the slowest core. With normal cache miss operations some of this scheduling overhead is masked by using large circular buffers, which allow the cores to execute sufficiently far ahead that the interconnect will not have to stall for long if the core thread is scheduled to sleep. This pathological behaviour is addressed in the next chapter with the introduction of user-space scheduling, which provides significantly lower context switch overheads and allows for finer



(a)



(b)

Figure 4.7: Histograms of simulator performance from a large scale design space exploration, showing that almost all of the 20204 design/workload combinations have simulated performance over 1 MIPS, while most perform significantly better than this.

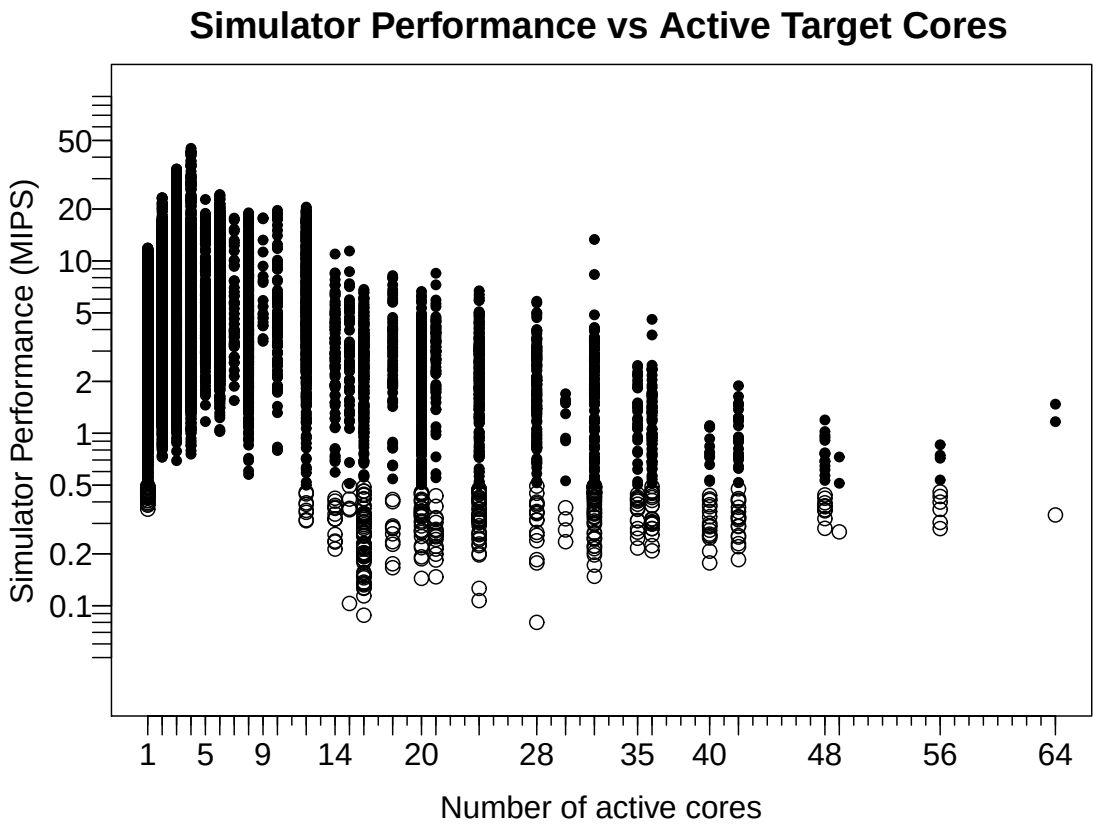


Figure 4.8: Simulation performance against number of active simulated cores. This is one less than the required number of simulation threads. Results less than 0.5 MIPS are highlighted with hollow circles.

control of the scheduling. Other simulators also use user-space scheduling for the same reasons [129].

It is worth noting that while the simulator supports JIT compilation to accelerate simulation of the individual cores, it was disabled for the large-scale performance and accuracy results presented in Figures 4.7 & 4.11. This is because there are a few cases where the JIT compiled simulation model produces slightly different timing of events, due to a microarchitectural detail that has been updated in the interpreted mode only. This does not lead to a significant error in the simulated accuracy (<1% difference to interpreted with the 3-stage pipeline), but does lead to non-determinism, as events will be generated at slightly different times. Since the cluster-based simulations were run primarily for purposes of design-space exploration rather than performance evaluation of the simulator, it was decided that leaving the feature disabled was preferable. Since the extra performance provided by using the JIT has the most impact when the interconnect has little work and can fast-forward to keep up, its use would only stretch out

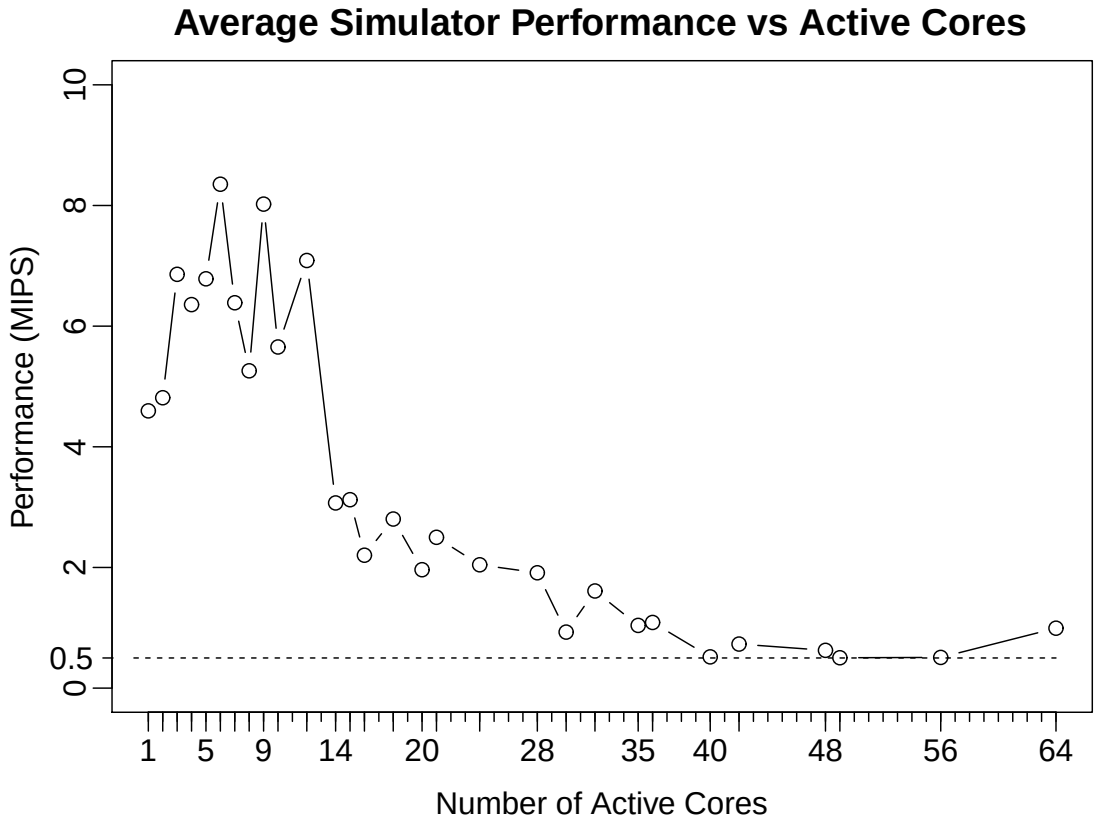


Figure 4.9: Average simulation performance against number of active simulated cores, showing the impact of simulation with more threads than there are physical cores in the host.

the upper tail in simulation performance towards 100 MIPS per core. It would not significantly affect the body of the results, which are limited by the performance of the interconnect model.

4.6 Accuracy Evaluation

To construct and verify the detailed microarchitectural switch and memory controller models, detailed tracing output from the simulator was compared manually against Verilog simulation, confirming that the interconnect switch and memory controller implementations were indeed cycle accurate under a variety of test conditions. To empirically measure accuracy of the whole simulator several designs were synthesized to a Xilinx Virtex6 FPGA and test workloads were run on these platforms. The run-times in cycles for each core were collected and the results compared with the simulator, using the generated simulator configuration file.

To demonstrate how simulator accuracy varies with the different EEMBC benchmarks, and when scaling up the core count, the simulated cycle counts from the performance tests in Figure 4.5 were compared with the same design running on the FPGA. Once again solid bars represent direct mapped caches, and striped are 2-way set associative, with each benchmark being run on 1-, 3-, 6- and 12-core designs. The bar for 12-core 2-way set associative is absent because this design does not fit into the available Virtex6 FPGA. The results, shown in Figure 4.10 clearly show that the error is most affected by the benchmark, rather than the hardware configuration. Another feature which can be discerned using Figures 4.5, 4.6 and 4.10 is that benchmarks with good cache behaviour (almost no interconnect traffic) such as FBital can have poor simulation accuracy, while those with poor simulation speed due to higher interconnect traffic like AutCor have very little error, indicating that the core microarchitectural model is more responsible for the error. The fact that error is most affected by benchmark and not size of the design supports this.

Secondly, as shown in Figure 4.11, the randomly generated multiprogrammed workloads as described in Section 4.5 were run on a range of different MPSoC configurations, listed in Table 4.3, to evaluate error across different design options and more interesting workload combinations. Each shaded region represents the results for one workload, with each bar being the RMS error of the cores compared to the same core on the FPGA for a given MPSoC design. The designs are listed in order in Table 4.3. Here the trend that accuracy is determined by benchmark mostly continues, al-

Design	1	2	3	4	5	6	7	8
Cores	1	2	2	2	2	2	2	2
Clusters	1	1	1	1	1	1	1	1
RAMs	1	1	1	1	1	1	1	1
Complexity	2	2	2	2	2	2	2	2
Fifo Depth	2	2	2	2	2	2	2	16
Core Freq(MHz)	12.5	12.5	12.5	12.5	12.5	25	50	12.5
NoC Freq(MHz)	12.5	12.5	25	50	100	12.5	12.5	12.5

Design	9	10	11	12	13	14	15	16
Cores	2	2	2	2	2	2	2	4
Clusters	1	1	1	1	1	1	2	2
RAMs	1	1	1	2	4	8	1	1
Complexity	4	8	16	1	1	1	1	1
Fifo Depth	1	1	1	1	1	1	1	1
Core Freq(MHz)	12.5	12.5	12.5	12.5	12.5	12.5	12.5	12.5
NoC Freq(MHz)	12.5	12.5	12.5	12.5	12.5	12.5	12.5	12.5

Table 4.3: NoC configurations used for accuracy analysis.

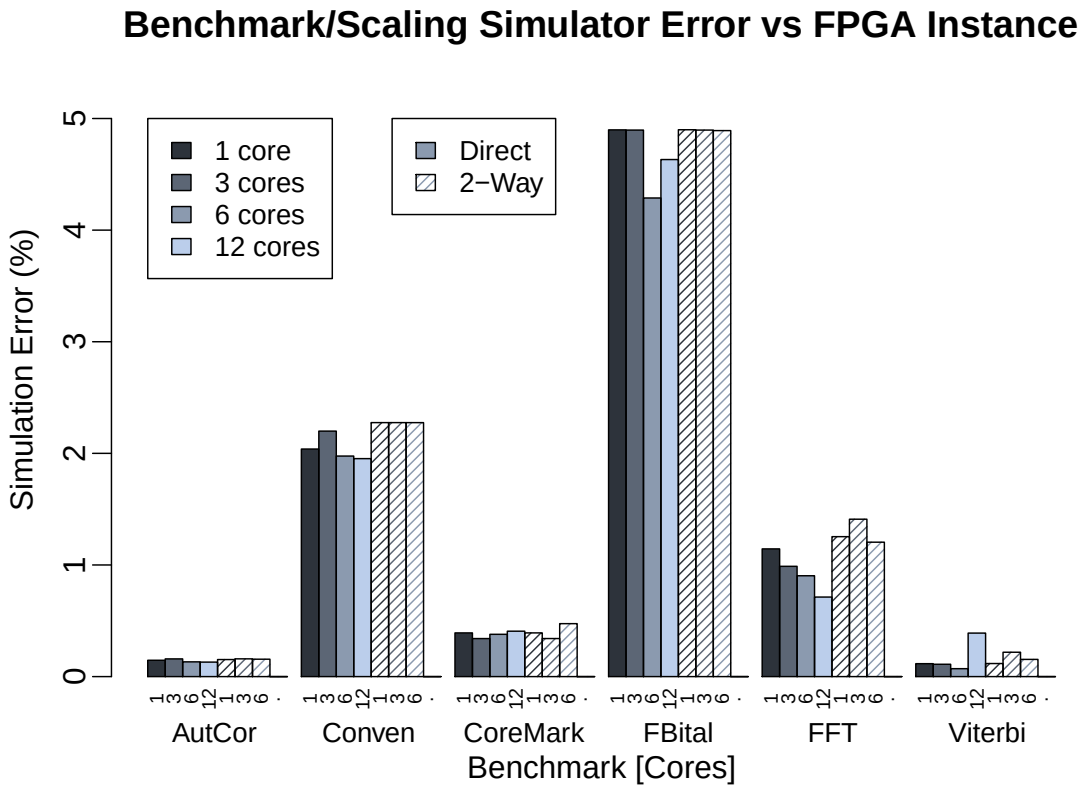


Figure 4.10: Error from performance graph Figure 4.5. Cycle count error from 1-, 3-, 6- and 12-core simulations of direct mapped and 2-way set associative configurations, for standard embedded benchmarks.

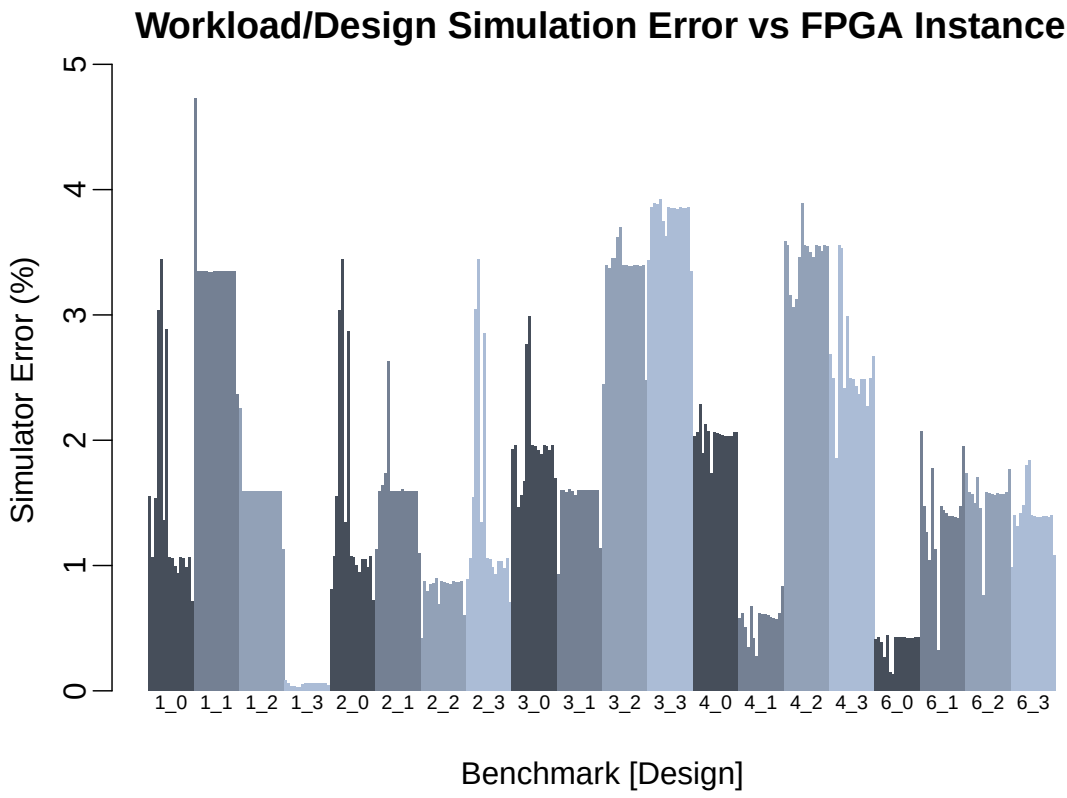


Figure 4.11: Cycle count error across the small scale design space in Table 4.3 for mixed benchmark workloads detailed in Table 4.2.

though more per-design variation is demonstrated than in Figure 4.10 due to the shift in execution time spent in the interconnect and core respectively across the different designs.

The mean error from this larger experiment set is only 1.8%, with an RMS error of 2.1%, comparing well to the single core FaCSim, which achieves an average 7% error relative to its reference platform, and GEM5, which was recently evaluated to provide an average RMS error of 8.8% relative to a more complicated dual-core reference platform [82]. The most similar simulation system, a parallelised SystemC simulator, manages an average error of 6% [129] over a cycle accurate software model.

4.7 Conclusion

Having demonstrated simulation speeds of up to 377 MIPS, with an average execution time error of only 2.1% relative to hardware reference implementations, the presented simulator clearly provides a flexible, powerful tool for embedded systems development. With unrivalled performance for software based, multicore, full system, cycle accurate simulation the simulator is not only useful for application development (for timing accurate functional behaviour, debugging, and performance evaluation) but also large scale design space exploration. This was leveraged to perform over 20,000 simulations on a shared cluster service in under three weeks.

This was achieved through novel exploitation of the cache-incoherent nature of embedded systems to decouple simulation components with significant slack, allowing for efficient parallelism. The performance potential was only fully realisable through the presented packet tracking and counting optimisation techniques, which, along with the cache efficient interconnect model and cycle skipping techniques, allows the NoC based interconnect simulation to keep up with the high speed parallel core simulations.

Chapter 5

A Simulation Architecture for Cache-Coherent Manycore Systems

5.1 Introduction

While a manycore embedded system without cache coherency may be an option for some applications, there are a large number of existing applications and operating systems which require cache coherency to function, which one might wish to run on a manycore architecture. As with the target architectures of the previous chapter, it is highly beneficial to investigate detailed micro-architectural decisions involved in the design of a platform before committing to a final design, and novel micro-architectural changes require detailed, accurate, simulation to verify their impact.

This chapter extends the work in the previous chapter to enable the cycle accurate simulation of systems up to 1024 cores, while fully simulating a complex cache coherency protocol and collecting a large number of important statistics.

5.2 Architectural Summary and Assumptions

Continuing the theme of the previous chapter, this work remains focused on the use of embedded general purpose cores, because to reach the high density required to build a 1024-core chip the cores themselves must not only be very small, but also very power efficient. The architecture itself is discussed in more detail in the next chapter, but this section will highlight the important points as they relate to simulation.

The small 3-stage core is retained, but to reduce the size of the tag RAM, and coherency overhead per byte of memory, the cache line size was increased to 64 bytes for

both the instruction and data caches. The in-order interlocked pipeline assumptions are also retained, allowing the use of time-offset tracking discussed in the previous chapter, although synchronisation must now happen on all data accesses. To better simulate an application processor, a pseudo MMU has been added to the data memory hierarchy, featuring a 128-entry 2-way data TLB of 8KB pages, trapping to a software handler. The TLB is not used in simulation to actually translate between virtual and physical memory, the simulation still runs in a single address space, but it produces the traffic and performance characteristics equivalent to a full MMU. An instruction TLB is not modelled, for simplicity, under the assumption that the small benchmarks used will easily fit within one or two pages and not produce a significant number of TLB miss events. The data cache is now coherent, so all memory operations must be ordered in interconnect timing order, reducing the slack in the simulation. To accurately model the coherence protocol also requires full data caching, not just tags and timing information, to correctly model any errors or behaviours which are more relaxed, or differently timed, than the host memory consistency model. The timing accurate functional caches, which store not only tags but also data, are one feature which GEM5 for example lacks, which makes it difficult to model relaxed memory consistency models, although recent contributions have implemented some cache shadowing to help address this [130]. The data interconnect contains multiple levels with different bus and switch types, from a narrow tree hierarchy, through a flit merging FIFO, to a wider 4×4 mesh giving 16 isolated data trees. An overview of the architecture is provided in Table 5.1 along with Figure 5.1, which is featured again and explained in more detail in the next chapter.

As a simplification measure, the multiple clock domain support was removed and all components are run at a synchronous 800MHz simulated clock rate. The directory is designed to match the 64 entry set size of the L1 caches, with configurable associativity, allowing experimentation of the associativity requirements of the directory ranging from the same associativity as the L1 cache system, down to highly constrained small, low associativity directories. For many of the later presented experiments the directory is configured to match the L1 associativity, to remove it as a source of noise and highlight the features of interest in the experiment. Similarly the last-level caches are modelled as "perfect" caches, which always result in a hit; this removes noise artefacts from the simulation that would be caused by DRAM accesses which are orthogonal to those attributes being investigated. While off chip memory bandwidth is a significant problem for manycore processors [131] it is outside the scope of this thesis. The direc-

Design Parameter	Configuration
Core Architecture	ARC700 32-bit RISC
Pipeline	3-stage in-order
I&D-Cache	4 KB Direct Mapped
Cache line size	64 Bytes
Interconnect Protocol	Custom unified packet based.
Coherency Protocol	MESI Directory with Coarse Vector

Table 5.1: Architecture design parameters.

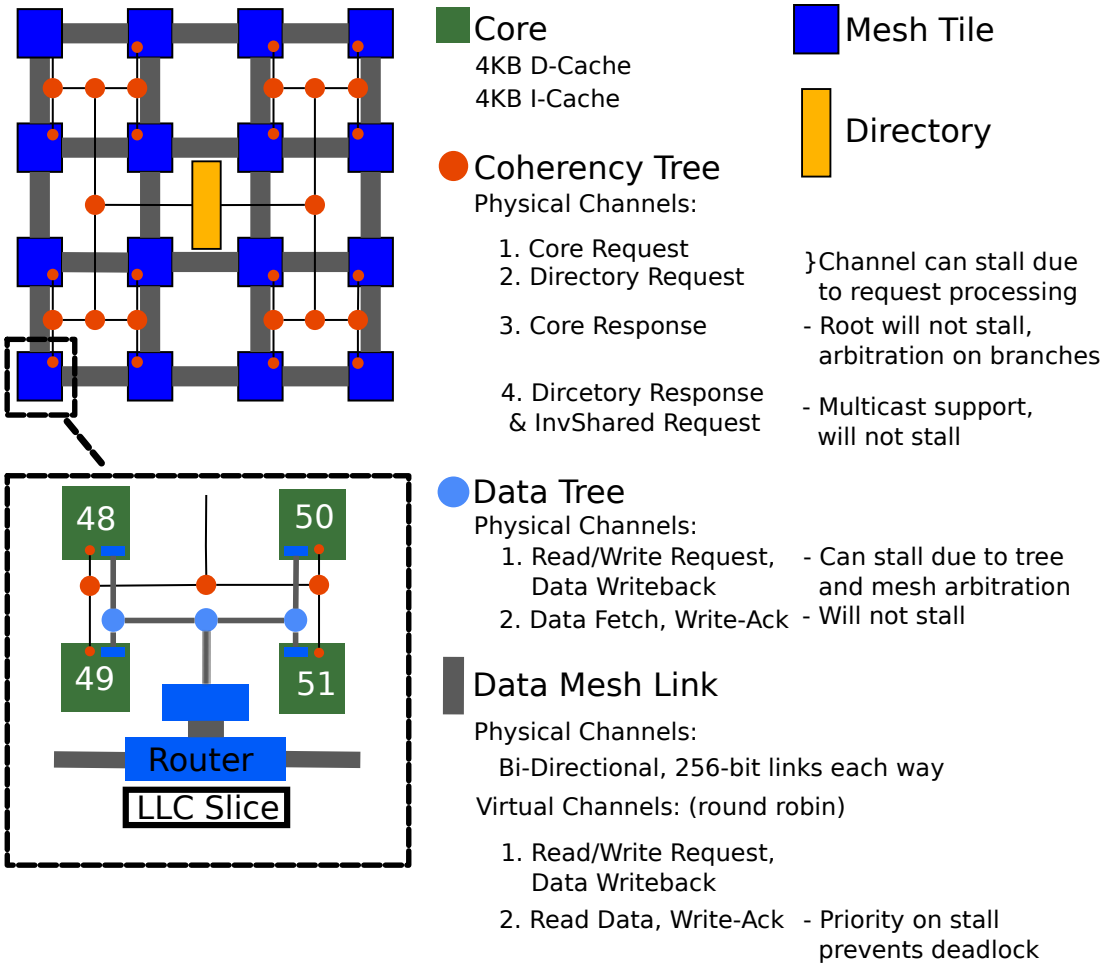


Figure 5.1: Overview of the simulated architecture.

tory uses a simple mask on the lower address bits to generate the index as found in a traditional direct mapped cache. The directory is internally sharded into four pipelines, with cache lines assigned based on the lowest two index bits. This was done to distribute the directory better between smaller, and as such faster, memory blocks; and to allow operations to be processed in parallel while one or more of the four pipelines is stalled on an operation. Each pipeline has a four entry transaction buffer, indexed on the next two index bits. This is to enable certain operations to be suspended which would otherwise require the pipeline to stall while the pipeline continues operations on other independent transactions. The directory and coherence protocol are discussed in more detail in the next chapter.

5.3 Simulator Construction

Each of the NoC switch and router elements are modelled as a simple class containing some internal data structures for arbitration state and any buffering, a C++ 'struct' instance for each output port of the appropriate type for the bus protocol it transports, a pointer to a similar struct for each input port, and a procedure to call which will execute a simulation cycle in the switch. A diagrammatic representation of the up and down tree routers can be found in Figre 5.2. The full NoC model is created by allocating large arrays of each type of switch and router as required, then procedurally assigning the input pointers to the appropriate output ports in the source switch. To reduce computational complexity and the number of iterations through the arrays on processing the ordered interconnects (data and coherency trees) are allocated in the array in such a way that a linear traversal either forward or backwards through the array will process the elements in data flow order, so by traversing the array in the opposite direction a simple registered bus is easy to simulate without requiring extra buffers or handshaking – each switch can propagate its result to the output in the same cycle knowing that its output was computed before it, so the result will not be accidentally propagated. This is a small simplification over the switch models in the previous chapter, in which the registered switches in the incoherent platform used asynchronous FIFO buffers, which introduced longer processing delays and requiring more complicated internal microarchitecture. The simplification allows lighter weight simulation without a significant impact in accuracy, and still provides the full resource contention modelling which is so important in a manycore model.

Because the switches communicate through structures and pointers, any switch

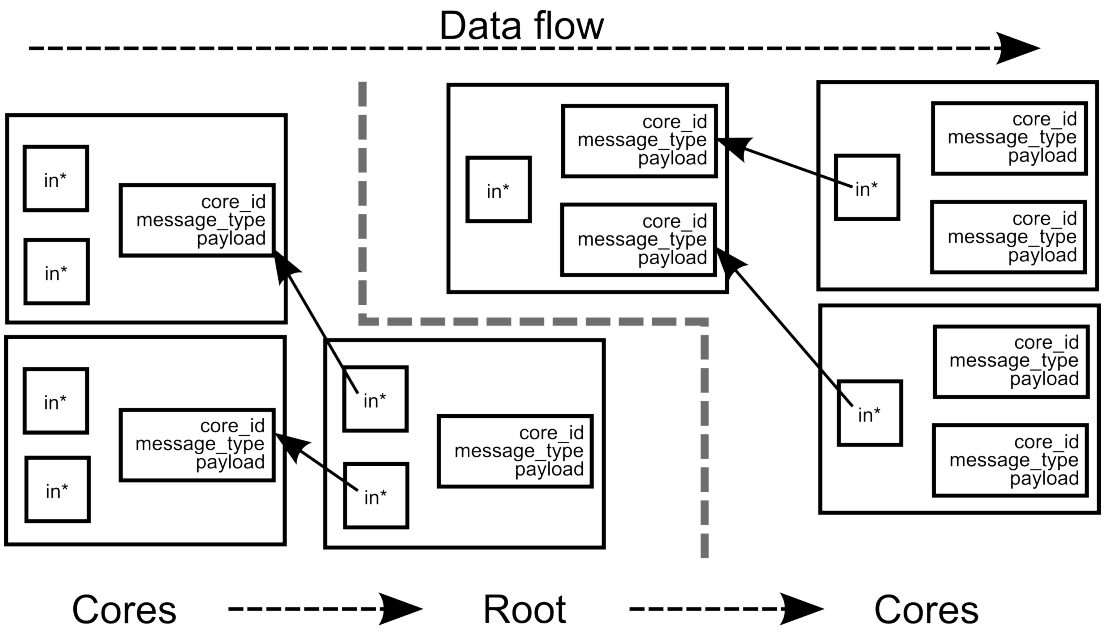


Figure 5.2: Memory representation of interconnect elements.

can be connected to any other switch so long as they use the same struct for communication. This mechanism allows different parts of the network to be connected using converter switches as in the real hardware, for example the flit splitting component between the data trees and the mesh, which take in wide flits from the mesh into a FIFO buffer, and over multiple cycles produce smaller flits exposed to the data tree. This construct allows for memory efficient modelling, because there are almost no extra memory overheads than the registers used in the FPGA or ASIC implementation, which in turn allows for large interconnect simulations which are not bounded by memory constraints, and fairly cache efficient simulation.

5.4 Instrumentation and Statistics Gathering

One of the great benefits of developing an interconnect model from scratch is the potential to include instrumentation which is understood to be an accurate measure of the features one is trying to measure. Because the data structures used for switch to switch communication effectively expose the wire level communication, it is trivial to collect a measure of wire energy for the NoC model. A good first approximation for bus wire energy is the capacitive model, which within a fixed voltage and process scaling, has a fixed cost for flipping an individual wire from one state to the other, scaled linearly with the length of the wire, i.e. the cost is calculated as bit flips per unit length. To

calculate the number of bit-flips that occurred on an individual link in the interconnect, when a new value is about to be written into the output registers representing that link the hamming distance between the new and old value is computed and accumulated in a counter variable stored in the switch. This is performed efficiently by combining the new value with the original using a bitwise XOR operation, so the resulting bit pattern holds a 1 for every bit which has changed, then counting these efficiently using the popcount instruction. Thus for every wire link in the interconnect, the number of bit-flips is cheaply recorded. To arrive at an energy model from this the final count at the end of simulation is multiplied by the estimated wire length for that bus. The number of flits and packets is also recorded to give a separate indication of energy used in the switch routing and overall network traffic, along with an indication of the average number of wires which change state for each flit. A common model simplification made is that on average half of the wires change state, and collecting this information allows us to demonstrate the validity or fallacy of this simplification.

In addition to the usual cache and TLB statistics, the simulator also uses a histogram to track the time to serve memory requests for data and instructions, and time spent idle on sleep instructions explained in the later chapters, providing powerful view into the behaviour of the memory hierarchy and platform performance as software and hardware parameters are modified. The number of active cores is also sampled every 10,000 cycles along with the directory occupancy, to provide a runtime profile of the processor activity and directory utilisation, giving further insights to how a program uses the hardware over the course of its runtime.

5.5 Performance Optimisations

To keep the simulation of coherent architectures fast, many of the techniques from the previous chapter can still be applied to good effect. Whereas in the future, with architectures like AMD's Fusion APUs, and the HSA combined memory space, it will be much more feasible to execute the apparently data parallel NoC switch simulation on an accelerator card, the data transfer and kernel invocation overhead is still far too high with discrete accelerator cards. As such it is faster to compute the NoC serially on a conventional processor, and take advantage of the serial processing nature to make the simulation more efficient. As discussed in Section 5.3, by simulating the switch micro-architecture in a controlled order the requirement for externally visible state to remain unchanged during computation is removed. This allows any form of state shadowing

and duplicating, which would almost certainly be required for GPGPU aided computation (such as using a simulation kernel which computes from one copy of system state, to another, and switches each cycle like the traditional graphics front and back frame-buffers), to be avoided. One of the best techniques for reducing interconnect simulation discussed in the previous chapter was the use of network partitioning and packet counting. Unlike the AXI based network of the previous architecture, the proposed manycore architecture uses a unified data bus, so the method must be applied differently. Each of the 16 tiles contains its own independent data tree, with two separate directions of traffic, giving 32 networks which can be simulated independently. Not only can these be packet counted to avoid simulating the network when possible, but they can be simulated in parallel across a small number of general purpose processors. Similarly the mesh can be treated as another network to be packet counted, but it must be run in its entirety (including the components which couple it to the data trees) before the tree networks can be simulated in parallel. The four coherency trees can also be simulated in parallel to these, with three of them being amenable to packet counting, and the multicast tree (due to its non-blocking nature) able to be put to sleep after a known time-out after the last packet being injected. Another two phases of the simulation which can be executed in parallel are the handling of the core cache controllers and the L2 cache controllers attached to the mesh nodes. After the statistics collecting phase, run every 10,000 cycles, the simulator first runs the core communication and cache controller phase, optionally in parallel, because after this phase it is known if any IO activity must be performed or the entire cycle (or several) may be skipped. It then evaluates whether coherency or data must be simulated and runs the coherency networks, followed after a barrier by the directory simulation. Then, the parallel phase of the data network simulation (including L2 cache output and mesh switch output simulation phases), and finally after another barrier, the mesh and L2 cache output phases of simulation.

The described method is definitely not optimal for a parallel simulation, and it would make more sense to batch all of the parallel network simulation jobs at once (coherency and data trees) perhaps using a task farm style of parallelism rather than the static partitioning used in this implementation, then run the directory and mesh phases of the simulation afterwards, but in parallel with each other. Such a design may be taken by future work which emphasises simulation performance, but the limited time available for this work meant that efforts had to be focussed on architectural exploration using the simulator, rather than further refinement of the simulator itself.

It should now be apparent that there is available parallelism to be exploited in the interconnect, and a small number of cores, from 2 to 4, shows meaningful speedups for most configurations. Unfortunately it is not possible to keep the same degree of core simulation parallelism with the coherent simulation that was seen with incoherent simulation, due to the stricter synchronisation requirements. To tackle the large reduction in simulation slack, and handle the significantly larger number of simulated processors, the switch was made from OS managed multithreading to user-space context switching. A fixed number of OS threads were allocated for running the core simulations (in a similar fashion, a fixed number was allocated to interconnect simulation) between which the simulated cores were statically partitioned. Each core simulation is now running in a user-space managed thread, which is transparent to the running simulation, except it must now yield using a different function which enables a cooperative multithreading based scheduling to take place. This yield function returns the thread from the core simulation, to the thread manager, which then picks the next core simulation to run, and switches context again into that thread, restoring its register state and switching to its thread stack. This approach is similar to how the SystemC kernel switches between running Threads and Methods when they issue Wait statements, context switching to the next Method or Thread which is scheduled to run and moving the thread handler which just suspended into the appropriate queue for the event it called Wait upon. Unlike the parallel SystemC runtime presented by Mello *et al.* [129] the parallelisation presented in this thesis does not sacrifice simulation accuracy, despite the initial similar appearance.

The user-space scheduler used was a simple round robin scheduler, which potentially wasted many cycles switching execution state back to a processor which was still waiting on a response from the interconnect thread. An obvious optimisation which could be implemented in the future would be to maintain a list of active core simulation threads, and remove a core when it yields, leaving only active cores in the list. When the interconnect thread has processed its request, it pushes the core back onto the active threads list. This could also allow work balancing between the threads which handle simulation, because the interconnect could push it back onto the scheduler thread which was quietest, rather than the one the thread came from. This could result in poor cache performance because of threads being regularly migrated however, and is left for future work.

5.6 Performance Evaluation and Conclusions

It is difficult to form an apples-to-apples comparison between the presented simulator and others, because the presented simulator has greater potential accuracy, but less overall flexibility and a restriction on simpler in-order pipelines, when compared to the nearest competition. However taking the best performing simulators capable of simulating a manycore target with some form of timing model gives Sniper [101] and the Monchiero *et al.* modified COTSon [49] as the best competitors. These both have performance claims for high speed multicore coherent simulation. These respectively are 2 MIPS instruction throughput for a 16-core target on an 8-core host for Sniper, and 1 MIPS scaling down to 0.7 MIPS for the COTSon based simulation.

In this section performance results are presented from simulations used to produce the experimental results in the subsequent chapters. As such they were not run in a timing controlled manner, were often running on the same machine as other workloads and simulations, and did not all run on the same hardware configuration or with the same number of threads allocated. However, all simulations which form the same line segment were run on the same machine, with the same thread allocation, so trends amongst the results are still valid. The results can also be considered a lower bound on the performance expected from the highest performing simulation environment and configuration used, and as such can still be compared against the Sniper and COTSon competition. The simulations were all run with either three or five threads, configured as two or four threads executing core simulations, and a single thread running the interconnect simulation. This arrangement was chosen to simplify statistics instrumentation in the interconnect (so thread-safety was not a concern) and to get the best simulation throughput out of the multi-socket workstations and compute-servers used to perform the experiments (simulation performance does not scale linearly with more cores, while running more simulations in parallel achieves close to linear improvements in throughput). The fastest machine configuration was a dual socket 6-core Xeon X5650 machine, providing 12 cores running at 2.6GHz, of which each simulation used at most five cores.

Figure 5.3 presents the instruction throughput for the simulator developed in this chapter. There are three distinct sections to the graph: No-TLB No-Sleep refers to simulation where no MMU is modelled, no coherence filtering takes place (all memory accesses require coherency) and cores may not sleep (active spin-wait is requires for synchronisation). TLB No-Sleep introduces a TLB and coherence filtering pre-

sented in the next chapter, which allows conservatively detected non-coherent pages of memory to be accessed without involving the coherency protocol. Finally TLB Sleep adds the Wait-on-Address instruction presented in Chapter 7, which allows cores to be suspended while awaiting synchronisation operations, rather than use spin-wait operations. This has the side affect of drastically dropping the apparent IPC and instruction throughput of the simulator, because spin-wait instructions otherwise fill this waiting time with "work" for the simulator.

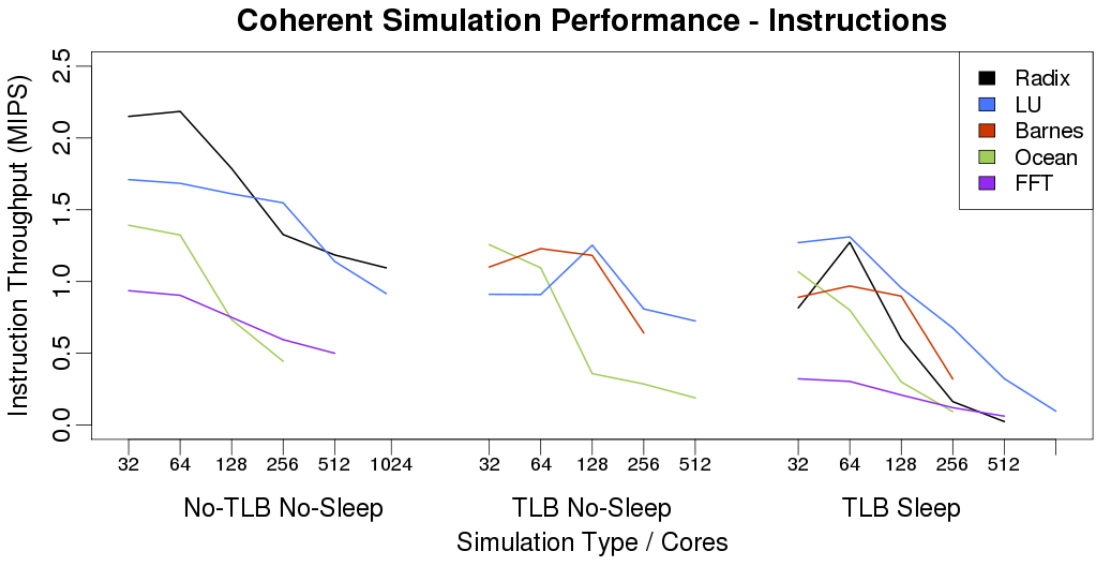


Figure 5.3: Instruction throughput of the simulator in different operating modes.

It is apparent from the initial No-TLB No-Sleep results that the presented simulation infrastructure, despite using only five host cores and using a tightly synchronised and cycle accurate interconnect model, is competitive with Sniper. The 32- and 64-core simulations of Radix, achieving 2.15 and 2.18 MIPS respectively, outperform the best quoted result from Sniper, although Sniper simulates a more complex core micro-architecture, while Arcsim provides greater interconnect accuracy. While the simulator appears to drop off in performance more rapidly than COTSon, this is due to the significantly more detailed interconnect model becoming a bottleneck, resulting in a drop off in IPC; this means that for a relatively constant simulation rate in core-cycles per second, the effective instruction throughput will drop. Despite this only Ocean and FFT drop to performance slightly worse than COTSon, simulating at 0.44 MIPS for 256-core Ocean, and 0.5 MIPS for 512-core FFT. Moving on to the TLB No-Sleep results, performance drops slightly, although unfortunately there are only two benchmarks in common between these data-sets. This is due in part to the increased simulation com-

plexity of resolving a TLB lookup and resolving the coherency filtering for every memory access, and because more instrumentation was added to the interconnect. However, simulation performance is still maintained at around 1 MIPS, on-par with the relaxed COTSon and Sniper simulations. This is while providing full, cycle accurate, timing correct data duplication for cache models, and calculating wire-accurate energy models based on the data transmitted over each wire in every flit, amongst other detailed modelling features. Finally the TLB Sleep results, which enable the cores to be suspended while waiting on synchronisation primitives, appear to perform significantly worse. As explained in more detail in later chapters, the SPLASH-2 benchmarks all spend an ever increasing proportion of their time in spin-wait synchronisation as the number of threads is increased. As a result, removing these instructions gives the appearance that the simulation is doing no work. The truth will become apparent on analysis of Figure 5.4.

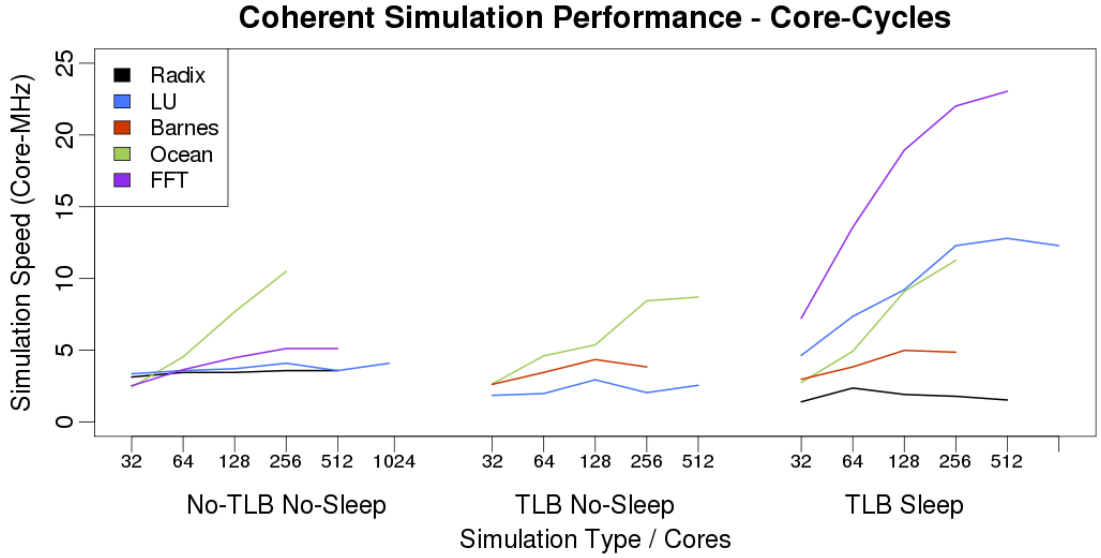


Figure 5.4: Cycle throughput of the simulator in different operating modes.

In Figure 5.4 results are presented for the core-cycle throughput, i.e. the number of cycles per second simulated by the simulator, multiplied by the number of cores being simulated, to give a figure in core-MHz. This metric is a good metric for how well the simulator is actually performing, since many aspects of the interconnect model also scale up linearly with the number of cores. A good result for a single threaded simulator would be a horizontal line. This would represent constant performance, with the increase in work for the extra cores resulting in a linear increase in the time taken to process it. A parallel simulator with a fixed number of worker threads such as the one

presented in this chapter should expect similar behaviour. Indeed, when WoA is not enabled, so cores cannot sleep, the simulation performance remains mostly constant at around 3-5 core-MHz, with the exception of Ocean which is likely to be spending an increasing proportion of its time fulfilling cache requests, running at maximum interconnect processing rate while the core threads are relatively idle. Because of the extensive optimisations performed on the network model, interconnect simulation does not take proportionally longer as more cores are added, and can remain relatively constant under the right circumstances. As such the core-MHz actually increases when cores are waiting proportionally longer on interconnect simulation. This behaviour is unique to Ocean however, with the TLB No-Sleep results corroborating this.

The interesting results are those where WoA is enabled. Although discussed in detail in Chapter 7, the WoA optimisation was initially conceptualized as a means for increasing simulation efficiency and performance. After witnessing the large number of processor cycles spend simulating the instructions for a synchronisation spin-wait a new mechanism for communicating to the simulator the intent behind the inefficient waiting was clearly needed. Wait-on-Address fulfils this need by explicitly informing the simulator that the processor is waiting for a specific cache line to change, enabling the core to be completely suspended from both the core simulation loop, and skipped over when simulating the core side of the cache model (external coherence events must still be handled for example). This has the extra benefit of dedicating proportionally more resources to the simulation thread doing the work upon which other cores are waiting, drastically increasing the simulation performance, and greatly shortening the time taken to complete the work. The TLB Sleep portion of Figure 5.4 shows that simulation performance can be over 4 times greater for the equivalent simulation, despite the initial appearances of Figure 5.3. Accounting for this the effective simulation rate of the presented simulator, for a suitable compiled benchmark, can in fact simulate significantly faster than the less accurate Sniper or COTSon simulators, with the only simulators providing better performance for the same accuracy provisions requiring FPGAs to perform the timing model, or both timing and functional model. Even these do not provide the same simulation detail, especially with regards to incorrect or relaxed coherence protocols, as the simulator presented here.

In conclusion, the presented cycle accurate, multi-threaded, full system simulator for large scale cache coherent architectures demonstrates that it is possible to build a performant coherent simulator for large scale milticore systems; especially in the case where the simulated cores have simple interlocked pipelines. The presented simulator

is fast enough, accurate enough, and collects enough information to be a more useful tool than the existing simulators when designing new features for new multicore processor systems.

Chapter 6

A Latency-Bandwidth Balanced Manycore Architecture

6.1 Introduction

Since technology scaling no longer yields increases in single-thread performance, future scaling relies on integrating more cores in a single chip. This is particularly prevalent in the power and thermally constrained mobile world, where over the past few years there has been a rapid progression from single low-powered ARM cores, up to eight relatively high-powered ARM cores with a handful of DSPs and other accelerators. However, despite moving on to multicore architectures, as of 2013 the individual cores were still an order of magnitude less powerful than the desktop and server processor found today [132; 133; 134]. By 2015 this gap has been closing, especially with newer 64-bit ARM processors such as the A57, but many of these benchmarks are performed at maximum operating frequencies, which cannot be sustained in the thermally restricted operating environment of mobile processors. This move to multicore processors is because larger cores require significantly more energy per unit of computation that their smaller brethren and so, where sufficient parallelism exists, near-linear throughput gains can be made with near-linear power expenditure by adding more cores. Most programs written today assume a coherent view of memory, and a fairly strict memory consistency model. Keeping each core's view of memory "coherent" has many possible solutions, which revolve around either direct core-to-core communication (e.g. snooping, which works well for small core counts), keeping sharer information in the last level cache tags, or the introduction of a new on-chip structure, the coherence directory.

With modern fabrication nodes and tiny embedded microprocessor cores delivering compelling performance, it is possible to design a homogeneous general-purpose manycore, with 1024 cores on a single die of similar area to a typical server processor [73]. In order to run existing parallel code, with minimal modification, cache coherency must be a first class citizen, and power consumption will be very important.

However, without accurate, high performance simulation tools, such as those developed in the previous chapters, performing accurate evaluations of any hypothetical designs must either take far too long to enable iteration and design space exploration, or be so inaccurate that any performance comparisons have low statistical confidence.

As systems are scaled ever larger, maintaining cache coherence with strong memory consistency models becomes an increasing bottleneck to parallelism and system performance [34], so a new platform should provide a means to avoid the coherence system if possible enabling future programs to be written with as little strong coherency as possible. Eventually this could allow future manycore processors to provide only a simple, small coherency mechanism for occasional use, rather than provisioned for use with every memory access.

This chapter proposes such an interim architecture, which provides strong sequential consistency by default, but allows memory regions to be accessed with the coherency infrastructure disabled, and can automatically handle the migration of memory regions from private and shared incoherent and shared coherent states without violating sequential consistency. The novel simulation technology developed in the previous chapter is then used to asses the new architecture's performance and power scalability.

6.2 Architecture

The proposed design is a hybrid tree-mesh architecture, connecting many small embedded embedded RISC processors based on the ARCompact instruction set [135]. These cores combine low die area with high cycle efficiency, and are thus ideal for manycore systems. Each core has 4KB of direct-mapped instruction and data cache, with only the data cache connected to the coherency infrastructure. At 28nm it is possible to fit 1024 cores on a single die. These cores can operate at frequencies of up to 1GHz, so the proposed architecture is operated at a conservative 800MHz. The data network features a narrow bandwidth tree, truncated close to the roots with a wide high-bandwidth mesh, while the coherency network is separate, and unusually is based on a binary tree with a single root. Many mesh architectures distribute their directory

with their last-level cache (LLC), and insist on an inclusive cache architecture [136]. Instead a separate, centralised directory is utilised, which allows optimisation of the LLC contents, not explored in this thesis. An overview of the 64-core example architecture can be seen in Figure 6.1, and a summary of the configurations simulated can be found in Table 6.1. Milo *et al.* argue that a hierarchy of fully inclusive caches, with coherency state tracked in the tags would be a good way to achieve full chip coherency for a manycore processor [136], but when applying their three level cache hierarchy to a 1024 core design of even small 4 KB caches like those in this thesis results in a 4 MB 1024-way set associative last-level cache (LLC), without accounting for instruction caches. Ignoring at first the associativity problem, a three level cache hierarchy would use 12 MB of cache for this processor, which for a very data parallel problem would only provide 4 MB of useful cache. To avoid the associativity problem Milo *et al.* suggest using an over-sized but less associative cache. This only solves the problem when there is extensive data sharing or the data is carefully aligned to avoid conflicting in the last level cache, otherwise the utilisation of the L1 caches will be restricted. Even if a four times over-provisioning addressed the associativity concerns, a three tier cache hierarchy with 4 MB of total L1 and L2 cache now requires $4+4+16=24$ MB of total cache capacity, while still effectively only providing each core with 4 KB of usable cache. By separating the coherency concerns into a separate directory (with potential for a hierarchy of directory caches), the cache hierarchy is free to be non-inclusive, and the die area can be retained for cache which can be better utilised, removed to shrink the processor, or used to provide other features such as accelerators or complex IO interfaces. By isolating the coherency and data-caching concerns it is also possible to use less on-chip memory to store the coherency information, because data can be stored usefully in cache, without the need for tracking coherency information, so long as the software takes responsibility for tracking it, or it is only in the LLC. It also moves the associativity problem from the data hierarchy, where much more information-destructive compression techniques can be used not just on the sharer state, but also on the address tags [64].

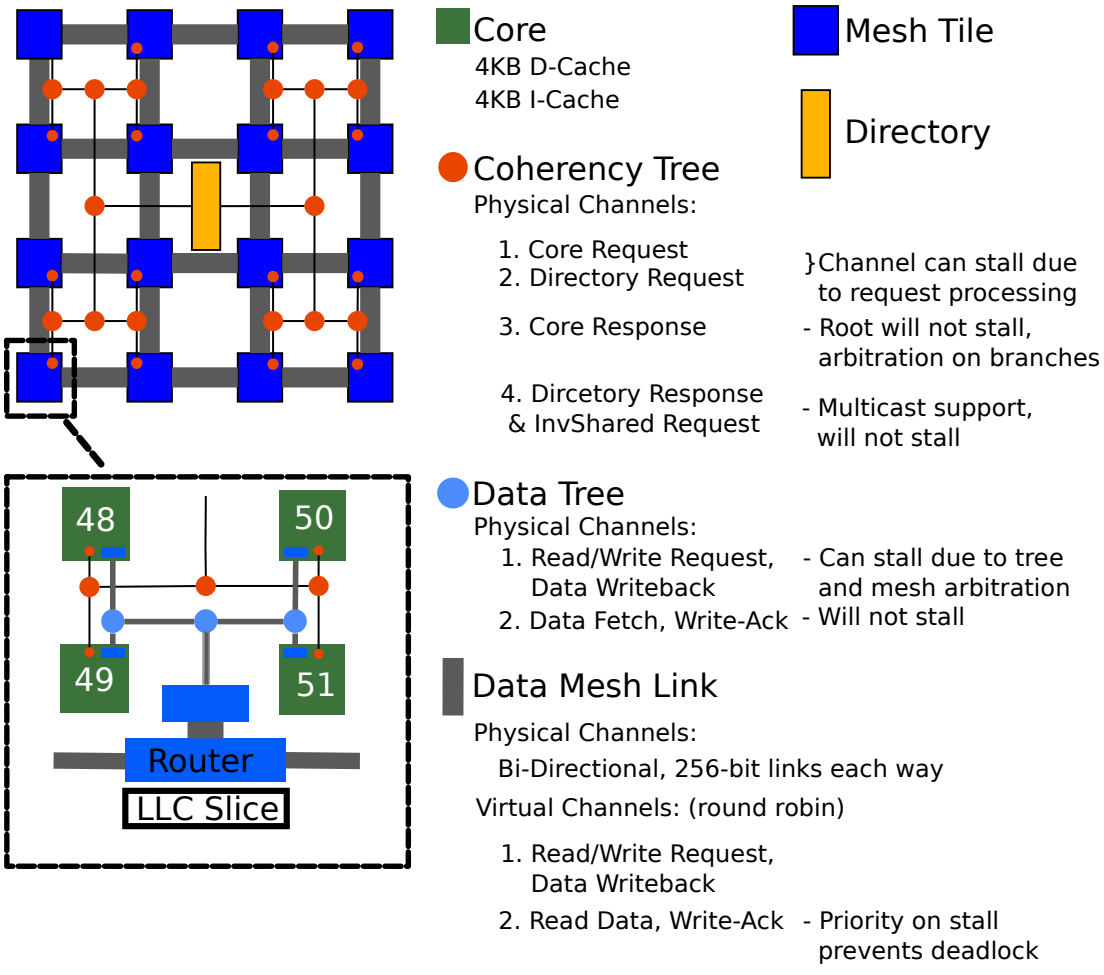


Figure 6.1: Overview of the simulated architecture.

