J Sign Process Syst (2015) 80:103–120
DOI 10.1007/s11265-014-0944-6

Exploiting Tightly-Coupled Cores

Daniel Bates · Alex Bradbury · Andreas Koltes · Robert Mullins

Received: 20 December 2013 / Revised: 31 July 2014 / Accepted: 7 August 2014 / Published online: 26 August 2014

Abstract The individual processors of a chip-multiprocessor traditionally have rigid boundaries. Inter-core communication is only possible via memory, and control over a core's resources is localised. The specialisation necessary to meet today's challenging energy targets is typically provided through the provision of a range of processor types and accelerators. An alternative approach is to permit specialisation by tailoring the way a large number of homogeneous cores are used. The approach here is to relax processor boundaries, create a richer mix of inter-core communication mechanisms and provide finer-grain control over, and access to, the resources of each core. We evaluate one such design, called Loki, that aims to support specialisation in software on a homogeneous many-core architecture. We focus on the design of a single 8-core tile, conceived as the building block for a larger many-core system. We explore the tile's ability to support a range of parallelisation opportunities and detail the control and communication mechanisms needed to exploit each core's resources in a flexible manner. Performance and a detailed breakdown of energy usage is provided for a range of benchmarks and configurations.

Keywords Computer architecture · Many-core · Parallelism · Homogeneous

1 Introduction

Current multi-core approaches provide a rigid target for the programmer and compiler. This inflexibility and the predetermined partitioning of resources complicates the writing of parallel programs. The hard boundaries given to cores also exposes the limitations described by Amdahl's law by forcing the mix of sequential and parallel capability to be fixed at design-time. Furthermore, computation and communication are often controlled by hardware mechanisms, making it difficult to streamline the implementation of a particular program to overcome increasingly severe power constraints. Perhaps surprisingly, while such concerns persist, the architecture of most multi-core chips diverge little from older multi-node machines, even though the design space on-chip is far less constrained. In this paper, we introduce the Loki architecture as a testbed for exploring solutions that address many of the upcoming problems faced by computer architects.

We explore a new approach to embrace the abundance of new parallel programming and compilation techniques and to achieve the necessary step-change in energy efficiency. By allowing greater control over the placement of data, placement of execution, and of how communication takes place, higher performance and more energy-efficient solutions can be built than are possible on a traditional multi-core architecture. We suggest the programmer and compiler specify an application-specific virtual architecture

This work was supported by EPSRC grant EP/G033110/1 and ERC grant 306386.

D. Bates (✉) · A. Bradbury · A. Koltes · R. Mullins
Computer Laboratory, University of Cambridge,
Cambridge CB3 0FD, UK
e-mail: Daniel.Bates@cl.cam.ac.uk

A. Bradbury
e-mail: Alex.Bradbury@cl.cam.ac.uk

A. Koltes
e-mail: Andreas.Koltes@cl.cam.ac.uk

R. Mullins
e-mail: Robert.Mullins@cl.cam.ac.uk

or *overlay* for their target application. This is a network of the best processors, helper engines, accelerators, memories and routers for that application. The overlay can take advantage of parallelism inherent in the application's structure, rather than requiring that the program be modified to suit the architecture, and it is possible to implement a range of overlays with different area-energy-performance tradeoffs. The ability to describe this overlay purely in software offers further advantages, as it becomes possible to dynamically adapt it in response to changing conditions at run-time. This saves power by minimising superfluous switching activity, for example by providing a direct low-cost communication path between certain components or by specialising the computation resources (and their control) to a particular task.

This approach requires an architecture that is able to provide a sea of resources that can be combined into the required overlay. We achieve this by allowing a large number of simple cores and memory blocks to communicate freely over a single on-chip network. This logical network is partitioned into multiple physical networks, each optimised to reduce costs for a particular communication pattern. In accordance with Pollack's rule, we reduce the resources devoted to each core, and use the saved area and energy to place many more cores on-chip, resulting in an increased overall potential for computation. The design allows cores and memories to be composed to form larger computation structures, and provides more direct access to on-chip resources, effectively exposing individual datapath components to others on the network. To achieve the desired level of flexibility while maximising energy efficiency, the design additionally supports bypassing of resources when they are not required.