Cores	32	64	128	256	512	1024
Mesh	4×4					
Mesh Width	256 bits					
Cluster Size	2	4	8	16	32	64
Tree Width	64 bits					
Directory Associativity	32	64	128	256	512	1024

Table 6.1: Architecture Summary.

6.2.1 Design Decisions

There are many reasons for selecting this tree based architecture, mostly relating to latency, logic area, and energy. Simplicity was also a key factor: because the coherency interconnect is ordered, it simplifies the coherency protocol, making it easier to reason about and also enabling protocol optimisations.

With a 2D mesh architecture the node to node communication latency is on average $(2/3)\sqrt{N}$, while a binary tree (or H-tree) scales as $\log_2(N)$. For 16 cores, a mesh has a better average hop count of only 2.7 (maximum 6), while the tree is 4 in all cases. However, the tree is more attractive at 64 cores, with a mesh giving 5 hope on average, but a maximum of 14 compared to a tree's 6 hops. For 1024 cores the mesh has more than double the average hop count of a tree, with a worst case 6 times greater than the tree, which still only requires 10 hops. The latency and wire distance comparison for various sizes of mesh and tree can be found in Table 6.2.

Mesh Size	**Mesh Hops**	**Tree Hops**	**Tree Distance**
2×2	1.3	2	1
4×4	2.7	4	3
8×8	5.3	6	7
16×16	10.7	8	15
32×32	21.3	10	31

Table 6.2: Average hop distance and wire distance for mesh and tree networks.

The logic area advantage comes from the fact that the binary tree router is extremely simple compared to a 5 port mesh router. While a tree and 2D mesh architecture both require approximately the same number of routers, each mesh router must be significantly larger than the corresponding tree router. This is caused not only by the fact that the mesh router must arbitrate significantly more physical directions, it must handle virtual channel arbitration on each bidirectional link, while in this case the tree does not require virtual channels.

With many cores attached to each mesh node, there can easily be enough request packets sent to a particular mesh node's LLC slice that it fills the local buffer and jams the network on links heading towards the node. If this happens to two nodes, A and B, and a response packet from B must head to A to be routed back to its target core, the network can deadlock. To avoid this, "up" and "down" packets must be separated into virtual channels, with buffer allocation in the routers to allow for "up" packets returning to cores, to proceed even when the "down" channel has saturated and jammed.

Because the tree does not share links between "up" and "down" channels, directionality is built into the tree; there is no need for virtual channel allocation, or buffering (other than to drive the flit onwards on the next cycle). If coherency traffic ran over the same wires as data then VC would be needed to arbitrate between them, and due to the coherence messages being much smaller than the data bus flit size (to allow fast cache line transfers) it would also be a large waste of potential bandwidth, and incur a higher routing energy overhead than necessary.

The energy difference comes again from the complexity of the larger switches required in a mesh. Unfortunately the wire distance travelled on average for meshes is less than for large trees – while the mesh architecture requires $2/3\sqrt{N}$ hops on average, an H-tree requires exactly $\sqrt{N}-1$ hop-length equivalents, so the latency and routing energy must pay off against the 50% greater energy cost for the tree.

As a tree cannot provide the required bandwidth for such a large architecture, it makes sense to leverage the 2D mesh at the lowest level of the tree, to handle the high bandwidth data movement across the chip. A 4×4 mesh provides 4 times the bisection bandwidth using links of the same length as the tree roots it would replace, while an alternative such as a butterfly or omega network would introduce even longer wires. A mesh also provides natural power islands allowing sections of the chip to be turned off to conserve power.

6.2.2 Data Interconnect

The 4×4 mesh carries two virtual channels, one going from cores to L2 caches, and the other returning from caches to cores. They are given equal priority, using a toggling arbiter when a whole packet has transmitted, but if a core to cache packet is blocked and there is a cache to core packet waiting for the link, it will back-off into a storage buffer in the source router and be exchanged for a cache to core message. This is done to prevent deadlock in the network, when too many requests prevent the responses from exiting the mesh. The mesh routing uses fully provisioned buffering for both virtual channels, once a packet has been accepted it can receive the whole packet even if the next link is not available yet. This allows the virtual channel back-off to function without splitting packets, which can be up to 3 flits for a 256 bit bus width, and 64 byte cache line size.

Each router consists of a 6×6 crossbar (5×5 from the perspective of each virtual channel) with each virtual channel output buffer doing its own round robin arbitration of the input sources, and the final output multiplexor arbitration performed on these buffers, as shown in Figure 6.2.

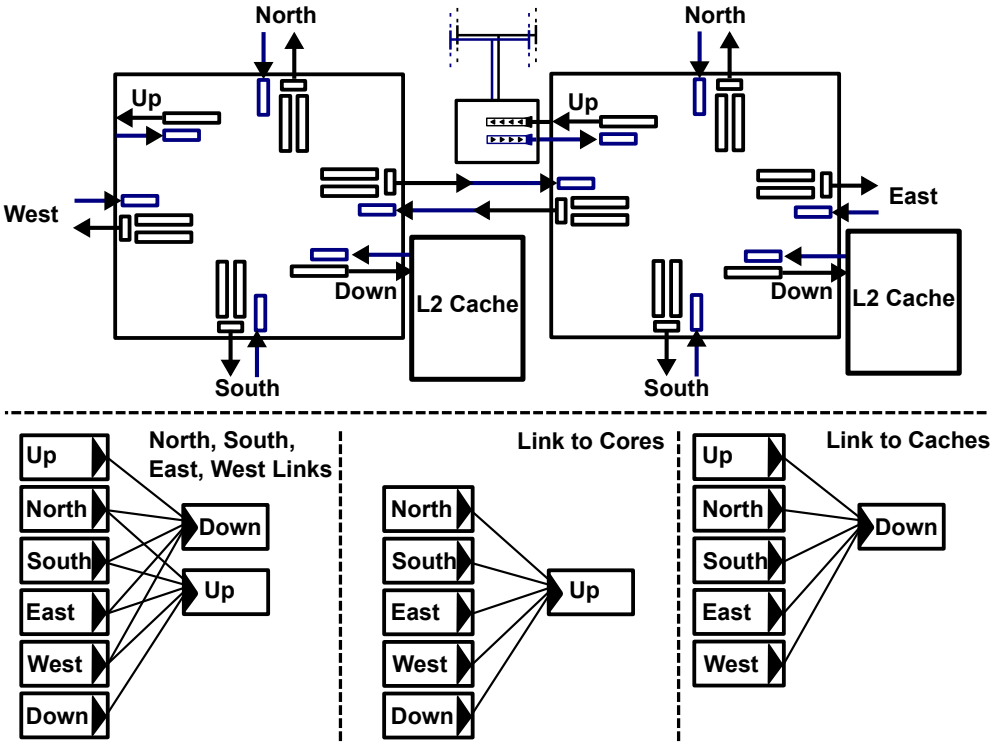


Figure 6.2: Overview of Mesh Router Micro-architecture.

6.2.3 Coherency Tree

A point of potential controversy in the proposed design is the use of a single tree to provide the directory. At first glance this is a serious bottleneck and might appear to have long wire issues which were avoided in the data interconnect via the use of a mesh. The longest wires in the tree, however, are exactly the same length as those used in the mesh (and significantly narrower) and the use of a single directory in the design brings a large simplification to the coherency protocol: invalidation acknowledgements are not required for the invalidation of a "Shared" cache line (AckIS messages). The tree architecture also enables efficient multicast opportunities, which combined with the lack of AckIS messages, allows for both a large saving in interconnect packets, and subsequently energy, and the streamlining of the directory operation itself, which can greatly reduce the latency of requests.

To quickly address the bandwidth fears, the tree architecture is amenable to a hierarchy of directory caches at higher points in the tree, which operate like the caches in a traditional memory hierarchy. These reduce the strain on the central directory without violating the (effective) single serialisation point at the root of the directory. Once again this is a feature omitted from the simulation to reduce complexity and help ensure correctness of the implementation. The bandwidth required by the proposed directory is also significantly less than that required for a traditional distributed directory in that the directory does not need to process AckIS packets, nor does it have to send multiple unicast messages for shared invalidations. Other opportunities to reduce coherency bandwidth demands are discussed in Section 6.3.

The coherency tree is separated into two down channels, and two up channels. From core to directory there is a channel for coherency requests, and a channel for responses. The separate response channel is required to allow the responses to overtake requests, to prevent deadlock; it would be possible to implement these as virtual channels on the same physical network, but for simplicity it is simulated as a separate physical network. From directory to the cores the two channels are quite different: there is a non-blocking channel which progresses every cycle, cores guarantee to handle messages off this network within one cycle, and a blocking channel which carries messages which cannot guarantee a single cycle processing time, or which might have a dependency on putting a message onto a return channel, which could be blocked. The non-blocking channel supports multicast, and carries acknowledgement messages back to core messages, along with invalidate-shared messages (which are multicast)

and invalidate Shared Read-Only TLB entry (which is full broadcast). The blocking channel is left to carry requests which could result in cache write-backs: invalidate exclusive and flush page messages. Again it would be possible to use virtual channels, but it is much easier to manage the strict requirements of the non-blocking channel if it does not also have to handle virtual channel arbitration, so it is modelled as a distinct physical network also.

It should be noted that the definition of non-blocking used in this chapter is not the same as that traditionally used with interconnects. In the context of this chapter non-blocking means that a single unit of forward progress is made on every cycle, with no potential for arbitration or stalls. This allows all routes in the interconnect to progress synchronously.

6.2.4 Coherency Protocol

The proposed coherency scheme is based on the MESI protocol, with directories maintaining the states Invalid, Shared or Exclusive for each stored line, and L1 caches containing the states Invalid, Shared, Exclusive and Modified. The Exclusive state could be merged with the Modified state by sending a "shared" acknowledgement (AckS) from the directory upon read requests even when it is the first and only sharer, which could save on latency in an application with a large component of shared read only data.

The key distinction between other protocols and the one presented here is that the directory transition Shared→Exclusive has no intermediate state, where one must wait for invalidation acknowledgements from the sharers. The directory immediately sends out an invalidation multicast on the first cycle (ignoring directory lookup latency) after a request, and on the subsequent cycle can send the exclusive acknowledgement (AckEx), storing the new state immediately.

This works because the acknowledgement messages are sent on the same non blocking channel as the invalidate-shared message, and from the serialisation point at the centre of the tree all cores will receive the InvS command at least one cycle before the AckEx message arrives at the requesting core.

Like any sequentially consistent (SC) protocol, the Exclusive to Shared transition must still wait on the response from the current owner, to indicate that the LLC value is current and it has downgraded its copy to Shared. Here the protocol must only wait for the response of an individual core, however, not potentially 1024 responses.

This brings us to the internal storage of the proposed directory: to store every sharer exactly would require N bits per line, which is quite infeasible on the scale of 1024-core systems. Because the proposed policy does not use shared invalidation acknowledgements, it does not matter if sharer cannot be removed from the sharer vector, such as a with coarse vector representation. This allows the protocol to use a lossy sharer compression without worrying about keeping track of the number of sharers (AckWise) or requiring AckIS messages from all receivers of an invalidate message – in fact, the protocol does not even need to know how many cores received the message – it can simply send and forget. If the directory was distributed and/or AckIS messages were required it would still be fairly straightforward to perform acknowledgement aggregation in the response trees to reduce energy and bandwidth requirements. The directory requires a minimum of $\log_2(N)$ bits per cache line, to store the owner of an Exclusive cache line, and 2 bits for coherency state. This makes the best case space storage $O(\log_2(N))$. The naive storage requires N bits, one for each sharer, but there are a few existing ways to represent the sharer vector in an $O(\log_2(N))$ representation as discussed in Chapters 2 and 3:

1. Store a limited number of core pointers (enforce this with invalidations, ADir$_k$NB, or fall back to broadcast when full, ADir$_k$B [56], and the extension of this which counts sharers to reduce the required AckIS messages AckWise [57]).

2. Store a coarse vector representation [61] (easiest with power of 2 bits allocated to the sharer vector, but possible with any number of bits).

3. Store a hash of the sharer pointers, like a Bloom filter to filter out cores that should not receive invalidations. For example, expand each sharer pointer into a $2\log_2(N)$ bits representation where $0 \rightarrow 01$, and $1 \rightarrow 10$, then OR these symbols together to generate a mask of cores which should receive the message, as proposed by Agarwal [56] and later given the names DirX [61] and Tristate [59; 60]. This can be multicast on the tree quite easily – 01 means go left, 10 means go right, 11 means take both branches.

Each of these has its own set of problems:

1. These must send a unicast message for each sharer, or fall back to full broadcast, neither of which are ideal from an energy or bandwidth perspective.

2. If $2\log_2(N)$ bits are used for the vector, then each bit represents $N/(2\log_2(N))$ cores, so as N grows, the number of sharers represented by each bit increases

dramatically – meaning many false invalidation requests for a small number of sharers.

3. This method is prone to pathological sharer combinations, which can result in only two sharers aliasing to cause a full chip broadcast; Section 7.4.2 demonstrates this problem.

6.3 Reducing Bandwidth and Directory Size/Associativity Requirements

Coherency bandwidth is an important problem when scaling up an architecture, and a centralised directory, even when throughput optimised, can only handle about one request per cycle. The proposed scheme as described so far works to reduce the traffic in the protocol: a single packet is sent for any invalidation (multicast), and at most only requires processing a single invalidation or share acknowledgement for each transaction (no need for AckSI). This still means each core can only really make a request to the root directory about once every 1024 cycles. With small data caches, which are required to fit 1024 cores on a chip, the rate of cache misses is likely to be much higher than this, but there are three obvious ways to reduce directory strain:

- Introduce directory caches in the tree, to merge and filter requests to shared data, and to catch capacity/conflict victims.
- Introduce another localised coherency mechanism, such as Proximity Coherence [52].
- Track whether or not data needs to be kept coherent at all [67].

The first option would not be very difficult to incorporate in the proposed architecture, with perhaps 4, 8, or 16 caches placed at the appropriate level of the tree, each entry would hold the local sharer vector, the coherency state, and an additional bit to indicate if the cache line was currently private to this branch of the tree (seen as Exclusive to the root, allowing the cache to transition between Shared, Exclusive, and Modified states independently) or present in other branches (requiring a request to be propagated to the root for escalation of a cache line to Exclusive). The architecture above this point would be identical to that described so far, but the issue of the

inclusiveness of the cache must be decided. The options are to either enforce inclusiveness, requiring a large cache or limiting the local coherent data capacity, or to be non-inclusive and make a best effort to maintain local sharer state. When an eviction is required it would likely make sense for cache lines which are broadly shared to be evicted, where resulting in a broadcast would not be much more inefficient than the current multicast state. The root directory would simply hold a full sharer vector of the directory caches, and not the tree representation above them. This architecture would also allow the operation of a more distributed directory such as the ring based ATAC-1000 to operate with an AckSI based protocol at the lowest level, but getting those small number of Ack messages quickly from the directory caches, which use the same non-blocking multicast/broadcast scheme described above.

Because the architecture demonstrates scaling potential without these caches, and their presence is not necessary for the research investigated in the rest of this thesis, the implementation and benefits of these caches are not investigated further.

The second option, as discussed in Section 3.1.3, is initially incompatible with the ack-less shared-invalidate phase of the proposed coherency protocol. However if the sharer encoding used a coarse vector or other clustered approach then because all cores within a cluster would receive the invalidation request, the chaining of Shared cache lines would be unnecessary, they can be forwarded freely within the cluster. This change requires the Proximity coherence only forward Shared cache lines within a cluster, but in doing so simplifies some of the Proximity Coherence protocol, and enables it to be used with an architecture like the one proposed here. Evaluating such a complex addition to the protocol is outside the scope of this thesis however, and would require significant modification to enable data-forwarding within the architecture.

This third option has the least overhead in introduced chip area, and can be implemented explicitly at the programmer/compiler level, or transparently, at the OS and MMU level. This was implemented in the proposed platform by adding a software handled TLB, which additionally tracked the state of each 8KB page, either Private, Shared Read-Only, or Shared Read/Write. All pages initially start in the Untouched state, moving on first access to the Private state when a core accesses them. In this state the page table entry not only stores the sharing state (and physical address), but also the core which has private access to the page. When another core wishes to access the page, it must lock the page table entry, then transition to one of the shared states by sending a special page flush message to the core which is holding the page private. This causes the target core to invalidate the TLB entry, and walk its L1 cache writing

back all of the dirty cache lines from this page. The core then sends back an acknowledgement informing the new core it has completed, and the core may now insert the page into its own TLB, and update the page table state before removing the lock on the page table entry. If it is a read operation the page transitions to Shared Read (e.g. single core initialised read only data), and then any writes will transition it to Shared Read/Write. If a Private page is written to by another core it is directly transitioned to Shared Read/Write. With this scheme in place only Shared Read/Write traffic needs to go to the directory, private memories such as program stacks and other thread local storage, along with static read-only data, are automatically detected, and the MMU can dynamically determine if it needs to perform a coherency operation on a cache miss. This also has the added side affect of reducing the required directory storage, and as shown later in this section, greatly reduces the required associativity. Without this, an alternate scheme for dealing with directory associativity would have to be used, such as one of the related works in this area discussed in Chapter 3. The page transitions are summarized in Table 6.3.

Page State	Read	Write
Untouched	→Private Owned	→Private Owned
Private Owned	Success	Success
Private Other	→Shared R-O	→Shared R/W
Shared R-O	Success	→Shared R/W
Shared R/W	Success	Success

Table 6.3: Transition events for Read and Write accesses to different Page states.

Implementing this scheme, the TLB sharing state is checked upon every cache access. With a TLB state of Private or Read-Only, a cache line in any of the valid coherence states, or the Incoherent state, is valid for a read, while in the Read/Write state only coherent states are valid. For writes, the TLB state must be Private or Read-/Write, or a page fault is raised, and similarly Read/Write requires the cache line to be in a coherent state (specifically Exclusive of Modified), while the Private state may write if the cache line is Incoherent, or Exclusive or Modified. Arriving at a cache line in the Shared state for a Private line is only possible with an MSI coherence protocol (when a private cache line is accessed with filtering disabled, which can happen to the

stack on TLB faults), or through an error of the runtime system, and this cache line would be migrated to Incoherent (dirty), invalidating the coherency state of the line.

To handle thread migration, a solution could enforce some pages to be Private (such as the program stack), and instead of transitioning them to a Shared state upon access from another core, force the cache entries to be flushed from the current owner (as would happen anyway) and transition the page to new ownership, instead of a Shared state. Alternatively, when preparing to migrate a thread, the current owner thread could flush its caches and walk the page table to clear ownership of any Private pages it is holding. This would allow any new core it was scheduled on to take ownership of the pages it needed, without having to wait for inter-core page flush commands to complete.

Cache lines can become temporarily marked with coherency information even if they are in the Private state, because upon page faults the runtime system must disable

On Read	Cache Line Coherence State				
Page State	Invalid	Incoherent	Exclusive	Modified	Shared
Invalid	TLB Miss	TLB Miss	TLB Miss	TLB Miss	TLB Miss
Private	Incoherent Fetch	Success	Success	Success	Success
Shared R-O	Incoherent Fetch	Success	Success	Success	Success
Shared R/W	Coherent Fetch	Coherent Fetch	Success	Success	Success

On Write	Cache Line Coherence State				
Page State	Invalid	Incoherent	Exclusive	Modified	Shared
Invalid	TLB Miss	TLB Miss	TLB Miss	TLB Miss	TLB Miss
Private	Incoherent Fetch	Success	→Modified	Success	→Incoherent
Shared R-O	TLB Miss	TLB Miss	TLB Miss	TLB Miss	TLB Miss
Shared R/W	Coherent Fetch	Coherent Fetch	→Modified	Success	Coherency Miss

Table 6.4: Transition events for accesses to different cache states when page filtering is active. Note that the Private page Shared coherence state can only be reached in a corner case when using the MSI protocol, and is not a reachable state when using MESI. When using a kernel stack to handle TLB misses, Private and Shared R-O cannot reach a state with coherent cache lines at all.

the coherency filtering and handle the fault conservatively, using coherency protocols for all memory accesses. When returning from the page fault handler some cache lines in the stack, which will likely be Private, may have been fetched as coherent. This only happens because the experimental runtime does not use a dedicated kernel mode stack when handling TLB misses, and instead pushes registers onto the application stack to execute the handler. If a proper kernel mode switch was used, or dedicated hardware logic handled the TLB transitions, then the Incoherent page states Private and Read-Only would not have the potential for coherent cache lines. The access permissions and actions are summarised in Table 6.4.

6.3.1 Benchmarks

To investigate scalability and other features of the proposed architecture the parallel benchmark suite Splash2 [137] is used, unfortunately architecture and runtime support have limited the experiments to a subset of the suite. The selected benchmarks and their configurations can be found in Table 6.5.

Benchmark	**Configuration**
Radix	1048576
FFT	2^{20} points
LU (contiguous)	512x512 Matrix
Ocean (contiguous)	258x258 Ocean
Barnes	16384 bodies

Table 6.5: Benchmark Configurations.

It has been shown that these benchmarks do not scale as far as 1024 cores [49], but the benchmarks should still usefully highlight performance trends up to 512 cores, and provide a platform for further innovations presented in Chapter 7.

6.4 Architecture Scalability Analysis

The experiments into scalability were conducted using a Course Vector sharer compression of $2\log_2(N)$ bits. Alternate encodings which provide multicast capability would have negligible impact on performance, but may be significantly better or worse

in terms of energy scalability for the multicast energy contribution. Unless otherwise stated the associativity of the directory is equal to the number of cores in the system, and each way contains 64 entries, corresponding directly to the 64 entries in the 4 KB direct mapped L1 data cache. The directory replacement policy when there are no free entries is pseudo-random, with a way-counter incrementing every time a capacity eviction must be performed. There are more intelligent schemes, such as preferentially evicting shared entries because they are lower latency and may be stale, where exclusive lines are definitely live and must wait for an acknowledgement (this policy would work well for the case where directory associativity is equal to the total L1 cache associativity, but does not address how the shared line is then selected for eviction). However common cache policies like least-recently used (LRU) do not always work well at a directory level, because the measure of activity at the directory does not correlate with the activity of this cache line in the L1 cache. Because of this pseudo-random was chosen as a simple to implement and fair invalidation policy, which should invalidate with an equal distribution across the sets, being relatively robust against pathological cases.

6.4.1 Performance

Accepting that an architecture so far from the traditional multi-core system that these benchmarks are targeted for will not perform equally well across the benchmark set, the initial analysis will focus on one of the best performing – LU. This benchmark is also the focus of more detailed analysis because its shorter runtime makes it more amenable to iterative experimentation, enabling some code generation issues to be analysed and corrected which were not for other benchmarks; the shorter runtime also enabled collection of results up to 1024 cores, which were not possible due to time or benchmarks constraints for many of the other benchmarks. Although the analysis focuses on LU, equivalent figures for most of the other benchmarks can be found in Appendix Section A.1.

Before the detailed look at LU however, Figure 6.3 presents the performance of the Splash2 benchmarks across the design space, giving an indication of performance scaling and a means of comparison with results in other work, while Figure 6.4 presents the instruction throughput per cycle of the target platform while executing the parallel section of the benchmark. The IPC results are from the simulations with efficient spin-waiting enabled, so instructions are almost entirely those used to do computation.

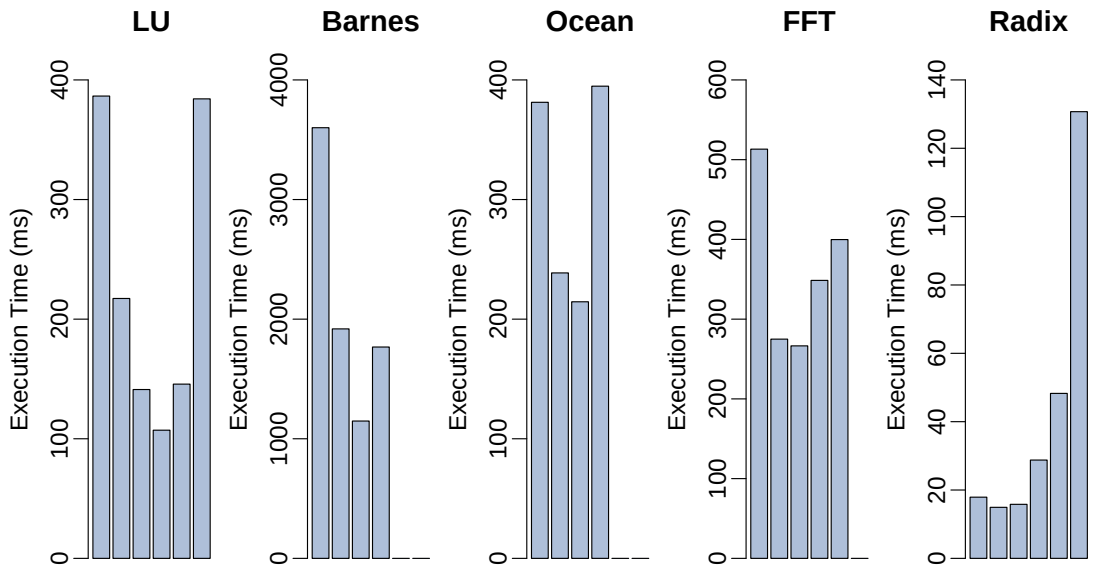


Figure 6.3: Execution times of the parallel sections of Splash2 benchmarks on 32 to 1024 core systems.

An increase in IPC should therefore be correlated with an increase in performance, and vice versa. The poor IPC performance of Radix suggests that it either exhibits extremely poor cache performance, or spends a large amount of time on spin-wait operations, either of which is likely to hamper it's performance and scalability on the proposed architecture.

Starting the analysis of LU with its scalability performance, shown in Figure 6.5, it is apparent that the proposed architecture does not scale particularly well with even with this benchmark, scaling sublinearly up to 256 cores, then actually reducing in performance with 512 and 1024 cores due to bottlenecks in the IO system. However, achieving a speedup of 71x over the single threaded version (which does not require any cache coherency overhead) is not an insignificant result.

6.4.2 Cache Aliasing

Using such small, direct mapped, caches introduces a problem not encountered often on modern high performance cores – instruction and data cache aliasing. This is where two frequently accessed data elements or instructions are mapped to the same entry in the cache, and the regular evictions and misses due to this conflict results in a large volume of memory traffic and very poor program performance. In the case of the LU

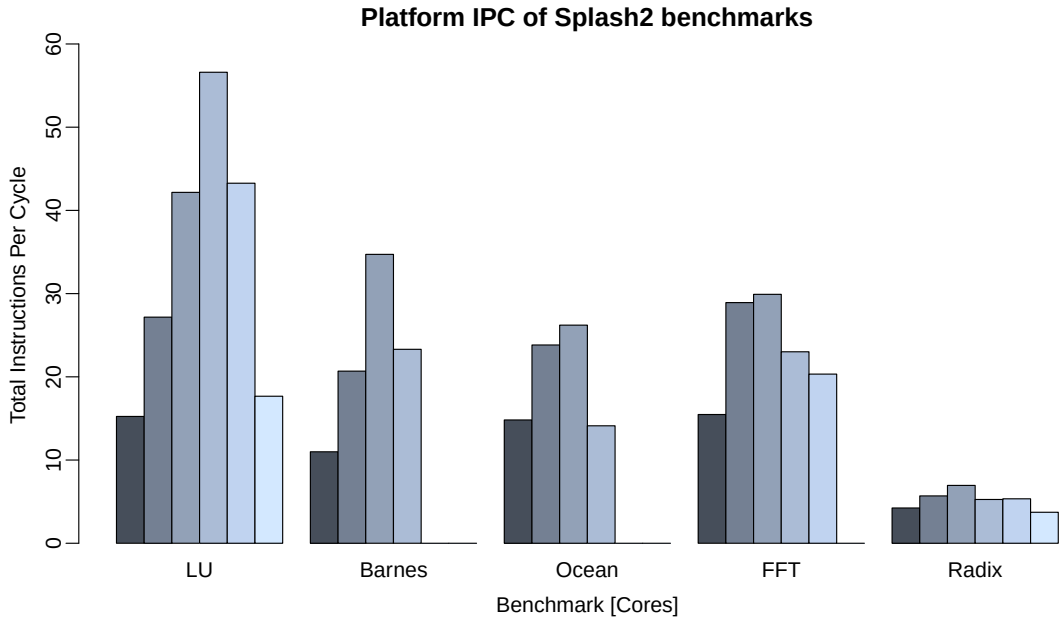


Figure 6.4: IPC of the parallel sections of Splash2 benchmarks on 32 to 1024 core systems, excluding spin-wait instructions.

benchmark the compiler naively places two of the main kernel functions either side of a large body of code which is either called once or never, for benchmark setup and validation purposes, resulting in the two main kernel functions largely overlapping in a small area of the L1 I-cache, while the rest goes unused. By manually relocating the code segments to be adjacent, the cache hit rate was improved from around 99.45% to 99.98%, and the affect of this can be seen most clearly by looking at the cache line fills requested from the data and instruction caches, shown in Figure 6.6. This shows the proportion of cache line fills performed relative to the total number of requests made by the I-cache optimised 32-core LU benchmark, across the design space of 32 to 1024 cores. It highlights the stress a small increase in instruction cache miss rate can put on the data interconnect, and helps solidify the argument made earlier that the instruction cache should be excluded from the coherency mechanisms. Cases where code coherency is required, such as self modifying code, JIT compiled code, and kernel mode execution loading a program binary, are already well handled by software coherency techniques. These use page protection schemes to trap on writes to code pages, and force an invalidation of any affected I-cache entries (on many architectures this requires a full I-cache flush).

Looking at the trends of Figure 6.6 it is also apparent that the total traffic require-

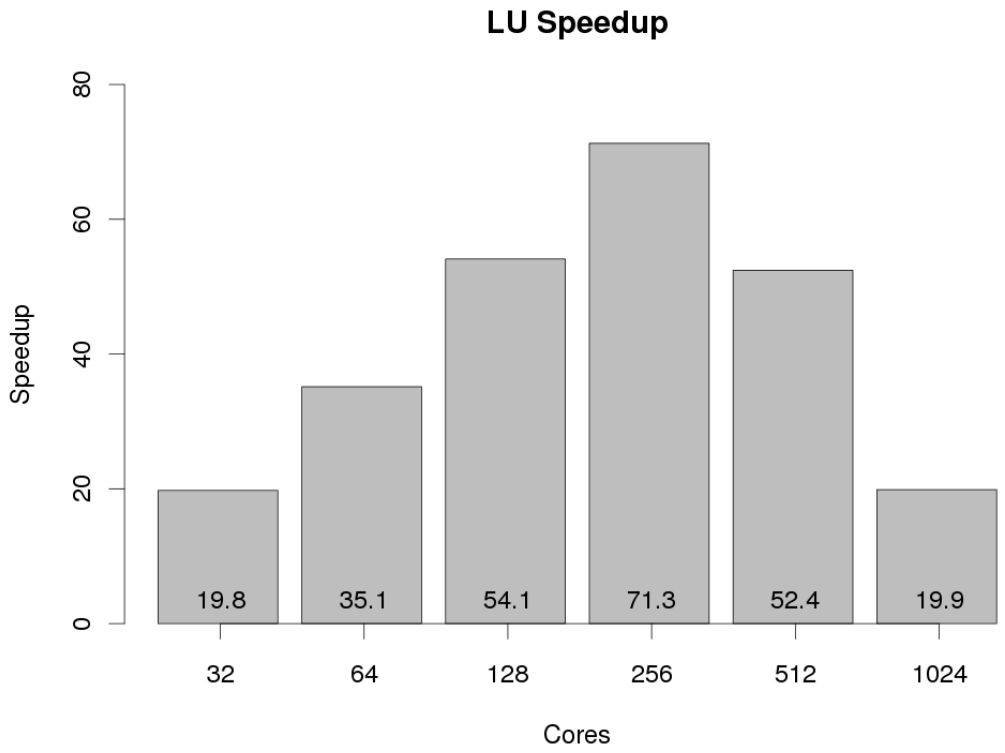


Figure 6.5: Speedup of LU benchmark relative to a single thread running on a 32-core architecture (after addressing instruction cache aliasing problems discussed in Section 6.4.2).

ment for the benchmark is almost constant as the number of cores scales towards 512. At this point it begins to increase, in line with the rapid deterioration of scalability shown in Figure 6.5. It is likely that these are related, and with 512 and 1024 cores the work elements assigned to each core are small enough that they either suffer from false cache line sharing, or the computations require data sharing that was unnecessary with the larger work elements. The increased I-cache misses also resulted in a 10 to 30% performance reduction.

Sadly LU was not the worst off by far, with Barnes suffering an abysmal I-cache hit rate of only 97.4%, i.e. every 40th instruction resulted in a cache fetch, which given the average instruction cache miss fetch latency on a 64-core system running Barnes of 92 cycles, means a CPI of over 2 before any coherency or data misses are considered. This extreme ratio of instruction to data traffic is shown in Figure 6.6 from which it is clear interconnect bandwidth and overall performance will be severely impacted by the volume of instruction traffic.

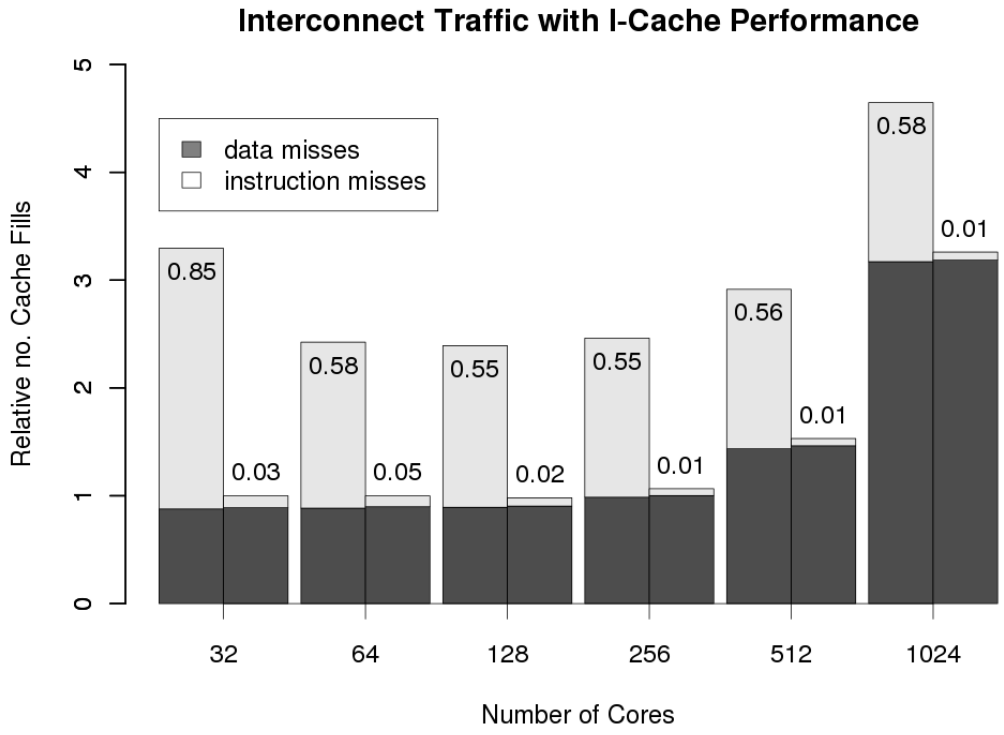


Figure 6.6: L1 cache fills generated by the LU benchmark before and after addressing I-cache aliasing problems. Bars are annotated with the I-cache miss rate (%).

Unfortunately the instruction cache issue was not noticed until after the experiments had been run and data analysis begun, so there was not enough time to manually tune and re-run all of the benchmarks. This means that the results for other benchmarks in this thesis are sub-optimal. There is however active research into helping compilers automatically pack program code in a cache aware manner, which could be used to avoid this problem on a real implementation of this platform. This research not only targets the direct mapped caches from the proposed architecture, but more associative caches, enabling the best use of the available L1 cache [138]. In the manycore scenario, using more associative or larger caches as an L2 I-cache for smaller clusters of four to eight cores may also help to alleviate the problem, and allow the cores to use even smaller caches, while leveraging the shared cache to avoid the large performance and energy penalty of fetching from the LLC. If this tier of cache was only used for instruction caching, the coherence protocol would not need to be involved at a hardware level, so long as instruction cache flush or invalidate operations were propagated to the shared caches. Other, currently unpublished, work suggests that support could

be provided to allow code generation, as occurs in a JIT compiler, to write directly to this shared I-cache to improve performance of JIT compiler based runtimes, like the JAVA virtual machine. In a production version of this design it is likely that a combination of compiler assistance and hierarchical caching would be used to maximise the performance of the resulting system. Fortunately many of the experiments conducted in the rest of the thesis are not badly affected by the I-cache performance, as they focus on the data memory operations in isolation.

6.4.3 Energy Scalability Analysis

It has already been shown in Figure 6.6 that for 32 to 256 cores the number of cache fills, and hence total interconnect traffic, remains constant, but this is not necessarily an indicator of constant power consumption. The four main energy contributions are:

1. Active core energy, cycles when the core is not sleeping or stalled on IO.

2. Static power (transistor leakage for the whole chip, constant power overhead).

3. Switch routing energy, the energy used to perform the routing of each network packet through each switch in the network.

4. Wire energy, the energy used to transition the state of an interconnect wire between states when sending a packet over a network link.

In the best case, the active core energy should be constant, with the same amount of work distributed fairly amongst all of the cores. Static power is a trade-off between the increased leakage of more transistors, versus the reduced runtime due to the increased parallelism. Unless the benchmark achieves perfect speedup, the static power will increase proportionally to the inefficiency of the parallelism in the benchmark. Because adding more cores requires extra switches in the network, and more switches for a packet to navigate when in flight, the switching energy will increase in line with the average hop count of the network. However, the proposed design increases the number of hops through light-weight radix-2 switches, and keeps the number of large radix-5 mesh routers constant. This means that with constant total traffic, the mesh routing energy will remain constant. Finally the wire energy is the area where energy growth is a necessity. Adding extra cores necessitates expanding the area which the network covers, making the wire links scale in length with $\sqrt{(N)}$; because wire energy

is linearly related to both traffic and length, even maintaining constant traffic results in the power consumption growing as $\sqrt{(N)}$.

Unfortunately, without performing detailed synthesis of the different components in the architecture, it is not possible to combine the different factors into a single figure, so instead these components must be compared in isolation; however this does ensure the areas where scaling is problematic, and those where it not yet a problem, are highlighted clearly. First, looking at the energy spent in routing flits through the network in Figure 6.7, it can be seen that the small data trees make up most of the routing events.

This is unsurprising given that each cache line requires nine flits compared to only three in the mesh, and single flit packets for the coherency networks. However, the biggest problem is growth in switch routing events (through increased traffic, and increased switch distance for trees) as the system grows. Normalising each traffic type to its 32-core value results in Figure 6.8, where it is more evident that while mesh traffic remains relatively constant even up to 1024-core systems, the data tree quickly grows in a close to linear relationship with the number of cores. This is expected since the number of tree switches is approximately equal to the number of cores. The unicast based coherency networks grow in a similar fashion, with a slower growth rate, but the multicast coherency network rapidly grows with a high power function and overtakes the data tree growth at 512 cores. This is most likely due to the fact that the lossy coarse-vector multicast scheme results in greater numbers of cores being messaged as the system scales, something which is addressed in the next chapter with a novel encoding scheme.

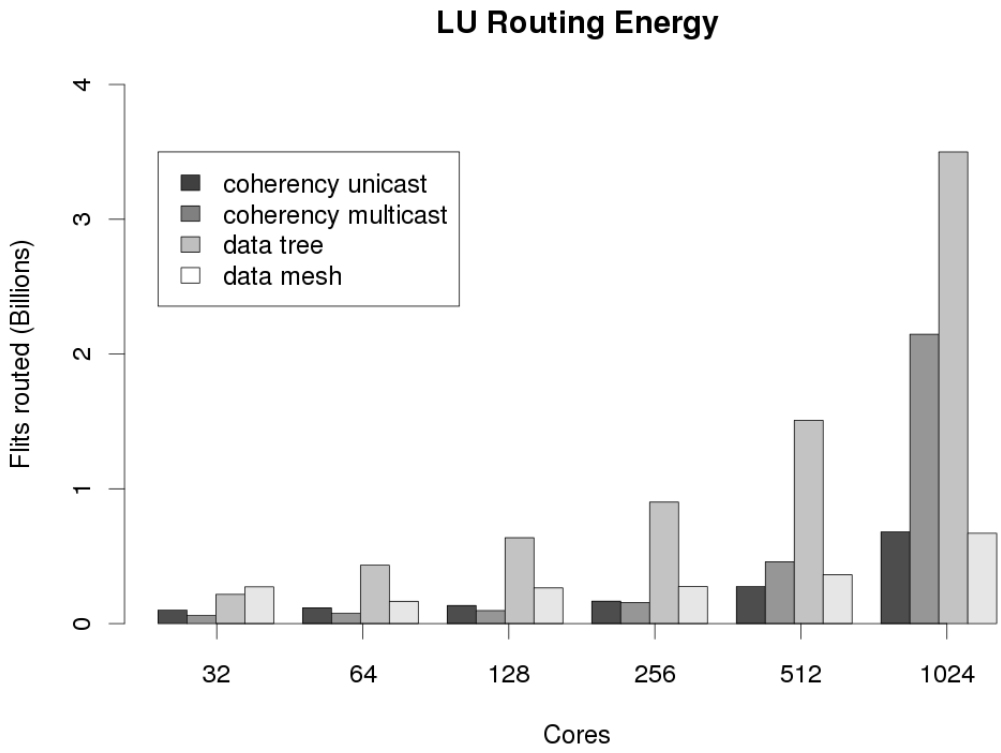


Figure 6.7: Interconnect flits routed through different classes of interconnect switch.

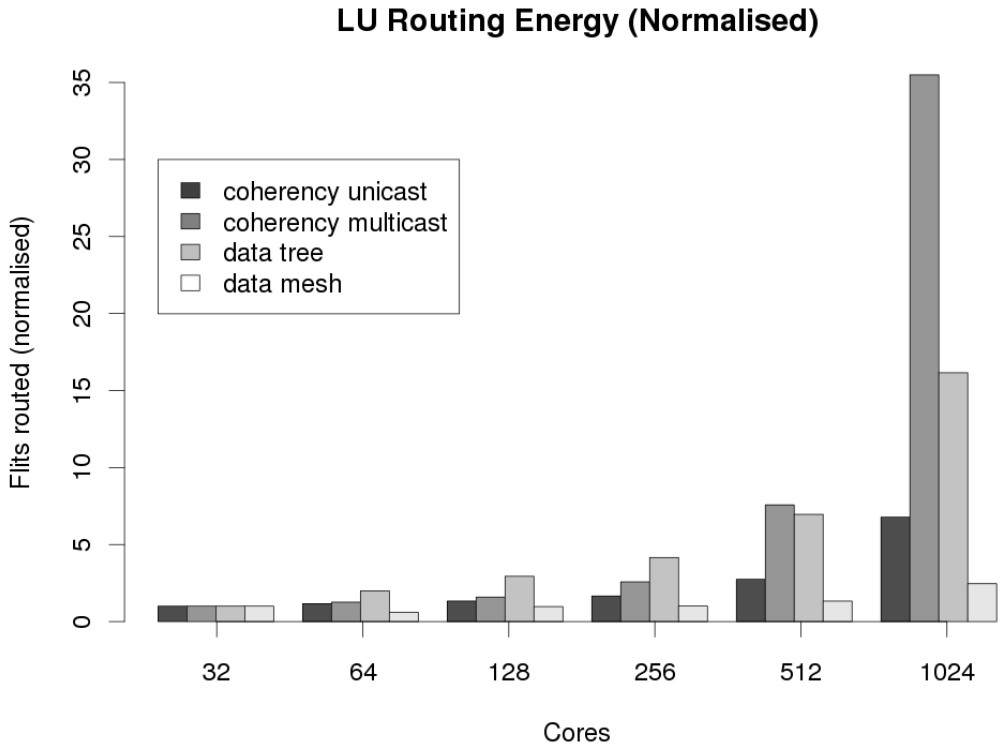


Figure 6.8: Interconnect flits routed through different classes of interconnect switch normalised to 32-core design.

The mesh network at first may appear to have less contribution to routing energy, since it processes fewer flits. However, the higher complexity of the switch micro-architecture, along with the significantly wider bus width and higher switch radix, means that each flit routed through a mesh router will consume many times the energy of routing a single flit through the simple tree switches. The other major contribution to interconnect energy is the wire switching energy. Assuming that all on chip networks operate at the same voltage, frequency, and wire spacing, the energy values given by multiplying the number of bit-transitions and wire lengths for each wire, should give results in the same, directly comparable, units for each bus type. This means that the results can be combined and directly compared, allowing us to see how energy scales overall, and how the each network type contributes to the overall totals for each network size. Figure 6.9 shows the wire energy for the LU benchmark and it is clear that the mesh dominates the network wire-energy consumption due to its longer wires, and significantly wider buses. Given the nature of these long, high bandwidth links it may be prudent to replace them with an alternative technology such as transmission-line or on-chip optical link such as those used in other work [66; 65; 57]. The data response network soon grows to be a large contribution to the network energy also, indicating that one of the best ways to reduce the network energy would be to introduce an intermediate level of caching, either at the core cluster level (such as 2-8 cores) or the per tile level (2-64 cores) to reduce the duplicate fetches of common data and instructions across the mesh, and handle local capacity or associativity shortcomings more gracefully.

A good result from Figures 6.7 and 6.9 is that the energy of the coherence networks is only a very small fraction of the network energy. The final contributions to chip energy, chip-wide static energy and dynamic processor energy, should stay constant in a perfectly scaling system. As the number of transistors scales linearly with the number of cores the static power should grow linearly, while the runtime should reduce at the same rate; resulting in constant static energy. Likewise, the number of cpu-cycles spent processing instructions should be the same, simply spread across more cores.

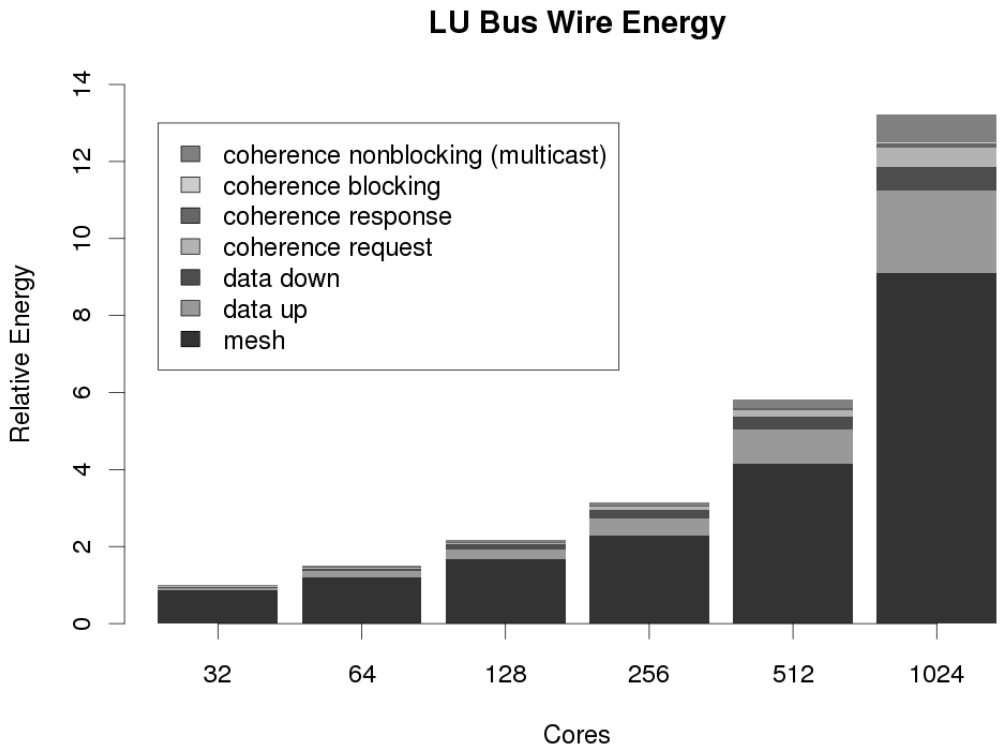


Figure 6.9: Interconnect wire energy generated by the LU benchmark.

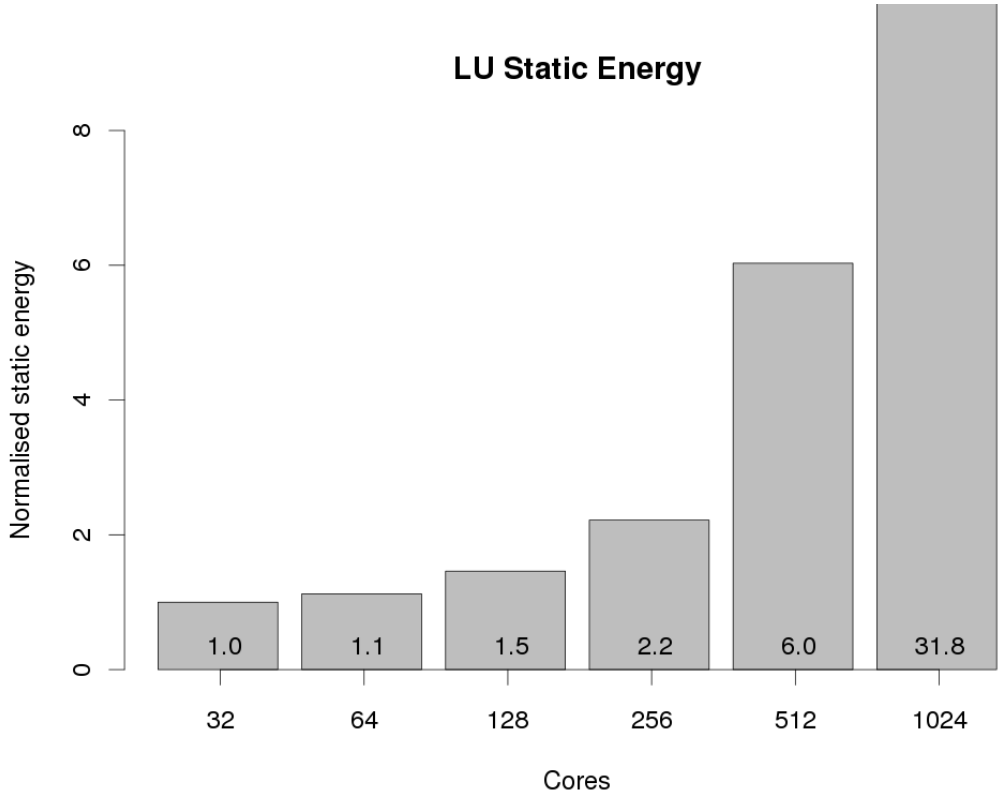


Figure 6.10: Static energy consumed by the LU benchmark.

Static energy, approximated by the number of cores multiplied by the runtime of the benchmark, can been seen in Figure 6.10, where it can be seen to scale well from 32 to 64 cores, which requires only 10% more static energy for a 1.75x improvement in performance, and even 256 cores still maintains a potentially acceptable energy consumption of 2.2x for a 3.6x speedup. Unfortunately scaling to 512 or more cores results in a drop in performance from the 256-core peak, resulting in significantly poorer energy efficiency because there are twice as many cores and interconnect switches consuming static energy, for an even longer period of time.

Finally, using "instructions executed" as a close approximation for the active energy of the core, Figure 6.11 shows that the active work required to compute the LU benchmark scales quite poorly past 128 cores, with a huge amount of energy being wasted for 512 cores. This is a reasonable approximation given the simple nature of the core and the fact the same benchmark and as such instruction mix is being run in all cases. Assuming the actual computational work remains constant, the extra energy must be due to synchronisation and communication overheads between the threads. Fortunately, the extra synchronisation energy can be easily addressed with some minor modifications to the processor architecture and thread libraries, as discussed in detail in Chapter 7. Doing so results in Figure 6.12, where it can be seen that the dynamic energy for LU is mostly constant, with extra instructions only being required for 512 and 1024 cores, and not nearly to the same extent as the interconnect or static energy has been seen to grow.

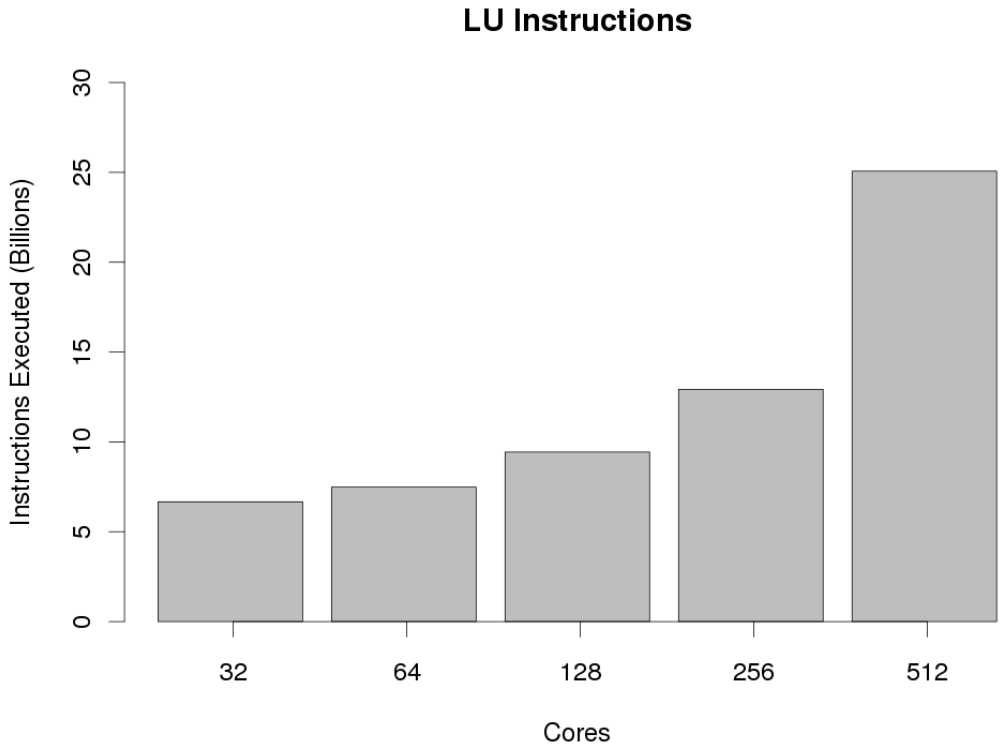


Figure 6.11: Number of instructions executed by the LU benchmark.