The choice of a homogeneous design means Loki is also well placed to tackle emerging challenges as we move to future fabrication nodes. This decision makes many aspects of the design simpler, including fault tolerance, design and verification, optimisation and scaling. Loki's support for software specialisation narrows the gap between its homogeneous structure and an optimised heterogeneous architecture. We aim to provide flexibility without imposing the limitations of reconfigurable architectures, such as FPGAs and CGRAs, in terms of limited virtualisation capabilities, poor control-intensive code performance and rigid on-chip communication structures.

Much as an FPGA provides a substrate for logic-level emulation, Loki and similar architectures provide a flexible processing substrate for executing software efficiently. These arrays of processing elements may provide support for a broader range of applications, where individual cores may be programmed as traditional processors but also viewed as configurable circuit-level components which perform a single task. Loki differs from other polymorphic chip multi-processors in its finer granularity and its greater scope for flexibly using datapath resources. The flexibility of the sea of cores and memories can also be exploited at run-time rather than requiring that overlays are static during execution or requiring an explicit reconfiguration phase.

Loki's novelty lies in the breadth of virtual architectures which can be implemented efficiently and the speed at which they can be configured. This is achieved by exposing many hardware elements through the instruction set and performing all specialisation in software. This article is an extension of a previous publication [5]; its main contributions are:

- An overview of the Loki architecture; one instance of the class of communication-centric architectures we describe (Section 2);
- A framework for high-level energy modelling, and a detailed performance, energy and area characterisation for the Loki architecture (Section 3);
- A demonstration that tightly-coupled cores, through the provision of software-controlled interconnect, allow a broad range of parallelisation techniques (Section 4);
- Evidence to suggest that flexibility is necessary in the types of parallelism which can be exploited by hardware, with different parallel structures allowing different energy-performance tradeoffs (Section 4).

2 Loki Architecture

Loki is a homogeneous, tiled architecture, composed of cores and memories connected through an on-chip network (Figure 1). Some of the tiles contain a number of processor cores and level-1 cache banks, as shown, while others are made up of banks of a distributed level-2 cache. Each core has a relatively simple 32-bit scalar pipeline (Figure 2). A traditional RISC instruction set is augmented with the facility to provide most instructions with direct access to the on-chip network. The studies in this paper focus on a single tile of the Loki architecture.

2.1 Software Specialisation

Loki aims to permit a programmer to exploit a wide-range of execution patterns, mirroring the techniques used by many different architectures, e.g. SIMD, fine-grain dataflow, task-level pipelines, ILP, etc. Such patterns are exploited at run-time through software rather than explicitly writing to a configuration memory. The aim is to tailor the execution and communication patterns to each program or phase of

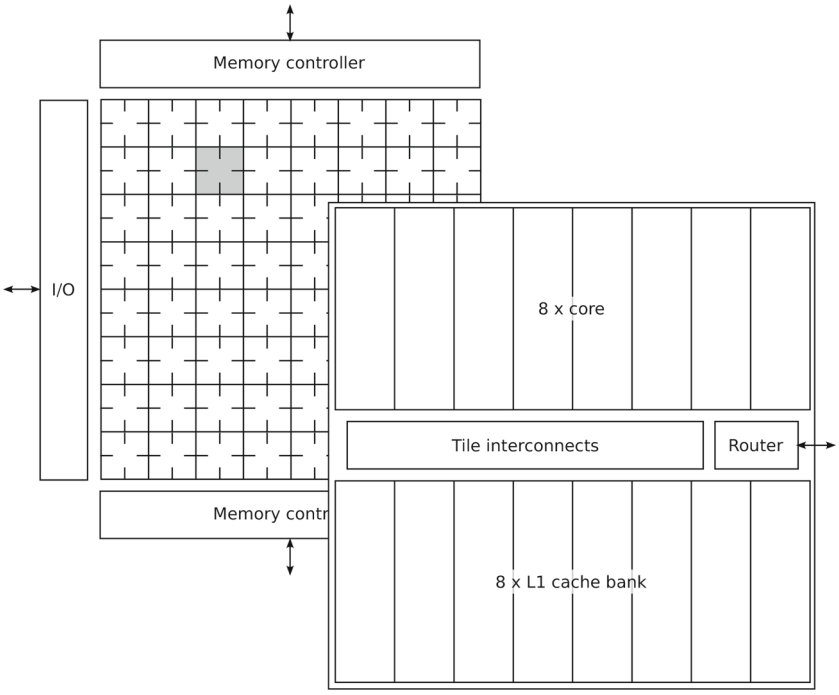


Figure 1 Loki's tiled architecture. *Left*: chip with one tile highlighted. *Right*: tile block diagram.

a program. Software management of each core's instruction and data stores is possible (though not compulsory), and network buffers are exposed to software through the instruction set.

2.2 Network-Centric Design

The network is central to the design and provides the basic mechanism by which resources can be accessed and composed in a low-cost fashion. The buffers that hold incoming data from the network are register mapped and the instruction set extended to allow instructions to place their results directly onto the network. The network is used to carry both instructions and data and allows arbitrary communication between both cores and memories.

A tile has a local network allowing communication between its constituent cores and memories. Each tile is also attached to a global chip-wide network, allowing access to more distant cores, L2 cache banks and main memory. The local network is implemented as a collection of networks each optimised for a particular communication pattern (Figure 3). Cores and memories communicate with each other over two fast crossbars (one in each direction) with half-cycle latencies to minimise memory latency. Each