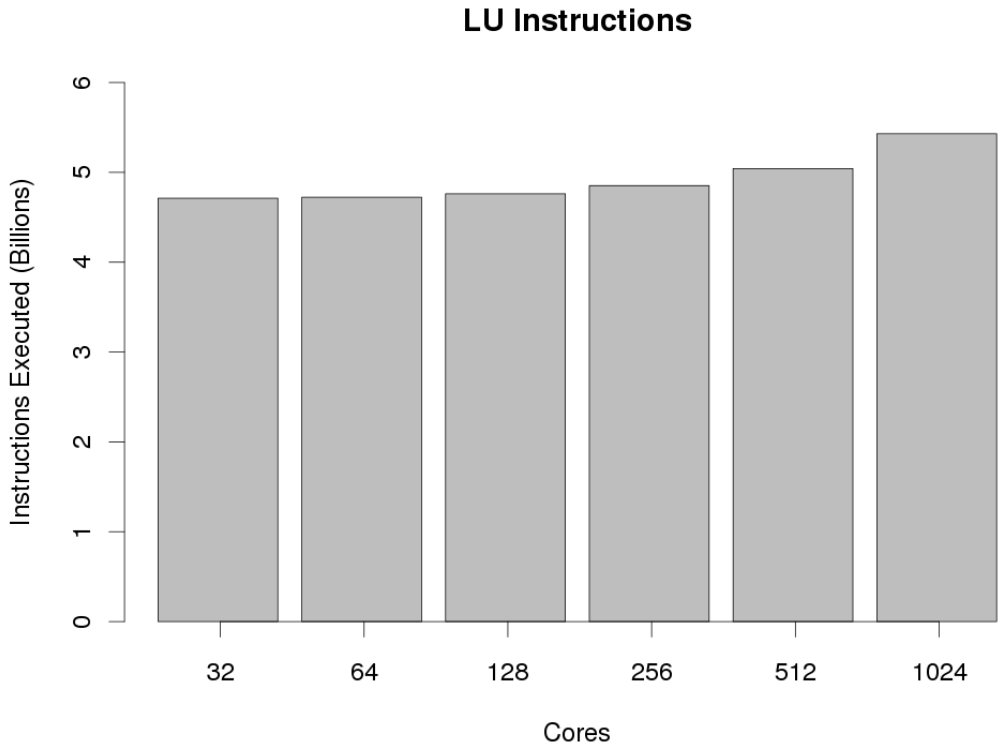


Figure 6.12: Number of instructions executed by the LU benchmark with optimised synchronisation.

6.4.4 Request Latency

Of course growing the number of cores on a chip not only increases the latency to LLC because of the greater hop distance, but the increased bandwidth demand for a fixed bandwidth resources necessitates that latency will increase because of contention. Looking at the distribution and 99th percentile of cache fetch requests gives another good indication of how well the chip design is scaling. Figure 6.13 shows the latency distributions for the same set of configurations as the other figures in this section, with overall latency obviously increasing from 32 down to 1024 cores, with each line representing the fraction of cache fill requests served within the time along the x-axis. It is obvious that the increasing traffic demands are too great for the infrastructure; adding more than 256 cores drastically increases the time taken to fulfil cache line requests, to the point where performance becomes worse than using fewer cores with less available parallelism.

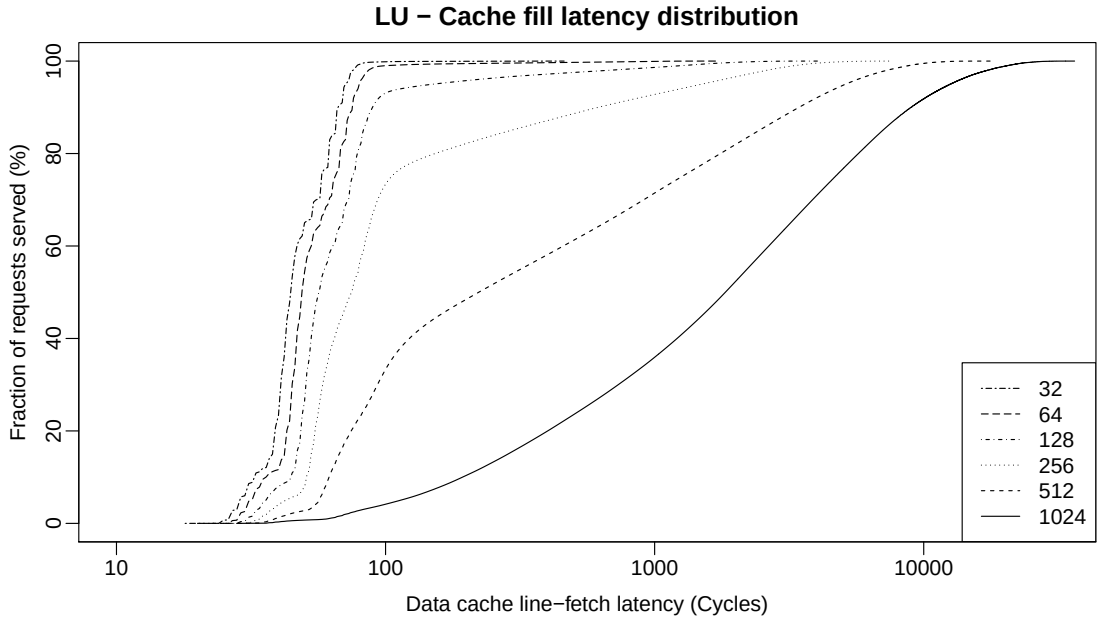


Figure 6.13: Distribution of times to fulfil data-caches requests for the LU benchmark.

6.5 Page Tracking Results

Figure 6.14 shows the total volume of memory accesses to each type of memory region, across a set of benchmarks and architecture sizes. It should be noted at this point that results are shown with the Wake-on-Address instruction (presented in the next chapter) enabled, this effectively removes excess memory instructions from spin-wait behaviours, so the number of instructions accessing Shared R/W pages are not inflated by locks and barriers. Looking first at the results from LU it can be seen can see that the Shared Read-Only and Read-Write regions are almost constant as the architecture is scaled up, while private access increase. This is expected in a workload where there is a fixed amount of computation divided equally between the active processing elements, as each element of computation requires an access to a fixed amount of Shared Read-Only and Read-Write data, along with a fixed amount of Private thread local storage. The private storage increases because each thread will have to do additional book-keeping work on top of the actual element processing, which will scale linearly with the number of active threads. This pattern is seen again with Barnes and Ocean, with only different ratios between the different access types. The small fraction of accesses shown with the filtering disabled are those produced by the TLB miss handler.

When looking at Figure 6.15, it can be see that cache misses are dominated by Shared Read-Write accesses, meaning that unfortunately little traffic is immediately saved by switching on page filtering. However the fact that that the number of accesses is almost constant when scaling from 32 up to 256 cores means that the total data throughput required over the period of the benchmark is not increasing, and the performance scaling results from Figure 6.5 show that the centralised directory can provide the bandwidth to allow good performance scaling. Unfortunately the benchmarks are reaching the limits of their scalability, with LU only algorithmically scaling up to 1024 cores with the current data set; each processing element has very little work to do and synchronisation and other non-computation accesses begin to dominate above 256 cores. This means that LU is only scalable to 256 cores on the proposed architecture, despite it being possible to partition the work further.

Bandwidth saving is not the only benefit however, as it turns out that without coherence filtering, the small fraction of memory accesses to Private and Shared Read-Only page types is very sensitive to the associativity of the directory. Blas *et al.* suggest that around 57% of directory space can be saved for a 12 core system [67], and by benchmarking performance across different directory size and associativity options with fil-

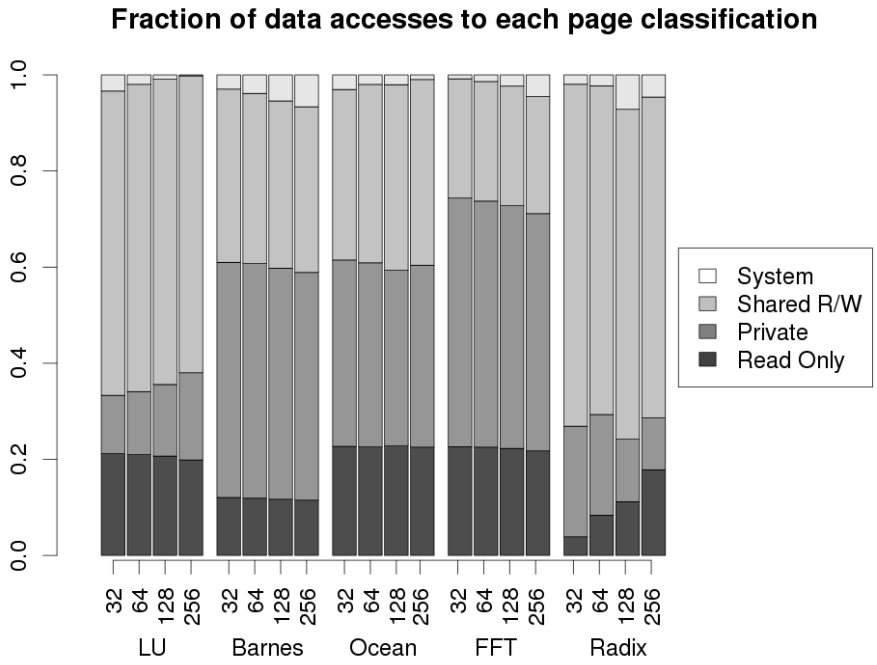


Figure 6.14: Fraction of memory accesses to each page classification across the benchmarks.

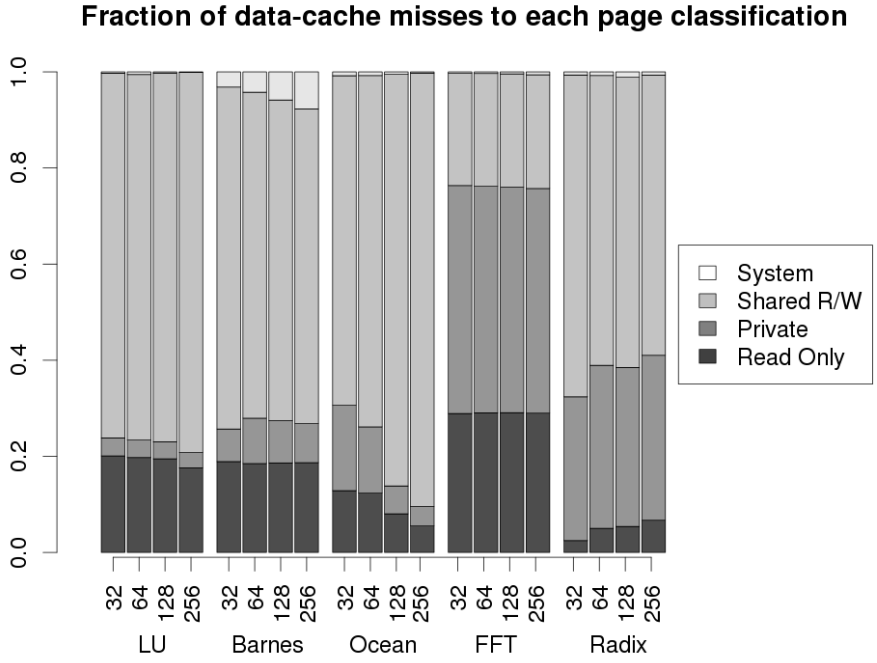


Figure 6.15: Fraction of cache misses to each page classification across the benchmarks.

tering enabled and disabled (by modifying the software handler to initialise pages into the Shared Read/Write state), Figures 6.16-6.25 show that many benchmarks are much less sensitive to directory associativity and size with MMU based page filtering.

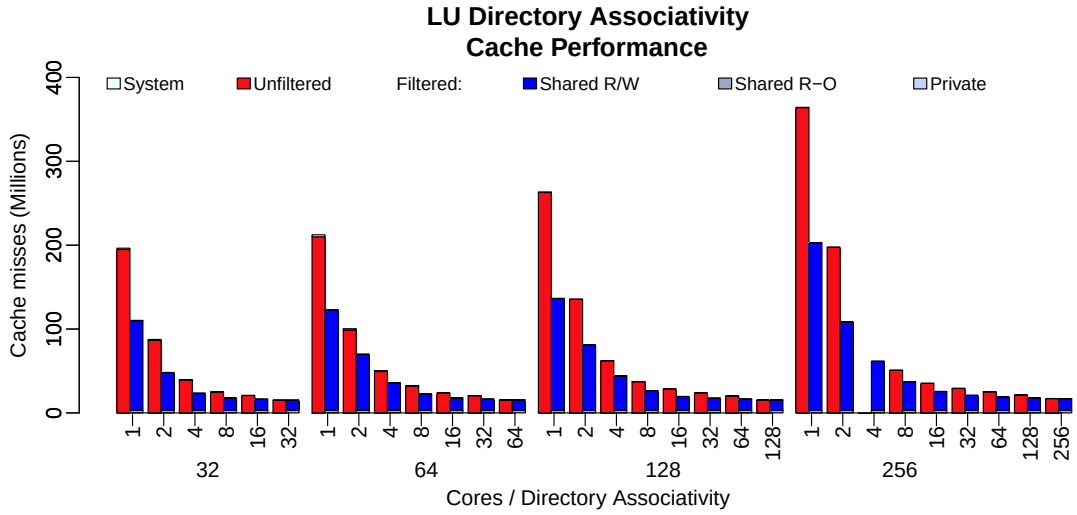


Figure 6.16: Effect of directory size & associativity on LU cache performance for 32, 64, 128 and 256 cores, comparing filtered and unfiltered accesses.

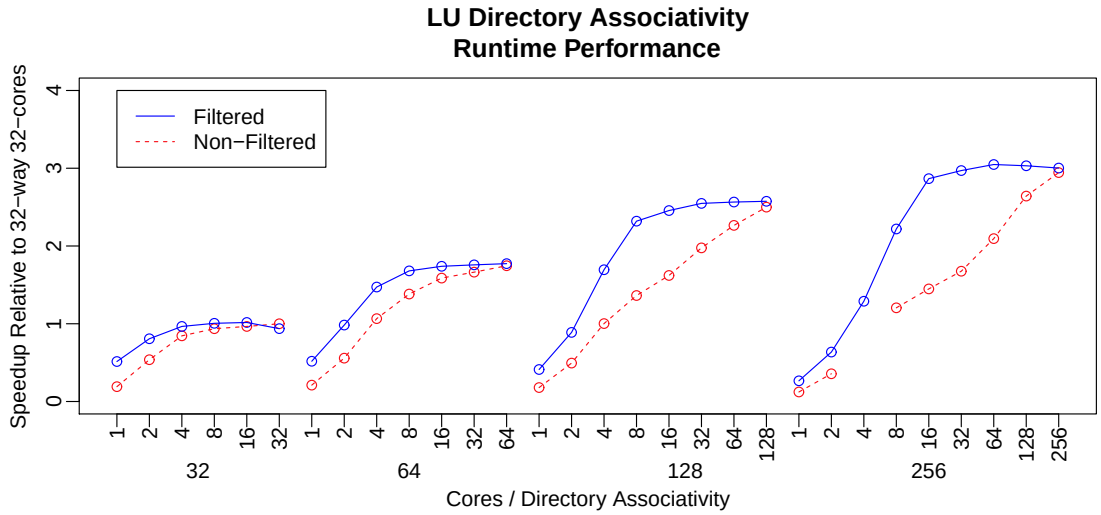


Figure 6.17: Effect of directory size & associativity on LU runtime for 32, 64, 128 and 256 cores, comparing filtered and unfiltered accesses.

Because each thread is performing similar work, the thread local stack behaviour will be similar in each core, resulting in a a subset of memory accesses which would require a directory of effective associativity the same as the number of active cores in

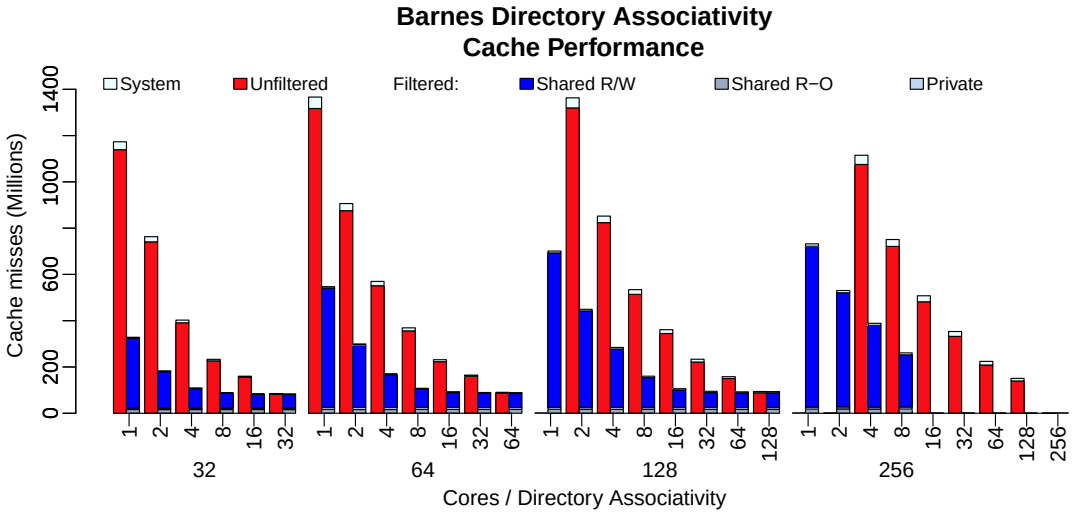


Figure 6.18: Effect of directory size & associativity on Barnes cache performance for 32, 64, 128 and 256 cores, comparing filtered and unfiltered accesses.

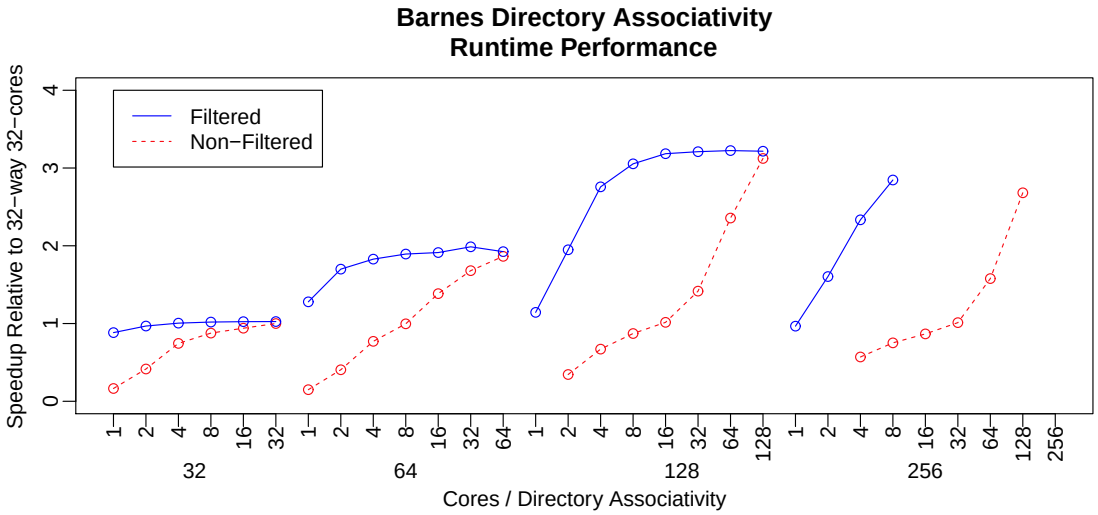


Figure 6.19: Effect of directory size & associativity on Barnes runtime for 32, 64, 128 and 256 cores, comparing filtered and unfiltered accesses.

the system to satisfy, or a coherence protocol which did not require a fully inclusive directory. It can be seen in all performance figures, but especially in Figures 6.19, 6.21, and 6.23, that without coherence filtering the performance drops significantly with less than this, and providing such a high associativity memory structure would be impractical.

LU and Barnes for example show that using MMU based filtering, a directory of only 1/16th the size and associativity of the full chip's L1 cache system can provide

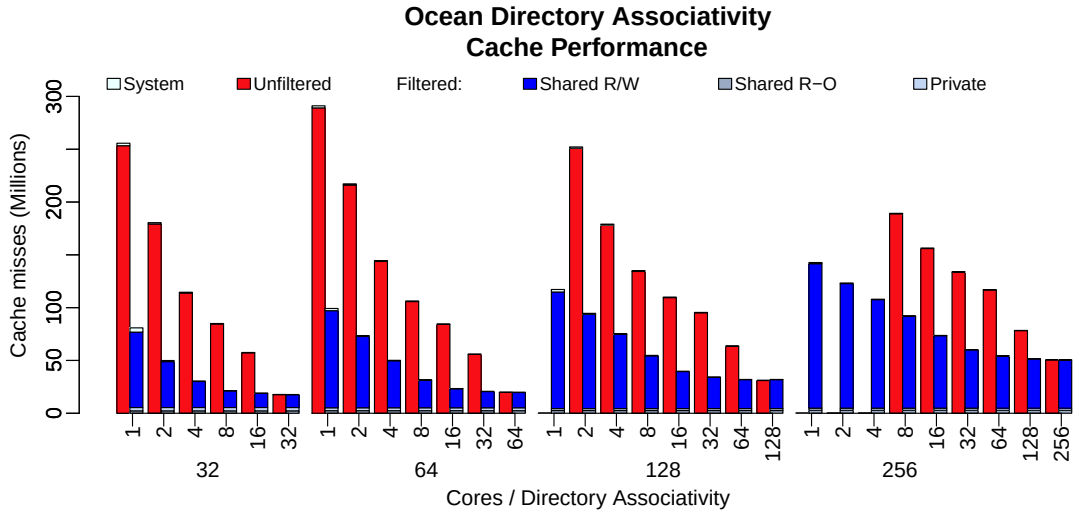


Figure 6.20: Effect of directory size & associativity on Ocean cache performance for 32, 64, 128 and 256 cores, comparing filtered and unfiltered accesses.

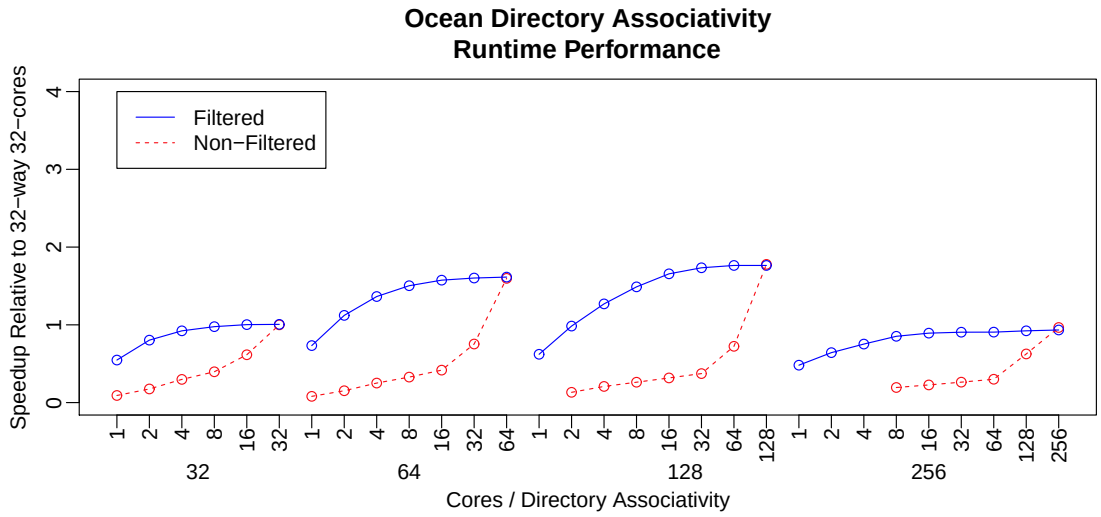


Figure 6.21: Effect of directory size & associativity on Ocean runtime for 32, 64, 128 and 256 cores, comparing filtered and unfiltered accesses.

almost the same performance as a directory which matches the capacity and associativity of the chip's L1 caches when no filtering is used. If that is increased to 1/8th then the performance exceeds the baseline unfiltered system even with full capacity and associativity. Seeing this trend in other benchmarks suggests that a directory with 1/16th of the L1 associativity would be sufficient, and with 256 cores this gives a 16-way associative directory, well within the limits of on-chip memory structures given that current L3 caches range from 16 to 48-way associative (Core-i7 and Phenom-II

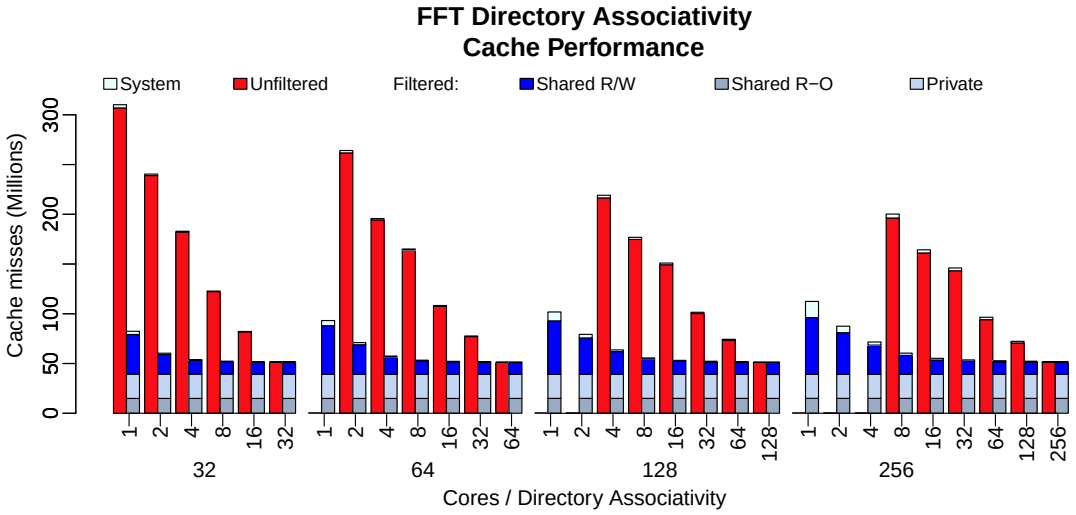


Figure 6.22: Effect of directory size & associativity on FFT cache performance for 32, 64, 128 and 256 cores, comparing filtered and unfiltered accesses.

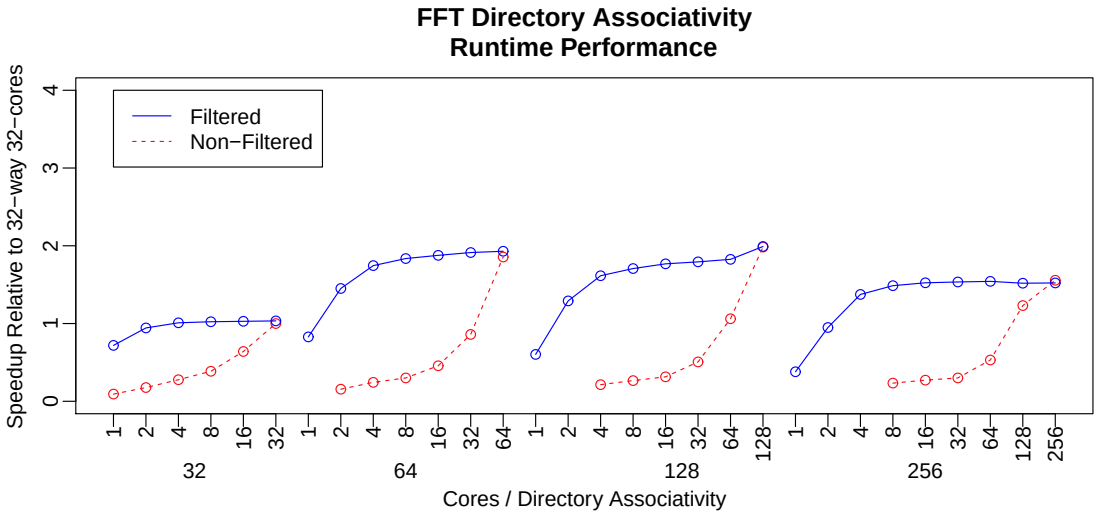


Figure 6.23: Effect of directory size & associativity on FFT runtime for 32, 64, 128 and 256 cores, comparing filtered and unfiltered accesses.

processors). Associativity requirements could be further reduced by using a technique such as with Cuckoo Caches discussed in Chapter 3 allowing larger systems or saving energy in the directory.

Unfortunately the Radix results in Figures 6.24 and 6.25 show that most of the shared Radix working set suffers from index-aliasing in the directory. This is clear from the similarity between the filtered an non-filtered performance scaling, which both increase linearly with the associativity of the directory. The relatively large frac-

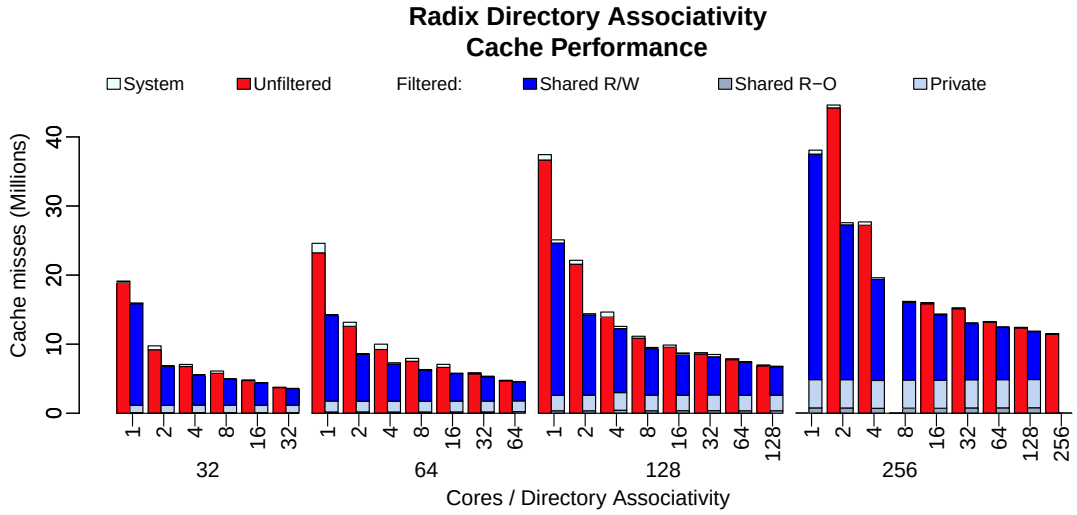


Figure 6.24: Effect of directory size & associativity on Radix cache performance for 32, 64, 128 and 256 cores, comparing filtered and unfiltered accesses.

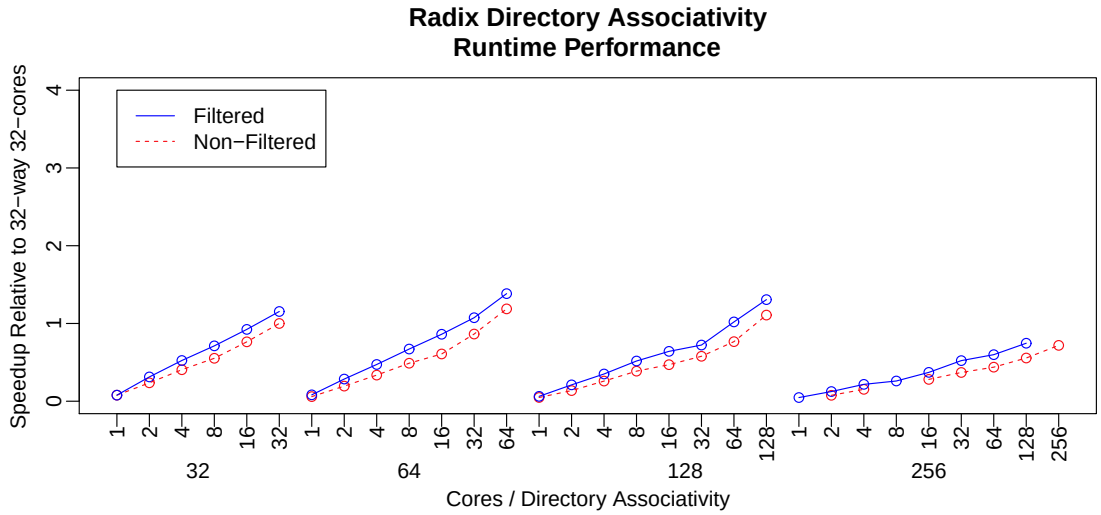


Figure 6.25: Effect of directory size & associativity on Radix runtime for 32, 64, 128 and 256 cores, comparing filtered and unfiltered accesses.

tion of non-coherent misses, shown in Figure 6.24 caused by capacity or conflict misses in the L1 data cache, are the cause of the difference in performance between the filtered and unfiltered results. Unlike most benchmarks, even at full associativity the filtered results are significantly better performing due to the coherency overhead being avoided for this large fraction of cache misses; the other benchmarks showed good L1 cache performance for non-coherent data.

6.6 Architectural Conclusions

The proposed architecture shows some scalability promise, but the performance depends very much on the application characteristics. This should be expected from a design which takes such an extreme compromise between core performance and interconnect bandwidth, to offer the large number of cores at reasonable power and area consideration. The proposed architecture is best suited to workloads with a high integer compute to memory bandwidth ratio, lacking the floating point hardware for typical scientific workloads. However, compared to GPGPU accelerator cards (which excel at data-parallel computation) the true CMP architecture provides significantly better support for algorithms with a high degree of control flow. The most suitable applications are probably cryptographic computations, where floating point hardware would be a waste of silicon estate, and algorithms are becoming increasingly more compute demanding for each data element processed. Cryptographic algorithms can also include conditional execution and complex control flow. Because the core architecture is highly configurable and extensible, it would be possible to design accelerator instructions for the cores, either in a homogeneous or heterogeneous manner, to produce a specialised accelerator for tasks such as encryption or video encoding.

Chapter 7

Saving Energy in Manycore Processors

7.1 Introduction

This chapter presents the novel contributions made on top of the architecture described in the previous chapter, designed to reduce energy use when running parallel programs.

It is imperative that common synchronisation primitives, such as mutex, barrier, condition variables, and task farms can be implemented in an energy efficient manner. Ideally one would like to be able to state the event conditions the program is waiting for, as in Figure 7.1 with the core using no energy until that event occurs, and without generating traffic about the event to any cores except those that need to know.

Additionally most applications, even parallel benchmarks, do not scale up to 1024 cores [139; 12], so it is more likely that such a processor would be used to run many smaller parallel applications, with the cores partitioned between applications [140]. To facilitate power scaling by turning off clusters of cores, and improve utilisation of the cache hierarchy, it is also important that applications have their threads geographically clustered on neighbouring processors. While the proposed architecture in this thesis does not use an intermediate shared cache between L1 and the LLC, there is a high probability that a real architecture would, along with a directory cache. It is also likely that a system might implement cache line forwarding from neighbouring cores, which is omitted from the presented architecture to simplify correctness guarantees, and reduce simulator development time. Both of these would require threads to be running close together for best performance, and a good scheduler would ensure this.

The main contributions of this chapter are:

1. An optimisation to the implementation of atomic instructions to reduce unnecessary cache line write-backs.

2. A new architecture-independent hint instruction Wait-on-Address to allow energy-efficient synchronisation primitives using traditional coherency mechanisms.

3. A novel spatially-aware, multicastable, and space efficient coherency sharer encoding.

This chapter is divided into three main sections for each of the contributions: Section 7 discusses a micro-architectural optimisation to the implementation of atomic instructions. Section 7.3 introduces a new hint instruction, designed to significantly reduce the energy spent on synchronisation, and discusses the ISA implementation optimisation. Section 7.4 introduces a new storage-multicast encoding, called Conservative Tree Encoding, and presents synthetic and scientific benchmark results.

It should be noted that, except where stated otherwise, the results presented in this chapter do not use a simulated MMU or TLB, and do not implement the coherence filtering presented in the previous chapter, due to time constraints. The simulated systems included a full capacity and associativity directory, which minimises any effect of directory pressure. This means that multicast invalidations from the directory should be almost entirely due to data modification, rather than directory capacity or conflict evictions of read-only data, as they would be with the filtering applied. The proposed features undergoing experiment in this section only involve cache lines which are shared by multiple cores, so the results will not differ significantly from those where coherence filtering had been performed.

7.2 Reducing Cache Write-Back During Synchronisation

When synchronising a large number of cores with conventional memory operations, most mutex code operates by waiting for a lock variable to change from locked to unlocked, then attempting to perform an atomic operation on the variable to acquire the lock (usually an atomic exchange, which returns the previous value of the variable, while setting it to a new value); the acquire succeeds if it returns the unlocked state

when setting it to locked. Unfortunately when this happens for a large number of cores, all of them will attempt to execute an atomic exchange simultaneously, requiring the coherency mechanism to transition each of them to exclusive and then modified in turn, and each core will have to write back the now modified cache line before the next core can read it (assuming cache line forwarding is not supported). The only core which actually succeeds is the first core processed by the coherency mechanism, but the other cores are not aware of this until they have attempted to acquire the lock themselves.

A simple modification to the implementation of atomic exchange can effectively eliminate this excessive write-back traffic: when performing an exchange operation, perform a bitwise xor between the previous and new values, and only change the cache line state to dirty/modified if the state actually changed. This means that all of the cores which fail to acquire the lock can simply discard their cache line and send an invalidated acknowledgement message, without the time and energy overhead of writing back the whole cache line. Although quantitative data on the reduction in interconnect pressure has not been collected, it seems clear that the interconnect energy saved will greatly outweigh the cost of the comparison, and the overall performance of the system under lock contention will be improved significantly.

7.3 Wait-on-Address Hint Instruction

In almost all parallel programs, there are communication and synchronisation actions which inherently depend upon one or more cores waiting for others to perform some action. The simplest example for this is the mutex – while one core holds it, any other cores which wish to operate on the protected structure must wait until the holding core releases the lock. There is already an "efficient" solution to this problem, in that network and coherence traffic can be negated by performing a "test and test-and-set" series of operations, where the core will spin reading the local cached value of the lock or signalling variable without generating coherence requests until the value is changed. However, this only reduces interconnect traffic, and leaves the core in a power hungry loop while it is waiting. This energy can be reduced by sleeping the core for a fixed period of time between checking the value, for example the Intel x86 PAUSE [141] instruction, but this can reduce performance since the processor might still be sleeping when the value changes. This still requires periodic checking of the memory value – there is still more energy to be saved.

To address this, a new hint instruction is proposed, referred to in this thesis as Wait-

on-Address. This instruction puts the core into a sleep state until a coherence message is received for the cache line indicated by the address argument to the instruction. This means that the core sleeps exactly the correct length of time before waking to deal with the change in memory value, providing the best energy efficiency possible (short of completely powering down the core and caches) while sacrificing almost no performance over a busy spin-loop. If the cache line is not valid in the L1 cache when the instruction is issued then it does not sleep – since it must have been invalidated between the last check and the issuing of the hint instruction. It is also perfectly valid to wake the core up for other reasons such as timer interrupts, since this is simply a hint instruction, and it is up to the programmer to check the new value and determine if the core should continue waiting, or carry on with some new action.

This new instruction is trivially implemented in the pthread semantics for mutex, barrier, and condition variables, as demonstrated in the code segments in Figure 7.1, and can also form a useful component of any other wait-on-event scenario, such as a task farm worker waiting on a new task, or waiting for a producer-consumer queue to become non-empty or non-full. It is also used in the pthread runtime implementation used in this work, to wait for a thread to be assigned to a core so that cores sleep and wait until they are assigned a thread without performing a busy wait. This has the secondary effect of improving simulation performance, since these instructions no longer need to be simulated.

Unlike some hardware-provided signalling methods, such as ARM's WFI or WFE [142] instructions (which suspend the core until an external interrupt, or dedicated event signalling channel, respectively), the proposed mechanism provides an essentially unlimited number of unique signals (one per cache line). These can be used for any form of one-to-one, one-to-many, or (less efficiently) many-to-one signalling between threads, so long as all of the signals that are being waited on can be stored in the same cache-line. For example the Cholesky benchmark involves a form of task-farm, where each worker must watch two potential work sources simultaneously. Because these, and their respective locks, all fit into the same cache line, it is possible to wait efficiently for work allocations, despite it being semantically two separate signals. An extension of the proposed instruction could take a bit-vector of cache lines instead, and wait for any given cache line to be invalidated to enable the monitoring of more simultaneous signals.

It may appear at first that a simple "wait for coherency event", which is not tied to an address or cache line could be sufficient, but this would miss the race condition

```
mutex_lock(mutex_t *m){
  //initially assume the lock is taken
  bool val = true;
  //spinwait until the exchange instruction returns false
  // i.e. (the mutex was free)
  do {
      bool testval = true;
      //read-only test loop avoiding exclusive access on every test
      while (testval){
          testval = m->lock;
          if(testval){
              //the mutex is locked, so wait until the value changes
              wait_on_address(&m->lock);
          }
      }//the mutex was free, so attempt to lock it
      val = atomic_exchg(&m->lock, true);
  } while (val);
  return;
}
mutex_unlock(mutex_t *m){
    m->lock = false;
}
```

```
barrier_wait(barrier_t *b){
  mutex_lock(&b->lock);
  b->present++;
  if(b->present == b->required){
      // reset barrier counter
      b->present = 0;
      // signal everyone that is waiting
      b->release++;
      mutex_unlock(&b->lock);
  } else {
      unsigned int release = b->release;
      unsigned int release_test = release;
      mutex_unlock(&b->lock);
      // wait until final core releases us
      while (release == release_test){
          wait_on_address(&b->release);
          release_test =  b->release;
      }
  }
  return;
}
```

Figure 7.1: Example use of Wait-on-Address.

where the cache line is evicted between checking the current address value, and issuing the wait instruction. There are two alternatives to WOA which could be imagined that produce the desired behaviour, instead using a pair of imagined instructions. Firstly (and most similarly) a load-linked operation (as is found on some architectures for speculative atomic operations) followed by a wait-on-linked instruction, which uses a existing instruction to encode the actual address of importance, and a subsequent instruction which may have a much shorter encoding to supply the wait command. Secondly, a "begin monitor" instruction, which instructs the processor to monitor the data cache for self-invalidation or coherency events, followed by a wait-on-coherency instruction. This second instruction pair allows the programmer to check as many addresses as required between the begin and wait instructions, but the wait will fail if any cache line was evicted (due to capacity, conflict, or coherence) during the period since the monitor instruction was issued. The core will awaken upon any coherency event that affects entries in the data cache, even if they were not explicitly accessed by the interim code. This allows multiple sparse memory locations to be monitored efficiently, which is unsupported by the proposed WOA instruction. A further optimisation would be to flag cache lines which were accessed during the monitor period, and only wake up the core on accesses to these, although this would require an extra bit of storage per cache line.

7.3.1 Results

The wait-on-address hint instruction was implemented in the simulated ISA, based on the ARC700 instruction set, and the pthread runtime [12] was modified to make use of it in all of the synchronisation primitives. Simulations were then run on the architecture and benchmarks described in the previous chapter, collecting statistics such as data accesses, instructions executed, and the time spent waiting on coherency events with new instruction.

For Barnes, LU, and Ocean the measurements were conducted over only the parallel portion of the benchmarks, with the full TLB based filtering enabled. Due to time constraints FFT and Radix results are presented from simulations which do not include this filtering and include a serial startup phase. As a result these two benchmarks may under-represent the relative reduction in instruction count and cache accesses, given the constant serial instruction count overhead. All cores not currently assigned work were suspended, so the startup phase does not contribute to spin-wait instruction counts.

Figure 7.2 shows the runtime profile of the average number of active cores for the 512 core Radix benchmark. The blue line represents average number of cores not "waiting" on the hint instruction in each 100K cycle timeslice, while the greyscale backdrop is a running histogram of the number of awake cores in each cycle of that timeslice. It can be see in the first half of the graph that there is generally about a hundred cores worth of parallelism, although all the cores are rapidly waking up and returning to sleep. The other benchmarks in Figure 7.3 feature more pronounced repeated barrier synchronisation, with the cores repeatedly being all active, then all asleep. Both patterns indicate ample opportunity to save energy.

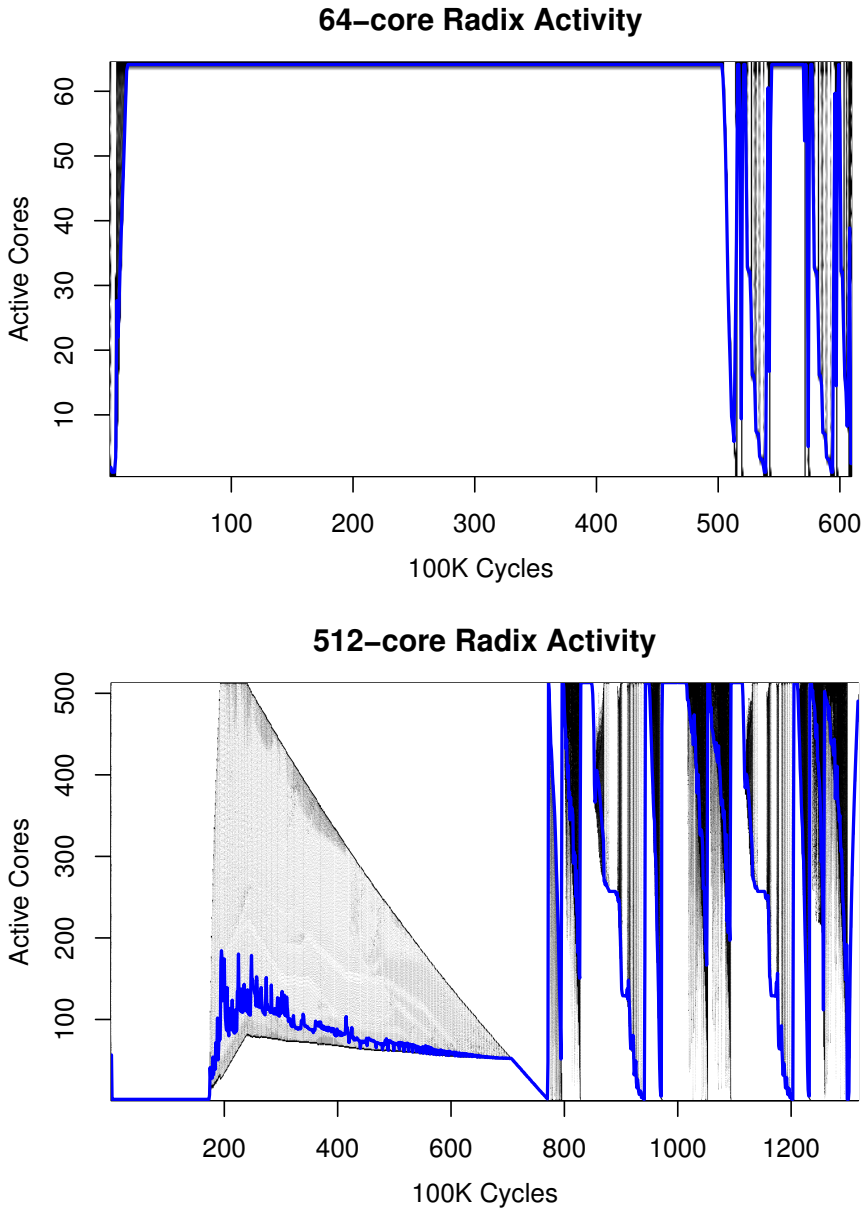


Figure 7.2: Runtime profile of the number of cores active through 64- and 512-core Radix. Blue line represents average number of cores active.

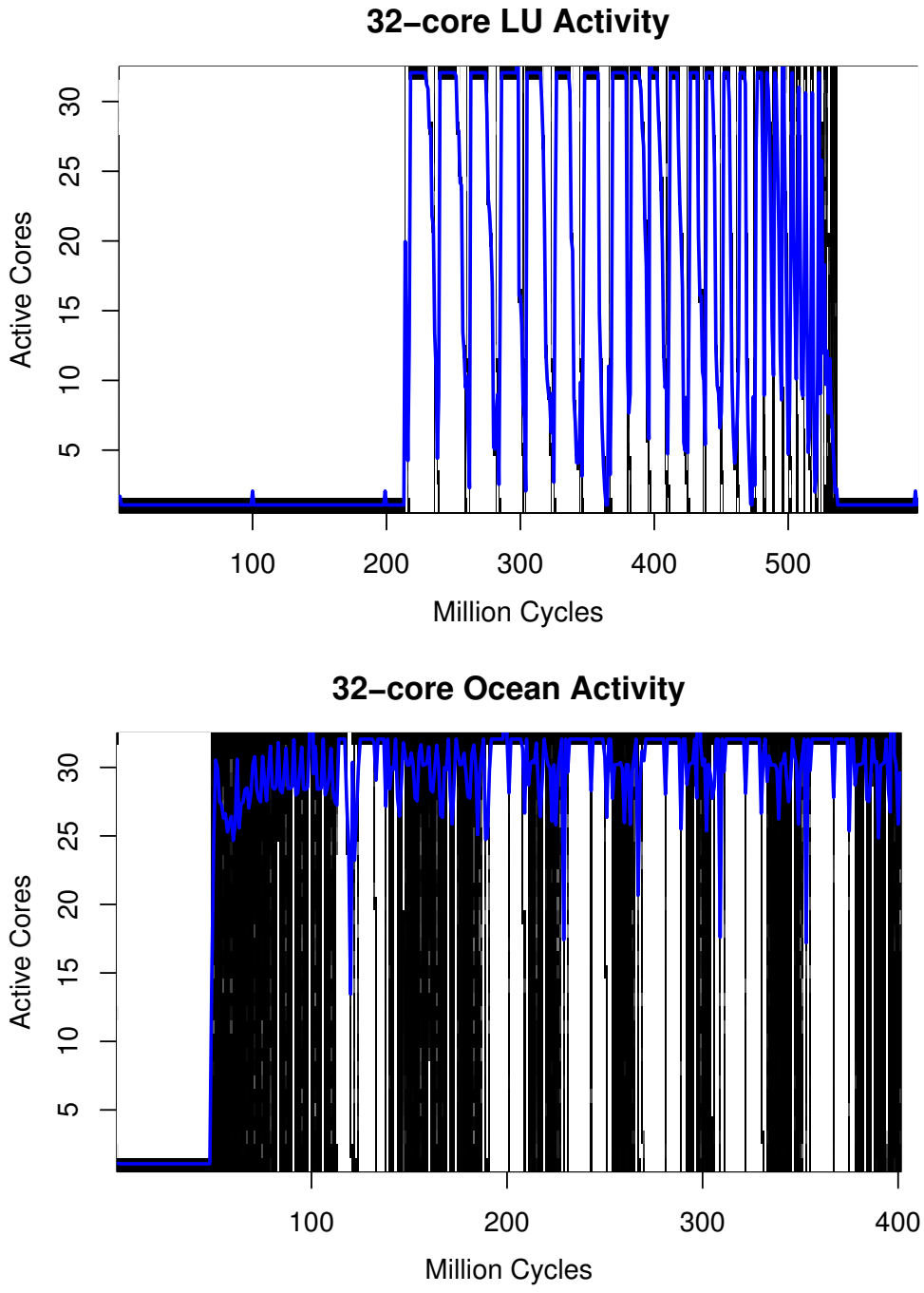


Figure 7.3: Runtime profile of the number of cores active through 32-core LU and Ocean. Blue line represents average number of cores active.

Re-visiting the scalability issues highlighted in Section 6.4.3, Figure 7.4 presents the number of instructions executed as each of the simulations is scaled up to larger systems. The red lines, representing simulations without WoA support, clearly demonstrate how synchronisation between application threads can rapidly grow to dominate the dynamic core energy of the processor. Conversely the blue lines, representing simulations with WoA support, demonstrate that by suspending these cores during spin-wait cycles the number of instructions executed – and as such computational work performed – is almost constant as more cores are added. This gives a perfect dynamic energy scalability result: adding more cores does not increase the dynamic core energy. As an estimate for the dynamic core energy saved, Figures 7.5 and 7.6 present the number of instructions and number of data accesses saved as a fraction of the total when not using the hint instruction, over the subset of Splash-2 benchmarks, and across a range of core counts. Looking at these results it can be seen again that all benchmarks spend an increasingly large proportion of their time waiting on synchronisation. The geometric mean of the relative number of instructions executed with WoA, relative to without WoA, was computed and subtracted from 100% to give an average saving in executed instructions of 53%, and a reduction in data accesses by an average of 83%.

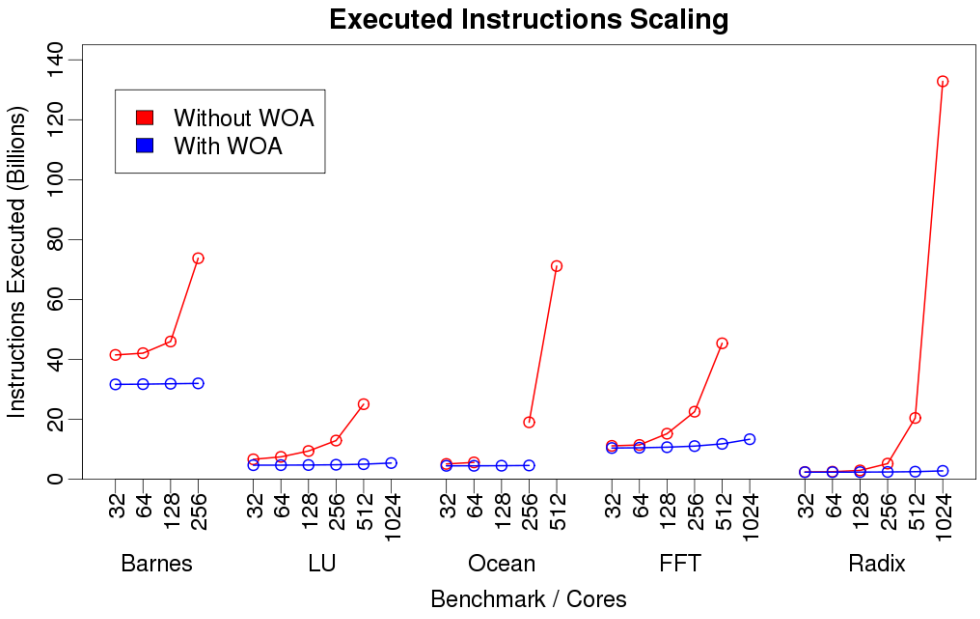


Figure 7.4: Scaling of instructions executed with and without WoA.