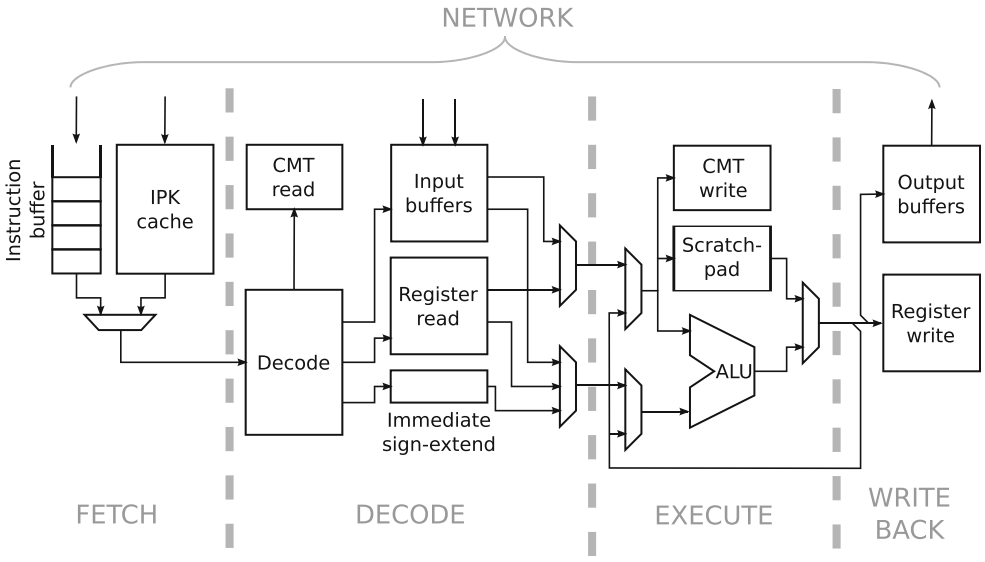
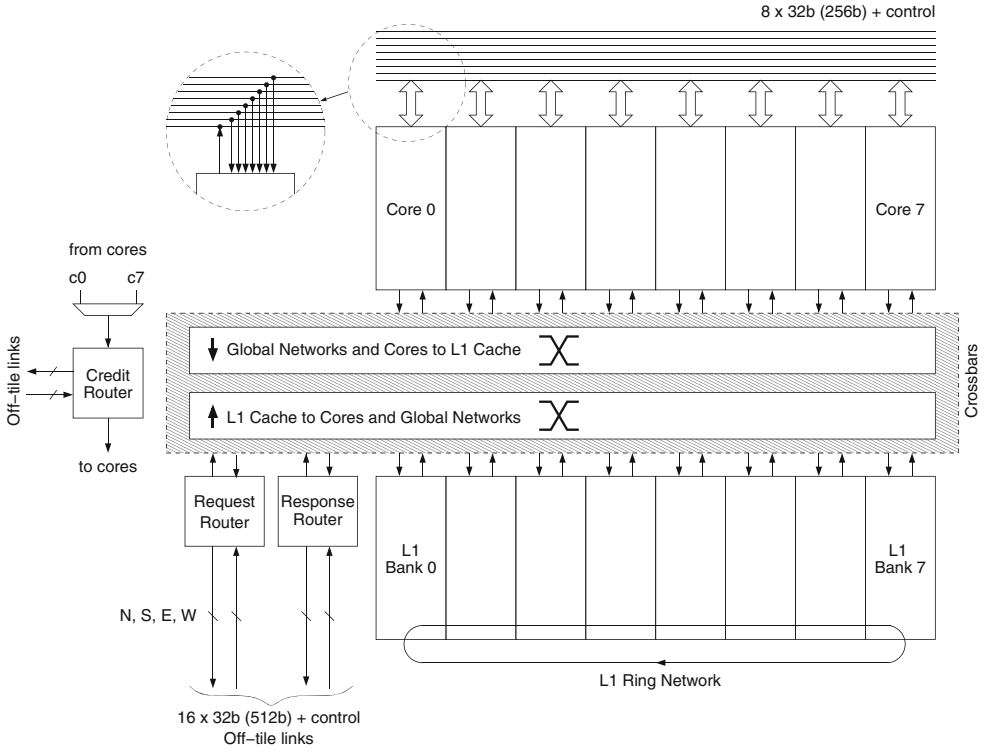


Figure 2 Loki core microarchitecture block diagram. IPK: Instruction Packet – atomic group of instructions similar to a basic block. CMT: Channel Map Table – mapping between logical and physical network addresses.

Figure 3 Loki subnetworks.



core also has a dedicated bus to which it can write to communicate with arbitrary subsets of other cores on the tile within one clock cycle. L1 cache banks are connected by a ring network to satisfy requests (e.g. for some instruction packets) that overflow into neighbouring cache banks.

The tiles are connected in a mesh toplogy by the global network. This consists of three physical subnetworks: a request network, a response network and a reply (or credit) network. Each network is constructed from simple wormhole routers with basic support to ensure equality-of-service. The possibility of deadlock is avoided through the provision of multiple networks and additional guarantees on the availability of buffer space. Buffer space guarantees are provided by preallocating buffer space for responses, providing dedicated buffer space for memory requests from each core at the memory controller and through the use of end-to-end flow control for all direct core-to-core communications. An additional virtual channel (or physical network) is required if L2 memory tiles are interleaved with compute tiles, rather than partitioning the two types of tile on either side of the chip.

Access to memory from each core is provided over the network in a decoupled fashion. This differs from the provision of a blocking memory access stage in a typical pipeline. A load instruction requests data from memory which is written into one of the core's input buffers. The pipeline will only stall if a subsequent instruction attempts to read this buffer when no data is present.

Instructions are grouped into atomic blocks called instruction packets (IPKs), which roughly correspond to basic blocks in the program. This approach is suited to the networked design: it allows a single memory request to result in a large transfer of instructions; and it makes prefetching simple, making it easier to hide memory latencies. Full compiler-based management of cache contents is also possible. Instruction packets may be sent directly between cores, enabling the execution of short instruction sequences at remote cores (e.g. to access or store data in a remote tile). This is achieved either by requesting that memory sends a packet to a remote core or by sending an inlined instruction sequence to a remote core's instruction buffer. When an instruction packet is fetched, it does not execute immediately, as in the case of a traditional branch instruction, but is queued up to execute when the current packet has completed. This behaviour is similar to the atomic instruction blocks used by the SCALE architecture [18]. Loki also supports predicated execution to reduce the amount of control flow, increasing the average size of instruction packets.

Channels are a fundamental design feature which allow components to communicate. Each core and memory has associated with it a number of channel-ends, to which it can read and write. Each channel connects a single source to one or more destinations. Channels are typically allocated at compile-time, though it is also safe to perform

run-time allocation if it is known that messages from different sources will not collide. Attempting to read from an empty input buffer, or channel-end, will cause the pipeline to stall. Writes also stall if the network is blocked or, in the case of longer distance communications, if no buffer space is available at the receiving core (end-to-end flow control is used). A layer of indirection is provided when writing to a channel in the form of a channel map table (CMT). This small table, present in every core, holds the full network addresses that data will be sent to, avoiding the need to encode these at the instruction level. The channel map table is also used to specify multicast groups and enable communications (or entire threads) to be remapped transparently at run-time.

Figure 4 lists a fragment of the kernel of the CRC benchmark. Before the kernel begins, *setchmapi* associates the logical network address `1` with the physical network address held in `r11`. The function itself begins with an instruction *fetch*: in this case, the next instruction packet to be executed is known immediately, and is fetched in advance. The load instruction (*ldw*) demonstrates the ability to send data onto the network with the `->` notation; most instructions are able to store their results locally, send them over the network, or both. The load works by sending a memory address over the network to the appropriate cache bank. The cache bank also has a channel map table which has been configured to send data back to channel 2 of the core. This data is used in the final instruction: registers 2-7 are mapped to the input buffers. The `.eop` marker denotes the end of the instruction packet and triggers the start of execution of the packet fetched previously.

2.3 Instruction and Data Supply

Instruction packets can be stored in each core's 64-word level-0 (L0) instruction packet cache to take maximum advantage of any available locality. Effective use of such L0 instruction caches has the potential to significantly reduce power consumption [17, 24]. The cache is fully associative and has a FIFO replacement policy to minimise the number of conflict misses and maximise utilisation of such a small store. If the position of an instruction in the cache is known statically any tag checks can be bypassed, saving energy.

Each core also has a 16-word instruction buffer. The buffer is used for instructions which will only need to be read once and for specialised code sequences which fit in the smaller store. This includes simple tasks sent between cores, but also includes code regions for which the cache will perform poorly, allowing the relatively expensive cache to be bypassed and reducing instruction supply energy. The buffer has priority over the cache: if there are pending packets in both structures, the one from the buffer is selected. Once an instruction packet from either source begins execution, it continues to completion.

A 256-word compiler-managed scratchpad is provided in each core to reduce the cost of accessing small tables of data, constant values, and sometimes sections of the stack. The scratchpad has the advantage that when a table is stored, element x of the table can often be stored at index x of the scratchpad, eliminating the need to generate a memory address.