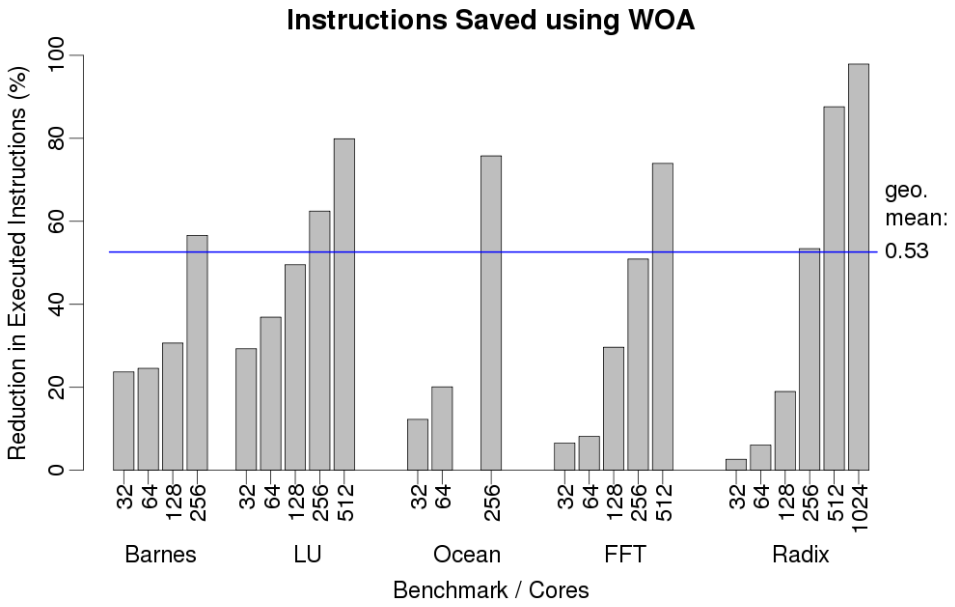


Figure 7.5: Reduction in instructions executed using WoA.

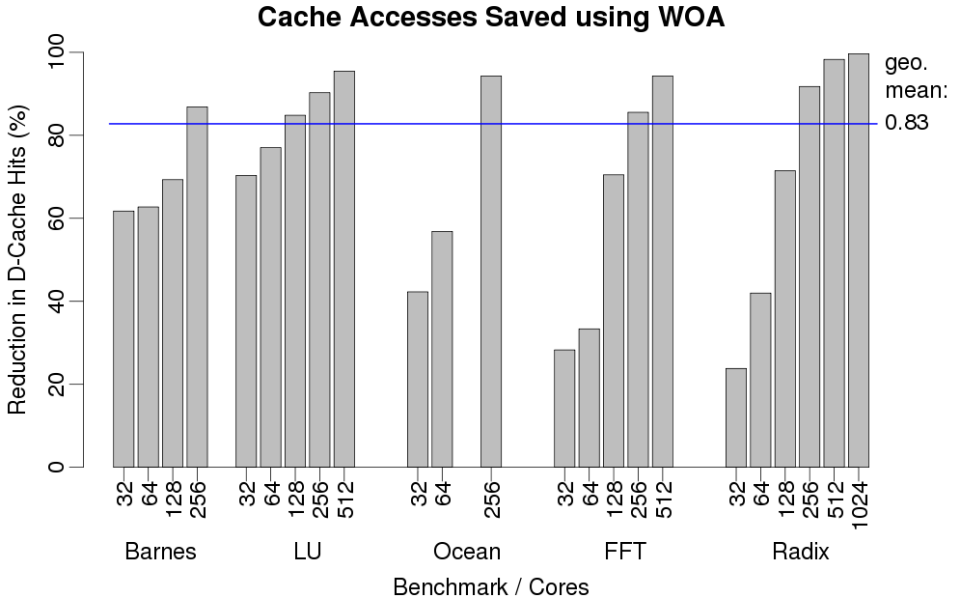


Figure 7.6: Reduction in data accesses executed using WoA.

7.3.2 Related Work

The Intel x86 architecture, for example, contains a "PAUSE" instruction [141] for use in spin-locks, which acts as a memory fence and a short sleep instruction, in order to save power. However, unlike the WoA hint instruction it does not take an address or cache-line to watch, it simply suspends the core for a finite period of time, then resumes to allow the program to continue, and then the programmer can check the value again. WoA greatly improves upon this by waiting indefinitely, until the exact moment that the value is changed, meaning the performance sacrifice is minimal, while still remaining in a low power state as long as possible.

ARM provides a very different feature, a broadcast messaging system with wait and send event (WFE, SVE) commands. This is a broadcast which could be implemented as an efficient dedicated hardware resource, but suffers from a lack of process isolation, and limited usability across multiple pieces of concurrently running code – two simultaneously running programs, or parts of the same program, will interfere with each other if they try to use it concurrently. The messaging is also out of band relative to the memory coherency and consistency model, so potentially extra coherency synchronisation would still be required in a data sharing scenario.

A third related concept is the design of the Propeller micro-controllers, which host a number of cores with each intended to be dedicated to a task. Rather than time multiplex a single core based on interrupts, each interrupt line can be assigned to an individual core, which is then put into a low power sleep state until awoken by its respective interrupt line. In this sense it embraces the physical task distribution ethos of the proposed manycore design, where one wants to be able to give a physical core to every thread, while also supporting a low energy wait-for-work state like the proposed hint instruction enables via the coherency mechanism. The similarities end here however, because this is a micro-controller architecture rather than a large scale cache coherent general purpose processor. ARM also supports a similar wait for interrupt (WFI) instruction along with the wait for event instruction, but this still does not enable a task isolated efficient messaging system.

The Rigel architecture paper hints at an efficient barrier implementation through their "update broadcast" support, but it reads like this simply replaces the coherency mechanism and allows the cores to spin on local cache, rather than use the global cache in their incoherent architecture [73].

In addition to the x86 WAIT, and ARM WFI and WFE instructions already men-

tioned, a very similar operation to the proposed load-linked and wait-on-linked pairing is mentioned in the design of the Alpha 21464 [143] in order to put SMT threads to sleep and avoid a waiting thread from consuming unnecessary resources in the shared pipeline. Although the functionality is similar, the motivation is quite different, with the Alpha design existing to improve performance (through improving efficient use of shared compute resources) while the proposed mechanism is to reduce power consumption in modern power and thermally constrained manycore designs. The analysis of thread sleeping performed in this thesis is significantly more extensive than that published with regards to the Alpha processor and highlights it's relevance for large scale CMP designs, not just for SMT systems.

Another processor design with this feature is the SMT-CMP PowerEN architecture [144] which provides a "wrlos" instruction: wait until reservation lost. This is exactly the WOA implementation described previously with the load-linked wait-linked pair, where "reserved" is the Power architecture term for the "linked" cache line state, and wrlos is the wait-linked instruction. Pasetto *et al.* [144] make a much more detailed analysis of the wrlos instruction in popular inter-thread communication, synchronisation, and work distribution algorithms on the heavily multithreaded PowerEN architecture, which provides 16-cores and a total of 64-threads. Their work demonstrates that for SMT architectures wrlos (equivalent to WOA) can provide significant performance benefits. While it also clearly demonstrates that the WOA instruction is not a novel concept, this thesis introduces the drastic energy savings which can be realised by using WOA. Unlike the discussed performance gains, which apply only to SMT architectures, these energy savings are applicable to all multi-threaded shared-memory architectures, from CMP, to multi-socket, and even distributed shared-memory architectures, regardless of their SMT support. For thermally constrained high density CMP architectures the energy savings can also translate into additional performance gains, making the importance of WOA clear for server environments where SMT-CMP processors are packed in a high-density environment. Pasetto *et al.*'s analysis of many of the popular inter-thread communication techniques extends not only to the impact of the wrlos instruction, but comparing many different algorithms on both PowerEN and x86 architectures, with different architectural atomic operation support. The work demonstrates that the simple algorithms used in this thesis for synchronisation and communication are sub-optimal, so better scalability results could probably be achieved for the proposed manycore architecture. However inter-thread communication algorithms themselves are not the focus of this thesis.

Tullsen *et al.* [145] make a similar analysis of thread synchronisation for an 8-way SMT architecture, demonstrating that the close coupling of the thread execution in the processor allows for much faster thread synchronisation than traditional multi-socket SMP systems, enabling a new class of algorithms to be effectively parallelised. Loops with inter-iteration dependencies and similar algorithms with short lived parallelism are unsuitable to thread level parallelisation when the synchronisation cost is as high as is common with inter-chip communications, so previously parallel benchmarks and applications have been focussed around more embarrassingly-parallel applications, where there is significant computation between synchronisation events. Their work shows clearly that integrated multi-thread architectures are capable of significant parallel speedup even when executing tightly synchronised parallel sections through techniques such as pipeline speculation. This new class of parallelism is something that should be exploited to make full use of high-density manycore architectures, so low-latency inter-thread communication is critical.

Li *et al.* [146] and Wells it et al. [147] describe methods for detecting spin-wait behaviour in hardware without explicit instructions such as WOA. While they use these to influence scheduler policies and power management, among other things, the same techniques could be used to substitute a classic spin-wait instruction stream with a WOA style spin-wait on-the-fly, giving most of the benefit, without the need to recompile existing applications. Similarly, using the WOA style instructions would enable much of the other work proposed in these two pieces of research without the effort of dynamically identifying core spinning.

The MIPS MT architecture, implemented in the MIPS32 34K core [148], defines a different mechanism for efficient inter-thread communication, using a "gating storage" called the Inter-thread Communication store (ITC). The ITC comprises of memory cells, each 16x64 bits long, which each contain a 64-bit memory element and a full/empty state bit. The each of the 16 possible 64-bit aligned indexes into this memory element implements a different access semantic, although the MIPS MT architecture leaves 10 of these reserved for future use. Between these 6 available "views" of the memory cell various synchronisation primitives such as semaphores, FIFOs and protected variables can be implemented without spin-wait behaviour. Instead of the pipeline-stall behaviour being handled by the core as in WOA, the ITC has blocking load and store semantics on some of the views, which cause the processor pipeline to stall because the memory operation has not completed.

Unlike WOA, which relies on the coherency protocol's tracking of the interested

core's "shared" cache line state, the ITC memory must be able to track outstanding operations from all possible cores, or deny requests (resulting in high traffic and active power while the core re-issues the request) or block the interconnect. This blocking is undesirable because it could lead to significantly reduced interconnect performance or even deadlock if the protocol is not designed well. Being able to track pending requests for all possible cores is wasteful however, and would grow linearly with the number of cores in the system. It might be possible for a single ITC to provide a single transaction tracker for a large number of cells, on the assumption that cores may not have a pending request on more than a single cell. Even if this could be addressed, the ITC limits the available shared storage for synchronisation primitives, unlike WOA, and is not easily mapped to existing software which is written with spin-wait behaviour. Because the ITC must be located on the physical address space it is also a shared resource between multiple concurrent applications, and this precious resource must be managed by operating system calls, potentially incurring overheads, and enabling one application to starve others of access to the required ITC resources.

7.4 Conservative Tree Encoding

Existing sharer representations discussed in Chapters 2 and 3 suffer from a number of shortcoming. When there are either a small, medium or large number of sharers the sharer encoding produces a sub-optimal result. Pointer based systems such as Dir_k[N]B, AckWise and SCI either require drastically growing storage for the sharer state, or have to resort to broadcast evictions once the storage space is fully consumed. Even if a large storage area is used to continue growing the pointer set, then a large number of invalidations must be sent one by one, as there is no convenient multicast mechanism. For lossy compression schemes there are other problems. For example, with coarse vector the large size of the sharer clusters results in many excess cores being sent invalidations when there are either a small number of sharers, or the sharer set is not well aligned to the clusters. Tristate/DirX has many pathological cases and software based Gray-coded Tristate results in sharer patterns with poor physical locality. Hierarchical coarse vectors suffer a similar, but less pronounced, growth problem to pointer based systems, but require an excessive degree of associativity in the directory, and also do not obviously provide a multicast scheme, although a multi-phase multicast can be constructed. To address the shortcomings this chapter presents a new sharer compression scheme, called Conservative Tree Encoding (CTE), which uses lossy tree

path encoding to make a best effort representation of the tree of sharers. This scheme can be scaled to the precision required, like the coarse vector representation, requiring $2\log_2(N)$ bits to be able to accurately represent a single sharer, in the same way as Tristate, but gradually degrading in spatial locality to full broadcast as more sharers are added. Adding more symbols (2 bits per symbol) will reduce the false invalidation rate, and even with only $2\log_2(N)$ bits will accurately represent power of 2 sized clusters of cores, aligned to the cluster size, with no false invalidation. This makes the proposed scheme a balanced trade off between the coarse vector and Tristate, with a configurable accuracy like the coarse vector, but with no minimum invalidation area.

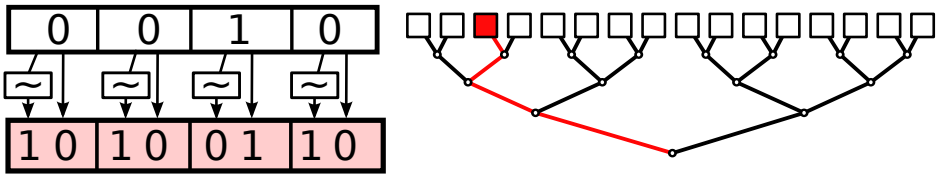
Symbol	**Meaning**
10	left branch only
01	right branch only
11	left & right branch interleaved
00	right & left branch interleaved

Table 7.1: CTE Encoding symbols.

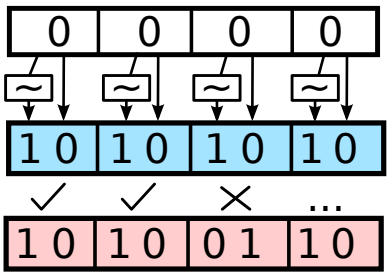
The encoding uses all four available states from the two bit symbol, listed in Table 7.1. To naively encode the tree, walk down from the sharing leaves, keeping a separate copy for each leaf, and at each node take the encoding so far, and prepend the relevant symbol for this node to the head of the encoding. If a shared node is encountered then the encodings for each subtree are interleaved, and the symbol 11 or 00 is used to indicate if the interleaving was left-right, or right-left alternating. To decide which to use the current head symbol on each subtree is examined, and if one of them has a 11 or 00 symbol the encoding that puts this on the right, or lower, in the interleaving pattern, is used. Eventually a limit on the available number of symbols is reached, and the encoding must be truncated, discarding the bottom symbols; it is for this reason that there are two encodings for a split in the tree. When symbols are discarded that whole subtree is lost, so it must be encoded with a fork node (00 or 11) when reconstructing on multicast, to produce a local broadcast on this tree. It is possible by using the 11 and 00 interleaving modes that one can chose to discard a 11 or 00 symbol in preference to a 10 or 01 symbol, reducing the amount of information discarded, and in doing so minimising the number of false invalidations sent. This selection between 00 and 11 is simply an optimisation that can be performed during

the encoding, it is not required to produce a "correct" encoding, but it helps address the asymmetry in encoding efficiency between left biased and right biased trees that otherwise occurs. A more advanced encoder could analyse beyond the next symbol to truly minimise the information loss, but most of the benefit can be achieved through the much simpler and less costly examining of only the next symbol.

To better explain the encoding a short example is presented, taking a 16-core system, and using 8 bits, i.e. 4 symbols, to store the sharer vector. This example walks through a series of insertions into the sharer vector, with core-2 making the initial request.

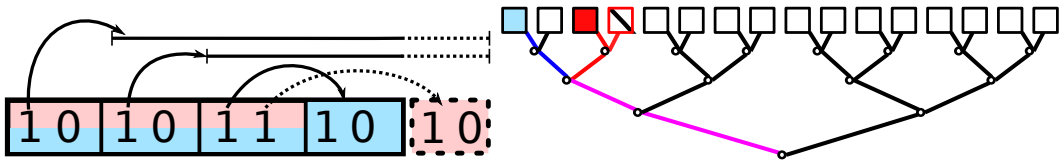


Core-2 is now initialised as the first sharer, the symbols trivially produced from the binary core-id. The upper boxes, in white, represent the core-id of the processor in standard binary notation, with the most significant bit on the left (core-0 is leftmost on the tree). By passing this through a series of logic inverters represented by the ' ' symbol the tree-encoding for this core-id is produced, with each symbol corresponding to a single bit of the core-id, with the left digit the inverse, and the right a direct copy. The tree diagram on the right demonstrates how this would be decoded climbing the tree, reading the symbols from left to right. The path to core-2 is represented in red for the remaining diagrams. Core-0 is added second, represented in blue, and will result in an overflow of the available symbol space.

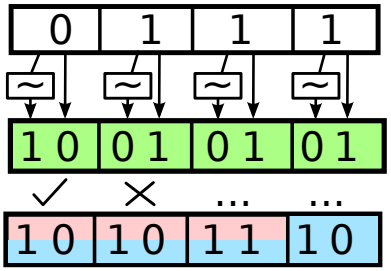


Walking the tree encoding, the first two symbols match, but the third takes the alternative branch. This means that the rest of the encodings must be interleaved. Since neither contains a 11 or 00 symbol in the next field, interleaving is performed left-

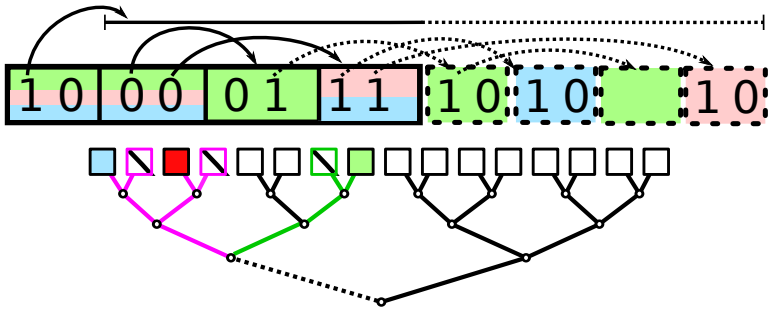
right, and truncates the path towards core-2. The sharer vector now conservatively must cover all cores on that subtree. The portion of the tree "owned" by each bit in the encoding is indicated by a curved arrow from this bit, either directly to the child nodes, or to an over-bar which extends over the node children. The symbols are shaded in a combination of colours to represent all of the sharers which contribute the that part of the encoding; because core-0 and core-2 share the bottom three levels of the tree in common (the root node counts as a level) the first two symbols are shared red-blue. After the split at the level three node the fourth symbol is entirely owned by the blue route to core-0, while the red route to core-2 has extended beyond the available space. Because this symbol is not available to store the '0' symbol filtering out core-3, it has become aliased into the encoding. Aliased cores are marked with a strike-through, and coloured with the sharer path which caused the aliasing. Truncated symbols are kept but represented with dashed boxes, to help with understanding the encoding and show where accuracy could be traded against encoding space.



Finally core-7 is added, represented in green, which will be followed by an explanation on how the tree gets easily decoded by the routers in the tree.



Core-7 only matches the first symbol, so once again interleaving is performed on the lower parts of the two encodings. This time the existing code has a branch symbol, 11, at the same level as the newly inserted 01 symbol. The encoding rules suggest that the next symbol on each branch should be compared, so that the interleaving order with the least information lost can be used. In this instance it might be beneficial to use the 00 encoding for this fork, because this places the existing 11 symbol further right in the encoding, preserving as much of the core-7 path as possible.



In this instance there are no additional savings using this interleaving, because only one symbol of each sub-tree will fit. However, if only three symbols were available in the first case, it would not have harmed the encoding to lose the 11 branch symbol, while it would have resulted in wasted traffic due to the significantly increased aliasing if the 01 symbol had been lost.

To decode this, each router in the tree needs to look at only the leftmost "head" symbol they receive, and does not need to do any processing of the remaining tail to determine what to do. The symbols 10 and 01 are simply interpreted as send the whole tail left or right, discarding the head symbol, while 11 and 00 tell it to reconstruct a left and a right message. This is done by concatenating every second symbol (e.g. 11 implies 2nd, 4th, 6th for left, 3rd, 5th, 7th for right), padding any remaining symbols with 00.

7.4.1 Advanced Insertion and Further Extensions

So far only inserting branch symbols has been discussed, but not how to easily insert into an encoding which has a branch as the first symbol that does not "match" the one being inserted. When walking the encoding to insert a new sharer, a branch symbol still counts as a match, but the stride is now doubled, to walk the interleaved path it is now inserting on, leaving the other branch intact. If another branch symbol is encountered, the stride increases to 4 symbols, and so on until there is no more available symbol space, or a difference is detected, requiring re-writing the path from this point onwards along the interleaved branch.

Insertion in this fashion is always $O(N)$ where N is the number of symbols allocated in the sharer vector. This means that the directory could take N cycles to insert a new sharer, but can send the AckS message before it has completed. The insertion is also simple enough that more than one symbol could probably be walked in a single cycle, and there is always the possibility to provide multiple instances of the insertion hardware into the directory logic, to handle multiple entries in parallel.

It is also possible to use CTE in a higher radix tree, for example a quad-tree could be encoded using 4-bit symbols, but the complexity increases with the need to handle 2, 3 and 4-way interleaved tails.

7.4.2 Synthetic Results

To better explore the proposed encoding, a set of synthetic tests were run using Tristate, Coarse Vector (CV), and CTE on a 1024-core tree, across a range of available encoding space. Tristate was only tested with 20-bits of storage as it requires precisely $2\log_2 n$ bits, where the scalable Course Vector and CTE encodings were tested with 16, 32, and 64, and 20, 32, and 64 bits respectively. The smaller comparison is slightly unfair as CTE and Tristate are allocated 25% more space than CV. However for 1024 cores CTE and Tristate can both identify a unique sharer with 20-bits, and CV makes the most sense implemented as powers of two, for extremely simple insertion and decoding, making 16-bits the closest encoding. With this power of two encoding CV provides exactly k equal coherency regions of size N/k, where k is the number of bits allocated and N is the number of processors. For each encoding and size combination two different tests were run: clustered, and random. The clustered pattern iterates through all possible contiguous sharer vectors, adding them one at a time then computing the number of cores which would receive a message upon invalidating them, to give 1024 data points for each of the 1024 possible numbers of sharers.

The random pattern operates similarly, except that for each number of sharers, 1-1024, 1024 random distributions of the sharers were generated, and the same multicast test performed as with clustered. The random pattern is the same for every test configuration. These two tests gives us a good picture of how the encoding will fare on programs of all sizes with both good and poor spatial sharer locality.

The graphs take two forms, a linear graph of true-sharers versus invalidations, and a log plotted (on the x-axis only) graph of the residual invalidations, i.e. those invalidations that occurred in addition to the expected true sharers. For the former the ideal result is a straight line of gradient one, from the bottom left to the top right, and for the latter a flat line along the x-axis. The residual plots are especially useful for comparisons of encoding quality, and the log transformed plot allows us to see detail for smaller sharer sets. Figure 7.7 summarises the poor Tristate results, with a linear plot of the clustered results at the top left already showing its failings, followed top right by a residual plot for easy comparison with the graphs in Figures 7.8-7.12. The bottom

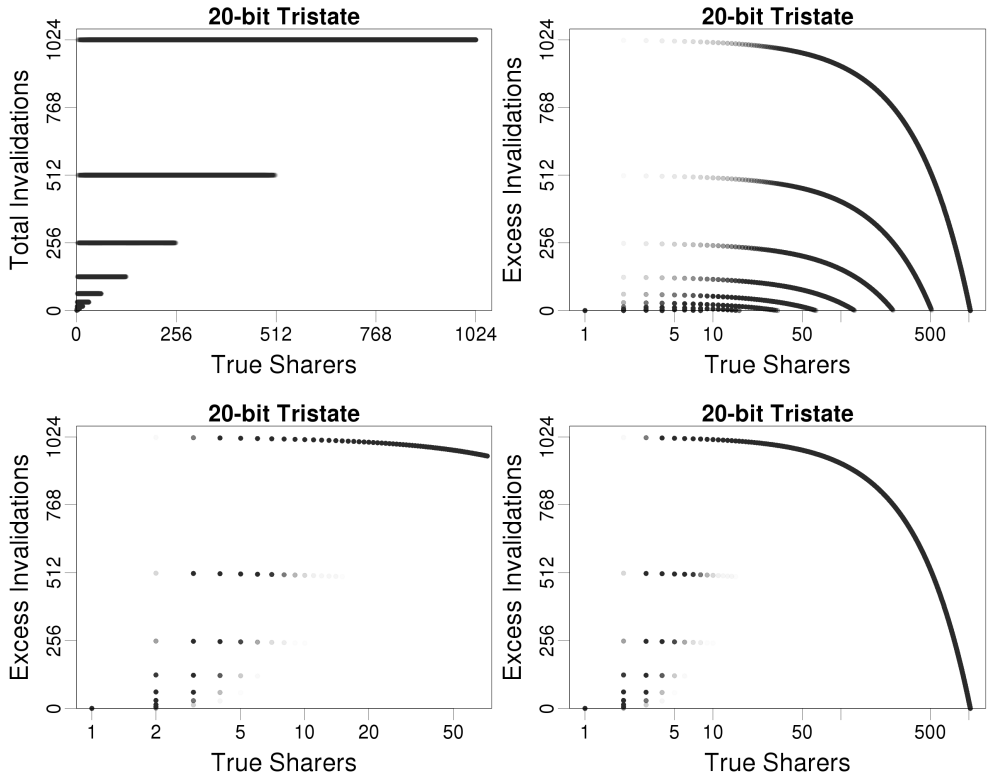


Figure 7.7: Tristate summary for 1024-core synthetic sharer simulations, each single result is an alpha-blended point giving a grey-scale density of the possible outcomes. Top-left: clustered, top-right: clustered residual, bottom-left: random residual 1-64, bottom-right: random residual 1-1024.

two graphs are a zoomed (1-64 sharers) and full (1-1024 sharers) view of the residual graphs for the random sharers. It is quickly apparent that a naive Tristate encoding is not a robust solution, and the nature of the encoding means it cannot be improved through additional encoding space.

A most contrasting summary of the differences between CTE and CV are to follow, but first it is good to get an overview of how each performs in absolute terms. The graphs in Figures 7.8–7.12 are arranged in paired columns, on the left are the CTE encoded results, and on the right are the equivalent CV results for direct comparison. Beginning with Figures 7.8 and 7.9 the clustered results for 20 and 32-bit CTE are presented, alongside 16 and 32-bit CV, with Figure 7.8 showing the linear plot of sharers versus invalidations. The initial impression is that CTE conforms much closer to the ideal line than CV for up to approximately 384 sharers, where the worst case performance for CTE begins to degrade. This is repeated in the 32-bit results, except with both encodings conforming much closer than their more constrained counterparts.

Switching to the residual graphs in Figure 7.9 gives a clearer picture of the performance with smaller sharer counts. CV's bipolar performance is due to the cluster alignment resulting in the cluster fitting either in the minimum number of CV regions, or the minimum plus an extra region as some sharers slightly spill into the next. This means that the worse case excess for CV is always two whole regions, minus the number of sharers modulo the number of cores in each region. This can be good for a large number of sharers, where it outperforms CTE, but the alternative solution, the best case for CV, is a whole region of cores minus the number of sharers modulo the number of sharers per region. This only provides a good result when the number of sharers is close to the size of a cluster, and must be well aligned to avoid transitioning to the worst case. For less than 64 sharers the 20-bit CTE is significantly better than 16-bit CV, and due to the 64-core cluster size of 16-bit CV up to approximately 20 sharers there are no contiguous sharer patterns where CV provides a better solution. Moving to look at the 32-bit results, take note that the Y-axes range has been halved with this increase in precision (and will be halved again for the later 64-bit results summary). Here it can be seen that while the CV results are twice as good across the board (to be expected from halving the size of the coherency regions), the CTE results for under 64 sharers are almost four times as good, degrading to only twice as good for the larger numbers of sharers. This shows that for smaller numbers of sharers (<100 on a 1024-core system) CTE not only provides much better tracking accuracy than CV for the same number of encoding bits, its accuracy improves at a much greater rate than CV when increasing the provision of more space. Unlike CV, the CTE encoding can be easily extended two bits at a time with minimal change, where CV makes most sense using power of two encodings. It should be noted that CV can be used with an arbitrary number of bits (as was done do for the real-world benchmark results) with a encoding scheme still slightly less complex than CTE.

The excellent small sharer performance scaling of CTE is exemplified by the almost perfect results for up to 10 cores with a 32-bit sharer vector (otherwise capable of only holding three core-pointers), and looking ahead to Figure 7.12, using 64-bits of sharer state allows any number of sharers, up to almost 64 cores, in any contiguous placement on the processor network, to be represented with negligible excess messages. This is not possible with any other fixed size sharer encoding other than full-map (requiring 1024 bits of storage) or Dir$k$[N]B with $k = 64$ for 64 core-pointers, using 640 bits of storage ($10\times$ that required by CTE) and lacking any multicast support.

Moving Down to Figure 7.10 and 7.11, the results of random sharer distributions

are compared. Beginning with a linear plot of invalidations versus sharers again it can be seen that neither encoding handles random sharer distributions well, and this will be an inherent problem of all compressed sharer formats – random data is incompressible. Again CTE is shown to perform better for small numbers of sharers, but above 5 sharers coarse vector begins to perform better, and although adding more bits of state sees CTE take a clearer lead for small numbers of sharers, coarse vector gets even better for a large number of random sharers. The trends continue with the 64-bit results of Figure 7.12, with CTE performing better for clustered sharers and small numbers of random sharers, while coarse vector performs best for more than a few randomly distributed sharers.

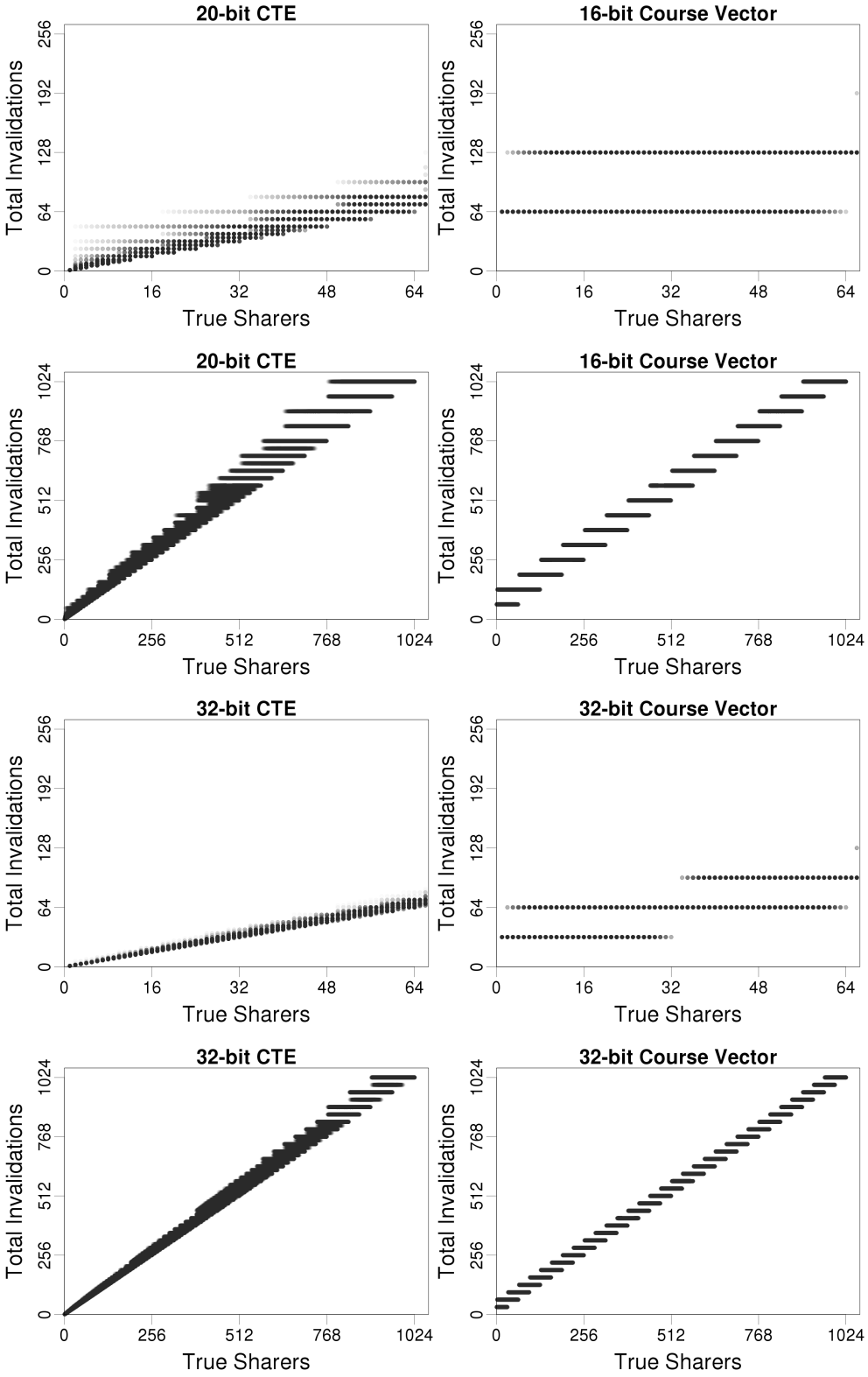


Figure 7.8: Clustered invalidations received on a 1024-core tree.

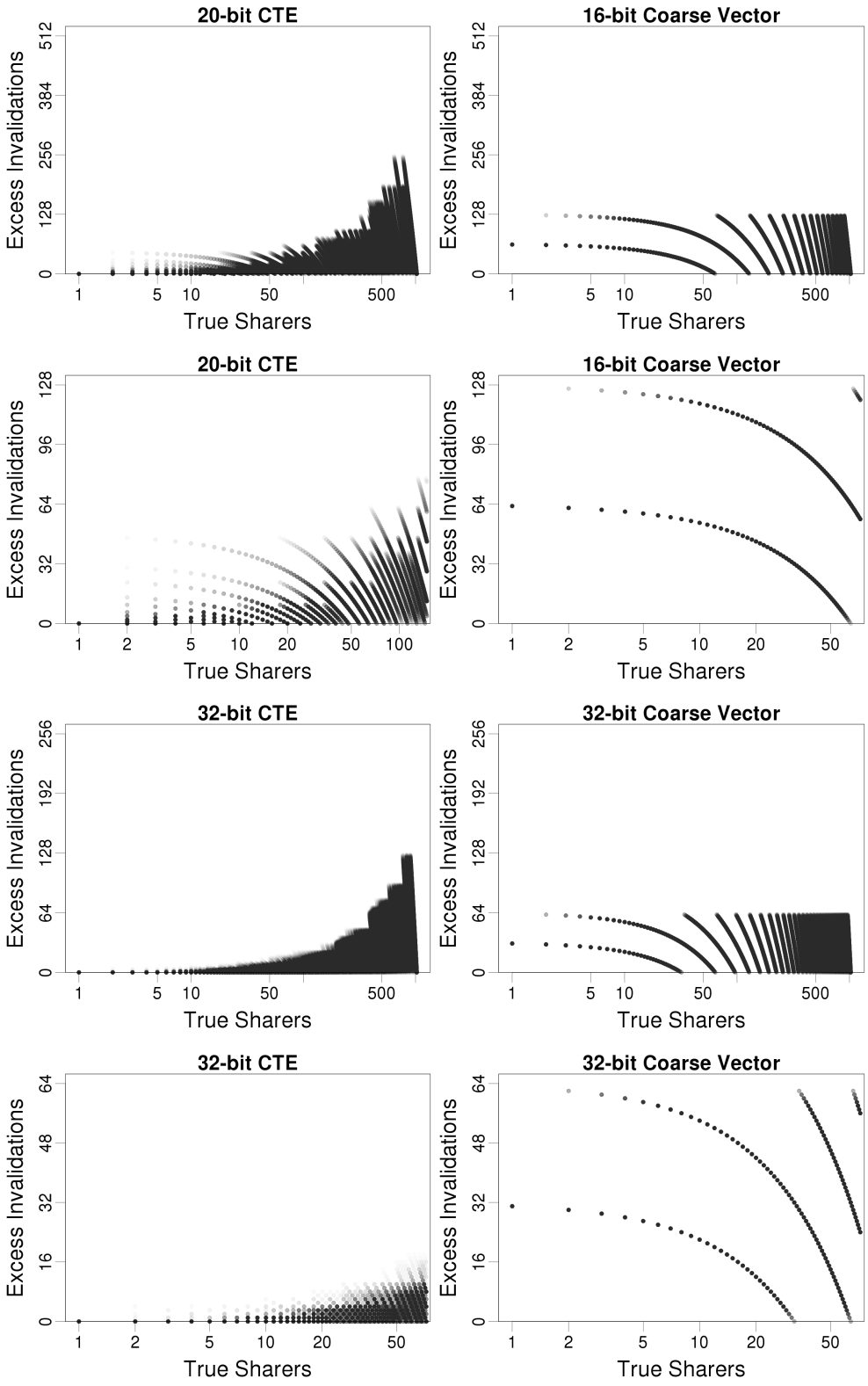


Figure 7.9: Excess clustered invalidations received on a 1024-core tree.

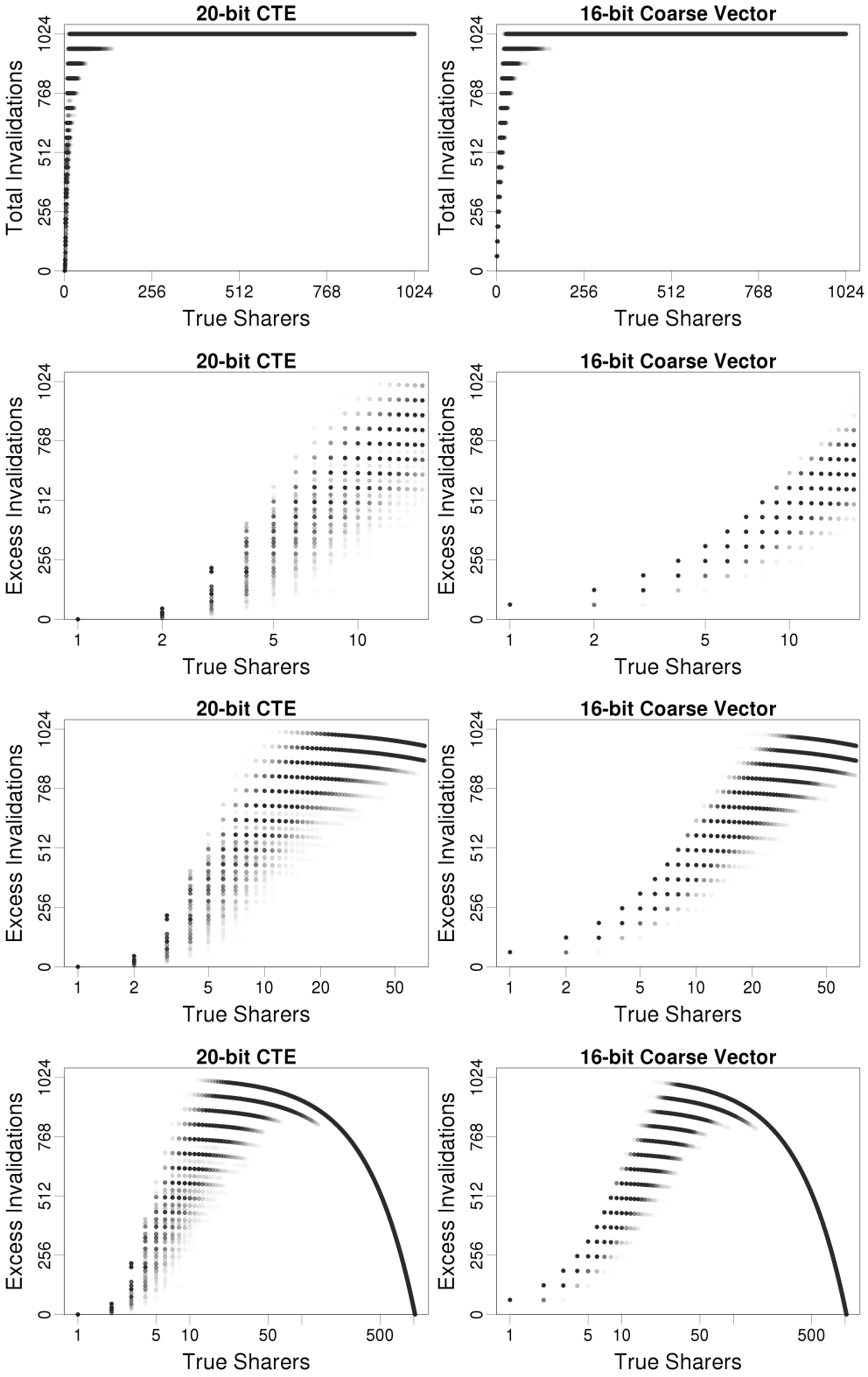


Figure 7.10: Random invalidations received for small sharer encoding space.

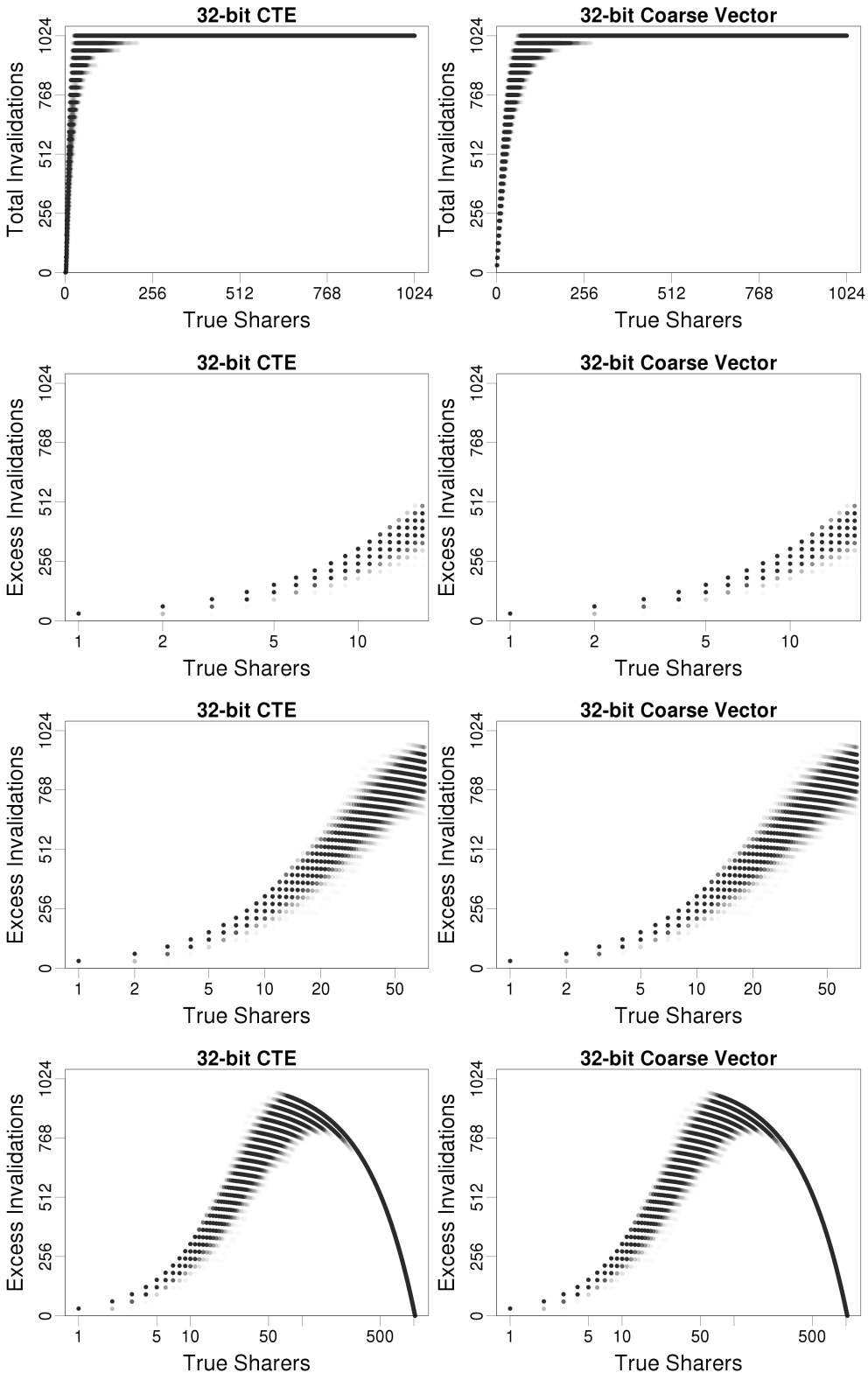


Figure 7.11: Random invalidations received for 32-bit encoding space.

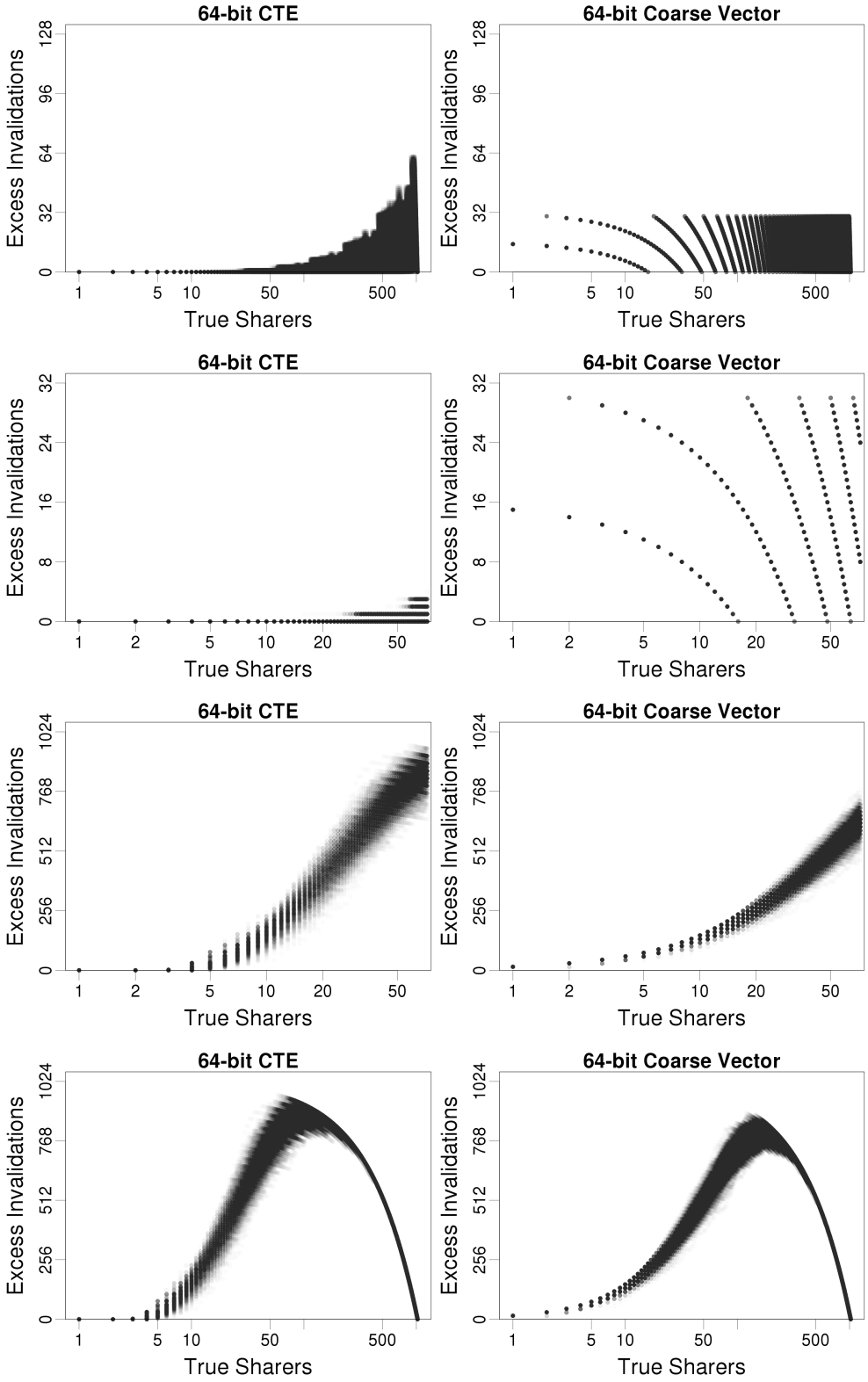


Figure 7.12: Clustered and random invalidations received for 64-bit encoding space.

Following the context provided by the absolute compression performance, the next series of figures focus on directly comparing 32 and 64-bit CTE and CV on a sample-by-sample basis. As with the previous figures, for each number of sharers (1-1024), 1024 simulated sharer additions and invalidations were performed, with the same sharer patterns used for all experiments of the same distribution. For each size-distribution combination the number of invalidations made by the CV encoding was subtracted form the number of invalidations made by the corresponding CTE encoding. This provided a distribution of the relative difference in effectiveness between CTE and CV for each number of true-sharers. Figures 7.13 and 7.14 show these distributions in terms of the percentage of sharer combinations in which CTE outperformed CV by the number of invalidations shown on the y-axis, or CV outperformed CTE on the negative axis. For example for 32-core clustered, 100% of two-sharer patterns had at least 30 extra invalidations for CV compared to CTE, while approximately 10% of these had 62 extra invalidations.

These results are further summarized in Figure 7.15 which plots the median differences from Figures 7.13 and 7.14. It is clear from these figures that the CTE encoding is superior for clustered sharer groups involving less that 512-cores, with little difference between the encoding above this, but random distributions are less clear cut. For the 64-bit encoding CTE performs best up to nine sharers, but beyond this CV proves to be much more robust, while with 32-bits CTE is even worse, losing out after only five sharers. As such the best choice of encoding really depends on the sharer distributions present in the target application. Figure 7.16, which shows the difference as a proportion of the true sharers rather than an absolute, shows that the advantage of CV is less pronounced when the excess invalidations are considered as a proportion of those being invalidated.

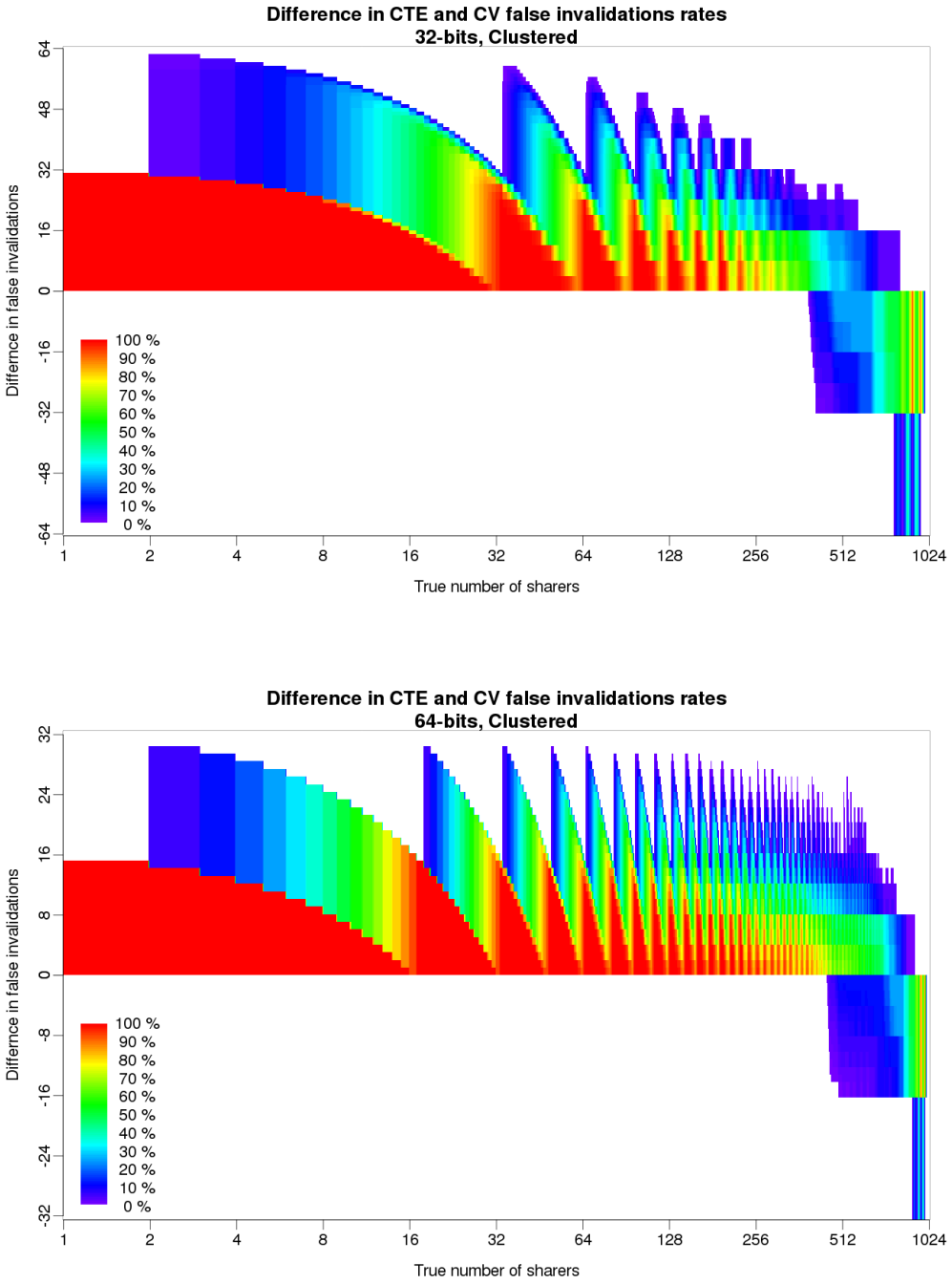


Figure 7.13: Distribution of the difference in false invalidations between CTE and CV for clustered sharers. Distribution represents the percentage of sharer combinations with at least as much difference as indicated on the y-axis. Positive difference represents how much better CTE has performed, negative represents CV. The median performance passes through the green band at approximately 50%.