Figure 4 CRC code example showing features of Loki's instruction set.

```
uint32_t updateCRC32(uint8_t ch,
                     uint32_t crc)
{
  return crc_32_tab[(crc ^ ch) & 0xff] ^
         (crc >> 8);
}
```

(a) C code

```
setchmapi 1, r11                # set up output channel 1
[...]
fetch     r10                   # pre-fetch next packet
xor       r11, r13, r14         # r11 = arg1 ^ arg2
lli       r12, %lo(crc_32_tab)  # lower 16 bits of label
lui       r12, %hi(crc_32_tab)  # upper 16 bits of label
andi      r11, r11, 255         # r11 = r11 & 255
slli      r11, r11, 2           # r11 = r11 << 2
addu      r11, r12, r11         # r11 = r12 + r11
ldw       0(r11) -> 1           # request data from memory
srli      r12, r14, 8           # r12 = r14 >> 8
xor.eop   r11, r2, r12          # use loaded data (r2)
```

(b) Loki assembly code

2.4 Memory System

Each tile holds eight 8kB memory banks which make up the unified L1 cache. To increase uniformity and flexibility, memory banks are also accessed over the network. This allows cores to masquerade as memories, e.g. in order to apply a transformation to memory addresses before accessing the banks themselves. It also makes the memory banks easily accessible to multiple cores, reducing the need for a hardware coherence mechanism. In order to reduce the impact of the network latency when accessing memory, arbitration is done in parallel with computation or memory access – the total time required to access what is effectively a 64kB banked L1 cache is two clock cycles in a zero-load system.

The L2 memory system is left undefined for this work as it is outside of the local tile. We are currently experimenting with a configurable L2 memory system that would allow the L2 cache memory to be used in a number of different ways. Loki does not currently support hardware cache coherence between tiles; we are exploring various low-complexity approaches to providing coherence when necessary.

3 Methodology

3.1 Performance Modelling

The architecture is modelled in SystemC. Together with performance data, fine-grain event counts are collected in order to estimate energy consumption. Simulation is cycle-accurate apart from the modelling of system calls, which complete instantaneously. For this reason, we lightly patch some benchmarks to remove system calls from inner loops, to reduce their impact on performance results.

The L2 cache is not fully modelled: it has a latency of ten cycles (beyond the L1), consumes no energy, and is large enough to hold all data required to execute a benchmark. The impact of this on our current compute intensive benchmark suite is minimal.

3.2 Benchmarks

Our experiments are performed using the MiBench benchmark suite [9]. We use only integer benchmarks, since Loki doesn't yet have hardware floating point support, and we use only those benchmarks which compile (some require libraries which are not yet supported on Loki). We simulate ten benchmarks in total covering all six of the MiBench categories: automotive, consumer, network, office, security and telecom.

All benchmarks are compiled using the settings suggested by the MiBench makefiles and are executed using the "small" inputs. We execute the benchmarks with the aid of the Newlib [13] C standard library implementation.

We use a custom LLVM-based [19] compiler. Since the compiler is not yet able to perform some optimisations, we hand-modify the most frequently executed regions of each benchmark. The modifications are expected to be within reach of a standard optimising compiler, and include simple optimisations such as removal of no-ops and filling branch- and load-delay slots. Parallelisation of benchmarks is also performed by hand at the source code level.

3.3 Energy Modelling

We describe all of the major datapath main components in SystemVerilog and implement them using the Synopsys Design Compiler and IC Compiler tools. Parasitics are extracted using StarRC and power is measured on a cycle-by-cycle basis using Primetime. Simulation event logs are then combined with energy consumption data in order to form an energy model using a multiple regression analysis for each component. Events of interest include the types of operation performed and number of bits toggled. Power is estimated assuming perfect clock-gating at the datapath component level. Energy models for interconnects are extracted in a similar way for fast, slow, well spaced and congested scenarios. We use Orion 2.0 [14] to model the high-level clock tree and validate it against a 1-bit bus of comparable length. We use a commercial memory compiler to obtain energy models for each of the SRAMs.

All results are obtained by targeting a commercial low-power 40nm process. In particular, we use cells from a general-purpose nominal-V_t library. Leakage is subsequently low and is not reported here. Timing is closed using a multi-corner PVT analysis where 0.99V and -40°Cis usually the worst-case corner. Energy results are reported for the typical case (1.1V, 25°C). We target a 435MHz clock rate due to timing constraints imposed by the instruction packet cache, register file, and L1 cache banks. The design is conservatively margined at the WC corner including foundry recommendations for on-chip variation and clock jitter. Our clock period is ∼42 FO4 delays: within the typical range of 40-60 FO4 delays for modern system-on-chip designs. We note that synthesising and modelling each datapath component separately will likely overestimate costs slightly due to the lack of cross-boundary optimisation.

The floorplan of a single tile is shown in Figure 5. A tile size of 1mm^2 permits 8 cores and 8×8kB memory banks, with a crossbar latency of half a clock cycle and a multicast latency of one cycle. A larger tile would increase the latency and energy costs of communication: adding an extra cycle of L1 cache latency reduces performance by

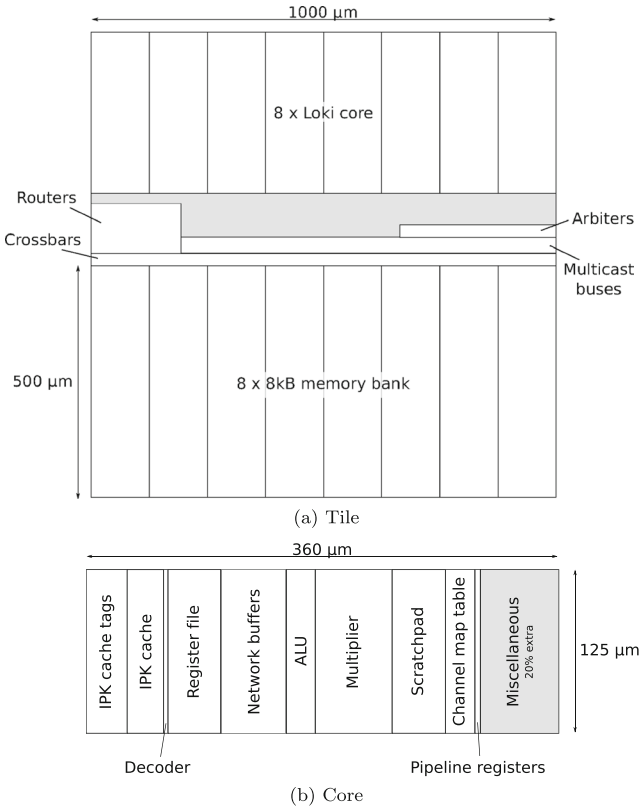


Figure 5 Floorplans for tile and core after all major subcomponents have been placed and routed. The tile occupies an area of 1mm×1mm and each core occupies 360µm × 125µm.

an average of 15%. A smaller tile would reduce the gains from coupling cores, as more communication would involve traversing a higher level of the network hierarchy. It is interesting that the network structures consume such a small area – this highlights the opportunities for dense interconnects and a rich variety of communication patterns. Each core was configured as follows after a simple exploration of the design space: 64-word instruction cache (with 16 cache tags); 32-word register-file; 7 network buffers of 4 entries each; 256-word scratchpad memory; 16-entry channel map table.

4 Evaluation

In this section we explore some of the many parallel execution patterns possible when fast and efficient inter-core communication is available. Mapping code across multiple cores can be used to increase both performance and energy efficiency. Three case studies are performed into different types of parallelism, using subsets of the benchmarks which are able to make use of each. Small studies are performed to identify the effects of additional hardware changes which could further improve the profitability of particular execution patterns.