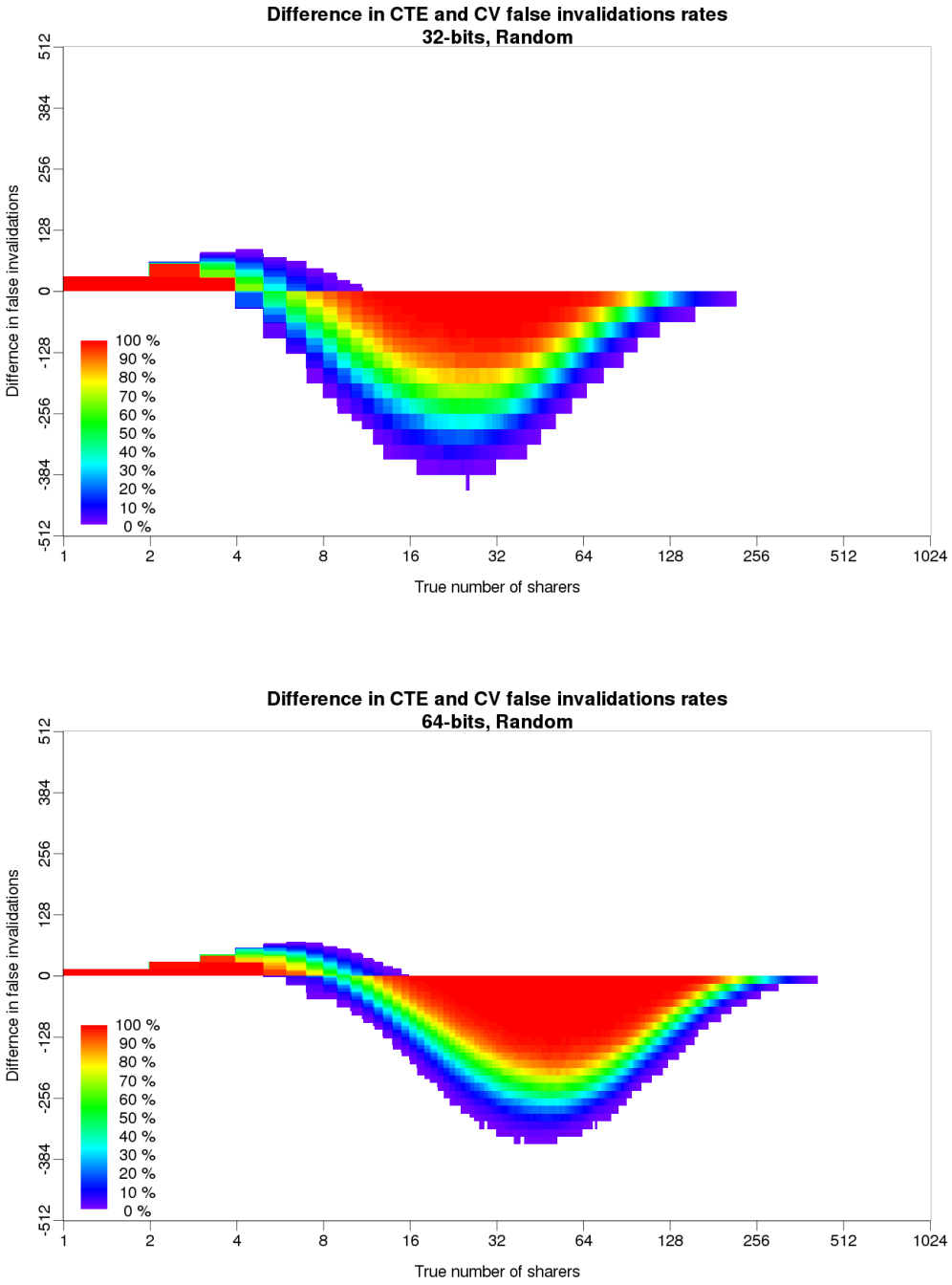


Figure 7.14: Distribution of the difference in false invalidations between CTE and CV for random sharers. Distribution represents the percentage of sharer combinations with at least as much difference as indicated on the y-axis. Positive difference represents how much better CTE has performed, negative represents CV. The median performance passes through the green band at approximately 50%.

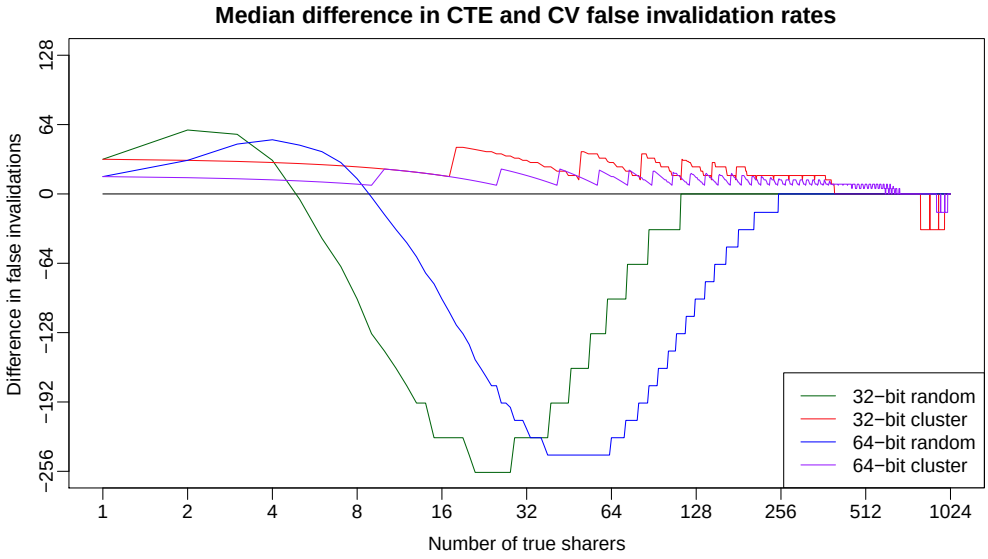


Figure 7.15: Median difference in excess invalidations between CTE and CV, positive numbers indicate that CV has a greater number of false invalidations.

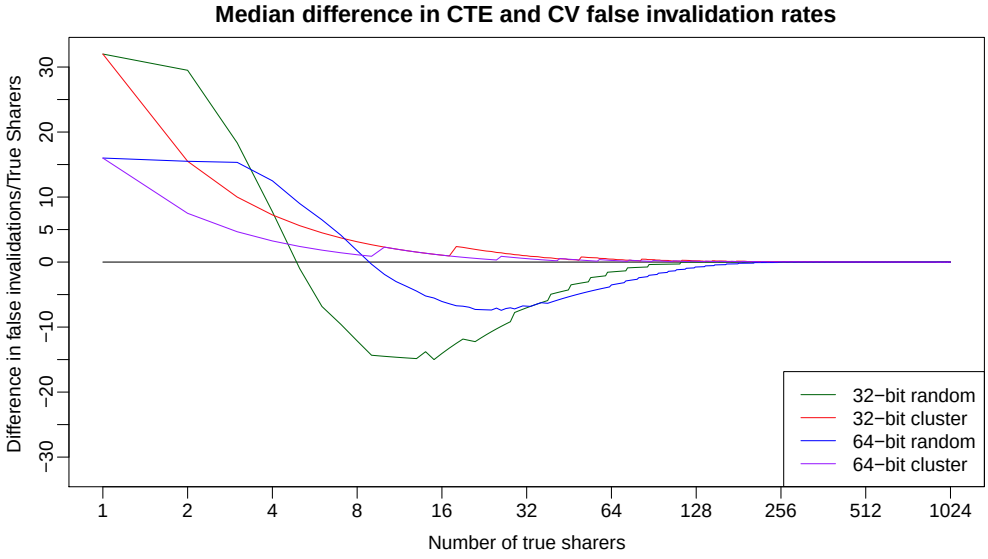


Figure 7.16: Median difference in excess invalidations between CTE and CV, as a proportion of the number of true sharers. Positive numbers indicate that CV has a greater number of false invalidations.

7.4.3 Benchmarks

Due to limits on architecture support, scalability of benchmarks, and available time for simulation, experiments were performed on a small subset of the Splash2 [137] benchmark suite which had previously indicated that they might support running on up to 1024 cores, and which indicated performance scalability at least as far as 32 cores [12]. The selected benchmarks and their configurations can be found in Table 7.2.

Benchmark	**Configuration**
Radix	524288 Keys
FFT*	2^{20} points
LU (contiguous)	512x512 Matrix
Ocean (contiguous)	258x258 Ocean

Table 7.2: Benchmark Configurations (*FFT was modified to parallelise the dataset initialisation, and a buffer of random numbers was pre-computed to further reduce this phase).

7.4.4 Benchmark Results

The final test comes with the simulated results from the four Splash2 benchmarks listed in Section 7.4.3. Simulations were run using $2\log_2(N)$ bits of sharer state for both CTE and coarse vector, although since these are not all even powers of two, the coarse vector represents a balanced tree distribution of the available bits, with each fork in the tree taking half the remaining bits (where there is one extra it always gets allocated to the left branch, towards lower numbered cores).

From these simulations traffic statistics were collected for all multicast packets, which represent shared line invalidations. From Figure 7.17 you can see CTE can reduce the multicast traffic by up to 12.7%, for 32-core Ocean, but the results are extremely benchmark specific, both LU and FFT suffer increased traffic, indicating many sparse sharers.

To combat this problem a hybrid scheme was devised, which on inserting a new sharer into a CTE vector, will determine if the new sharer state would be better suited to coarse vector. The current implementation does this by using the current CTE to construct a coarse vector representation of the current sharer list, then inserting the

new sharer into both; the number of cores which would be invalidated in a multicast is calculated, and the encoding with the lowest value is selected. Once the representation has switched to coarse vector it does not switch back, working on the assumption that coarse vector handles large numbers of sharers better. The only instance where it may be beneficial to switch back is 32-core Ocean, where CTE beats the hybrid scheme, however this difference is well worth the prevention of cases where CTE is outperformed by coarse vector. On average (geometric mean) the hybrid scheme reduces multicast traffic by 7.7% over coarse vector alone, and almost always outperforms both CTE and coarse vector alone.

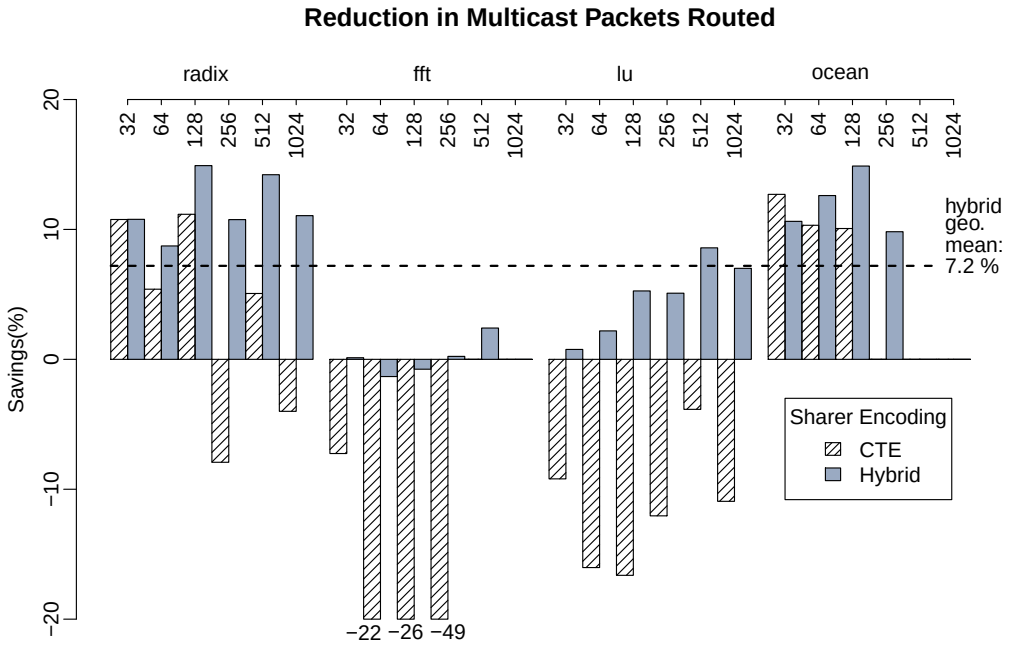


Figure 7.17: Reduction in multicast packets using CTE and hybrid CTE-CV over coarse vector alone.

7.5 Further Work

There are many potential extensions of this work which merit investigation. For example the CTE multicast can be applied to virtual trees, allowing the directory to be split out across the mesh, sacrificing the need for Ack messages in return for allowing the directory to initiate a memory request for the core, rather than delegating that to the core after it has received permission from the directory. This trades off bandwidth and energy in one part of the protocol, for bandwidth and energy in another. Perhaps

the most interesting and important future work will be evaluating the multicast CTE support with a multi-program workload, as its biggest advantage is with small groups of geographically clustered cores.

An additional piece of work which could be investigated is another take on "tagless" directories. Currently the nature of L1 caches means that a directory must be highly associative, or have some means of spilling into a secondary data structure to maintain a scalable coherency protocol [34]. Instead, one could have a very large direct mapped directory, without address tags ("tagless" but not in the same sense as Zebchuk [64] proposes). By indexing into the directory with just the lower address bits (as in a simple direct mapped cache) the lookup is extremely cheap, with no tag check required, and the set of all mapped addresses can be easily reconstructed by prefixing a "match-all" state to the index bits. This structure would have very cheap indexing, at the expense of an invalidation being aliased to not just multiple cores, but to multiple addresses too. When a directory line must be evicted to satisfy an exclusive request the set of all cores stored as sharing the directory line must be messaged with a multi-address invalidation, which matches all potential addresses mapped by the directory line. Since the directory should have a greater number of lines than any individual set in a processor L1 cache, the processor can uniquely map the request to a single index, and then must compare the lower tag bits with the request address bits formed from the directory index. This large multi-processor multi-address aliasing is easiest to handle with a silent eviction protocol such as the one presented in this thesis, where upon multicasting the invalidations the directory does not need to know how many acknowledgement messages to wait for, and can immediately continue processing the request. Predominantly read only data can be aliased quite safely, but private write-heavy data could be problematic. Two potential solutions to this are: to make good use of page-tracking to exclude both private and read-only pages, and/or have a small associative tagged backup directory which high-contention entries switch over to under pathological conditions. This proposed directory could provide significantly more tracking for the same given silicon area, due to the absence of tag RAM, nor the need for multiple parallel-lookup sets required for associative systems. These benefits would also reduce the time and energy required to service a request. In general the aliasing problem will be reduced by over-provisioning, but it is still likely that a mechanism for handling pathological cases would be required to get the best performance.

Chapter 8

A Machine Learning Based Approach to MPSoC Design

8.1 Foreword

This chapter was originally produced as a collaborative work with Dr. Oscar Almer. Inspired by his previous work on predicting interconnect features [121; 46] we worked together to integrate the simulation infrastructure developed in Chapter 4 into the machine learning techniques Almer had used previously. In this work the MPSoC design space and benchmark configurations were based upon the previous work, but decided upon and developed by myself. Simulation orchestration was entirely my responsibility, along with feature selection to be provided to the machine learning models. Almer took responsibility for running the machine learning models, due to his experience with them already, and produced the initial regression figures, however the subsequent histogram based analysis and ranking analysis was my own work. Almer also contributed to the related work section of this work, which has been collated with the other related work sections in Chapter 3.

8.2 Introduction

Increasingly capable MPSoC (Multi-Processor System-on-Chip) devices are powering much of the social revolution in computing. The scope of these devices does not stop there; MPSoC devices are increasingly being used as the paradigm of choice for delivering new products. Whether shrinking existing multi-chip designs into a single chip or expanding from a single processor on a die, the MPSoC trend is currently prevalent

throughout the electronics industry.

With the integration of multiple compute cores and IO devices on a single die, the configuration and joint capabilities of the system as a whole become first-class concerns for designers. Unfortunately, the sheer size of MPSoC devices preclude exhaustive evaluation of the design options, even for relatively modest designs. In addition, many MPSoC designs are manufactured for specific tasks, and are intended to run only specific software, something which traditional partitioned optimisation does not address. The space of MPSoC designs for even a single task is larger than can be effectively evaluated.

At the same time, power has become a primary design constraint, giving another goal for the optimisation of MPSoC systems along with area, functionality, performance and cost. Achieving optimality on all these metrics simultaneously is becoming increasingly difficult with increasing on-chip resources and configurability. The only way to address all these constraints at once is to co-design hardware and software to achieve an efficient application-specific MPSoC design. As software is considerably more mutable than hardware, and can be changed significantly in shorter time, it is imperative to be able to quickly find a good MPSoC hardware design given some software. Changing the hardware to suit the software reactively in this manner enables software development to go ahead with less regard to the hardware architecture, knowing that an efficient hardware solution can be found given the software implementation.

This chapter addresses the problem of exploring the multitude of hardware options available for even a single new piece of software. The ultimate goal is to have an automated system which can be given a new piece of software, analyse it, and quickly suggest an optimal hardware MPSoC based on learned heuristics about the hardware design space. To this end, this chapter discusses machine learning methods for finding optimal hardware design points for new programs given incomplete data about the hardware design space. This work evaluates several methods of machine learning over a generated set of MPSoC designs drawn from a large design space by attempting to predict the performance of new programs both in terms of runtime and approximate power consumption.

It is concluded that machine learning is applicable to this problem, and that Random Forest is the most appropriate for the task of those algorithms investigated. This work also concludes that by using machine learning for this task, solving the design-selection problem can be efficiently automated.

Section 8.3 and 3.4 discuss this work's relationship to other work in this area.

Section 8.4 outlines this work's approach to the problem in detail and discusses the example design space used. Section 8.5 describes the experiments performed in applying the approach to the MPSoC design space. Section 8.6 discusses the results of applying this methodology to the example design space. Section 8.7 contains the conclusions.

8.3 Related work

One of the main differences between the previous discussed in Section 3.4, especially the work of Almer *et al.* [121; 46], and this work is the platform used to run the simulations, and the scale of the design space covered. While targeting a similar NoC architecture to Almer *et al.*, a pure software simulation approach is used, allowing the computational work to be scaled out to a compute cluster to cover a larger portion of the design space (1025 designs rather than 71). In using software simulation this work is also not constrained to the size of designs which can be synthesised to a single FPGA, allowing exploration of designs up to 64 cores. Although fewer workloads are simulated in this thesis (64 rather than 82), a greater range of workload size is covered, ranging from 1 to 64 tasks, and greater emphasis is placed on memory heavy workloads, by providing a workload which taxes one of the IO devices on the target platform, and an extremely traffic-heavy memory-thrashing benchmark. This work achieves slightly better average accuracy predicting when the best designs for runtime and EDP, with fewer results being especially badly predicted. It also presents results across a variety of machine learning techniques, and gives a better insight into the distribution of the design space in Figure **??**.

8.4 Methodology

To address the problem of finding methods for predicting the performance of MPSoC designs, a data set correlating design features to performance is required. Such comprehensive data sets are not readily available, and consequently one was generated using own using the hardware-calibrated multi-threaded fast MPSoC simulator presented in Chapter 4.

The simulator has been verified against MPSoC systems implemented in FPGA hardware, and has been found to generate runtime and interconnect switching counts within 3% of the actual MPSoC designs. It is therefore considered as accurate for the purposes of this work. The main benefit of the simulator over the FPGA implementa-

tions is in synthesis speed; the simulator can be reconfigured significantly faster than the test devices, fractions of a second relative to several hours, and therefore leads to faster evaluations. The second is that training data can be generated in parallel on a compute cluster, providing significantly greater design throughput than a limited number of FPGAs.

To address the hypothesis that the proposed approach is applicable to any new software, a large number of computational benchmarks were generated and run on the calibrated MPSoC simulator.

Section 8.4.1 describes the design space parameters used for the MPSoC design space as well as for the benchmarks. Section 8.4.2 describes the design space parameters used for benchmarking these MPSoC designs. Section 8.4.3 discusses the machine learning methods evaluated for predicting performance.

8.4.1 Hardware Configurations

For this work the design space of MPSoC configurations ranging from 1 to 64 cores, with an binary-router based NoC architecture is considered. As in the previous work by Almer *et al.* [121], the cores are grouped into clusters on the interconnect, through a cluster level arbiter. To confine the design space it is restricted to homogeneous designs, those with the same number of cores in each cluster. The configurable parameters are:

- The number of cores per cluster (1 to 8).
- The number of clusters (1 to 8).
- The number of RAM banks (1, 2 or 4).
- The core to interconnect clock frequency ratio.
- The FIFO buffer size in the interconnect switches.
- The interconnect "complexity" – the depth of the interconnect network connecting the cores to memories and devices.

This gives a design space of 12288 unique hardware configurations, from which 1025 designs were chosen randomly, and with uniform distribution, to use for evaluations.

The design parameters of the MPSoC are identical to those presented in Chapter 4, summarised again in Table 8.1

Design Parameter	Possible Configurations
Core Architecture	ARC700 32-bit RISC
Pipeline	3-stage in-order
D-Cache Size	4 KB
D-Cache Associativity	Direct Mapped, 2-Way
I-Cache Size	4 KB
I-Cache Associativity	Direct Mapped, 2-Way
Cache line size	32 Bytes
Interconnect Protocol	AMBA AXI
Interconnect Topology	32-bit wide binary-routing network
Coherency Protocol	None – Cache-Incoherent
Cores per cluster	1 – 8
Clusters	1 – 8
Block RAMs	1, 2, 4, 8
Total Block RAM Size	2 MB
Complexity	1, 2, 4, 8, 16
Fifo Depth	2, 16
Core Freq(MHz)	12.5, 25, 50
NoC Freq(MHz)	12.5, 25, 50, 100

Table 8.1: MPSoC design configurations.

8.4.2 Benchmark Configurations

To evaluate the machine learning methods on this design space a set of multiprogrammed embedded benchmarks was generated.

As in the verification section of Chapter 4, the components of these benchmarks were chosen from:

- 5 EEMBC benchmarks: AutCor, Conven, FBital, FFT and Viterbi.
- CoreMark.
- Two in-house synthetic workloads: a cache-thrashing benchmark which generates a dirty cache miss every two out of three instructions, and a panning image benchmark.

The panning image benchmark not only generates significantly higher cache miss rates than the EEMBC benchmarks, but also produces uncached IO to the on-chip display controller.

Four workloads were generated each for 1, 2, 3, 4, 6, 8, 12, 16, 20, 24, 28, 32, 36, 48, 56 and 64 tasks, making 64 workloads in total. The workloads were generated with a two-phase random selection algorithm. For any workload size, the first phase selects a subset of 2, 4, 6 or 8 tasks from the full set of tasks at random. For each of the chosen subsets a workload is then generated using only the tasks in that subset. For each size of workload, this method generates both workloads that contain little variation between tasks (more homogeneous workloads as might be found in a parallelised signal processing application), and workloads with many different tasks (heterogeneous workloads as might be found in a process controller for a realtime system or the main processor in a multi-media device). This provided a diverse set of workloads for good training coverage while ensuring that there are no duplicate workloads.

The runtime scheduling of tasks to cores was done statically on each simulated MPSoC system, so that each core in the system was assigned a fair number of benchmarks to run. This means that the maximum runtime of the system depends on the core(s) with the most and/or longest tasks. The task selection method therefore generates varying runtime from relatively similar benchmarks, exposing the complexity in predicting runtime for arbitrary software.

It is important to get a set of workloads which feature a good mix of compute heavy and memory and I/O heavy benchmarks, because these features can drastically alter the ideal MPSoC selection. A workload with all threads performing compute tasks

will favour a design with a high core frequency, while one with high memory bandwidth requirements will favour a higher bus and memory frequency. Similarly an I/O intensive application may favour lower latency to memory even if overall throughput is sacrificed, compared to a streaming memory application where latency is amortized. On their own the design choice for each is fairly straight-forward, but with a mixed workload the trade-off must be made between each of the threads to find the best balance. The workloads used are designed so that on an average system they all execute within approximately the same length of time. This is to ensure that the problem is not dominated by accelerating a single thread which runs for much longer than the others, but enough variation is ensured so that some mixtures will favour different design points. The design space is intentionally constructed so that maximum core speed is only possible with minimum interconnect speed, and vice versa. Homogeneous workloads are likely to favour one extreme or the other, while heterogeneous workloads will have to find a balance to suite all workload threads, else risk leaving a poorly matched thread to dominate runtime.

8.4.3 Machine Learning Methods

The main interest of this work lies in identifying suitable machine learning methods for predicting performance metrics of MPSoC systems. As a baseline for comparison, an empirically random prediction method is used, that simply returns a random value from the training set as the prediction. This method has the upside that the prediction(s) will match the distribution of the training data, but is otherwise random. It can therefore be regarded as a worst-case useful predictor, in that it does not actually use any of the features of the design space, but still returns results of the right distributions.

For actual predictions the following methods were chosen for evaluation:

- A standard linear model in N dimensions, to ascertain if results can be obtained even with so simple a model.
- A *wk*NN model, calibrated through internal cross-validation in the training data set.
- A Multivariate Adaptive Regression Splines (MARS) model built from the training set.
- An Artificial Neural Network tuned for regression. ANNs are calibrated through internal weight adjustments using the training set.

- A Decision Tree model in regression mode. This method has implicit internal feature selection.

- Random Forests of decision trees, again internally calibrated through cross-validation over the training set.

The linear, *wk*NN, MARS and ANN methods do not have internal feature selection, so for these methods, external feature selection is required.

To evaluate each machine learning technique leave-one-cross-validation was used: for each workload all data from this workload was excluded from the simulation data to provide a training set, which was then used to train the machine learning algorithm. Predictions were then made for each design for which data was available for the excluded benchmark, and the predicted result compared against the simulation result. This process is repeated for each workload and machine learning technique.

8.5 Empirical Evaluation

To evaluate the design space the benchmarks, discussed above, were simulated running on the 1025 MPSoC designs. This section discusses the generation of the MPSoC performance data through simulation in Section 8.5.1 and the machine-learning implementations used in Section 8.5.2.

8.5.1 Data Set Generation and Features

Due to constraints in the scheduler on the systems the simulator was executed on, simulated benchmarks with a small number of tasks were favoured over those with higher task-counts. This led to particularly simplistic benchmarks featuring more often in the evaluation data than may be expected.

A constraint added to help manage the time taken in generating training data was that workloads would not be run on designs that would result in more than 4 tasks being scheduled on a single core of the target design. The reasoning was that an engineer would make this decision anyway because the design would be unlikely to meet the real-time constraints of the task. Similarly, if the simulator detects that a simulation has executed for four billion core cycles on the design it will time out for the same reasoning – this design fails to meet the required performance for this workload. Unfortunately this means the machine learning system does not see this training data,

so must predict based upon designs that pass these requirements; it also means that during cross validation to evaluate the results the testing set is limited, because only predictions that have been simulated to completion can be verified.

In total 18602 different benchmark runs were generated, over 1025 simulated hardware designs. (This is different to Chapter 4 because additional design configurations were simulated in parallel with the compute cluster on other available workstations, in order to increase the design coverage as much as possible. However the compute cluster continued to run after the research for this chapter concluded, so eventually it produced more simulation runs, but over a slightly reduced number of designs. The extra results from these other simulations were not included in the simulation performance analysis in Chapter 4). The design space explored for each benchmark is therefore a subset of the 1025 chosen MPSoC designs discussed previously, but the constraints meant that not all MPSoC designs ran all benchmarks. One cannot say how large a part of the whole design space this explores, as the workloads used are effectively unbounded; any program that can be run using the presented system could be considered another benchmark. Taking only the narrow definition of workloads constructed as there were generated in this work, however, only 64 out of around 10^{37} possible benchmarks were used, and 1025 out of 12288 MPSoC designs, covering a minute fraction of the combined design space.

Training data generation time was on the order of a few weeks real-time, but as each simulation is independent from each other simulation, more data could easily be generated in parallel, subject to computing resource availability. The simulation time of the presented approach can therefore be very short, if parallel simulation infrastructure is available.

The features collected and provided to the feature selection algorithm (or machine learning algorithm directly for those with internal feature selection) were the following design features: clusters, cores per cluster, RAMs, complexity, fifo depth, core multiplier, interconnect multiplier, cores, and threads.

Along with the following runtime statistics collected from the workload run on two representative designs which form the input feature for prediction: max runtime, min runtime, average runtime, total runtime, max IPC, min IPC, average IPC, max IO ratio, min IO ratio, average IO ratio, max read request flits (AR), min AR, average AR, total AR, max write request flits (AW), min AW, average AW, total AW, max write data flits (W), min W, average W, total W, max read data flits (R), min R, average R, total R, max write acknowledge flits (B), min B, average B, total B, total switching events (sum

of flits×bus-width for all flits).

And along with the static frequency of each of the major instructions from the ARC700 ISA found in the binary: abs.f, add, add1, add1.n, add2, add2.c, add3, add3.n, add.c, add.f, add.nz, add_s, add.z, and, and.f, asl, asl.f, asl_s, asr, asr.f, asr.lt, b, bbit1, bc.d, b.d, bge, bge.d, bgt, bgt.d, bhi, bhi.d, bic, bl, bl.d, ble, ble.d, bls, bls.d, blt, blt.d, bmsk, bnc.d, bnz, bnz.d, breq, breq.d, breq_s, brge, brge.d, brhs, brhs.d, brlo, brlo.d, brlt, brlt.d, brne, brne.d, brne_s, bset, bxor.f, bz, bz.d, cmp, cmp_s, extb, extb_s, extw, file, flag, j, j.d, jeq_s, jl, jl.d, j_s, j_s.d, ld, ld.a, ld.ab, ld.as, ldb, ldb.a, ldb.ab, ldb.di, ldb_s, ldb.x, ld.di, ld_s, ldw, ldw.ab, ldw.as, ldw.x, ldw.x.ab, ldw.x.as, lp, lr, lsr, lsr.f, lsr.nz, lsr_s, lsr.z, max, min, mov, mov.f, mov.ge, mov.gt, mov.hi, mov.le, mov.ls, mov.lt, mov.nc, mov.nz, mov_s, mov.z, nop, nop_s, not, or, or_s, pop_s, push_s, ror.f, rsub, rsub.hi, rsub.lt, rsub.nz, rtie, sexw, sexw.f, sleep, sr, st, st.a, st.ab, st.as, stb, stb.ab, stb.di, stb_s, st.di, st_s, stw, stw.a, stw.ab, stw.as, stw.di, stw_s, sub, sub1, sub2, sub.f, sub.ge, sub.ls, sub_s, tst, tst_s, xor, and xor.lt.

The instructions frequencies were extracted with the following shell code:

```
objdump -d --no-show-raw-insn executable
| sed "s/^.*\://"
| awk  '{print \$1 }'
| perl -e 'while(<>){
             chomp; $1{$_}++;
           };
           foreach $i (keys %1){
               print "$i -> $1{$i}\n";
           }'
```

8.5.2 Machine Learning Execution

Implementations of the machine learning methods from the R [149] packages ‘kknn’, ‘nnet’, ‘earth’, ‘rpart’, ‘randomForest’ and ‘e1071’, were used, as well as feature selection using the package ‘FSelector’. These packages are available as downloadable and installable libraries from within R itself, and each package is publicly available code. As such these packages are considered to be accurate and efficient implementations of their respective machine learning methods.

The feature vector for each simulated program consisted of information about the MPSoC system used, i.e. the hardware configuration, together with benchmark-specific information. For each benchmark generated data from two runs of the bench-

mark on sample designs were used as features, together with data about the instruction mix in the benchmark. Taken together these features formed a feature vector of some 200 individual features.

Feature selection was performed by computing Spearman's correlation over the training set, obtaining scores for the linear dependence of features against the regression target. A cutoff in this ranking was then chosen as the point of largest difference between scores, and features above this cutoff were used to build the models. This means that only features that are well-related to the regression target are used. In general, and depending on the training subset used, feature selection chose to utilise around 70% of the features present, omitting primarily instruction mix features.

For Decision Tree and Random Forest methods the machine learning algorithms provide their own internal feature selection, so this was used directly without applying Spearman's correlation and cuttof. The algorithms were provided with the whole feature vector in all instances.

8.5.3 Design Space Transformation

Initially the simulations and machine learning were performed on core and interconnect frequencies of 12.5, 25, and 50 MHz core, with 12.5, 25, 50, and 100 MHz interconnect, paired up as core:interconnect ratios 50:12.5, 25:12.5, 12.5:12.5, 12.5:25, 12.5:50, and 12.5:100, for relative ratios 4:1, 2:1, 1:1, 1:2, 1:4, and 1:8. This design space obviously has two best design choices, either maximum interconnect or maximum core frequency, there is no benefit to choosing any of the intermediate changes. To make the design space more interesting, the resulting run-times were scaled by the square root of the larger side of the clock ratio, (divided then by two for more realistic clock frequencies), giving effective clock ratios of 50:12.5, 35:17, 25:25, 17:35, 12.5:50, and 8.9:71. This transformation has been applied after the prediction, but should not drastically affect the prediction accuracy because the transformation is linear for each clock ratio configuration, and applied to both the predicted and measured run-time values. The transformation is visualised in Figure 8.1.

8.6 Results

Figure 8.2(b) presents the accuracy of the best prediction method, Random Forest, at predicting the runtime for an unseen benchmark across all explored points of the

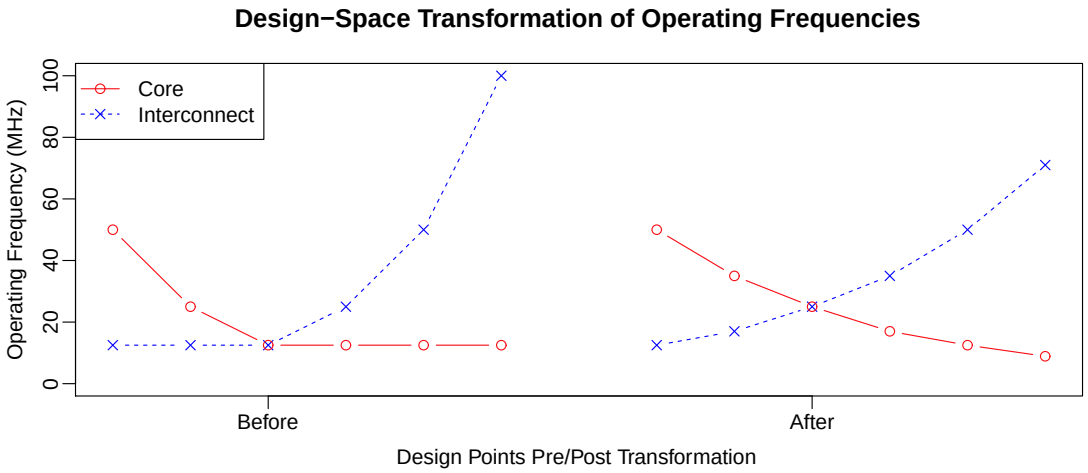
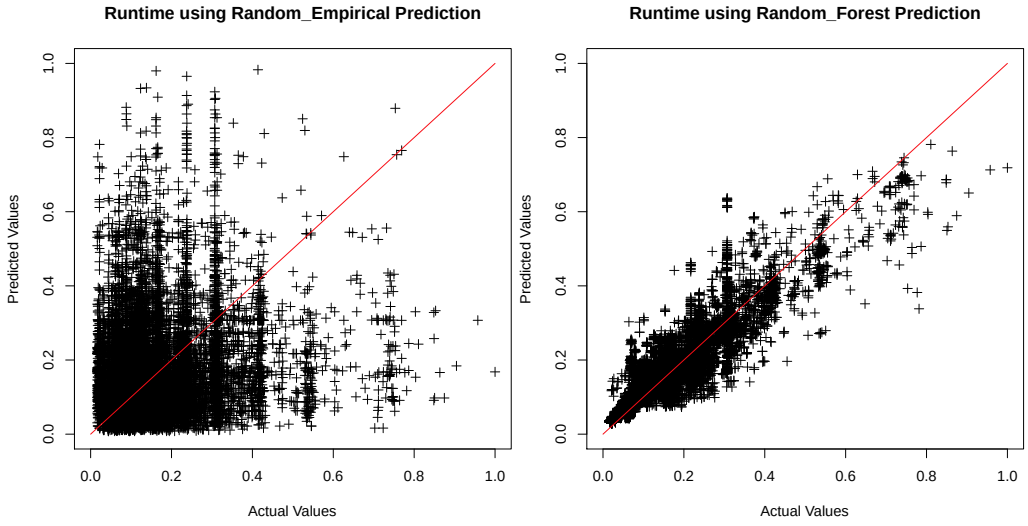
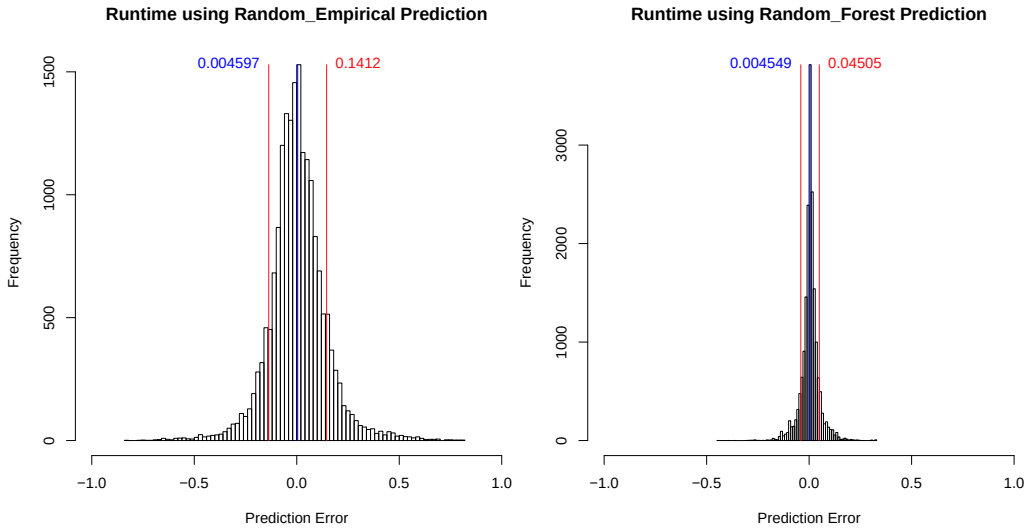


Figure 8.1: Transformation of the operating frequencies used in the design space. The transformation exposed the trade-off between core and interconnect performance, and design points which would otherwise have never been optimal choices.

design space; Random Empirical results are presented alongside for comparison to demonstrate that the accuracy is not due to the design space being condensed to a small performance distribution. Results presented are normalised to the maximum actual value measured for each variable, runtime and number of switching events, so numbers are relative to this. The Random Empirical results are included as the baseline for observations, and the performance of Random Forest should be viewed in light of this baseline. The relative standard deviations of 29.6% and 119% respectively, as annotated on the figures, are presented in Table 8.2 along with the results for the other machine learning techniques. Figure 8.3(b) depicts the capacity for the Random Forest to predict the best design choice for a given benchmark. Each vertical column represents the performance distribution of the (explored) design space for a single benchmark, normalised to the performance of the best design for that benchmark; the best performing design is at 1 on the Y axis, and a design which takes infinite time would be at 0. Darker regions indicate a concentration of designs at that performance point, while lighter regions have few to no designs in that region. This condensed form of "population" density is provided to give the reader a sense of the performance distribution exhibited by each benchmark, and to give the data presented by the points on the graph a greater context.

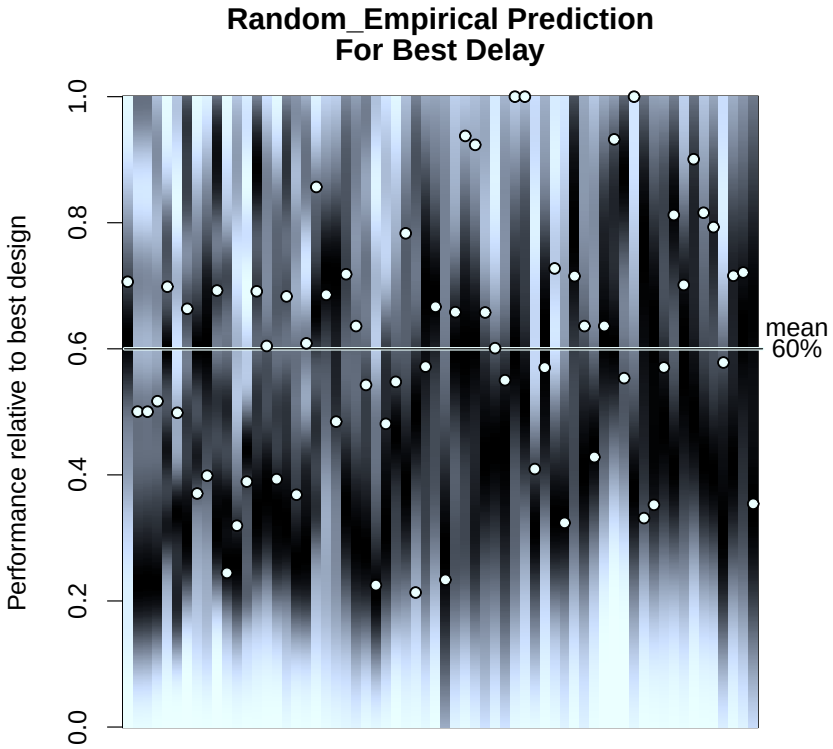


(a)

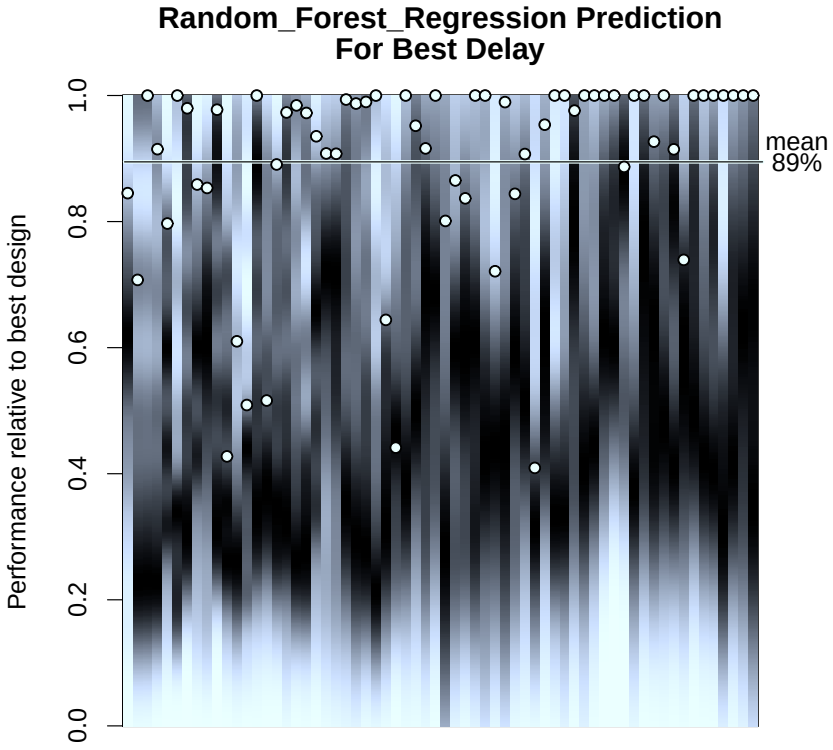


(b)

Figure 8.2: Empirical Random and Random Forest prediction accuracy when predicting for runtime of a new application.



(a)



(b)

Figure 8.3: Prediction quality of Random Forest versus Random Empirical. Each column represents one benchmark. The greyscale background shows the distribution of designs within each benchmark; white dots show the predicted designs.

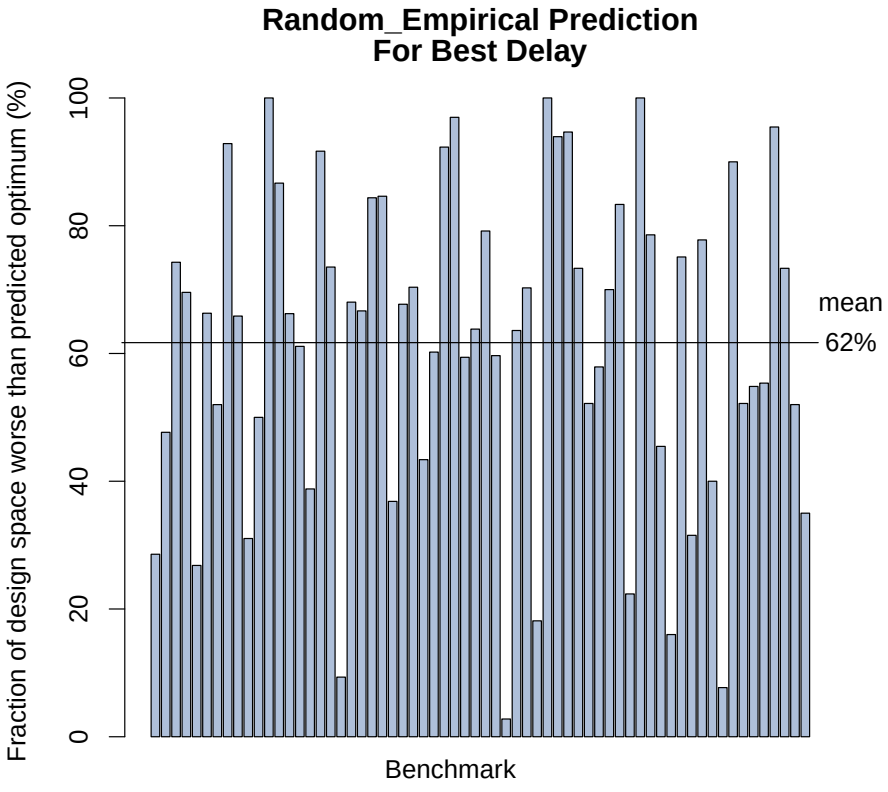
These points are the performance of the design which the presented machine learning techniques have identified as the best performing design for that benchmark, by generating a performance prediction for every design point and selecting the best design. For benchmarks which have multiple best predictions, the selection which corresponds to the worst possible choice from those selected was plotted.

Although it would probably be feasible for a designer to evaluate all of the choices, it was felt that presenting the worst was the fairest representation of the machine learning's capabilities, given it still on average predicts a design with at least 89% of the performance of the best design, even in benchmarks where almost the entire design space is below 40% of the maximum performance.

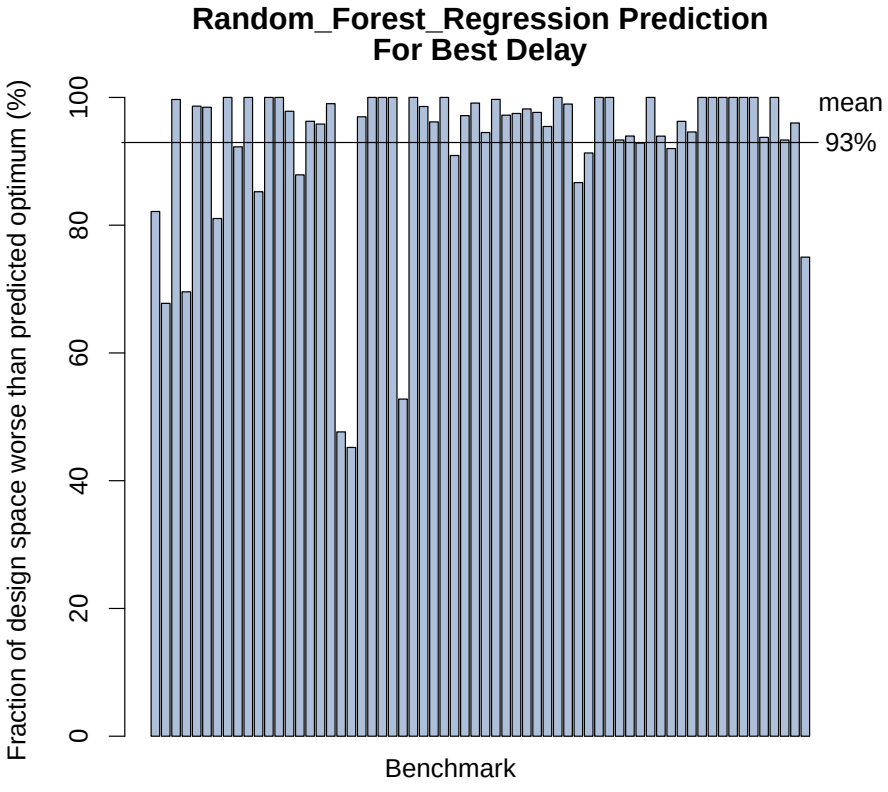
Figure 8.4 presents corresponding bar graphs for the points in Figure 8.3, representing the fraction of the design space which, for that benchmark, are worse than or equal to the performance of the design which performed the best. From these it can be

	ML Technique					
	RE	RF	DT	NN	*wk*NN	MARS
Runtime relative std-dev (%)	92.6	29.6	43.3	42.0	46.2	41.8
Switching relative std-dev (%)	261	119	293	105	130	113
Best Runtime as % of actual best	60	89	60	61	79	52
Best Runtime better than %	62	93	60	59	81	45
Best EDP as % of actual best	39	82	42	37	69	26
Best EDP better than %	55	93	53	49	86	34
Best Area×Delay as % of actual bes	69	93	85	88	89	83
Best ADP better than %	85	97	93	93	95	90
Best Area×EDP as % of actual best	69	93	85	88	89	83
Best AEDP better than %	85	97	93	93	95	90

Table 8.2: Numerical results for Random Empirical (RE) baseline, Random Forest (RF), Decision Trees (DT), Neural Network (NN), *wk*NN and MARS. Linear Regression is omitted as it behaves worse than RE, with a relative standard deviation of over 10$\hat{1}$07% for runtime and 10$\hat{1}$38% for switching.



(a)



(b)

Figure 8.4: Prediction performance of Random Forest versus Random Empirical showing the proportion of the designs that are the same or worse than the predicted value for each benchmark. 100% indicates that no designs are better than the predicted.

seen that on average this work is able to select a design in the top 7% of the explored design space for a given benchmark. Conversely the Random Empirical predictor does poorly, picking designs about half way through the design space, close to the median performing design as expected.

To evaluate how the methods would fare when predicting for energy, a similar energy model to Almer *et al.* [121] is used, predicting dynamic energy based upon the the total number of switching events in the interconnect. The standard deviation for switching event predictions is somewhat lower than total runtime, results also presented in Table 8.2. To evaluate this work's capacity to pick the design with the best EDP a similar prediction is performed across the design space to that used to generate 8.3, but predicted both switching events and runtime, selecting the design which was predicted to have the minimum product of the two. The final results of these are present in the results table, and show that Random Forest is still the best choice of prediction method, achieving on average an EDP within 82% of the optimal design, and in the top 7% of the design space.

The final concern for an MPSoC designer is the total silicon area, which largely dictates the cost of the device. As an approximate measure for area the number of cores in the design is used, as they will often dominate the silicon area used. When factoring in area, energy and performance the design space shows obvious changes in the trade-offs for benchmarks with greater task counts in Figure 8.6, where the presented technique is demonstrated to achieve 93% of the area-energy-delay product of the best design, selecting designs which are on average in the top 3% of the design space.

When optimising for an equation containing the area, any prediction which can identify the other parameters (runtime or energy) in the right order of magnitude will perform reasonably at excluding the larger designs. This is because unlike performance or energy, the area is not guessed by a machine learning algorithm, it's a known fact, and it is possible to perfectly weight the results based upon this. Because designs with small thread counts run well on all designs, guessing constant runtime and scaling by the area will produce a ranking with the best design correctly out-performing almost the entire design space in the metric. Because of this, the metric assessing the fraction of the design space that is worse than the selected is less meaningful, instead consideration should be given to what fraction of the maximum metric is achieved, as in Figure 8.5, which still relies on a strong prediction for the other metrics involved.

It is probably worth taking a closer look at the sort of predictions that the machine

learning is making. For simplicity's sake the analysis is restricted to the first three single-threaded benchmarks where the trade-off between different resources is easily reasoned about, and the final benchmark as the most diverse and complex. Among the design options, for a single thread the FIFO depth and number of RAMs will have negligible impact, because a single core cannot produce parallel requests to introduce contention or to leverage any increase in bandwidth. This leaves the number of cores (having too many cores will require the interconnect complexity to grow beyond that specified by the complexity parameter, increasing the distance between the core and the RAM and increasing memory latency), the interconnect complexity (increasing the memory latency), and the core and interconnect operating frequencies as variables worth consideration.

The first three benchmarks are image_display, the panning image benchmark, and Fbital and Conven from the EEMBC suite. Since the image benchmark requires a lot of IO, as well as compute to perform a greyscale transformation and dithering, while the Fbital and Conven benchamrks have been shown to be quite cache efficient in Chapter 4.

From Figure 8.3 it is already known the the prediction for the first two benchmarks are sub-optimal, but not terrible, but the third is close to ideal. Table 8.3 shows that for the compute/IO balanced image benchmark ideal configurations have a balanced 1:1 core:interconnect frequency ratio, with a low interconnect complexity and core count to keep the memory latency as low as possible. The machine learning misses the mark slightly and allocates a little bit too much compute relative to the memory performance, but only by a single decision point, and it keeps the core count and interconnect complexity low. Training with more examples of workloads which tax the interconnect, or on the transformed design space initially rather than transforming post-training and prediction, may have helped the machine learning do better at this workload. Moving on to Table 8.4, Fbital clearly favours compute over interconnect, with the best designs employing maximum core frequency, but still keeping complexity and core count low. The machine learning fails to recognise the compute potential, and suggests solutions one design point back towards a core:interconnect frequency balance, at the expense of pure compute performance. It does manage to keep complexity low, but not as low as the ideal design, and with more cores than necessary (although area is not a consideration for this design selection). The third benchmark, Conven, is one which the machine learning manages to predict extremely well for. In Table 8.5 it can be seen once again that the idea configuration has low core count and complexity, but most

	Predicted Result														
Cores	3	6	3	4	6	4	5	4	4	6	5	7	4	8	5
Complexity	1	2	1	2	2	1	1	1	4	4	4	2	4	2	4
Core MHz	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35
NoC MHz	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17
	Correct Result														
Cores	1	1	1	1	2	2	2	1	2	2	3	2	2	2	4
Complexity	1	2	1	2	1	1	1	4	1	1	2	1	4	4	2
Core MHz	25	25	25	17	25	25	25	17	17	17	25	25	25	25	25
NoC MHz	25	25	25	35	25	25	25	35	35	35	25	25	25	25	25

Table 8.3: Image Display benchmark top 15 predicted designs, compared with the correct top 15 designs.

importantly puts all of the emphasis on compute performance again. This time the machine learning gets it right, and suggests designs in line with the true best performing designs.

Perhaps most impressively, the 64-thread workload, which consists of a mixture of Autocor, Conven, Coremark, Fbital, Viterb and image_display instances. Although there were only ten configurations produced by the training data simulations, all ten designs were ranked perfectly by the Random Forest predictor. As seen in Table 8.6, despite the ranking aligning neither perfectly with the number of cores, not the ratio between compute and memory performance.

	Predicted Result														
Cores	8	7	14	8	7	7	7	12	12	21	21	12	12	12	8
Complexity	2	2	2	2	1	2	2	2	2	2	1	2	2	2	1
Core MHz	35	35	35	35	35	35	35	35	35	35	35	50	50	35	35
NoC MHz	17	17	17	17	17	17	17	17	17	17	17	13	13	17	17
	Correct Result														
Cores	3	7	7	12	2	2	15	3	3	4	5	10	6	6	8
Complexity	1	1	1	1	1	2	1	2	1	2	2	2	2	2	2
Core MHz	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
NoC MHz	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13

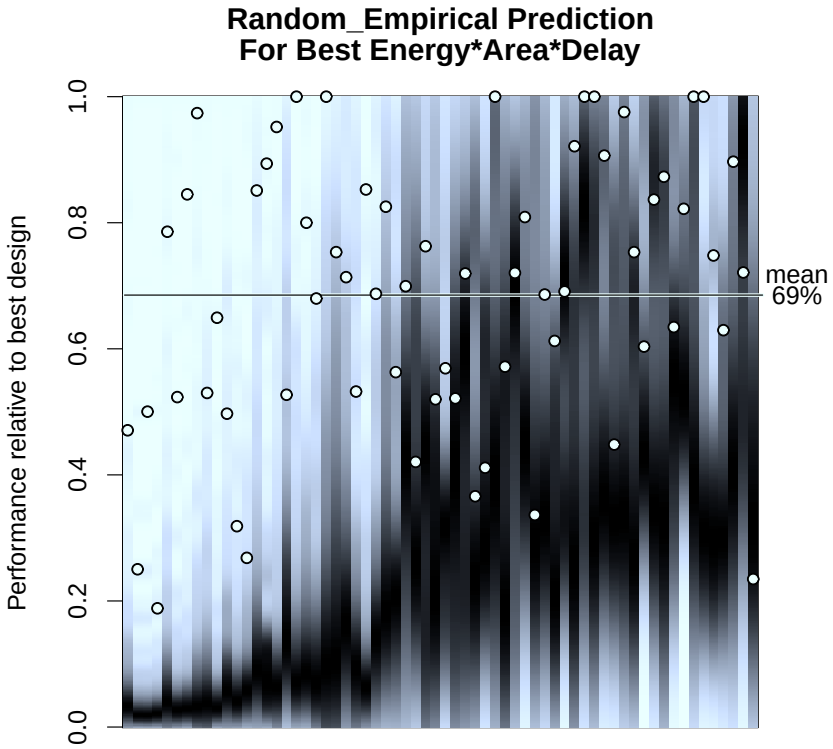
Table 8.4: Fbital benchmark top 15 predicted designs, compared with the correct top 15 designs.

	Predicted Result														
Cores	10	6	10	12	6	6	12	3	6	10	6	12	2	9	18
Complexity	2	1	4	2	8	2	2	2	8	1	2	2	1	4	2
Core MHz	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
NoC MHz	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
	Correct Result														
Cores	3	7	7	2	2	12	3	3	4	5	6	6	8	8	10
Complexity	1	1	1	2	1	1	1	2	2	2	2	2	2	1	2
Core MHz	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
NoC MHz	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13

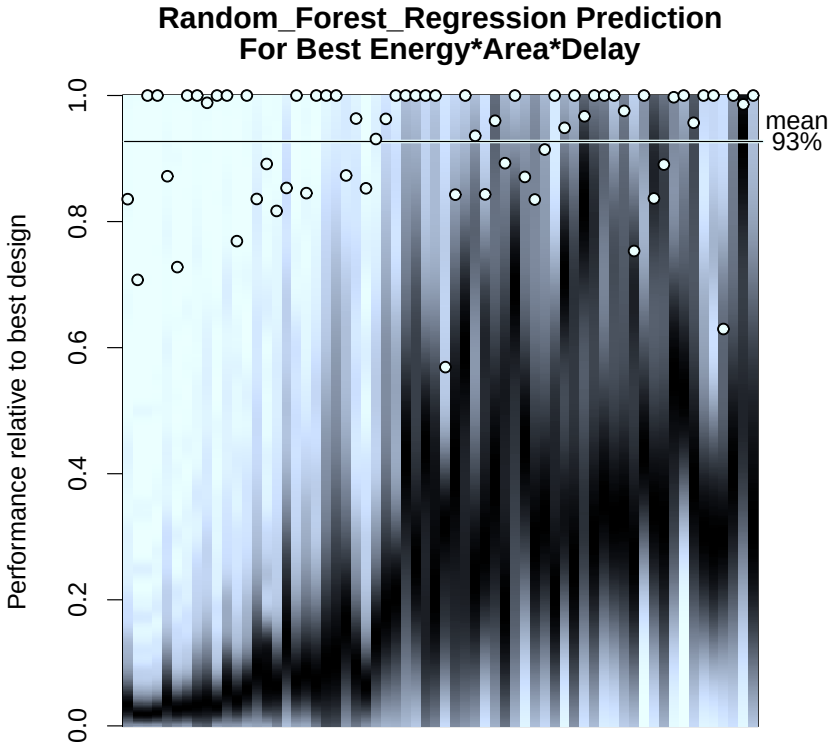
Table 8.5: Conven benchmark top 15 predicted designs, compared with the correct top 15 designs.

	Predicted Result									
Cores	32	42	42	16	32	21	35	36	35	20
Complexity	2	4	2	8	8	8	8	4	1	1
Core MHz	35	35	17	25	17	13	13	9	9	9
NoC MHz	17	17	35	25	35	50	50	71	71	71
RAMs	4	2	4	2	2	2	4	8	2	1
FIFO Depth	2	2	2	2	2	2	2	2	16	16
	Correct Result									
Cores	32	42	42	16	32	21	35	36	35	20
Complexity	2	4	2	8	8	8	8	4	1	1
Core MHz	35	35	17	25	17	13	13	9	9	9
NoC MHz	17	17	35	25	35	50	50	71	71	71
RAMs	4	2	4	2	2	2	4	8	2	1
FIFO Depth	2	2	2	2	2	2	2	2	16	16

Table 8.6: All 10 predicted designs for 64-thread workload ranked left to right, compared with the correct ranked 10 designs.

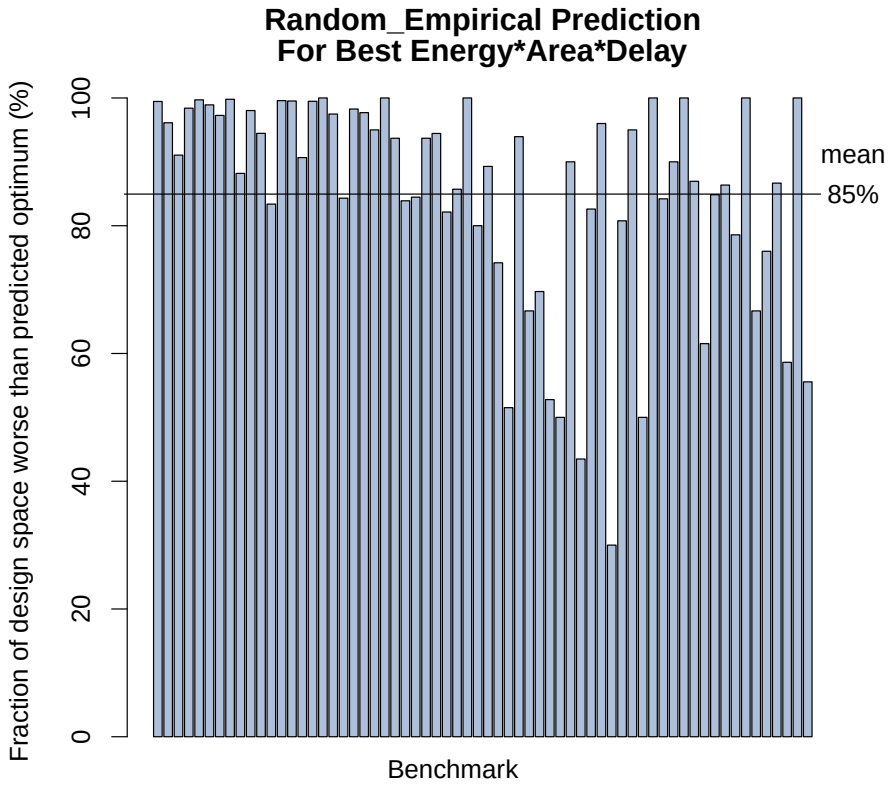


(a)

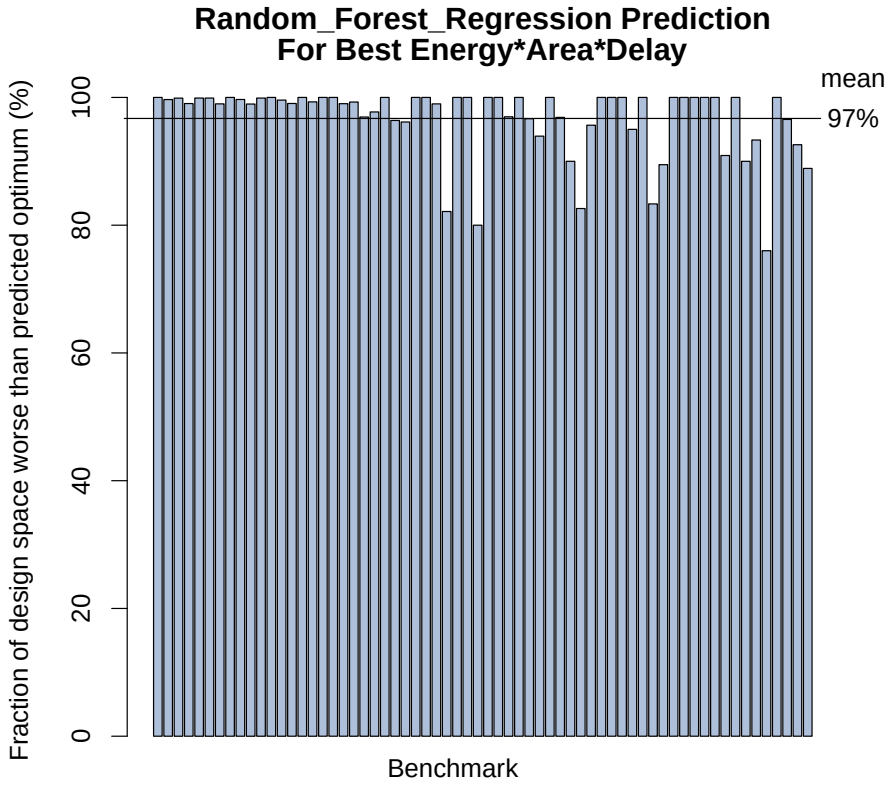


(b)

Figure 8.5: Prediction quality of Random Forest versus Random Empirical. Each column represents one benchmark. The greyscale background shows the distribution of designs within each benchmark; white dots show the predicted designs.



(a)



(b)

Figure 8.6: Random Forest and Random Empirical results when predicting for best Area-Energy-Delay product.

8.7 Discussion and Conclusions

From the results it is obvious that not all machine learning methods are as useful at predicting runtime or switching counts on the proposed design space. However Random Forest does a reasonable job of predicting the runtime for this difficult design space, with an average error of about 30%. Unlike the work of Ipek *et al.* [120] the work presented here is predicting for runtime, rather than IPC, and with the multiprogrammed workloads the task that dominates the runtime can shift depending upon the characteristics of the platform. A compute-heavy benchmark mixed with an IO-heavy benchmark will perform badly on a design with high core speed but low interconnect speed, and similarly badly on a design with high interconnect speed but low core speed. Conversely the homogeneous parallel workloads in the work of Ipek *et al.* [120] will have all cores affected in the same manner when design configurations are changed.

One of the most common use cases for the prediction methods presented is for an engineer to identify good candidate design choices for a new application, so it is important that the prediction scheme can at least rank the designs correctly – even if the actual prediction is incorrect – to allow the engineer to correctly select designs that perform well. Testing this by evaluating the best performing prediction reveals that most machine learning techniques are quite bad at this, with under-predicted outliers providing suggested designs which are far worse than the true best design, often on-par or worse than a random selection from the design space (Random Empirical). Fortunately Random Forest also does a reasonable job at this too, providing designs that give provide an average performance of 89% of the optimal design, that is at least as good or better than 93% of tested designs for that benchmark.

Since prediction results are also similar when energy and area are considered, the technique is a viable method for enabling engineers to rapidly identify a suitable choice of MPSoC, whether the requirements call for performance, energy efficiency, or cost minimisation, without the time-consuming, computation-intensive design space exploration traditionally required. It is also clear that Random Forest clearly performs the best for the explored design space, and should be considered a strong contender in any similar machine learning experiments.

Chapter 9

Conclusions

An engineer designing a modern MPSoC or CMP faces many challenges and difficult design decisions. This thesis has described and evaluated several tools and innovations that have been developed and proposed to make this designer's job easier, and provide them with design options to combat some of the primary challenges of large scale multicore processor design.

Chapter 4 presented novel simulation techniques enabling extremely accurate, high performance simulation of embedded MPSoCs. If implemented in a suitable simulator this could be a powerful, multifaceted tool that could be used throughout their design process. The high accuracy and simulation speed, coupled with almost instantaneous reconfiguration, would allow an engineer designing an embedded MPSoC to rapidly simulate many different designs under consideration. This could be performed in parallel on a large compute cluster to give the designer a detailed picture of the design space. Provided that their benchmark application was suitably representative of the final software targeting the platform, the simulations would not only be able to provide runtime performance estimates, but would also provide internal and external communication link statistics, giving insight into bottle-necks and power estimates. The modularity of the simulator design would also enable the engineer to add models for new components and quickly see how these would affect the runtime and interconnect performance. The high simulation speed of up to 383 MIPS, more commonly associated with functional only or un-timed models, provides an infrastructure to allow developers to begin software development while hardware design is ongoing, with a performance model which can be easily tuned to match the final hardware design as development progresses. Typical un-timed "Virtual Prototype" simulations do not give the developers any feedback about performance, which can be critical in embedded

systems, using the proposed simulation techniques allows the developers to estimate performance well ahead of hardware availability, with minimal impact on their productivity.

The main take-away from the simulator development has been that good simulation performance is only achievable through careful consideration and design, with care taken to perform processing in a cache friendly manner and taking care to avoid unnecessary computation where ever possible. Exploiting parallelism is important for high performance simulation, but doing it naively will result in sub-par performance; care must be taken to prevent part of the simulation becoming a severe bottleneck, to minimise synchronisation overheads, and to ensure that all threads have useful computational work to do at all times.

Chapter 5 demonstrates that the techniques from Chapter 4 enable a simulation infrastructure for fully cache-coherent manycore systems, that is fast enough for large-scale design space exploration. The simulation results presented in Chapter 5 support the claim that fast simulation models are able to deliver more detailed performance data, and in less time, compared with previous systems, such as GEM5, SystemC and FPGA-based systems. This includes the ability to deliver greater accuracy, improved micro-architectural visibility, better statistics gathering capabilities, increased flexibility, and higher performance. In contrast GEM5, COTSon, and Sniper lack the accuracy of the fast models presented in this thesis, SystemC has much lower simulation speed, and FPGA-based systems lack both the visibility of the presented models, and the capacity to simulate large scale systems.

Again the take-away is that it is possible to construct high performance multi-core simulations with as much detail as desired, so long as care is taken to design the work distribution and processing efficiently. When threads must stall it is important to switch the core to another thread which can do useful work as quickly and efficiently as possible, so that the host cores can continue to be fully utilised – so long as all cores are doing useful work the simulation is progressing efficiently, the goal is to maintain this state of maximum computation. Cooperative user-space scheduling is an effective way to implement this, and careful clustering of work queues can keep instruction and data cache contents hot between thread switches. For example by allocating the data structures for cores 0-3 together and assigning them to the same work queue, the instructions for running the core simulation will still be in the instruction cache, and it's likely that the data structures are still in one of the data cache hierarchies (certainly they won't have to be invalidated from another processor, or have been invalidated by

one).

Chapters 6 and 7 demonstrate how this simulator can be used to perform experiments into detailed micro-architectural issues relating to the scalability of manycore processors. In Chapter 6 it is used to assess the performance and energy scalability of a proposed novel manycore architecture, scaled from 32 to 1024 cores, and demonstrate the effectiveness of using software based coherency techniques to address one of the many scalability challenges of manycore processors – the associativity of the coherence directory. This chapter also proposes the use of a uni-directional non-blocking multicast channel to simplify coherency protocols and significantly reduce traffic involved in multicast invalidations, providing two techniques for an engineer to consider when designing a scalable architecture.

Chapter 7 introduces another three contributions to the engineer's scalability toolbox, aimed at reducing the energy consumed by the processor. The first of these addresses the cascade of atomic write accesses which occur after a contended lock is released, where all waiting cores see the lock freed simultaneously, and attempt to gain access through an operation such as atomic exchange. After the first core succeeds, all other cores will fail, but only after taking turns serially to load the cache line exclusively, modify it, making the cache line dirty, and then upon request from the next core, write it back to shared cache. This thesis suggests that introducing a comparison check on atomic exchange instructions – to test if modification took place, or the same value was written and stored – could remove all of these write-back operations. Although the coherency messages would still have to take place, a cache line write-back requires significantly more energy, although the trade-off against the energy of performing the check on every atomic exchange operation must be confirmed by more detailed power simulation. Secondly a new instruction is proposed, to address wasted energy performed during the spin-wait phase of many synchronisation events. By putting the core into a low power state until a monitored cache line is modified, it is possible to remove over 97% of instructions, and an even greater fraction of the instructions which activate the L1 d-cache, for synchronisation heavy benchmarks. All applications benefit by a large margin (on average 53% reduction in instructions executed) – which increases drastically as the number of cores is increased. This is because synchronisation is one of the problems which grows as an architecture is scaled, and this new technique has been shown in Figure 7.4 to effectively remove the associated in-core energy overhead (although not the communication overhead). The final contribution to scalability problems is a new sharer encoding scheme. Existing coherency protocols and sharer

encoding schemes often require either a large amount of storage space, broadcasting to many sharers indiscriminately, sending a large volume of unicast messages, or many or even all of the above. The new proposed protocol never requires more than one invalidation to be sent from the directory, by supporting multicast operations directly in the sharer encoding. The new scheme provides better than state-of-the-art support for tracking small to medium sharer clusters in large manycore architectures, degrading gracefully from multicast to broadcast as the sharer pattern becomes less compressible. The proposed encoding is the first to support exact tracking of sharer clusters from a single core to a medium number of sharers (depending on storage allocation) at any location in the processor, with a fixed number of encoding bits. For example 64-bits of storage can exactly represent any cluster of 1-32 cores, located anywhere on the processor, without an invalidation being received outside of this cluster. Because the encoding performs worse than alternatives which can also be used in the same multicast environment under certain randomised sharer distributions – like those found in some highly parallel benchmarks, a hybrid encoding is proposed. When it is detected that inserting a new sharer degrades the novel CTE representation below that offered by the CV alternative, the encoding is transitioned and CV is used for this cache line until it is invalidated. By using this hybrid scheme false invalidations can be reduced by around 7% across a range of scientific benchmarks.

The final chapter of this thesis addresses the fact that not all engineers want to design their target MPSoC from scratch, and perform hours of detailed simulations to evaluate their decisions. The short time-to-market pressure of embedded designs, such as those for high-demand consumer products, requires a tool to enable designers to rapidly identify a best candidate design, without the delay of simulation driven design space exploration. Previous work has identified the potential for machine learning to guide an engineer towards the best design choice, and Chapter 8 demonstrates that using the accurate, high speed simulations enabled by the work in Chapter 4, existing work can be extended to much larger designs than had previously been attempted. Using the techniques presented an EDA company could provide the upfront simulation time investment, and develop a tool capable of predicting designs for an engineer, on average, within the top 3% of the available design space. The engineer need only spend the time to run two short software simulations.

There are several areas of this thesis which could be expanded with future work. The addition of hierarchical caching of both data and directory information, and the associated caching policies and data migration policies present an interesting and com-

plex space for innovation, and should significantly improve the architecture scalability. The platform also presents a platform for algorithmic experimentation and development, where existing synchronisation and work-distribution algorithms can be compared, and further developments investigated. Further work to benchmark the performance against real world architectures will also be valuable once further tuning has been performed in the synchronisation libraries, as the existing implementations are known to be sub-optimal. It would be of most interest to either focus on integer and fixed-point based benchmarks, or to add floating point support to the simulator to emulate an FPU, recompiling the benchmarks to target this.

The large number of simulated cores provides a good infrastructure for designing and investigating coherency scalability of novel protocols, but would require enhancements to the core simulation to enable the out-of-order effects that would tax a relaxed coherency protocol such as TSO. This simulation extension itself would be an area of work which could put the simulator closer to GEM5 in terms of simulation capability, although the limited ISA support is a further problem to be addressed.

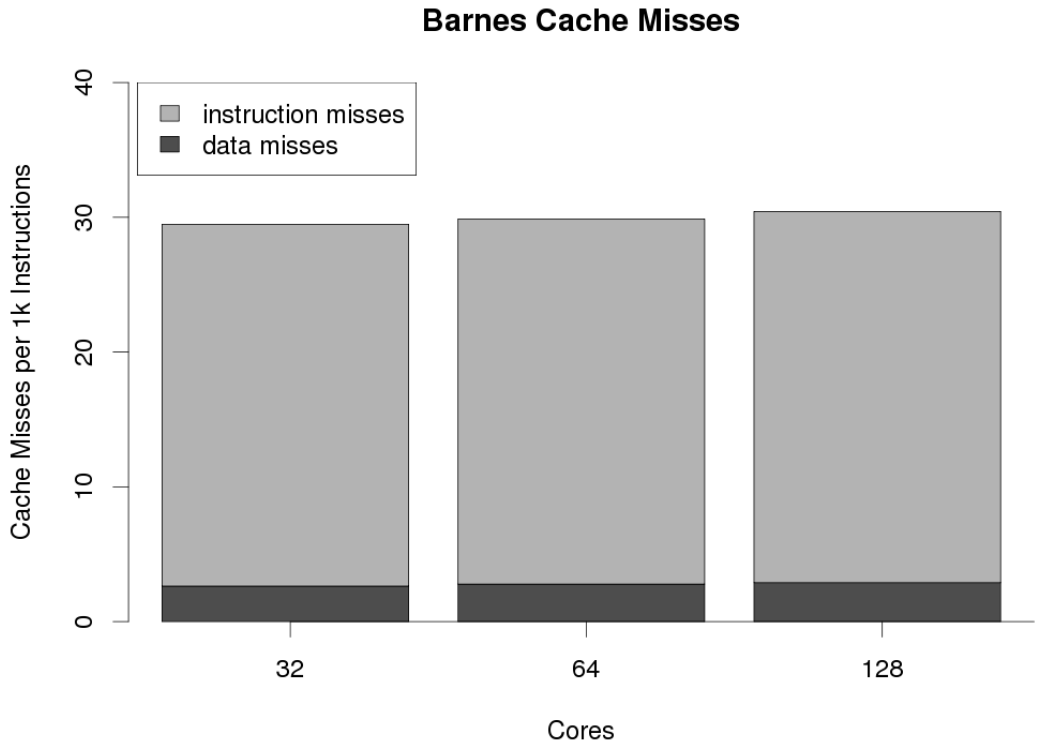
In summary, this thesis has advanced the state of the art in multiple frontiers important to the design of future manycore processors; addressing problems with scalability, energy consumption, and design effort. It also highlights how important careful design is to the scalability of both simulators and physical architectures.

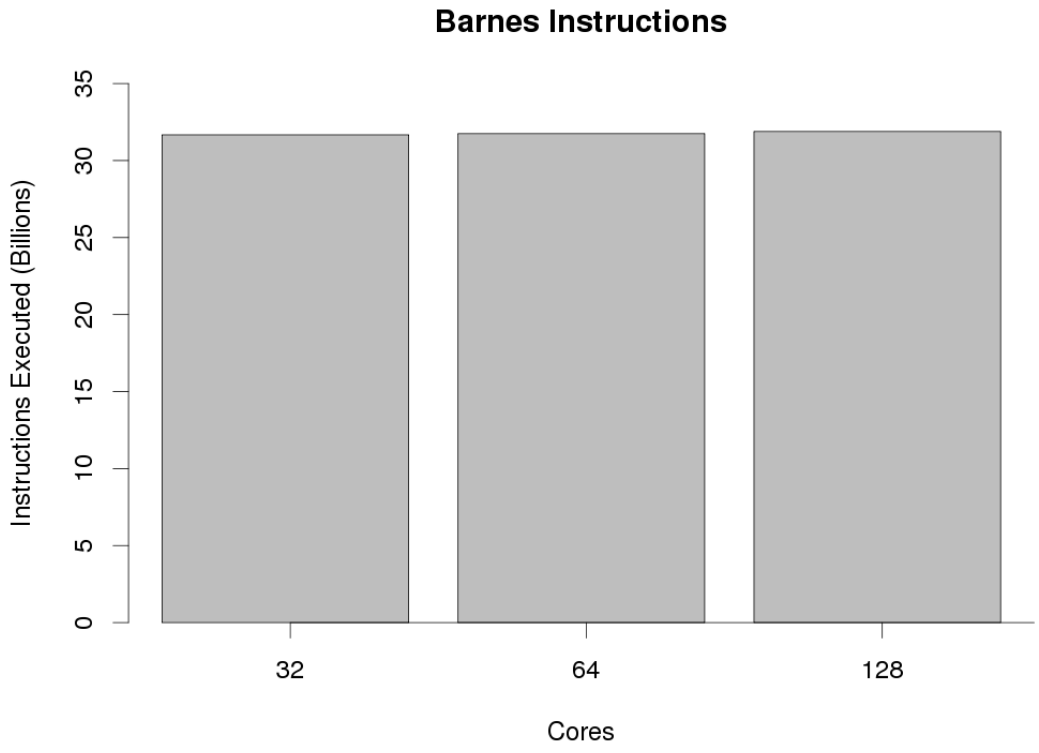
Appendix A

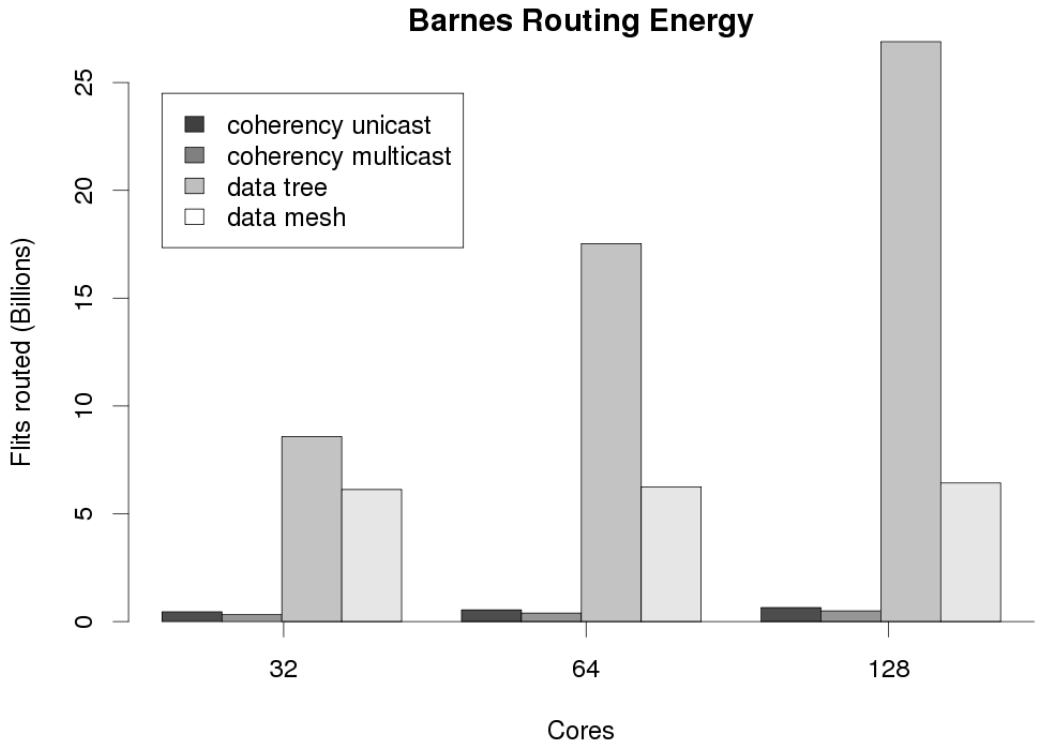
Appendix

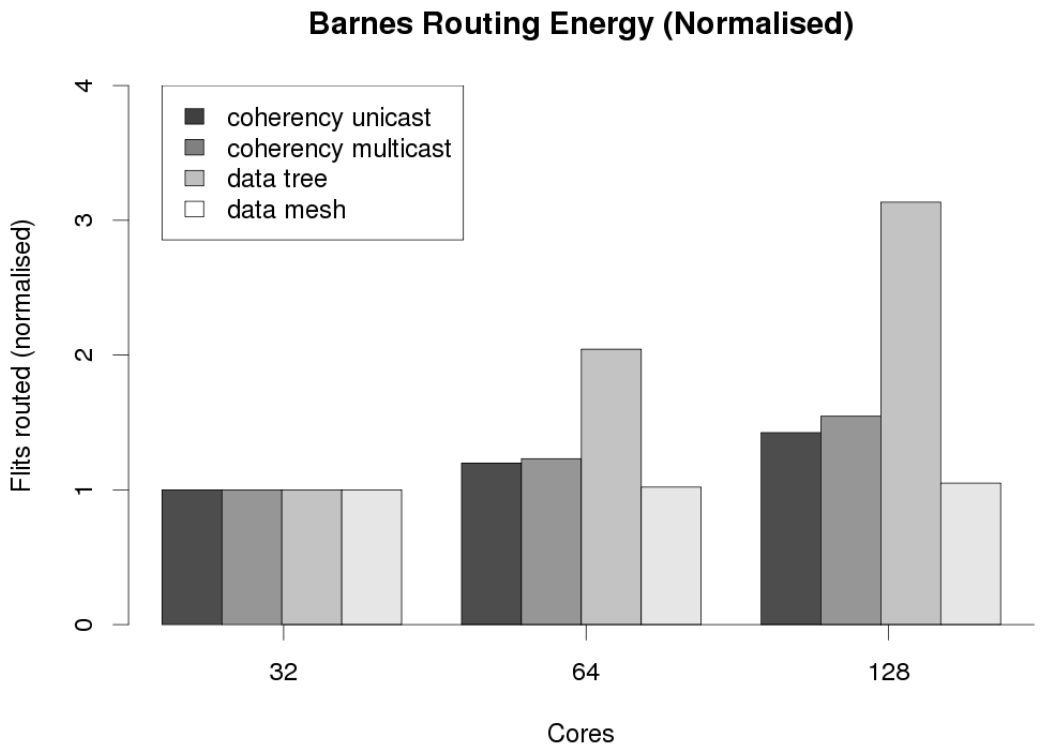
A.1 Architecture Scalability

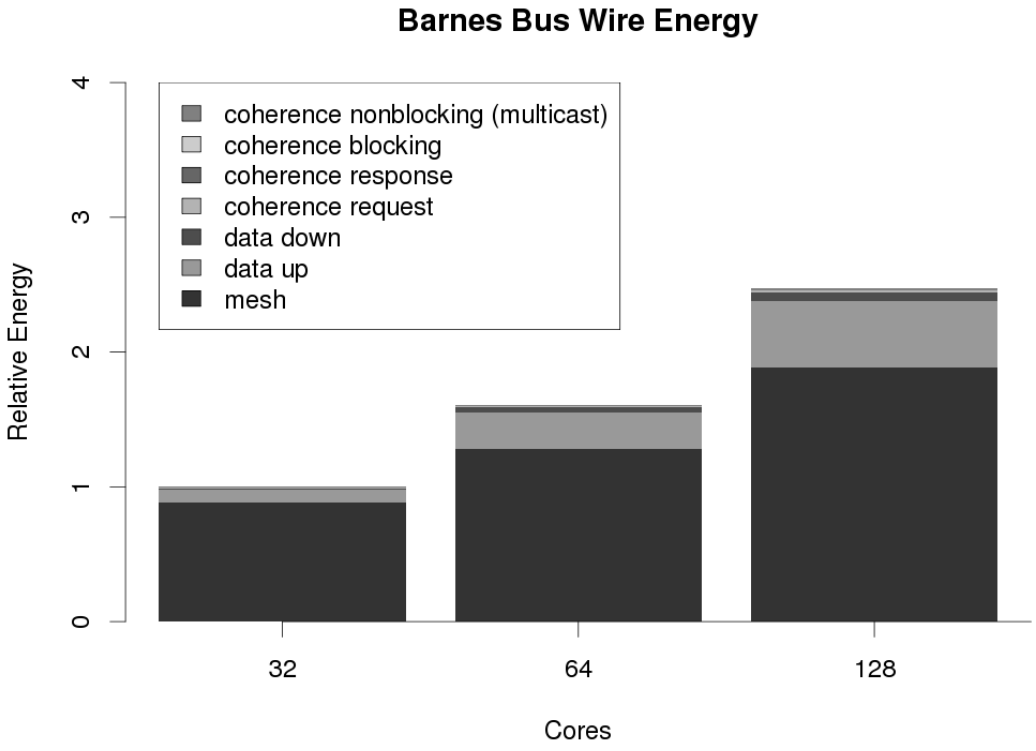
This section provides extra figures for extended architecture scalability analysis from Chapter 6. While discussion has already been provides for the LU benchmark, the equivalent figures from the remaining benchmarks are provided here for interest and further analysis on the part of the reader. Please refer to the equivalent figure captions in Chapter 6 for more details on each.

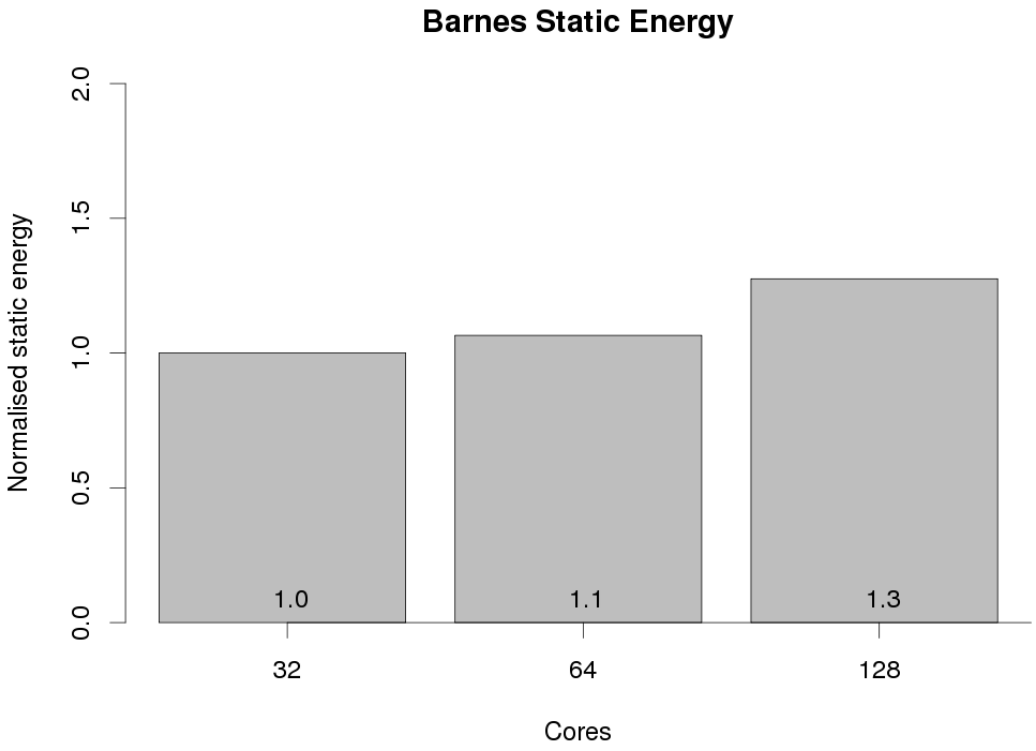

Barnes Cache Misses
instruction misses
data misses
Cache Misses per 1k Instructions
0
10
20
30
40
32
64
128
Cores

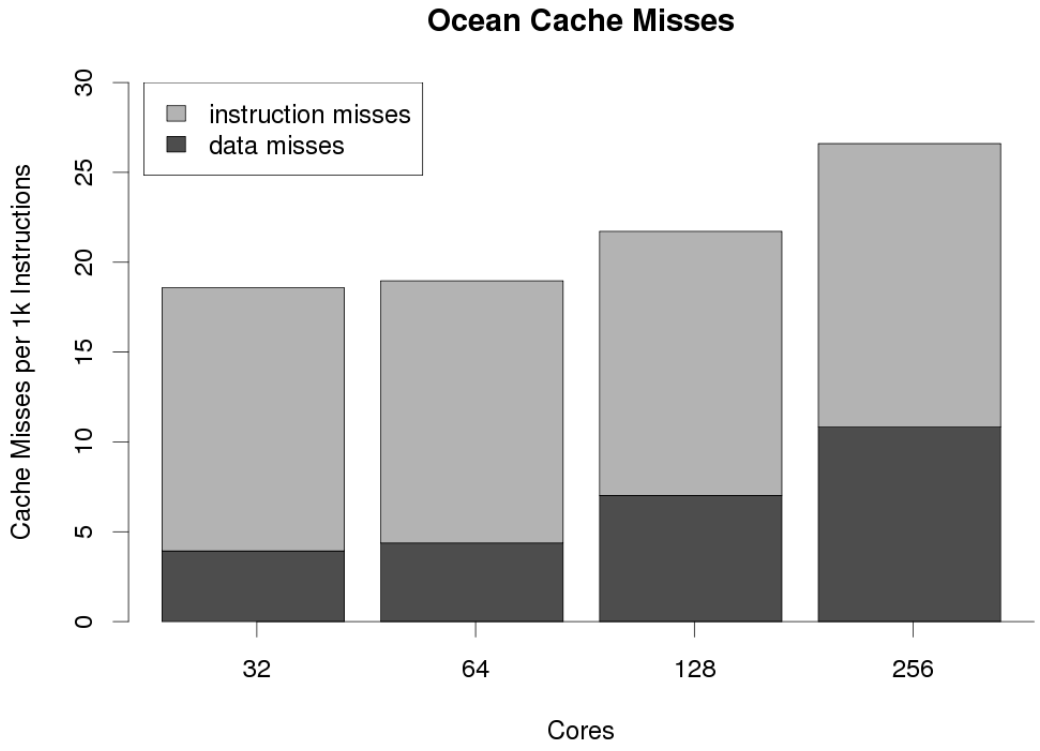

Barnes Instructions
Instructions Executed (Billions)
0
5
10
15
20
25
30
35
32
64
128
Cores

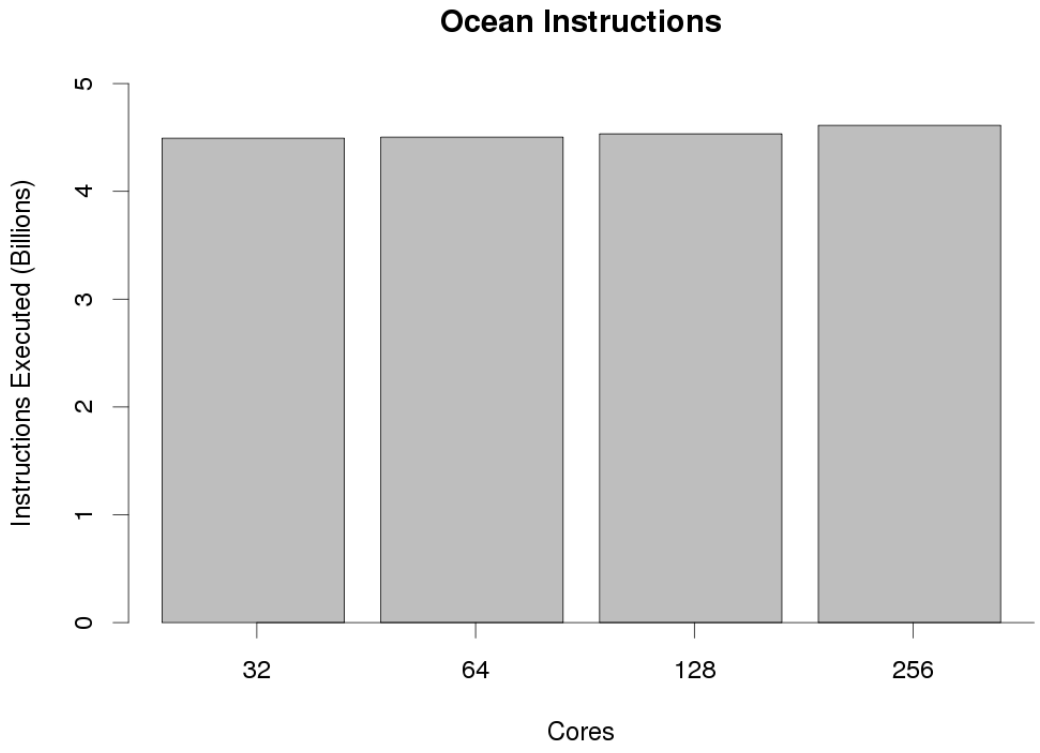

Barnes Routing Energy
coherency unicast
coherency multicast
data tree
data mesh
Flits routed (Billions)
0
5
10
15
20
25
32
64
128
Cores

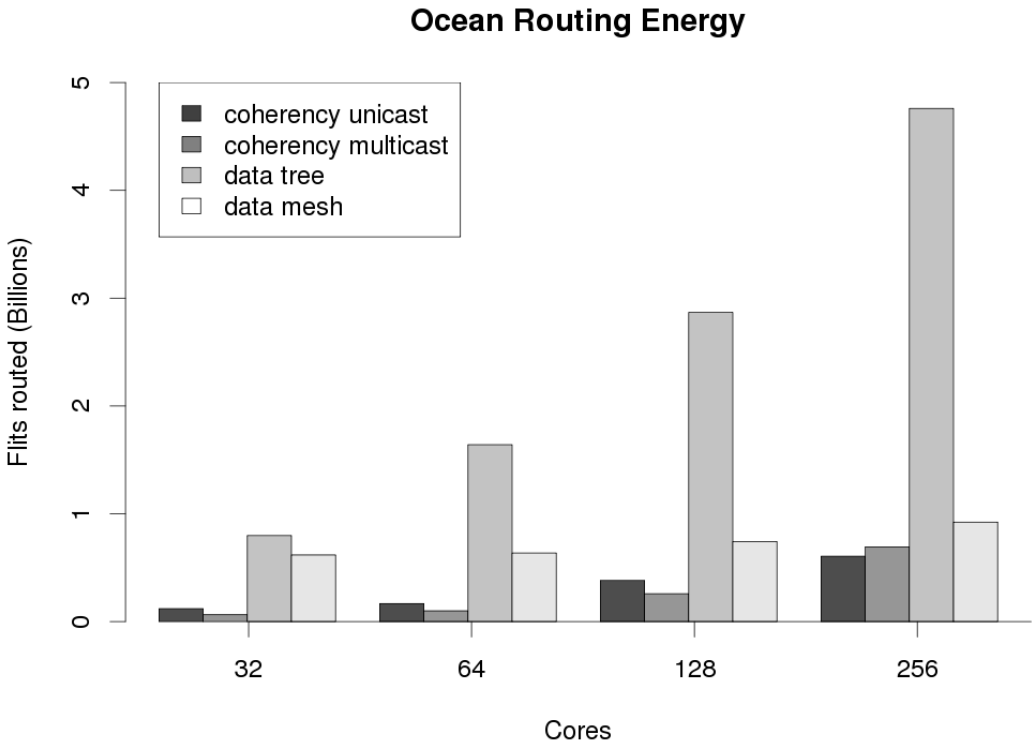

Barnes Routing Energy (Normalised)
coherency unicast
coherency multicast
data tree
data mesh
Flits routed (normalised)
0
1
2
3
4
32
64
128
Cores

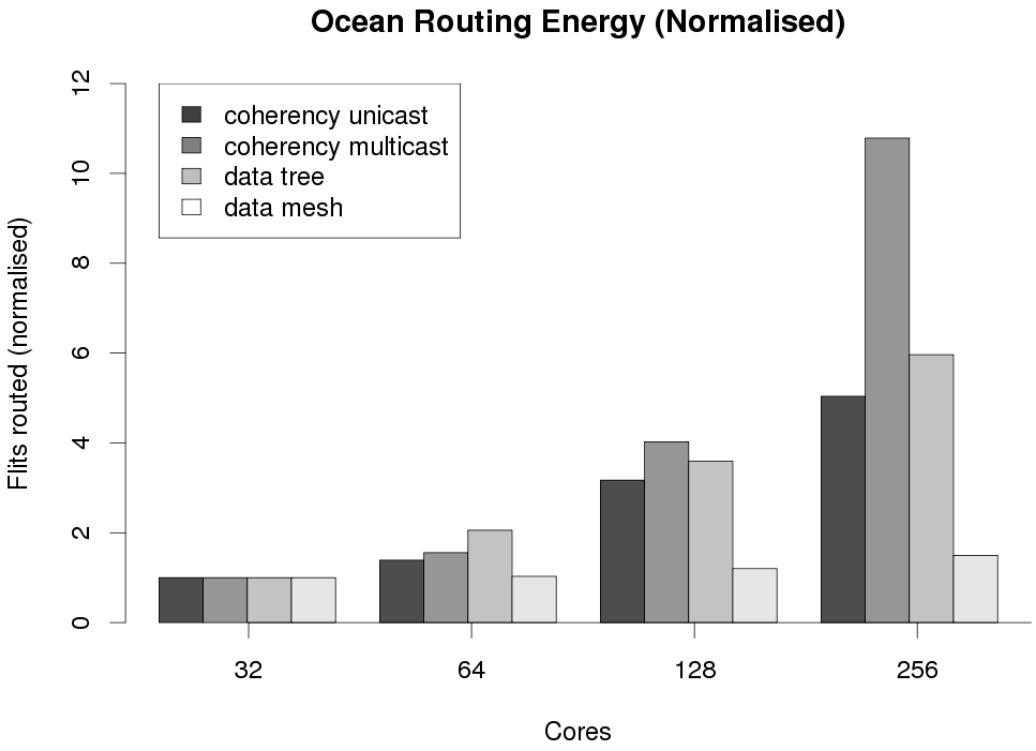

Barnes Bus Wire Energy
coherence nonblocking (multicast)
coherence blocking
coherence response
coherence request
data down
data up
mesh
Relative Energy
0
1
2
3
4
32
64
128
Cores

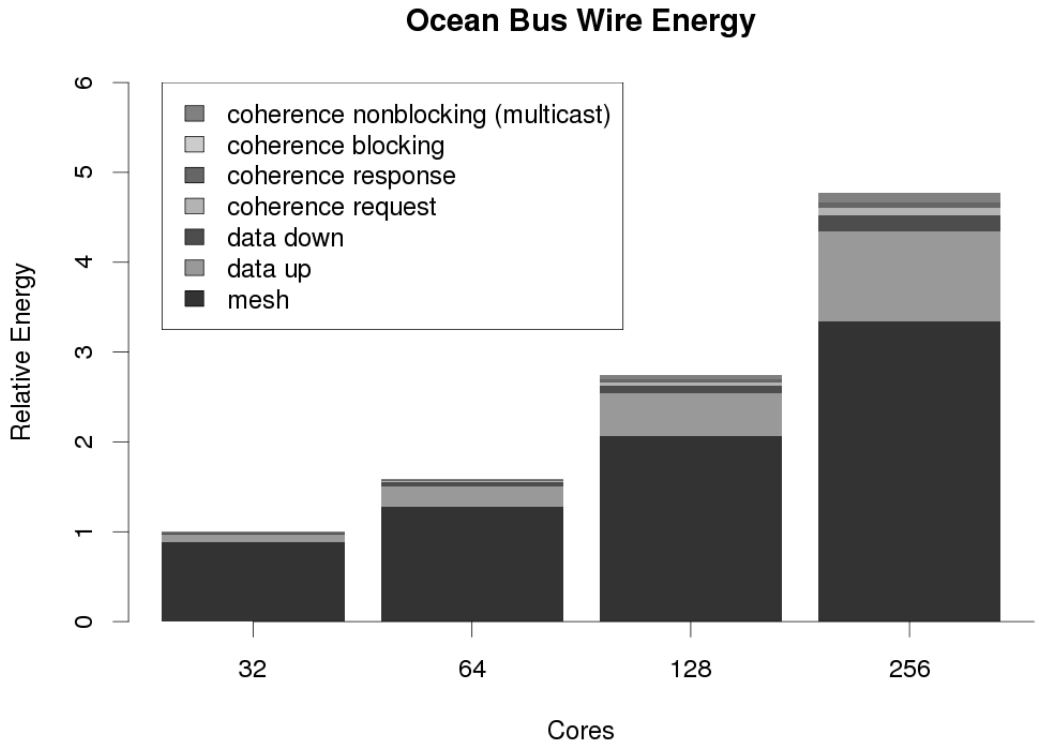

Barnes Static Energy
Normalised static energy
0.0
0.5
1.0
1.5
2.0
1.0
1.1
1.3
32
64
128
Cores

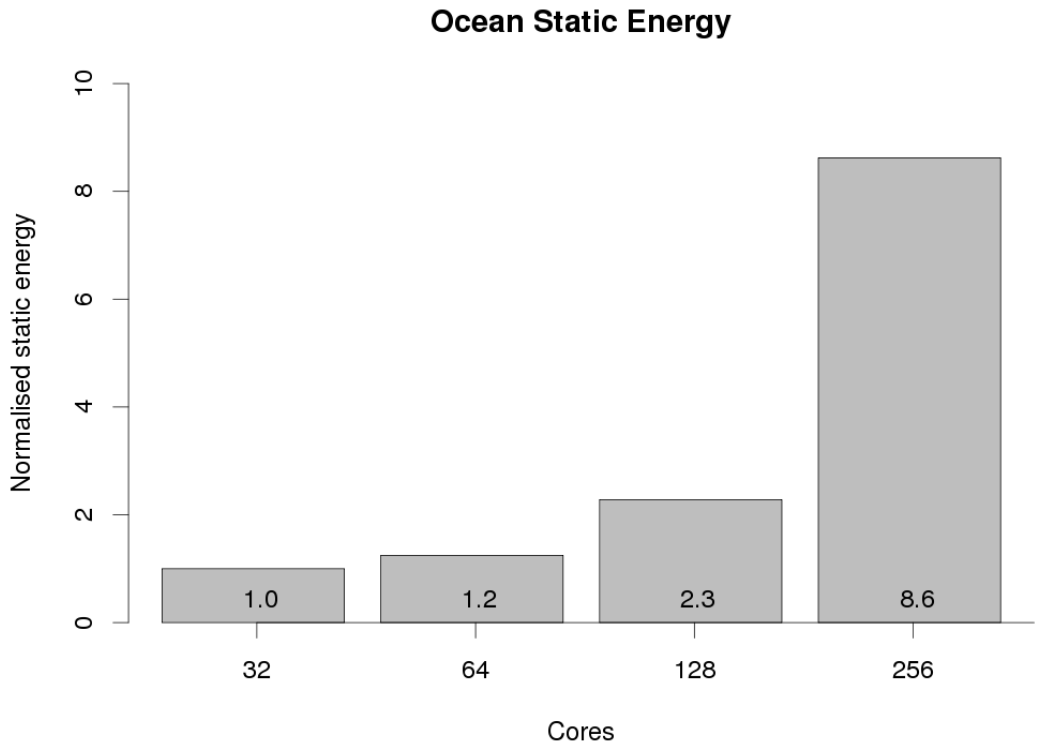

Ocean Cache Misses
instruction misses
data misses
Cache Misses per 1k Instructions
0
5
10
15
20
25
30
32
64
128
256
Cores

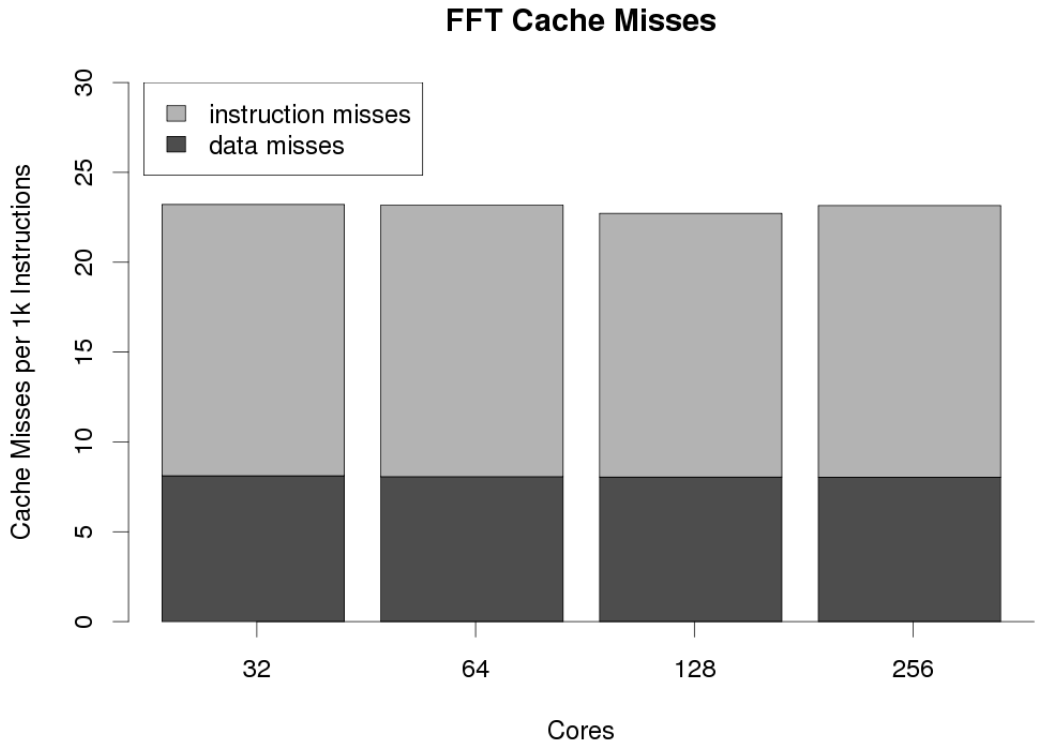

Ocean Instructions
Instructions Executed (Billions)
0
1
2
3
4
5
32
64
128
256
Cores

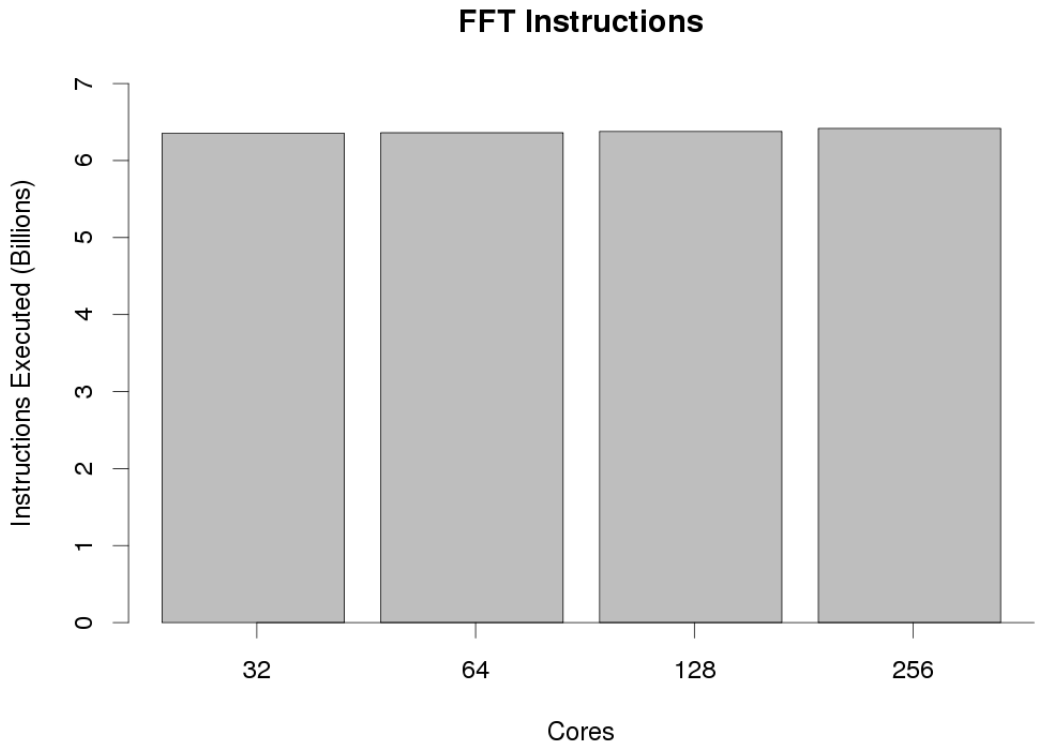

Ocean Routing Energy
coherency unicast
coherency multicast
data tree
data mesh
Flits routed (Billions)
0
1
2
3
4
5
32
64
128
256
Cores

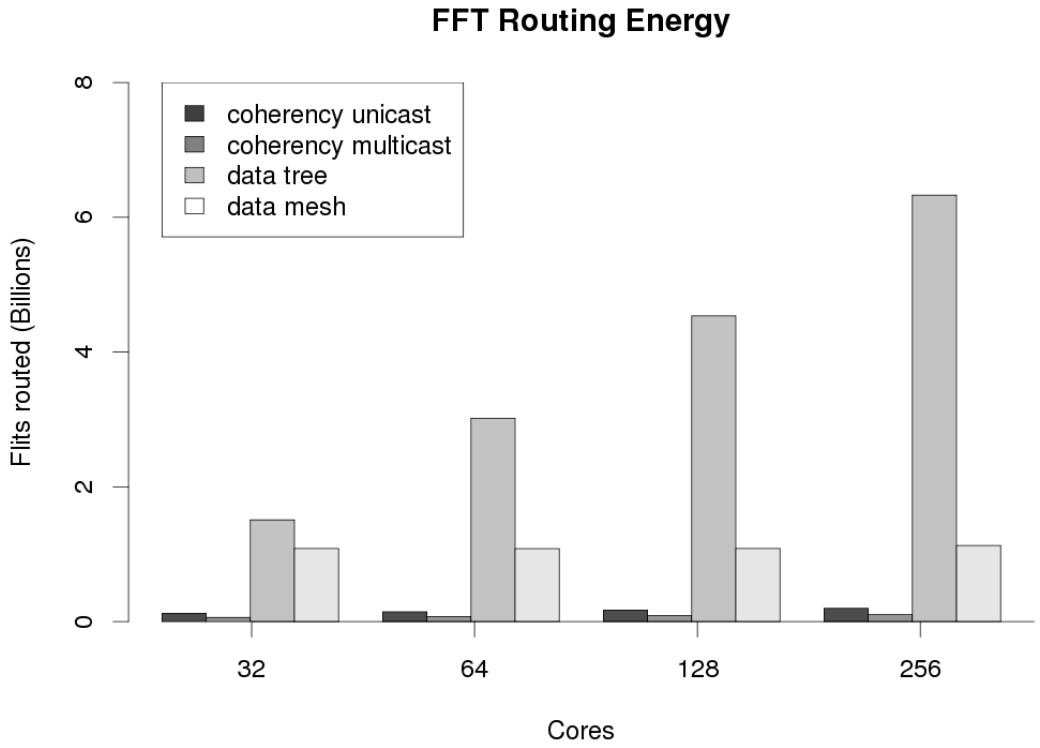

Ocean Routing Energy (Normalised)
coherency unicast
coherency multicast
data tree
data mesh
Flits routed (normalised)
0
2
4
6
8
10
12
32
64
128
256
Cores

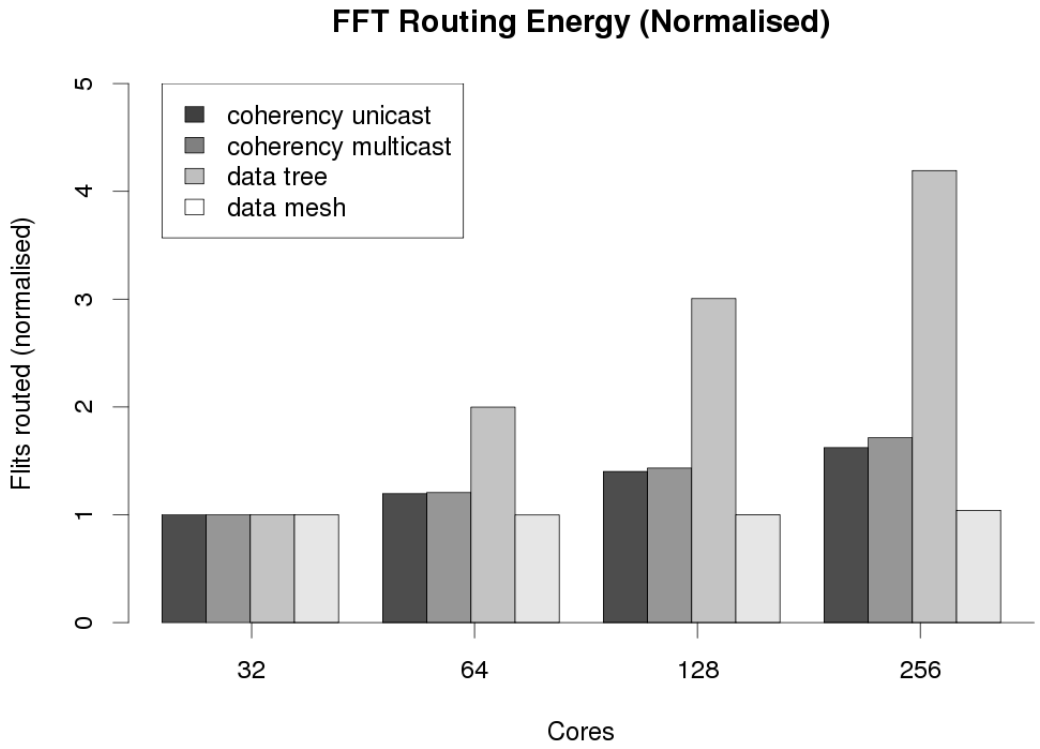

Ocean Bus Wire Energy
coherence nonblocking (multicast)
coherence blocking
coherence response
coherence request
data down
data up
mesh
Relative Energy
0
1
2
3
4
5
6
32
64
128
256
Cores

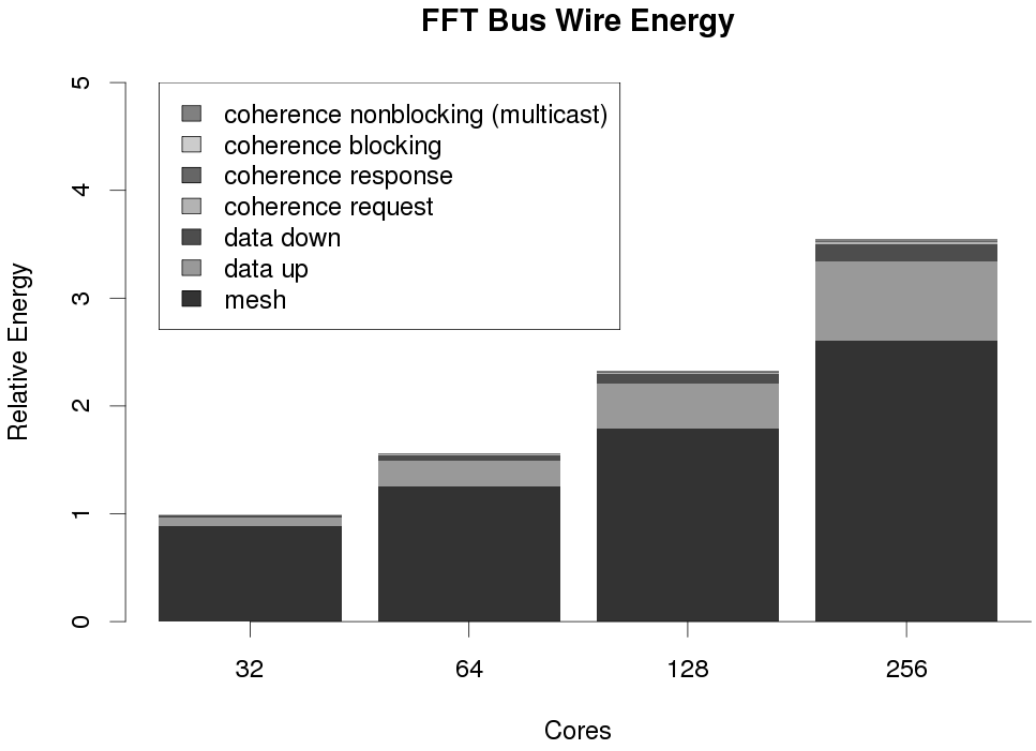

Ocean Static Energy
Normalised static energy
0
2
4
6
8
10
1.0
1.2
2.3
8.6
32
64
128
256
Cores

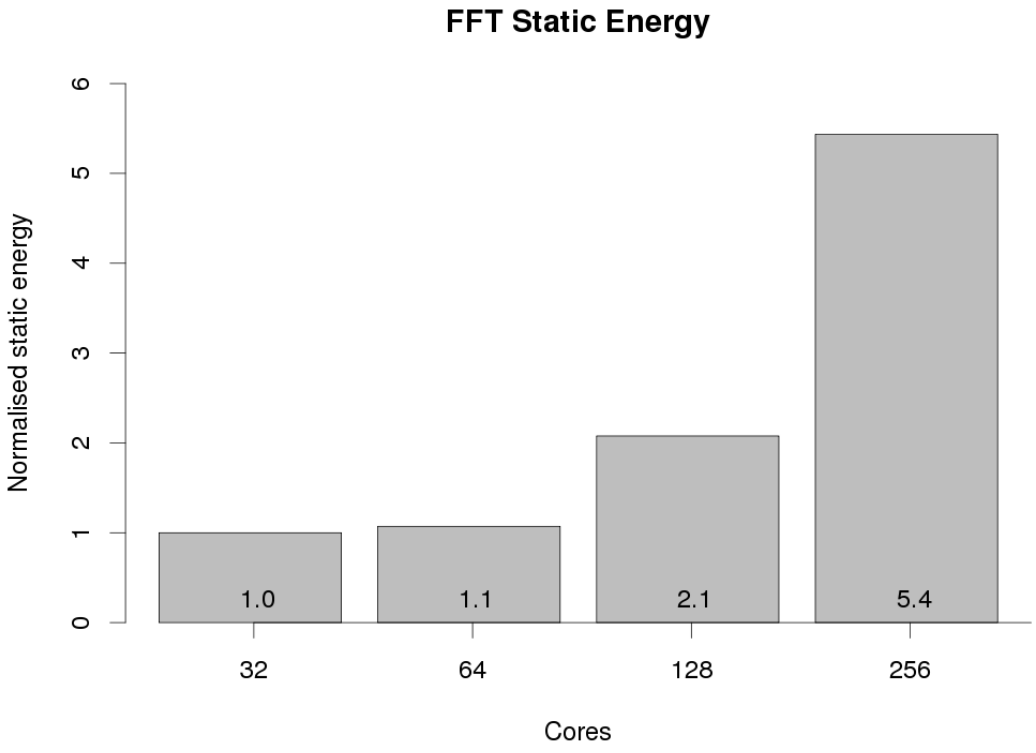

FFT Cache Misses
instruction misses
data misses
Cache Misses per 1k Instructions
0
5
10
15
20
25
30
32
64
128
256
Cores

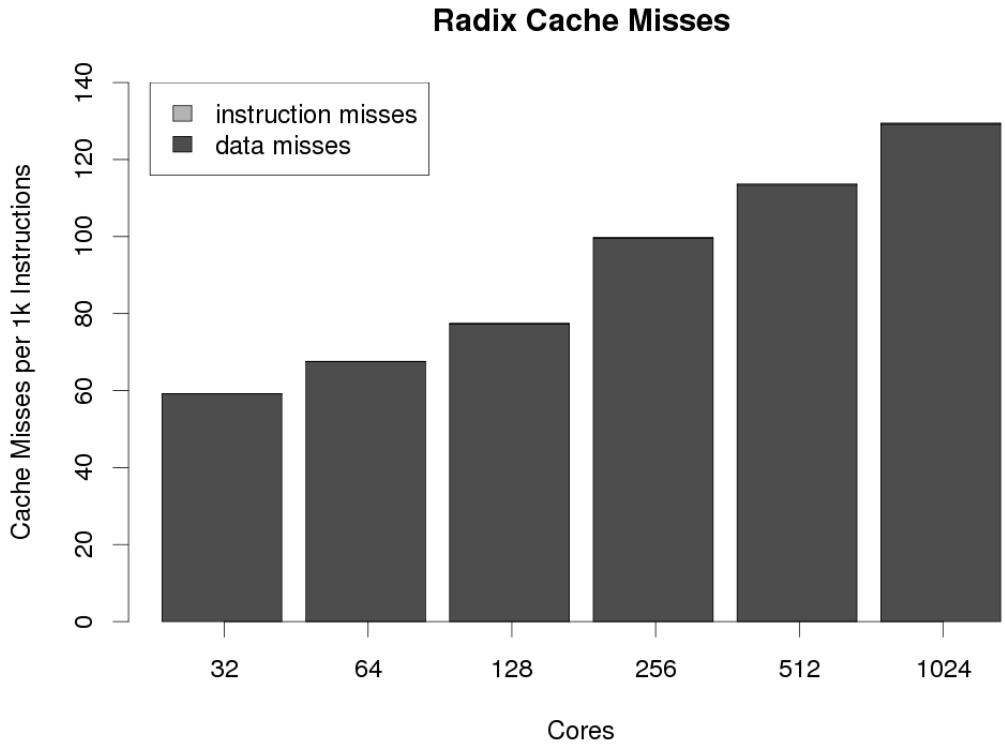

FFT Instructions
Instructions Executed (Billions)
0
1
2
3
4
5
6
7
32
64
128
256
Cores

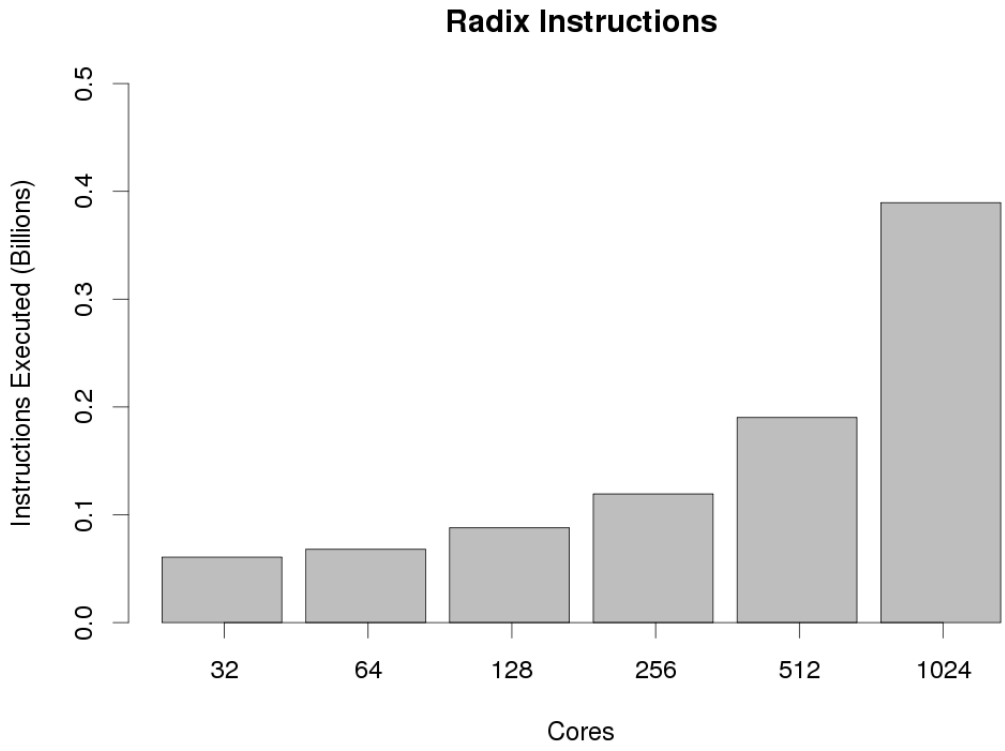

FFT Routing Energy
coherency unicast
coherency multicast
data tree
data mesh
Flits routed (Billions)
0
2
4
6
8
32
64
128
256
Cores

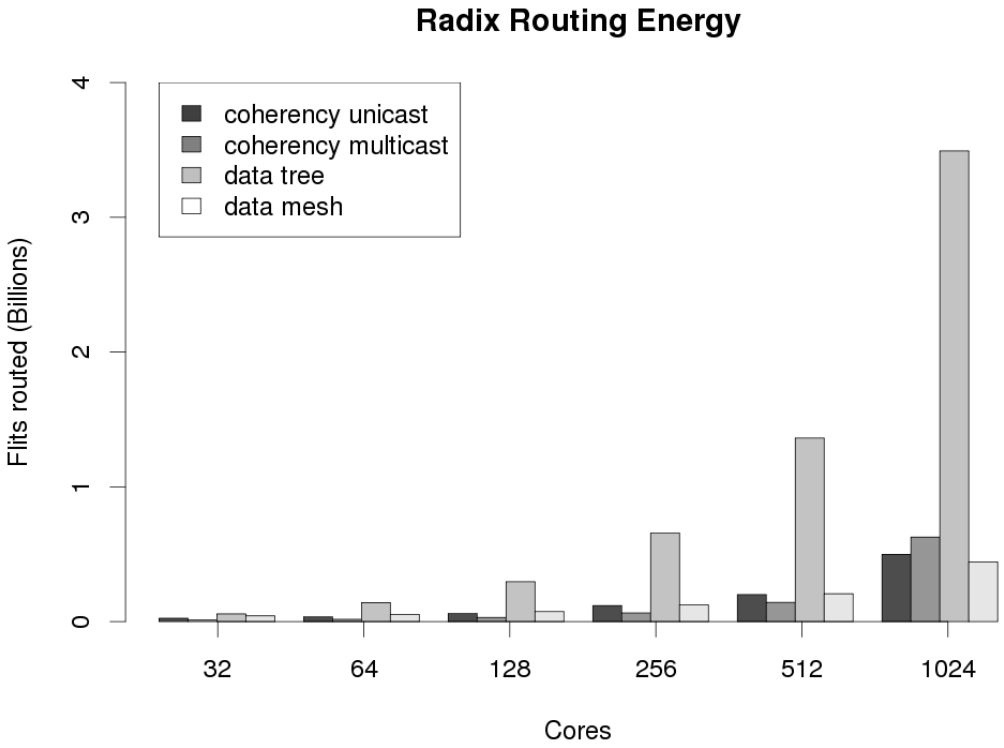

FFT Routing Energy (Normalised)
coherency unicast
coherency multicast
data tree
data mesh
Flits routed (normalised)
0
1
2
3
4
5
32
64
128
256
Cores

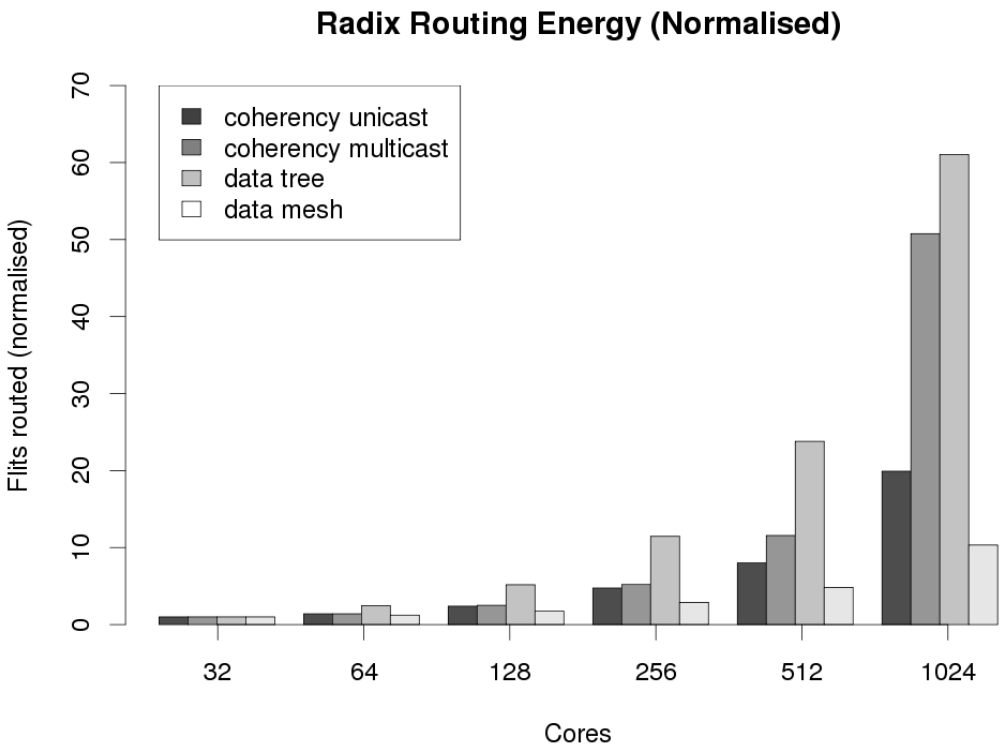

FFT Bus Wire Energy
coherence nonblocking (multicast)
coherence blocking
coherence response
coherence request
data down
data up
mesh
Relative Energy
0
1
2
3
4
5
32
64
128
256
Cores

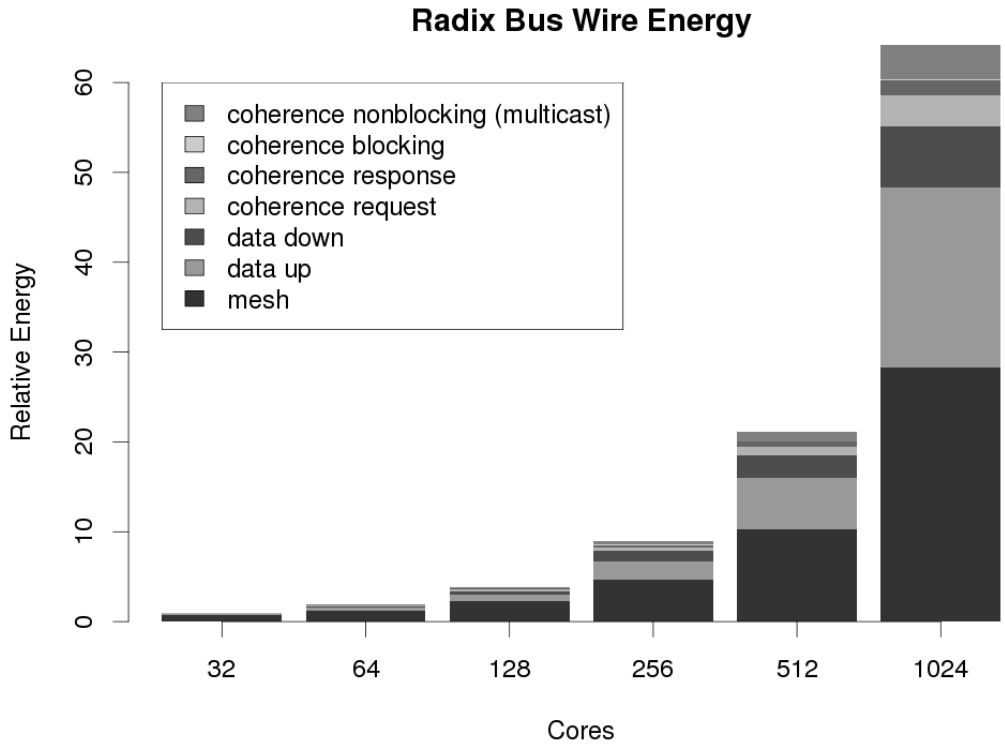

FFT Static Energy
Normalised static energy
0
1
2
3
4
5
6
1.0
1.1
2.1
5.4
32
64
128
256
Cores

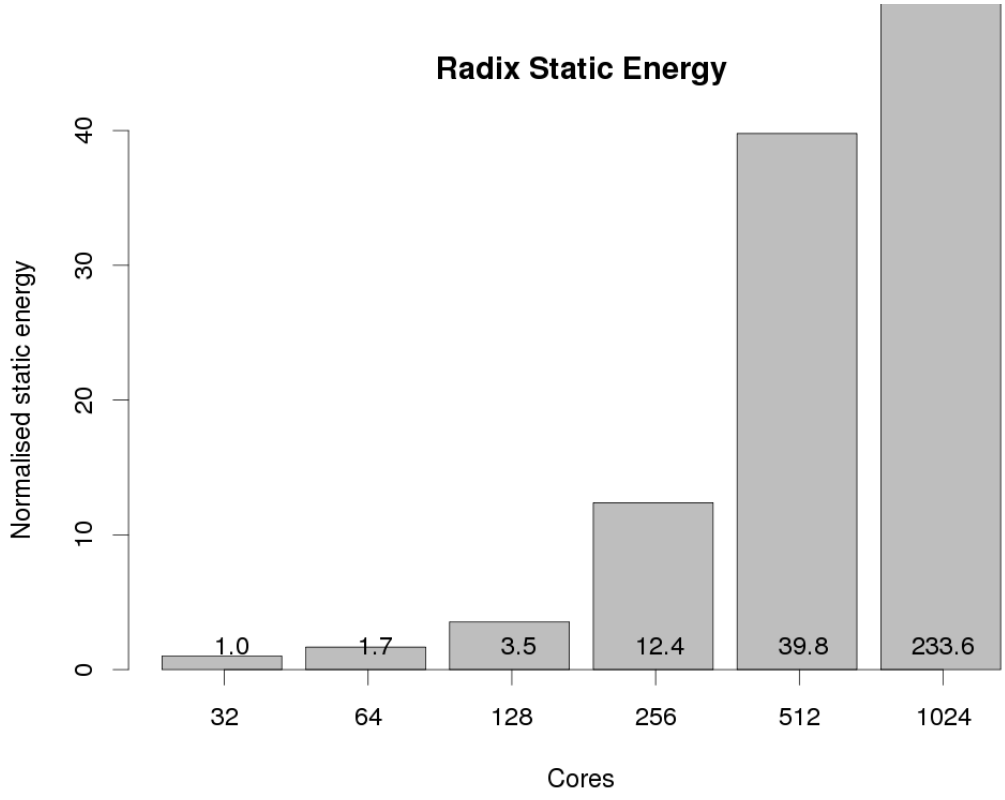

Radix Cache Misses
instruction misses
data misses
Cache Misses per 1k Instructions
0
20
40
60
80
100
120
140
32
64
128
256
512
1024
Cores


Radix Instructions
Instructions Executed (Billions)
0.0
0.1
0.2
0.3
0.4
0.5
32
64
128
256
512
1024
Cores


Radix Routing Energy
coherency unicast
coherency multicast
data tree
data mesh
Flits routed (Billions)
0
1
2
3
4
32
64
128
256
512
1024
Cores

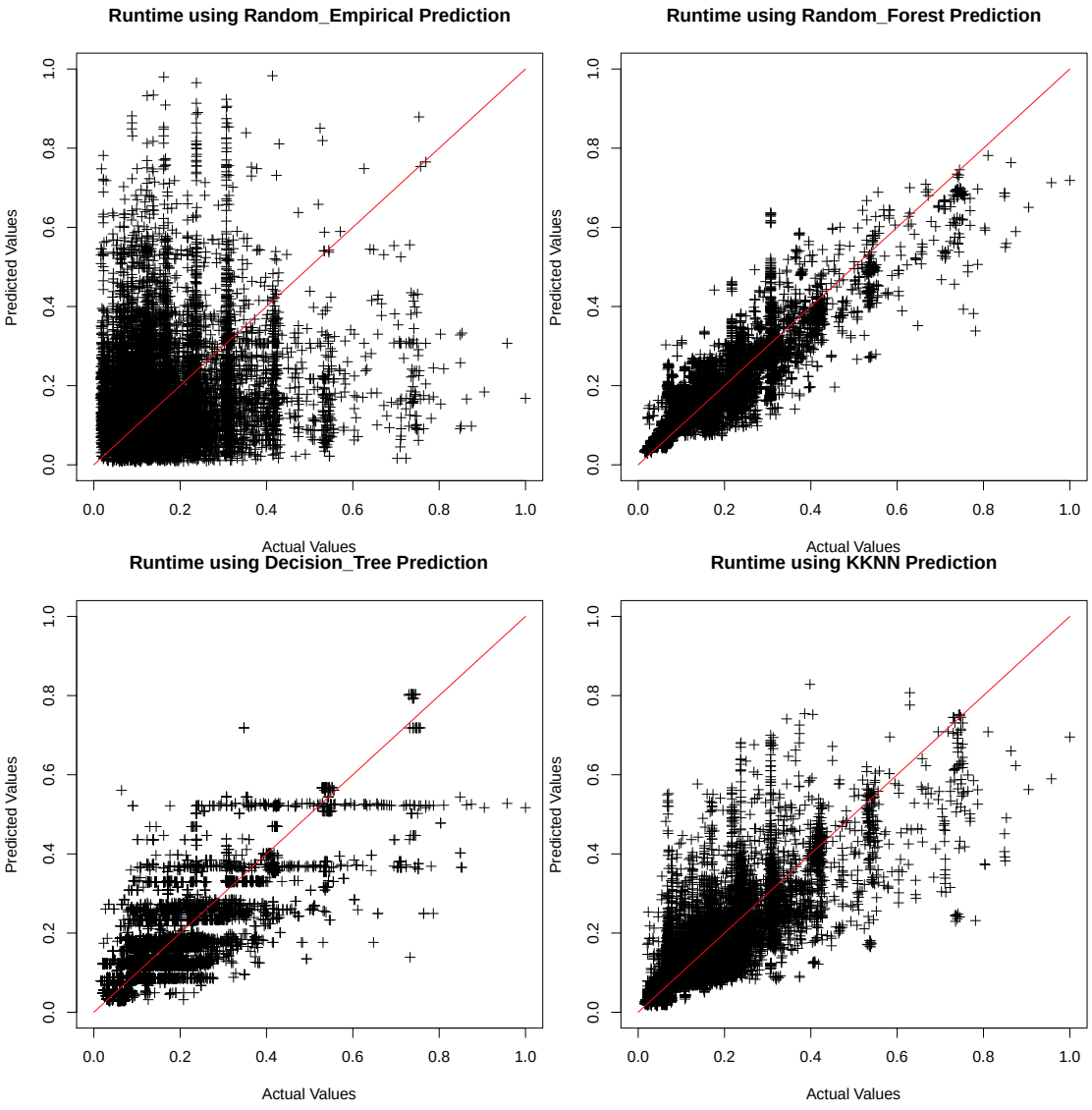

Radix Routing Energy (Normalised)
coherency unicast
coherency multicast
data tree
data mesh
Flits routed (normalised)
0
10
20
30
40
50
60
70
32
64
128
256
512
1024
Cores

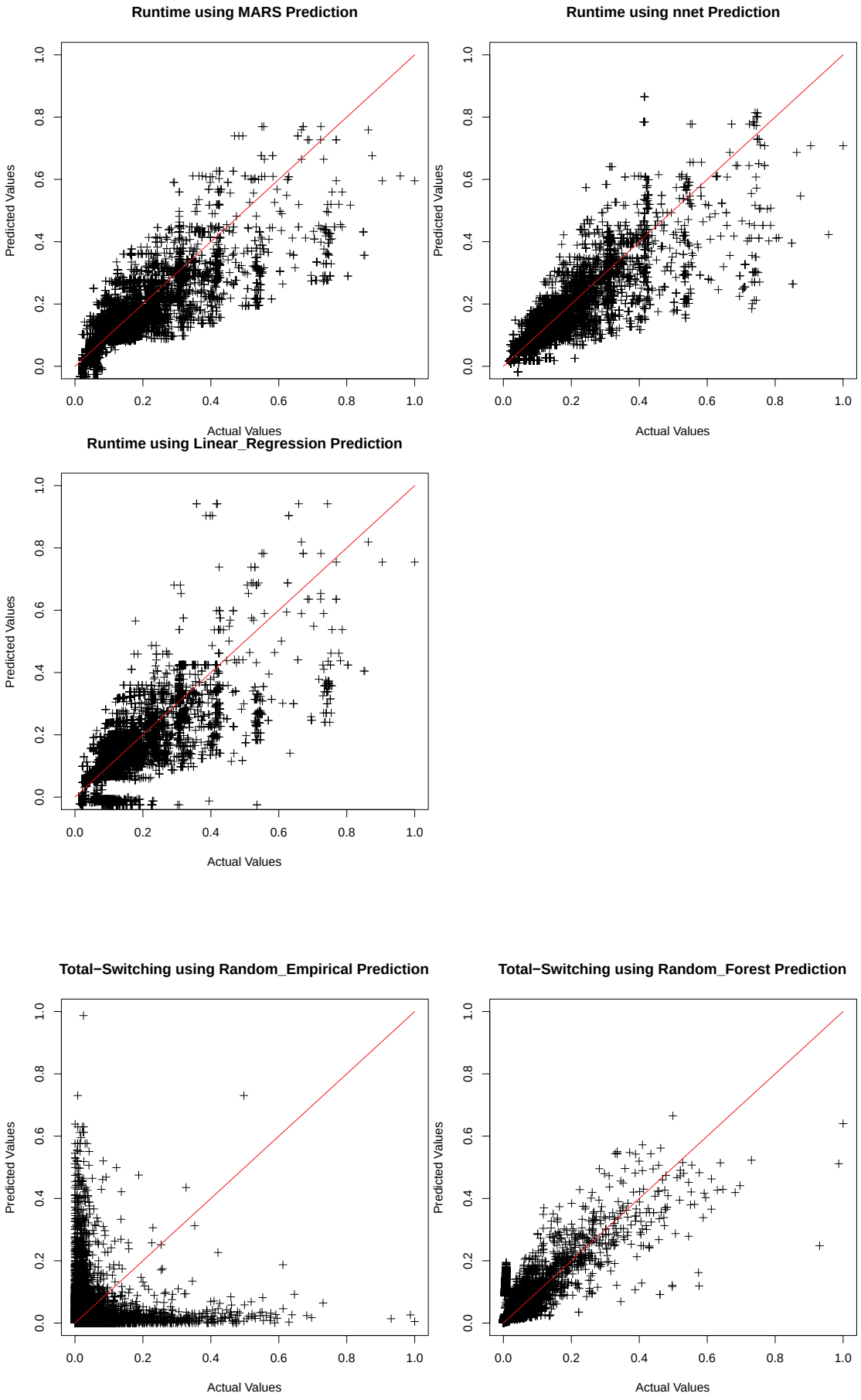

Radix Bus Wire Energy
coherence nonblocking (multicast)
coherence blocking
coherence response
coherence request
data down
data up
mesh
Relative Energy
0
10
20
30
40
50
60
32
64
128
256
512
1024
Cores

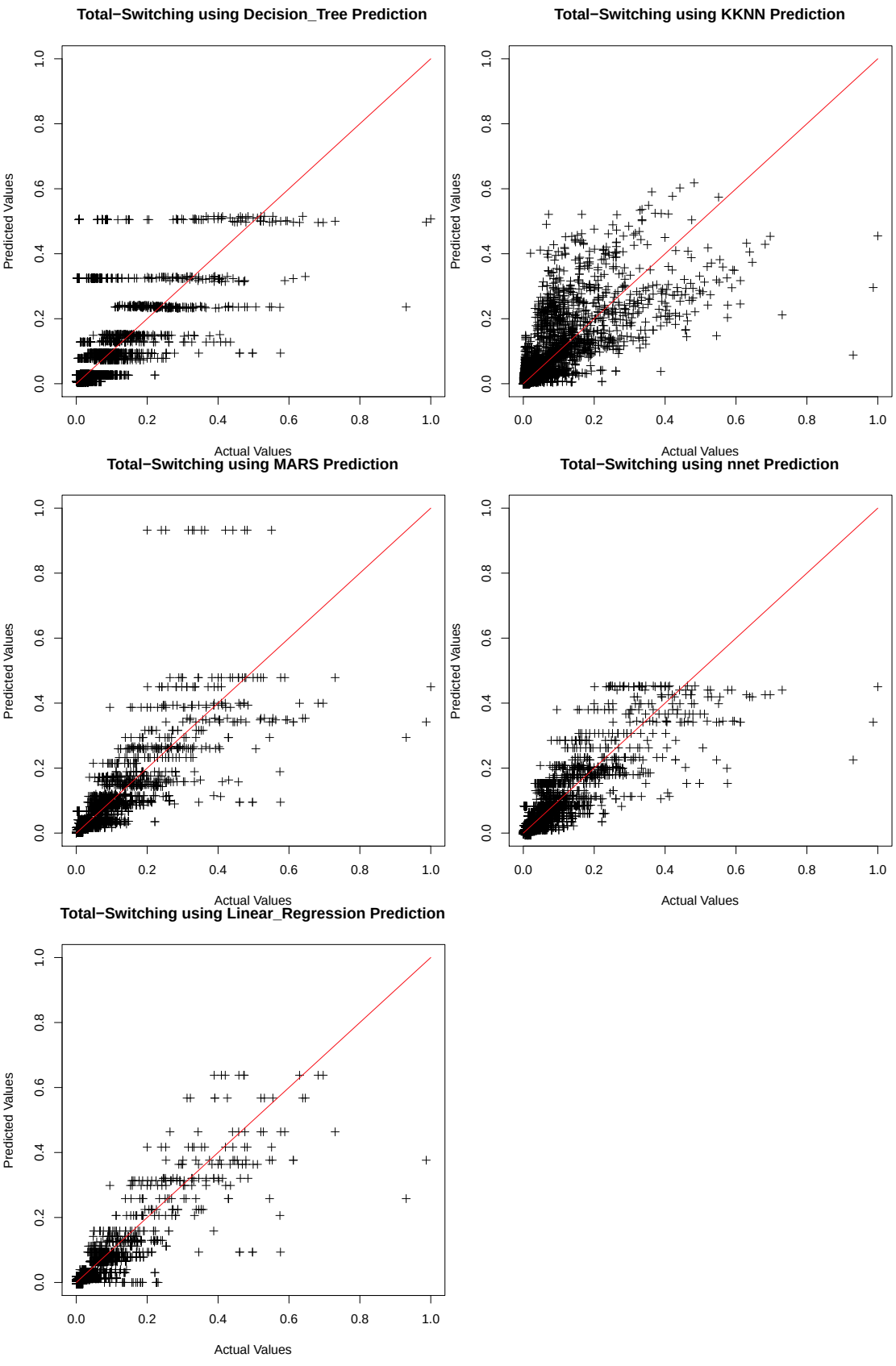

Radix Static Energy
Normalised static energy
0
10
20
30
40
1.0
1.7
3.5
12.4
39.8
233.6
32
64
128
256
512
1024
Cores

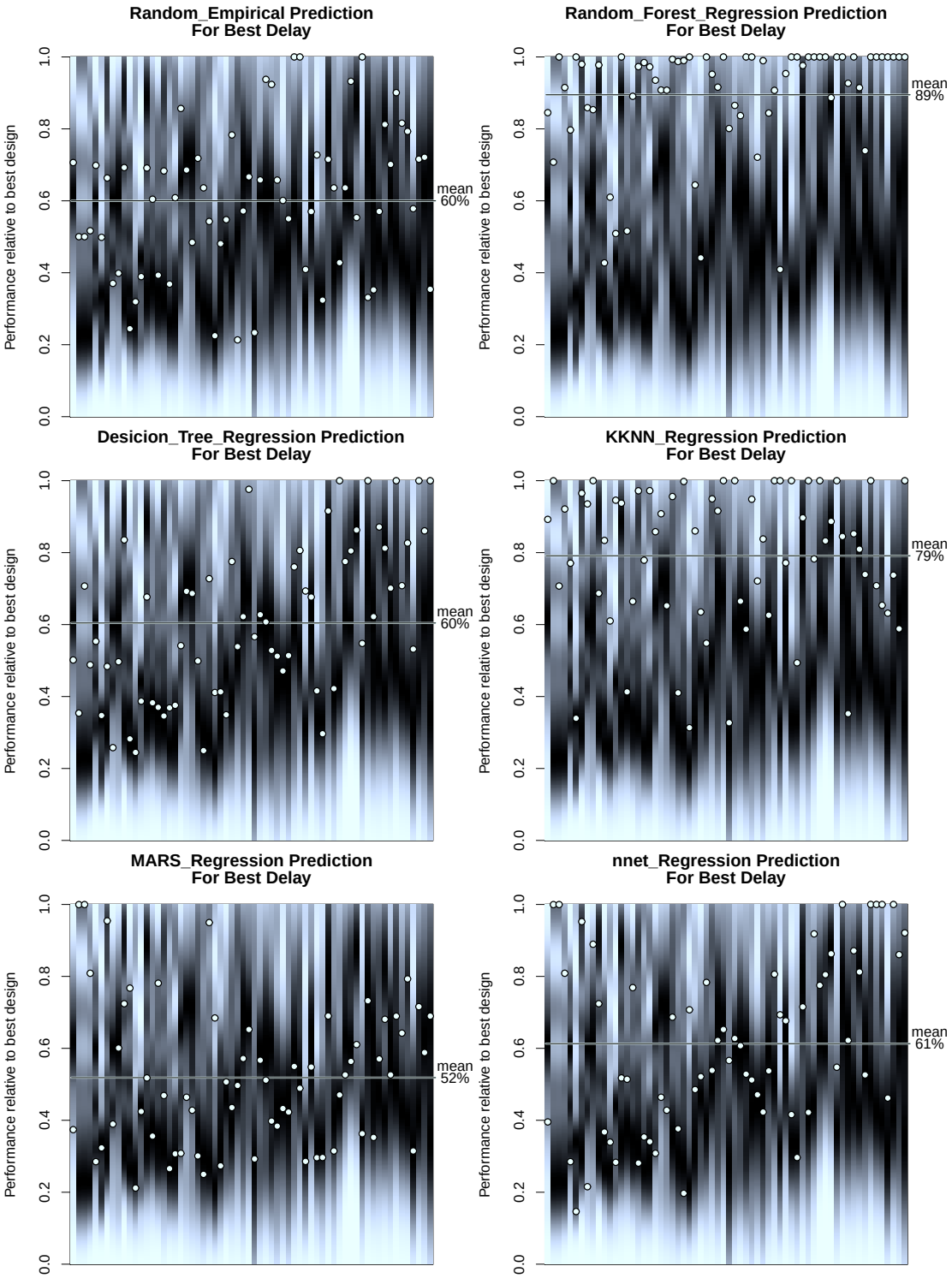
A.2 Machine Learning

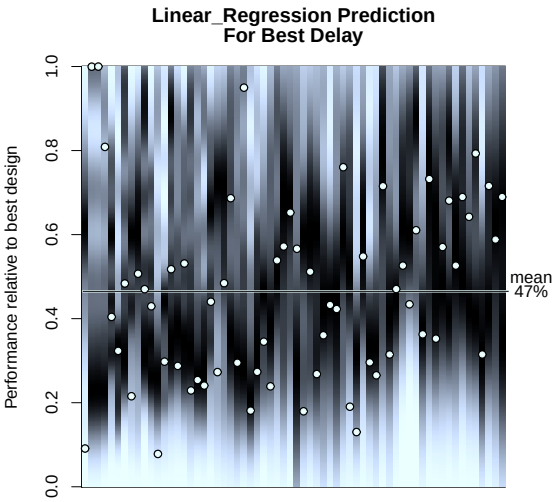
This section provides an extended set of figures for all of the machine learning methods in Chapter 8. Figures are provided for prediction of runtime and switching accuracy, as scatter-plots of predicted vs actual results, along with prediction quality figures for best runtime prediction, as a fraction of the performance of the best design and as a measure of the percentage of the design space which the prediction is better than. These two figure types relating to the "best prediction" are then provided again for energy-delay product, and again for energy-area-delay product. This provides the reader with a better insight into the performance of those methods not discussed in detail in Chapter 8 itself. Please refer to the associated figure captions in the source chapter for details on each figure type.

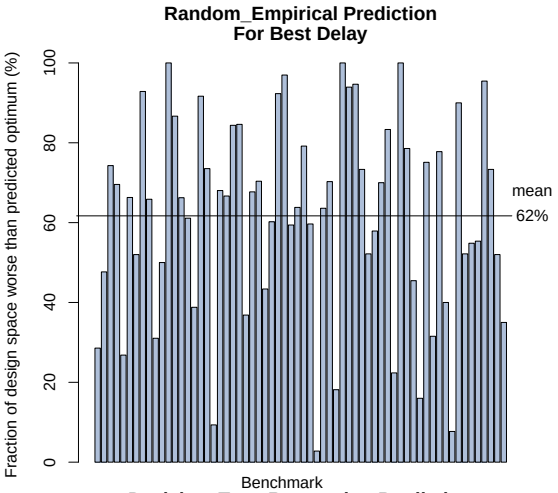


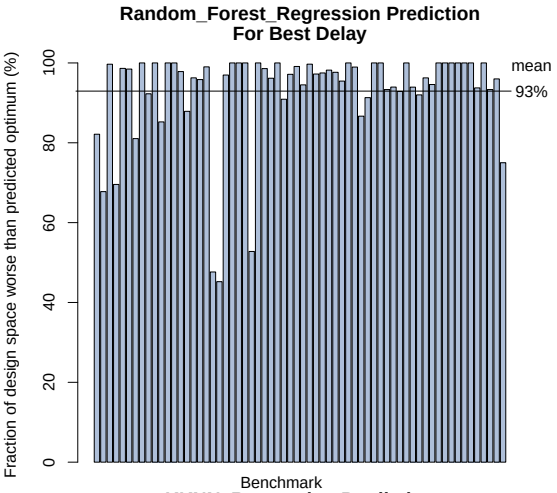

Runtime using MARS Prediction
Runtime using nnet Prediction
Runtime using Linear_Regression Prediction
Total−Switching using Random_Empirical Prediction
Total−Switching using Random_Forest Prediction
Predicted Values
Actual Values
0.0
0.2
0.4
0.6
0.8
1.0

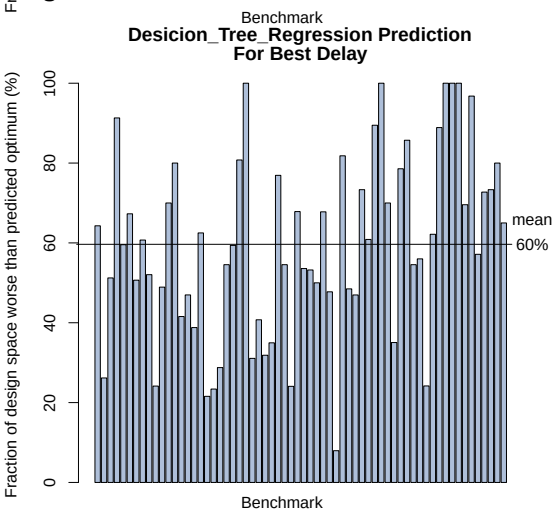

Total−Switching using Decision_Tree Prediction
Predicted Values
Actual Values
0.0
0.2
0.4
0.6
0.8
1.0
Total−Switching using KKNN Prediction
Predicted Values
Actual Values
Total−Switching using MARS Prediction
Predicted Values
Actual Values
Total−Switching using nnet Prediction
Predicted Values
Actual Values
Total−Switching using Linear_Regression Prediction
Predicted Values
Actual Values

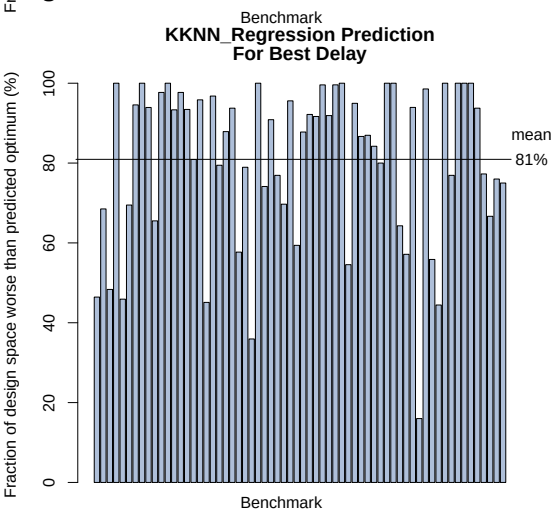

Random_Empirical Prediction
For Best Delay
Performance relative to best design
mean
60%
Random_Forest_Regression Prediction
For Best Delay
Performance relative to best design
mean
89%
Desicion_Tree_Regression Prediction
For Best Delay
Performance relative to best design
mean
60%
KKNN_Regression Prediction
For Best Delay
Performance relative to best design
mean
79%
MARS_Regression Prediction
For Best Delay
Performance relative to best design
mean
52%
nnet_Regression Prediction
For Best Delay
Performance relative to best design
mean
61%
0.0
0.2
0.4
0.6
0.8
1.0

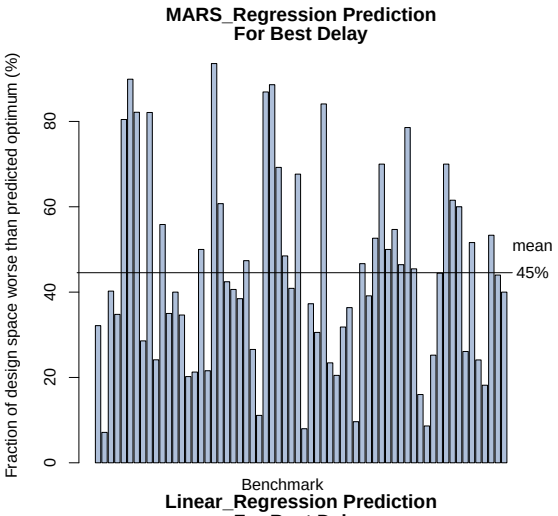

Linear_Regression Prediction
For Best Delay
Performance relative to best design
0.0
0.2
0.4
0.6
0.8
1.0
mean
47%

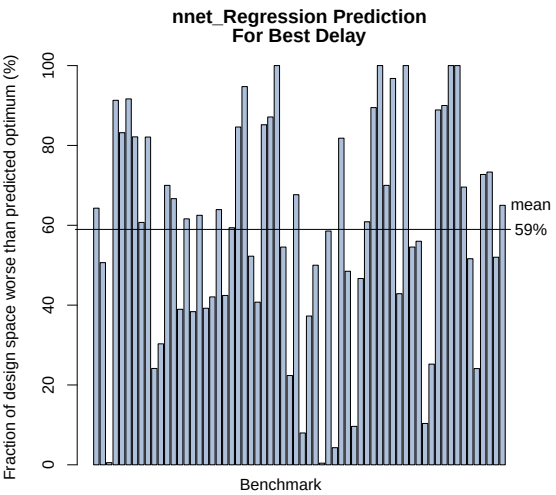

Random_Empirical Prediction
For Best Delay
Fraction of design space worse than predicted optimum (%)
0
20
40
60
80
100
mean
62%
Benchmark

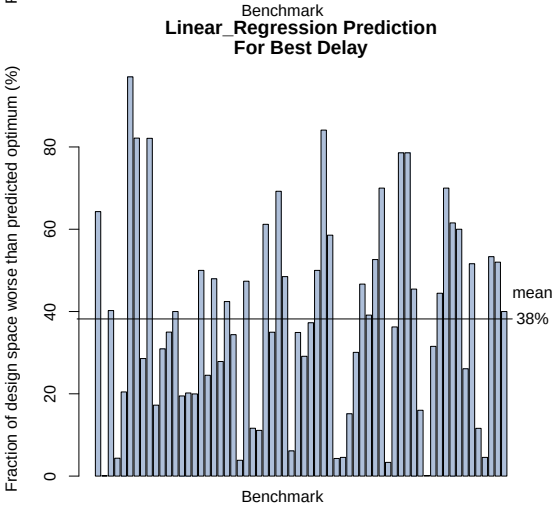

Random_Forest_Regression Prediction
For Best Delay
Fraction of design space worse than predicted optimum (%)
0
20
40
60
80
100
mean
93%
Benchmark

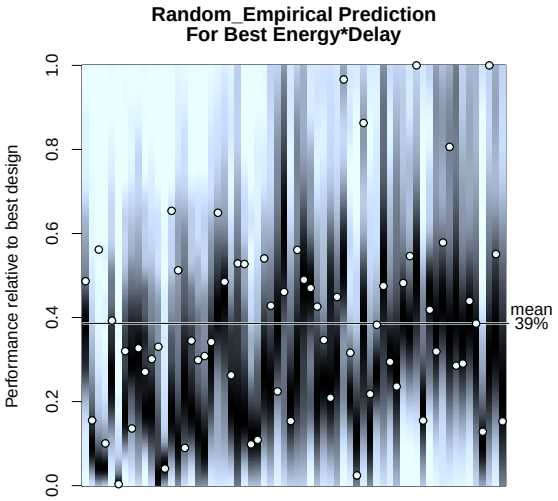

Desicion_Tree_Regression Prediction
For Best Delay
Fraction of design space worse than predicted optimum (%)
0
20
40
60
80
100
mean
60%
Benchmark

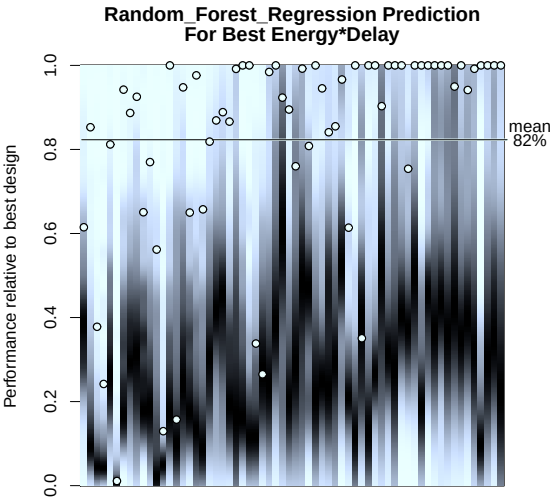

KKNN_Regression Prediction
For Best Delay
Fraction of design space worse than predicted optimum (%)
0
20
40
60
80
100
mean
81%
Benchmark

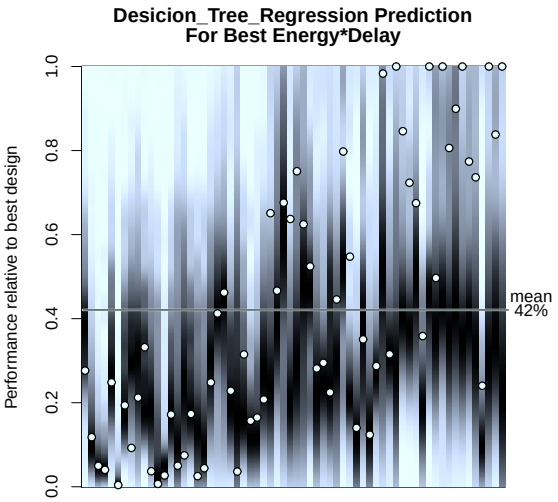

MARS_Regression Prediction
For Best Delay
Fraction of design space worse than predicted optimum (%)
0
20
40
60
80
mean
45%
Benchmark

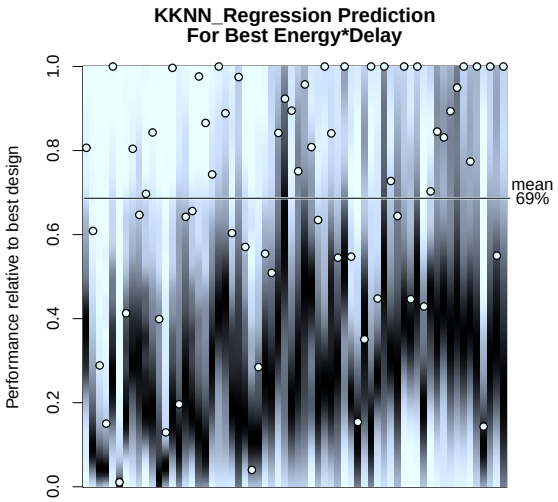

nnet_Regression Prediction
For Best Delay
Fraction of design space worse than predicted optimum (%)
0
20
40
60
80
100
mean
59%
Benchmark

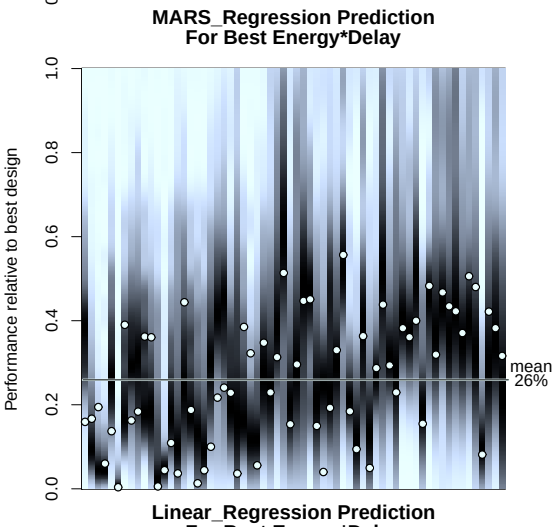

Linear_Regression Prediction
For Best Delay
Fraction of design space worse than predicted optimum (%)
0
20
40
60
80
mean
38%
Benchmark

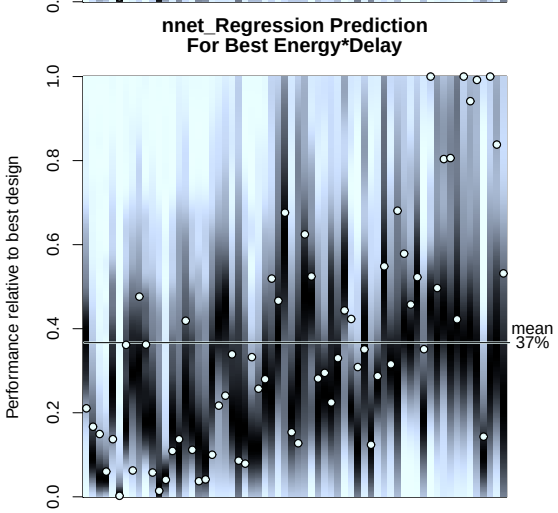

Random_Empirical Prediction
For Best Energy*Delay
Performance relative to best design
0.0
0.2
0.4
0.6
0.8
1.0
mean
39%

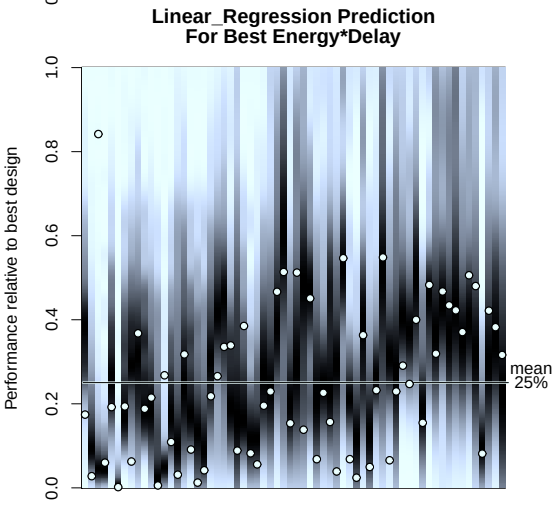

Random_Forest_Regression Prediction
For Best Energy*Delay
Performance relative to best design
0.0
0.2
0.4
0.6
0.8
1.0
mean
82%

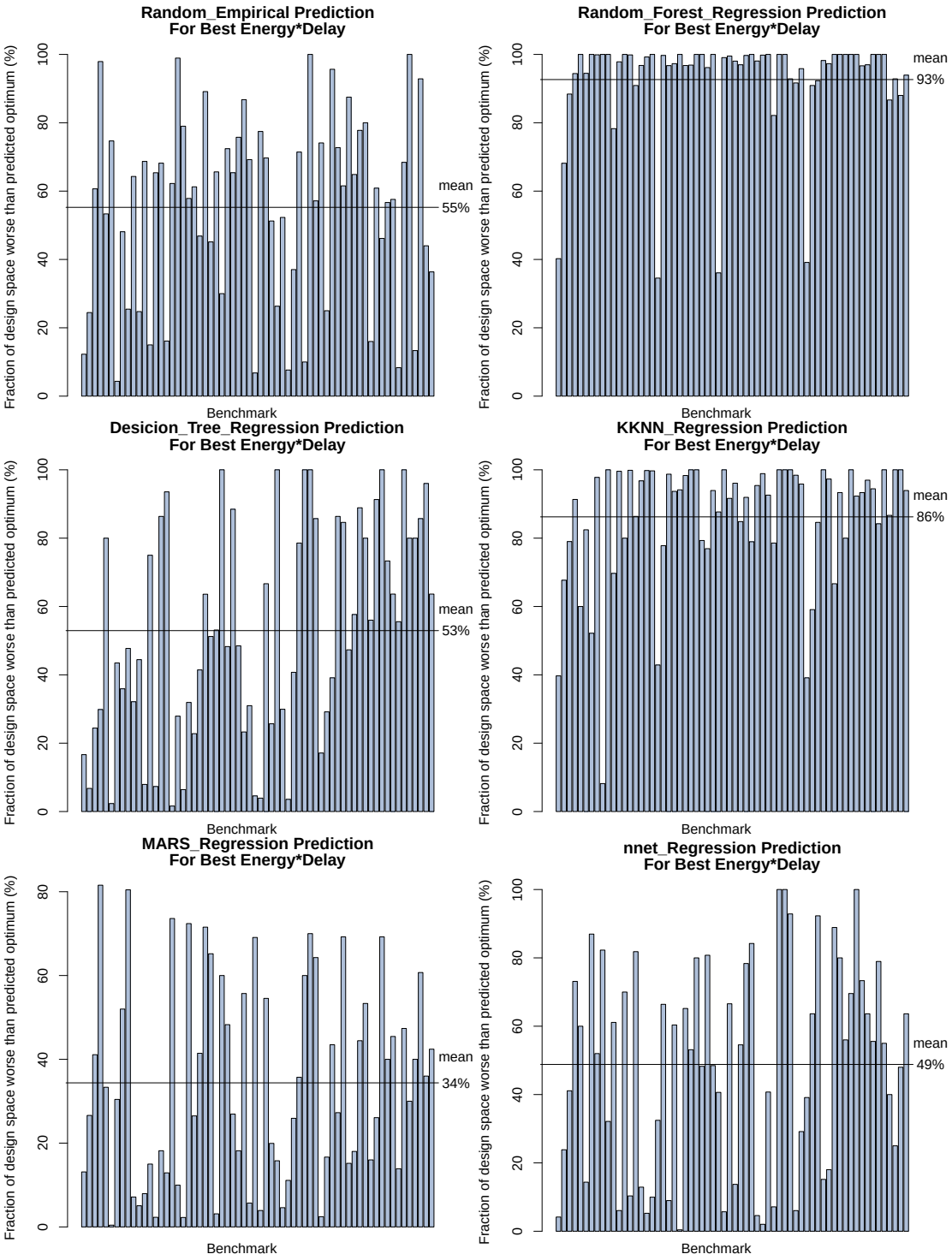

Desicion_Tree_Regression Prediction
For Best Energy*Delay
Performance relative to best design
0.0
0.2
0.4
0.6
0.8
1.0
mean
42%

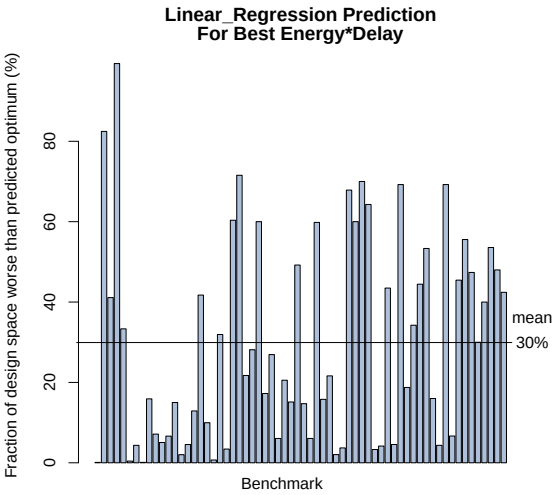

KKNN_Regression Prediction
For Best Energy*Delay
Performance relative to best design
0.0
0.2
0.4
0.6
0.8
1.0
mean
69%

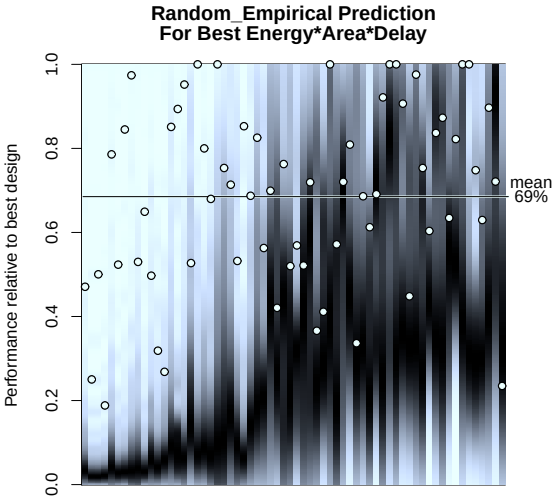

MARS_Regression Prediction
For Best Energy*Delay
Performance relative to best design
0.0
0.2
0.4
0.6
0.8
1.0
mean
26%

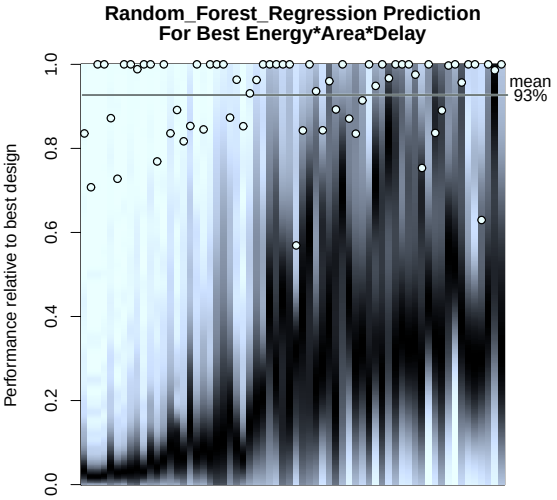

nnet_Regression Prediction
For Best Energy*Delay
Performance relative to best design
0.0
0.2
0.4
0.6
0.8
1.0
mean
37%

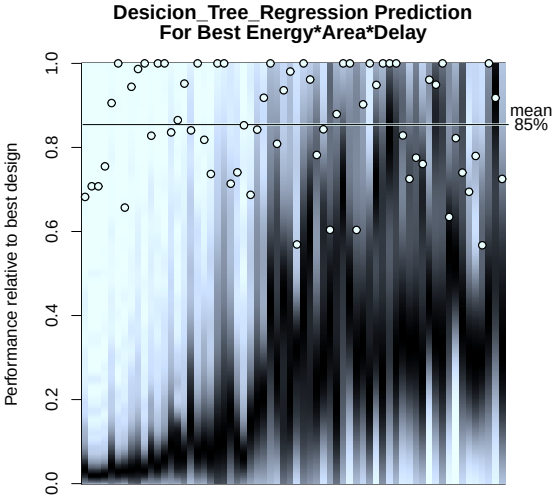

Linear_Regression Prediction
For Best Energy*Delay
Performance relative to best design
0.0
0.2
0.4
0.6
0.8
1.0
mean
25%

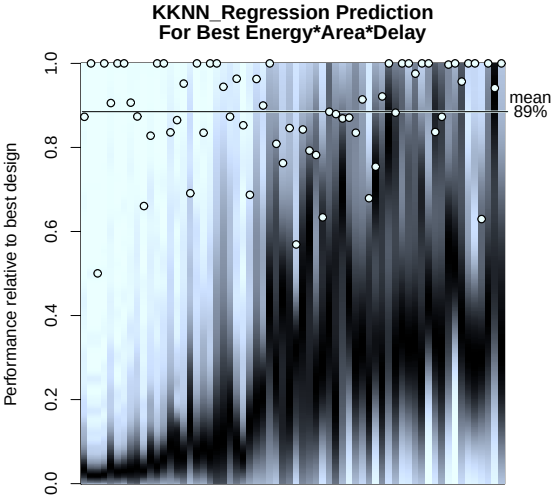

Random_Empirical Prediction
For Best Energy*Delay
Fraction of design space worse than predicted optimum (%)
0
20
40
60
80
100
mean
55%
Benchmark
Random_Forest_Regression Prediction
For Best Energy*Delay
Fraction of design space worse than predicted optimum (%)
mean
93%
Benchmark
Desicion_Tree_Regression Prediction
For Best Energy*Delay
Fraction of design space worse than predicted optimum (%)
mean
53%
Benchmark
KKNN_Regression Prediction
For Best Energy*Delay
Fraction of design space worse than predicted optimum (%)
mean
86%
Benchmark
MARS_Regression Prediction
For Best Energy*Delay
Fraction of design space worse than predicted optimum (%)
mean
34%
Benchmark
nnet_Regression Prediction
For Best Energy*Delay
Fraction of design space worse than predicted optimum (%)
mean
49%
Benchmark

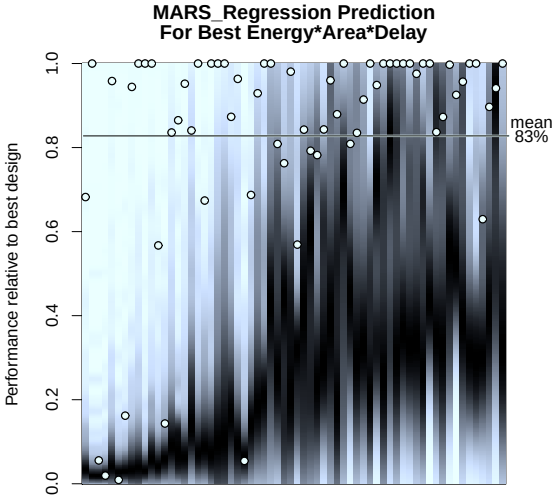

Linear_Regression Prediction
For Best Energy*Delay
Fraction of design space worse than predicted optimum (%)
0
20
40
60
80
mean
30%
Benchmark

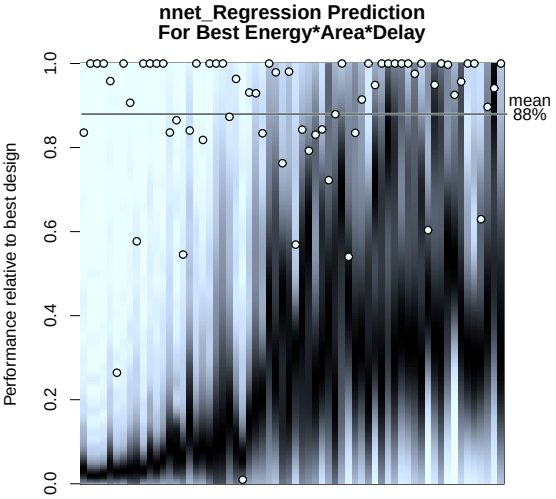

Random_Empirical Prediction
For Best Energy*Area*Delay
Performance relative to best design
0.0
0.2
0.4
0.6
0.8
1.0
mean
69%

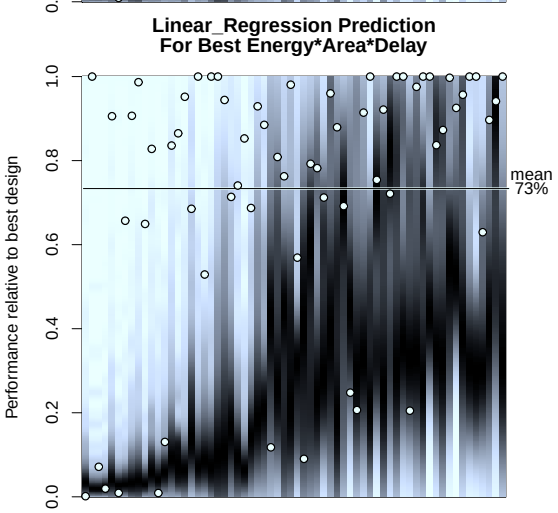

Random_Forest_Regression Prediction
For Best Energy*Area*Delay
Performance relative to best design
0.0
0.2
0.4
0.6
0.8
1.0
mean
93%

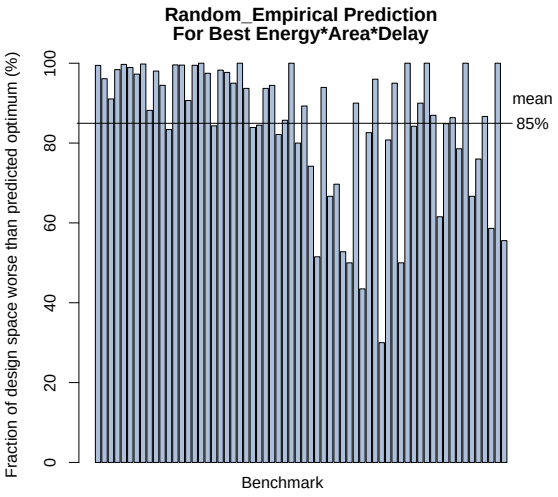

Desicion_Tree_Regression Prediction
For Best Energy*Area*Delay
Performance relative to best design
0.0
0.2
0.4
0.6
0.8
1.0
mean
85%

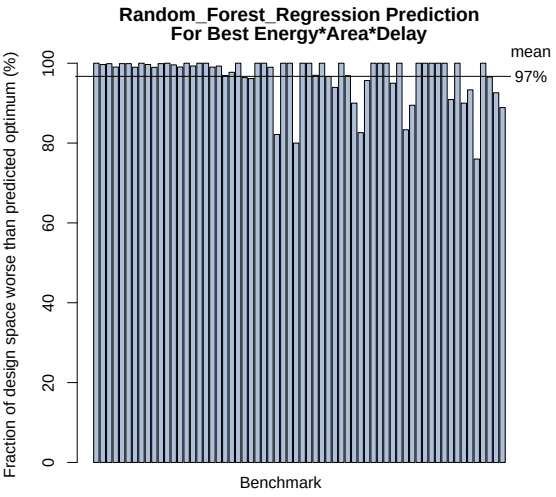

KKNN_Regression Prediction
For Best Energy*Area*Delay
Performance relative to best design
0.0
0.2
0.4
0.6
0.8
1.0
mean
89%

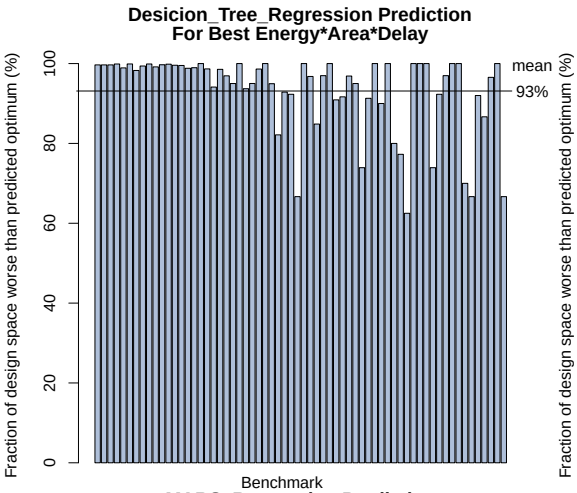

MARS_Regression Prediction
For Best Energy*Area*Delay
Performance relative to best design
0.0
0.2
0.4
0.6
0.8
1.0
mean
83%

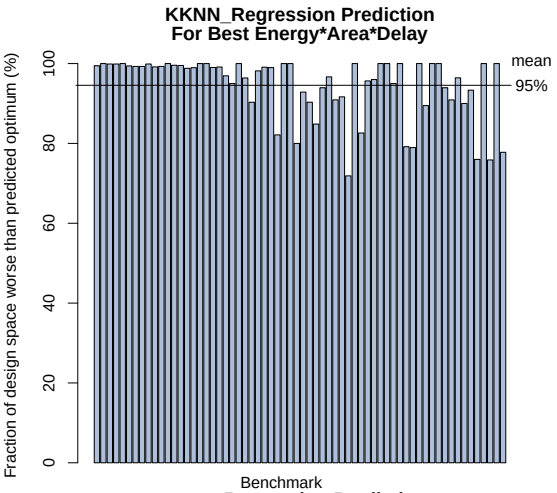

nnet_Regression Prediction
For Best Energy*Area*Delay
Performance relative to best design
0.0
0.2
0.4
0.6
0.8
1.0
mean
88%

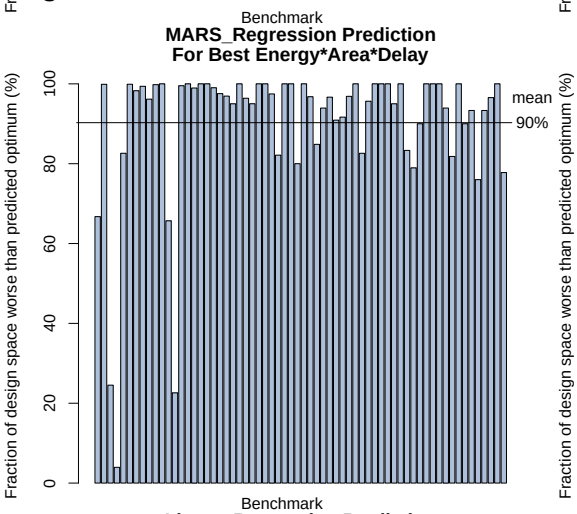

Linear_Regression Prediction
For Best Energy*Area*Delay
Performance relative to best design
0.0
0.2
0.4
0.6
0.8
1.0
mean
73%

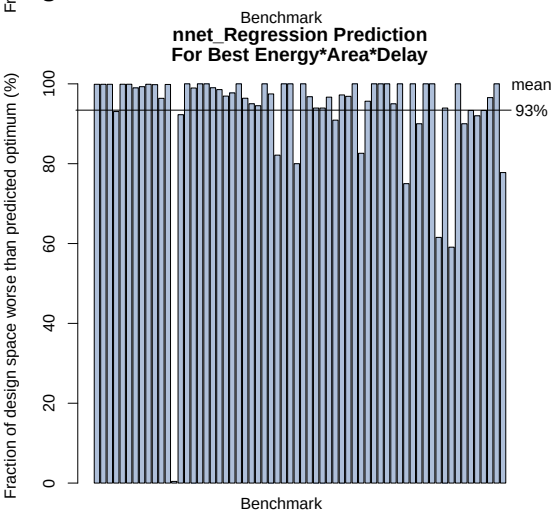

Random_Empirical Prediction
For Best Energy*Area*Delay
Fraction of design space worse than predicted optimum (%)
0
20
40
60
80
100
mean
85%
Benchmark

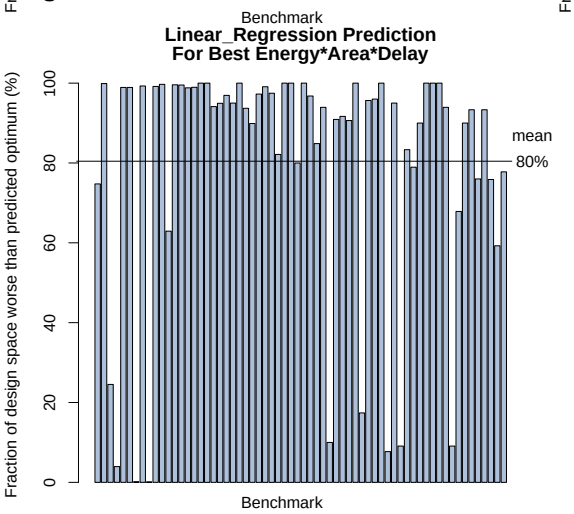

Random_Forest_Regression Prediction
For Best Energy*Area*Delay
Fraction of design space worse than predicted optimum (%)
0
20
40
60
80
100
mean
97%
Benchmark


Desicion_Tree_Regression Prediction
For Best Energy*Area*Delay
Fraction of design space worse than predicted optimum (%)
0
20
40
60
80
100
mean
93%
Benchmark


KKNN_Regression Prediction
For Best Energy*Area*Delay
Fraction of design space worse than predicted optimum (%)
0
20
40
60
80
100
mean
95%
Benchmark


MARS_Regression Prediction
For Best Energy*Area*Delay
Fraction of design space worse than predicted optimum (%)
0
20
40
60
80
100
mean
90%
Benchmark


nnet_Regression Prediction
For Best Energy*Area*Delay
Fraction of design space worse than predicted optimum (%)
0
20
40
60
80
100
mean
93%
Benchmark


Linear_Regression Prediction
For Best Energy*Area*Delay
Fraction of design space worse than predicted optimum (%)
0
20
40
60
80
100
mean
80%
Benchmark

Bibliography

[1] G. E. Moore, "Cramming more components into integrated circuits," *Electronics Magazine*, vol. 38, p. 4, April 1965.

[2] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger, "Clock rate versus IPC: the end of the road for conventional microarchitectures," in *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pp. 248–259, June 2000.

[3] D. W. Wall, "Limits of Instruction-level Parallelism," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, (New York, NY, USA), pp. 176–188, ACM, 1991.

[4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick, M. J. Demmel, W. Plishker, J. Shalf, S. Williams, and K. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," tech. rep., UC BERKELEY, 2006.

[5] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The Case for a Single-chip Multiprocessor," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, (New York, NY, USA), pp. 2–11, ACM, 1996.

[6] J. Held, J. Bautista, and S. Koehl, "From a Few Cores to Many: A Tera-scale Computing Research Overview," *Intel Research Whitepaper*, 2006.

[7] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar, "An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS," *Solid-State Circuits, IEEE Journal of*, vol. 43, pp. 29–41, Jan 2008.

[8] C. Moore, "Data Processing In Exascale-class Computer Systems," in *The Salishan Conference on High Speed Computing*, April 2011.

[9] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," *Computer*, vol. 41, pp. 33–38, July 2008.

[10] M. Loghi, M. Letis, L. Benini, and M. Poncino, "Exploring the Energy Efficiency of Cache Coherence Protocols in Single-chip Multi-processors," in *Proceedings of the 15th ACM Great Lakes Symposium on VLSI*, GLSVLSI '05, (New York, NY, USA), pp. 276–281, ACM, 2005.

[11] C. Thompson, M. Gould, and N. Topham, "High speed cycle approximate simulation for cache-incoherent MPSoCs," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 Int'l Conference on*, pp. 88–95, July 2013.

[12] O. Almer, I. Bohm, T. von Koch, B. Franke, S. Kyle, V. Seeker, C. Thompson, and N. Topham, "Scalable multi-core simulation using parallel dynamic binary translation," in *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pp. 190 –199, july 2011.

[13] O. Almer, I. Boehm, T. von Koch, B. Franke, S. Kyle, V. Seeker, C. Thompson, and N. Topham, "A parallel dynamic binary translator for efficient multi-core simulation," *International Journal of Parallel Programming*, vol. 41, no. 2, pp. 212–235, 2013.

[14] D. E. Culler, A. Gupta, and J. P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 1997.

[15] S. Owicki and A. Agarwal, "Evaluating the performance of software cache coherence," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS III, (New York, NY, USA), pp. 230–242, ACM, 1989.

[16] Y.-C. Chen and A. V. Veidenbaum, "Comparison and analysis of software and directory coherence schemes," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, (New York, NY, USA), pp. 818–829, ACM, 1991.

[17] A. V. Veidenbaum, "A compiler-assisted cache coherence solution for multiprcessors," in *International Conference on Parallel Processing, ICPP'86, University Park, PA, USA, August 1986.*, pp. 1029–1036, 1986.

[18] S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *Computer*, vol. 29, pp. 66–76, Dec. 1996.

[19] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, pp. 690–691, Sept. 1979.

[20] S. Owens, S. Sarkar, and P. Sewell, "A Better x86 Memory Model: X86-TSO," in *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, (Berlin, Heidelberg), pp. 391–407, Springer-Verlag, 2009.

[21] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "X86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors," *Commun. ACM*, vol. 53, pp. 89–97, July 2010.

[22] C. SPARC International, Inc., *The SPARC Architecture Manual (Version 9)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994.

[23] N. Njoroge, J. Casper, S. Wee, Y. Teslyar, D. Ge, C. Kozyrakis, and K. Olukotun, "ATLAS: A Chip-multiprocessor with Transactional Memory Support," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '07, (San Jose, CA, USA), pp. 3–8, EDA Consortium, 2007.

[24] J. Reinders, "Transactional Synchronization in Haswell." `https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell`, February 2012.

[25] R. Rajwar and J. R. Goodman, "Speculative lock elision: Enabling highly concurrent multithreaded execution," in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, (Washington, DC, USA), pp. 294–305, IEEE Computer Society, 2001.

[26] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[27] M. S. Papamarcos and J. H. Patel, "A Low-overhead Coherence Solution for Multiprocessors with Private Cache Memories," in *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ISCA '84, (New York, NY, USA), pp. 348–354, ACM, 1984.

[28] C. Ravishankar and J. Goodman, "Cache Implementation for Multiple Microprocessors," in *Proceedings of IEEE COMPCON*, pp. 346–350, February 1983.

[29] J. R. Goodman, "Using Cache Memory to Reduce Processor-memory Traffic," in *Proceedings of the 10th Annual International Symposium on Computer Architecture*, ISCA '83, (New York, NY, USA), pp. 124–131, ACM, 1983.

[30] H. Hum and J. Goodman, "Forward state for use in cache coherency in a multiprocessor system," July 26 2005. US Patent 6,922,756.

[31] D. Kanter, "The Common System Interface: Intel's Future Interconnect." `http://www.realworldtech.com/common-system-interface/5/`, August 2007.

[32] J. Archibald and J.-L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. Comput. Syst.*, vol. 4, pp. 273–298, Sept. 1986.

[33] E. McCreight, "The Dragon Computer System," in *Microarchitecture of VLSI Computers* (P. Antognetti, F. Anceau, and J. Vuillemin, eds.), vol. 96 of *NATO ASI Series*, pp. 83–101, Springer Netherlands, 1985.

[34] J. H. Kelm, M. R. Johnson, S. S. Lumettta, and S. J. Patel, "Waypoint: Scaling coherence to thousand-core architectures," in *Proceedings of the 19th Int'l Conference on Parallel Architectures and Compilation Techniques*, PACT '10, (New York, NY, USA), pp. 99–110, ACM, 2010.

[35] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling Ways and Associativity," in *Proceedings of the 2010 43rd Annual IEEE/ACM Int'l Symp. on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 187–198, IEEE Computer Society, 2010.

[36] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo directory: A scalable directory for many-core systems," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th IntâĂŹl Symp. on*, pp. 169–180, Feb 2011.

[37] D. Sanchez and C. Kozyrakis, "SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding," in *Proceedings of the 2012 IEEE 18th Int'l Symp. on High-Performance Computer Architecture*, HPCA '12, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2012.

[38] M. M. K. Martin, M. D. Hill, and D. A. Wood, "Token coherence: Decoupling performance and correctness," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, (New York, NY, USA), pp. 182–193, ACM, 2003.

[39] N. Jerger, L.-S. Peh, and M. Lipasti, "Virtual circuit tree multicasting: A case for on-chip hardware multicast support," in *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, pp. 229–240, June 2008.

[40] D. B. Gustavson, "The scalable coherent interface and related standards projects," *IEEE Micro*, vol. 12, pp. 10–22, Jan. 1992.

[41] C. Fensch and M. Cintra, "An OS-Based Alternative to Full Hardware Coherence on Tiled CMPs," in *Proceedings of the 14 International Symposium on High-Performance Computer Architecture*, pp. 355–366, IEEE, Feb 2008.

[42] S. V. Adve, V. S. Adve, M. D. Hill, and M. K. Vernon, "Comparison of Hardware and Software Cache Coherence Schemes," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ISCA '91, (New York, NY, USA), pp. 298–308, ACM, 1991.

[43] Y. P. Zhang, T. Jeong, F. Chen, H. Wu, R. Nitzsche, and G. Gao, "A study of the on-chip interconnection network for the ibm cyclops64 multi-core architecture," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 10 pp.–, April 2006.

[44] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *IEEE Trans. Comput.*, vol. 34, no. 10, pp. 892–901, 1985.

[45] P. Newman, *Fast Packet Switching for Integrated Services*. PhD thesis, University of Cambridge, 1988.

[46] O. Almer, *Automated Application-Specific Optimisation of Interconnects in Multi-Core Systems*. PhD thesis, The University of Edinburgh, 2012.

[47] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, (Washington, DC, USA), pp. 249–261, IEEE Computer Society, 2007.

[48] J. Lee, J. Kim, C. Jang, S. Kim, B. Egger, K. Kim, and S. Han, "FaCSim: a fast and cycle-accurate architecture simulator for embedded systems," in *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '08, (New York, NY, USA), pp. 89–100, ACM, 2008.

[49] M. Monchiero, J. H. Ahn, A. Falcón, D. Ortega, and P. Faraboschi, "How to simulate 1000 cores," *SIGARCH Comput. Archit. News*, vol. 37, pp. 10–19, July 2009.

[50] I. Böhm, T. J. Edler von Koch, S. C. Kyle, B. Franke, and N. Topham, "Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator," *SIGPLAN Not.*, vol. 46, pp. 74–85, June 2011.

[51] I. Synopsys, "HAPS® Family of FPGA-Based Prototyping Solutions." `http://www.synopsys.com/Systems/FPGABasedPrototyping/Pages/HAPS.aspx`, 2014.

[52] C. Fensch, N. Barrow-Williams, R. Mullins, and S. Moore, "Designing a Physical Locality Aware Coherence Protocol for Chip-Multiprocessors," *IEEE Trans. Comput.*, vol. 62, pp. 914–928, May 2013.

[53] Y. Zhang, R. Chen, W. Ye, and M. Irwin, "System level interconnect power modeling," in *ASIC Conference 1998. Proceedings. Eleventh Annual IEEE International*, pp. 289–293, Sep 1998.

[54] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. Irwin, "The design and use of simplePower: a cycle-accurate energy estimation tool," in *Design Automation Conference, 2000. Proceedings 2000*, pp. 340–345, 2000.

[55] K. Sundaresan and N. Mahapatra, "Accurate energy dissipation and thermal modeling for nanometer-scale buses," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pp. 51–60, Feb 2005.

[56] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," in *Computer Architecture, 1988. Conference Proceedings. 15th Annual Int'l Symp. on*, pp. 280–289, May 1988.

[57] G. Kurian, J. E. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. C. Kimerling, and A. Agarwal, "Atac: A 1000-core cache-coherent processor with on-chip optical network," in *Proceedings of the 19th Int'l Conference on Parallel Architectures and Compilation Techniques*, PACT '10, (New York, NY, USA), pp. 477–488, ACM, 2010.

[58] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "Limitless directories: A scalable cache coherence scheme," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, (New York, NY, USA), pp. 224–234, ACM, 1991.

[59] S. S. Mukherjee and M. D. Hill, "An evaluation of directory protocols for medium-scale shared-memory multiprocessors," in *Proceedings of the 8th International Conference on Supercomputing*, ICS '94, (New York, NY, USA), pp. 64–74, ACM, 1994.

[60] L. Fang, P. Liu, Q. Hu, M. C. Huang, and G. Jiang, "Building expressive, area-efficient coherence directories," in *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, (Piscataway, NJ, USA), pp. 299–308, IEEE Press, 2013.

[61] A. Gupta, W. Weber, and T. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," *Scalable shared memory multiprocessors*, p. 167, 1992.

[62] M. Acacio, J. Gonzalez, J. Garcia, and J. Duato, "A new scalable directory architecture for large-scale multiprocessors," in *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pp. 97–106, 2001.

[63] H. Zhao, A. Shriraman, and S. Dwarkadas, "SPACE: Sharing Pattern-based Directory Coherence for Multicore Scalability," in *Proceedings of the 19th Int'l Conference on Parallel Architectures and Compilation Techniques*, PACT '10, (New York, NY, USA), pp. 135–146, ACM, 2010.

[64] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, "A tagless coherence directory," in *Proceedings of the 42Nd Annual IEEE/ACM Int'l Symp. on Microarchitecture*, MICRO 42, (New York, NY, USA), pp. 423–434, ACM, 2009.

[65] Y. Xu, Y. Du, Y. Zhang, and J. Yang, "A Composite and Scalable Cache Coherence Protocol for Large Scale CMPs," in *Proceedings of the Int'l Conference on Supercomputing*, ICS '11, (New York, NY, USA), pp. 285–294, ACM, 2011.

[66] J. Oh, A. Zajic, and M. Prvulovic, "Traffic steering between a low-latency unswitched TL ring and a high-throughput switched on-chip interconnect," in *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*, pp. 309–318, Sept 2013.

[67] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato, "Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks," in *Proceedings of the 38th Annual Int'l Symp. on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 93–104, ACM, 2011.

[68] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar, "A 5-GHz Mesh Interconnect for a Teraflops Processor," *IEEE Micro*, vol. 27, pp. 51–61, Sept. 2007.

[69] B. Ames, "Intel Tests Chip Design With 80-Core Processor." `http://www.pcworld.com/article/128924/article.html`, February 2007.

[70] P. Kundu and L.-S. Peh, "Guest Editors' Introduction: On-Chip Interconnects for Multicores," *Micro, IEEE*, vol. 27, pp. 3 –5, sept.-oct. 2007.

[71] Tilera Corporation, *Tile Processor User Architecture Manual*, November 2011.

[72] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, "On-Chip Interconnection Architecture of the Tile Processor," *IEEE Micro*, vol. 27, pp. 15–31, Sept. 2007.

[73] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel, "Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator," in *Proceedings of the 36th Annual Int'l Symp. on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 140–151, ACM, 2009.

[74] X. Wang, G. Gan, J. Manzano, D. Fan, and S. Guo, "A Quantitative Study of the On-Chip Network and Memory Hierarchy Design for Many-Core Processor," in *Parallel and Distributed Systems, 2008. ICPADS '08. 14th IEEE International Conference on*, pp. 689 –696, dec. 2008.

[75] G. Tan, D. Fan, J. Zhang, A. Russo, and G. R. Gao, "Experience on optimizing irregular computation for memory hierarchy in manycore architecture," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, (New York, NY, USA), pp. 279–280, ACM, 2008.

[76] D.-R. Fan, N. Yuan, J.-C. Zhang, Y.-B. Zhou, W. Lin, F.-L. Song, X.-C. Ye, H. Huang, L. Yu, G.-P. Long, H. Zhang, and L. Liu, "Godson-T: An Efficient Many-Core Architecture for Parallel Program Executions," *Journal of Computer Science and Technology*, vol. 24, pp. 1061–1073, 2009. 10.1007/s11390-009-9295-3.

[77] D. Donofrio, L. Oliker, J. Shalf, M. Wehner, C. Rowen, J. Krueger, S. Kamil, and M. Mohiyuddin, "Energy-Efficient Computing for Extreme-Scale Science," *Computer*, vol. 42, pp. 62 –71, nov. 2009.

[78] 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," *Computer*, vol. 35, pp. 50–58, 2002.

[79] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Comput. Archit. News*, vol. 33, pp. 92–99, November 2005.

[80] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, "SimFlex: a fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, pp. 31–34, March 2004.

[81] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[82] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, "Accuracy evaluation of GEM5 simulator system," in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, pp. 1 –7, july 2012.

[83] , "The edinburgh compute and data facility (ECDF).." `http://www.ecdf.ed.ac.uk`, 2015.

[84] X. Zhu, W. Qin, and S. Malik, "Modeling operation and microarchitecture concurrency for communication architectures with application to retargetable simulation," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 14, pp. 707–716, July 2006.

[85] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," *SIGARCH Comput. Archit. News*, vol. 25, pp. 13–25, June 1997.

[86] R. Zhong, Y. Zhu, W. Chen, M. Lin, and W.-F. Wong, "An Inter-Core Communication Enabled Multi-Core Simulator Based on SimpleScalar," *Advanced Information Networking and Applications Workshops, International Conference on*, vol. 1, pp. 758–763, 2007.

[87] C. Pinto, S. Raghav, A. Marongiu, M. Ruggiero, D. Atienza, and L. Benini, "GPGPU-Accelerated Parallel and Fast Simulation of Thousand-Core Platforms," in *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pp. 53–62, May 2011.

[88] "HSA Foundation." `http://www.hsafoundation.com`, 2015.

[89] G. Zheng, G. Kakulapati, and L. V. Kalé, "BigSim: A parallel simulator for performance prediction of extremely large parallel machines," *Parallel and Distributed Processing Symposium, International*, vol. 1, p. 78b, 2004.

[90] S. Kanaujia, I. E. Papazian, J. Chamberlain, and J. Baxter, "FastMP: A multi-core simulation methodology," in *Proceedings of the Workshop on Modeling, Benchmarking and Simulation (MoBS 2006)*, (Boston, Massachusetts), 2006.

[91] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood, "The wisconsin wind tunnel: virtual prototyping of parallel computers," in *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '93, (New York, NY, USA), pp. 48–60, ACM, 1993.

[92] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, M. D. Hill, D. A. Wood, S. Huss-Lederman, and J. R. Larus, "Wisconsin Wind Tunnel II: A fast, portable parallel architecture simulator," *IEEE Concurrency*, vol. 8, pp. 12–20, October 2000.

[93] P. Ren, M. Lis, M. H. Cho, K. S. Shim, C. Fletcher, O. Khan, N. Zheng, and S. Devadas, "HORNET: A Cycle-Level Multicore Simulator," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 31, pp. 890 –903, june 2012.

[94] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi, "ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-stage Design Space Exploration," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, (3001 Leuven, Belgium, Belgium), pp. 423–428, European Design and Automation Association, 2009.

[95] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan, "Temperature-aware Microarchitecture: Modeling and Implementation," *ACM Trans. Archit. Code Optim.*, vol. 1, pp. 94–125, Mar. 2004.

[96] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSS: A full system simulator for multicore x86 CPUs," in *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pp. 1050–1055, June 2011.

[97] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. of the 2005 USENIX Annual Technical Conference.*, ATEC '05, (Berkeley, CA, USA), pp. 41–41, USENIX Association, 2005.

[98] J. Chen, M. Annavaram, and M. Dubois, "SlackSim: a platform for parallel simulations of CMPs on CMPs," *SIGARCH Comput. Archit. News*, vol. 37, pp. 20–29, July 2009.

[99] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 1 –12, jan. 2010.

[100] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools

with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, (New York, NY, USA), pp. 190–200, ACM, 2005.

[101] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, nov 2011.

[102] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 475–486, ACM, 2013.

[103] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "COTSon: infrastructure for full system simulation," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 52–61, January 2009.

[104] R. Lantz, "Fast functional simulation with parallel Embra," in *Proceedings of the 4th Annual Workshop on Modeling, Benchmarking and Simulation*, 2008.

[105] R. Lantz, "Parallel SimOS: Scalability and performance for large system simulation," *www-cs.stanford.edu*, Jan 2007.

[106] K. Wang, Y. Zhang, H. Wang, and X. Shen, "Parallelization of IBM Mambo system simulator in functional modes," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 1, pp. 71–76, 2008.

[107] X. Sui, J. Wu, W. Yin, D. Zhou, and Z. Gong, "MALsim: A functional-level parallel simulation platform for CMPs," in *2nd International Conference on Computer Engineering and Technology (ICCET), 2010*, vol. 2, p. V2, IEEE, Jan 2010.

[108] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi, "ProtoFlex: Towards scalable, full-system multiprocessor simulations using FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, pp. 15:1–15:32, June 2009.

[109] T. Spink, H. Wagstaff, B. Franke, and N. Topham, "Efficient Code Generation in a Region-based Dynamic Binary Translator," in *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '14, (New York, NY, USA), pp. 3–12, ACM, 2014.

[110] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanović, "RAMP gold: an FPGA-based architecture simulator for multiprocessors," in *Proceedings of the 47th Design Automation Conference*, DAC '10, (New York, NY, USA), pp. 463–468, ACM, 2010.

[111] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic, "RAMP: Research Accelerator for Multiple Processors," *IEEE Micro*, vol. 27, pp. 46–57, March 2007.

[112] D. Chiou, D. Sunwoo, H. Angepat, J. Kim, N. Patil, W. Reinhart, and D. Johnson, "Parallelizing computer system simulators," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1 –5, april 2008.

[113] D. Chiou, H. Angepat, N. Patil, and D. Sunwoo, "Accurate Functional-First Multicore Simulators," *IEEE Comput. Archit. Lett.*, vol. 8, pp. 64–67, July 2009.

[114] P. Flake, S. Davidmann, and F. Schirrmeister, "System-level exploration tools for MPSoC designs," in *Proceedings of the 43rd annual Design Automation Conference*, DAC '06, (New York, NY, USA), pp. 286–287, ACM, 2006.

[115] F. Angiolini, J. Ceng, R. Leupers, F. Ferrari, C. Ferri, and L. Benini, "An integrated open framework for heterogeneous MPSoC design space exploration," in *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, DATE '06, (3001 Leuven, Belgium, Belgium), pp. 1145–1150, European Design and Automation Association, 2006.

[116] M. F. S. Oliveira, E. W. Brião, F. A. Nascimento, and F. R. Wagner, "Model driven engineering for MPSOC design space exploration," in *Proceedings of the 20th annual conference on Integrated circuits and systems design*, SBCCI '07, (New York, NY, USA), pp. 81–86, ACM, 2007.

[117] M. Gries, "Methods for evaluating and covering the design space during early design development," *Integration, the VLSI Journal*, vol. 38, no. 2, pp. 131 – 183, 2004.

[118] J. Li, X. Ma, K. Singh, M. Schulz, B. de Supinski, and S. McKee, "Machine learning based online performance prediction for runtime parallelization and task scheduling," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pp. 89 –100, april 2009.

[119] E. Ipek, B. de Supinski, M. Schulz, and S. McKee, "An approach to performance prediction for parallel applications," *Euro-Par 2005 Parallel Processing*, pp. 627–628, 2005.

[120] E. Ipek, S. McKee, K. Singh, R. Caruana, B. de Supinski, and M. Schulz, "Efficient architectural design space exploration via predictive modeling," *ACM Trans. Archit. Code Optim*, vol. 4, no. 4, pp. 1–34, 2008.

[121] O. Almer, M. Gould, B. Franke, and N. Topham, "Selecting the optimal system: automated design of application-specific systems-on-chip," in *Proceedings of the 4th International Workshop on Network on Chip Architectures*, NoCArc '11, (New York, NY, USA), pp. 43–50, ACM, 2011.

[122] Renesas Electronics, *SH-4 Software Manual, Renesas 32-Bit RISC Microcomputer SuperH RISC engine Family*, September 2006.

[123] ARM, *AMBA AXI Protocol Specification*, March 2004.

[124] The University of Edinburgh, "PASTA-2 Receives £1.2m Funding from EPSRC." `http://www.icsa.informatics.ed.ac.uk/compilers/news_20104010.html`, 2010.

[125] A. Hopper and J. Wheeler, "Binary routing networks," *Computers, IEEE Transactions on*, vol. C-28, pp. 699–703, Oct 1979.

[126] I. Böhm, B. Franke, and N. Topham, "Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator," in *2010 International Conference on Embedded Computer Systems (SAMOS)*, pp. 1–10, July 2010.

[127] Accellera, "TLM-2.0 Reference Manual." `http://www.accellera.org/downloads/standards/systemc`, July 2009.

[128] T. E. M. B. Consortium, "MultiBench 1.0 Multicore Benchmark Software," 02 February 2010.

[129] A. Mello, I. Maia, A. Greiner, and F. Pecheux, "Parallel simulation of SystemC TLM 2.0 compliant MPSoC on SMP workstations," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pp. 606–609, March 2010.

[130] M. Elver and V. Nagarajan, "TSO-CC: Consistency directed cache coherence for TSO," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pp. 165–176, Feb 2014.

[131] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 371–382, ACM, 2009.

[132] E. Blem, J. Menon, and K. Sankaralingam, "A detailed analysis of contemporary ARM and x86 architectures," tech. rep., University of Wisconsin, 2013.

[133] E. Blem, J. Menon, and K. Sankaralingam, "Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2013.

[134] Hruska, Joel, "The final ISA showdown: Is ARM, x86, or MIPS intrinsically more power efficient?." `http://www.extremetech.com/extreme/188396-the-final-isa-showdown-is-arm-x86-or-mips-intrinsically-more-power-efficient/2`, 2014.

[135] Synopsys, Inc., "ARCompact instruction set architecture." `http://www.synopsys.com`.

[136] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why On-chip Cache Coherence is Here to Stay," *Commun. ACM*, vol. 55, pp. 78–89, July 2012.

[137] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings of the 22nd annual Int'l Symp. on Computer Architecture*, ISCA '95, (New York, NY, USA), pp. 24–36, ACM, 1995.

[138] Y. Liang and T. Mitra, "Improved procedure placement for set associative caches," in *Proceedings of the 2010 International Conference on Compilers,*

Architectures and Synthesis for Embedded Systems, CASES '10, (New York, NY, USA), pp. 147–156, ACM, 2010.

[139] M. Bhadauria, V. M. Weaver, and S. A. McKee, "Understanding PARSEC Performance on Contemporary CMPs," in *Proceedings of the 2009 IEEE Int'l Symp. on Workload Characterization (IISWC)*, IISWC '09, (Washington, DC, USA), pp. 98–107, IEEE Computer Society, 2009.

[140] H. Sasaki, T. Tanimoto, K. Inoue, and H. Nakamura, "Scalability-based Manycore Partitioning," in *Proceedings of the 21st Int'l Conference on Parallel Architectures and Compilation Techniques*, PACT '12, (New York, NY, USA), pp. 107–116, ACM, 2012.

[141] I. Corporation, *Intel®64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z*, Feb 2014.

[142] ARM, *ARM11 MPCore Processor Technical Reference Manual Rev r1p0*. ARM, February 2008.

[143] K. Diefendorff, "Compaq Chooses SMT for Alpha," *Microprocessor Report*, vol. 13, Dec. 1999.

[144] D. Pasetto, M. Meneghin, H. Franke, F. Petrini, and J. Xenidis, "Performance evaluation of interthread communication mechanisms on multicore/multithreaded architectures," in *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, (New York, NY, USA), pp. 131–132, ACM, 2012.

[145] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy, "Supporting fine-grained synchronization on a simultaneous multithreading processor," in *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, HPCA '99, (Washington, DC, USA), pp. 54–, IEEE Computer Society, 1999.

[146] T. Li, A. R. Lebeck, and D. J. Sorin, "Spin detection hardware for improved management of multithreaded systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, pp. 508–521, June 2006.

[147] P. M. Wells, K. Chakraborty, and G. S. Sohi, "Hardware support for spin management in overcommitted virtual machines," in *Proceedings of the 15th In-*

ternational Conference on Parallel Architectures and Compilation Techniques, PACT '06, (New York, NY, USA), pp. 124–133, ACM, 2006.

[148] K. D. Kissell, "MIPS MT: A Multithreaded RISC Architecture for Embedded Real-time Processing," in *Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC'08, (Berlin, Heidelberg), pp. 9–21, Springer-Verlag, 2008.

[149] R Development Core Team, *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2011. ISBN 3-900051-07-0.