4.1 Baseline

Energy consumption for each benchmark running on a single Loki core is shown in Figure 6 – data supply consists of register and scratchpad accesses, and the network interface consists of the channel map table and network buffers. Energy per operation varies between 10.2pJ and 20.6pJ, and is usually dominated by the supply of instructions from the cache hierarchy. In general, the benchmarks with the highest energy consumption per operation are those for which the L0 instruction cache performs poorly: *adpcm* (both compression and decom-

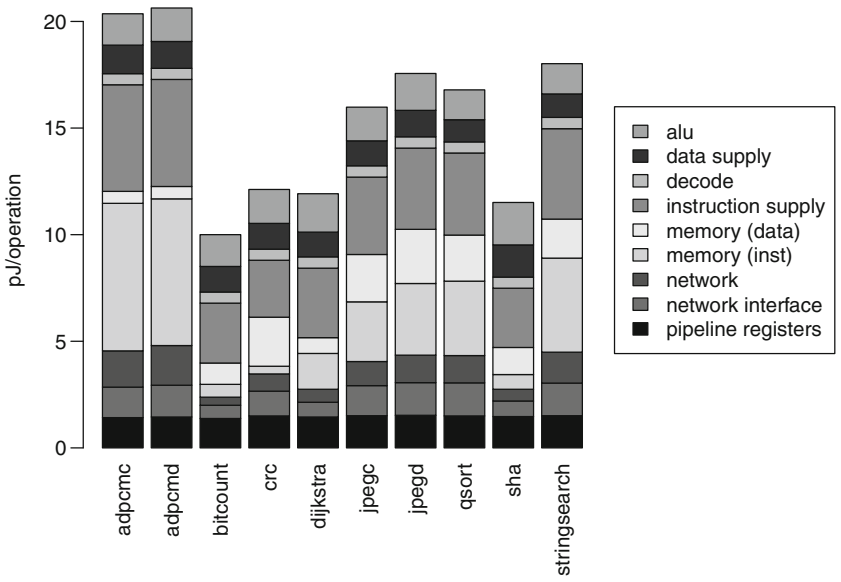


Figure 6 MiBench baseline energy distribution.

pression) consists mainly of a single loop which is too large to fit in the local instruction store, and *jpeg*, *qsort* and *stringsearch* contain extensive control-intensive code sections.

An ARM1176JZF-S processor in the same process consumes approximately 140pJ/operation (scaled from published data at 65nm [3] and confirmed through measurement) and consumes an area of approximately 1mm^2 with 32kB cache and a double-precision floating point unit. Dynamic instruction counts are 1.4-2.2× higher on Loki than the ARM processor at present. Overall execution on a single Loki core is typically 1-1.8× slower than the ARM core clocked at the same frequency. In most cases, Loki is able to close the performance gap when exploiting additional cores. The large difference in energy per operation suggests that it is possible to execute many Loki instructions in place of each ARM instruction to improve performance, while still consuming relatively little power.

4.2 Data-Level Parallelism (DLP)

When all iterations of a loop are independent (DOALL), executing them in parallel is trivial; the iterations can be sliced in whichever way is most convenient, and distributed across the cores.

When there are fixed cross-iteration dependencies (DOACROSS), it is necessary to set up communication channels before the loop begins, and modify the loop body to use the network when appropriate. On Loki, this can usually be done with zero performance overhead, as reading from the network replaces a register read, and sending onto the network is an optional feature of most instructions. The exception is that data must be copied into a register if it is needed multiple times since reads from network buffers are destructive. Also required are an initialisation phase to send the initial live-ins, and a tidying phase where any superfluous values are drained after the loop completes.

A number of loops exhibiting data-level parallelism were selected from the benchmarks. *adpcmc* contains DOACROSS parallelism, and all others are DOALL. Figure 7 shows how performance and energy scale as the number of cores used increases. The loops display a wide range of behaviours: some, such as *stringsearch* scale well, achieving a 5.4× speedup on 8 cores, and others such as *jpeg_dct* do not scale well because there are too few loop iterations for the execution pattern to be worthwhile. *adpcmc* converges on a speedup of approximately 2 when it uses 3 cores; this is limited by the dependencies between iterations and is not helped by the addition of further cores. For many of the benchmarks, energy remains roughly constant as more cores are used. This is because the same work is being done, but spread across more cores. The height of the line on the energy graph represents the overhead of the execution pattern: *bitcount_inner* has very tight loops, so the overhead is proportionally higher. For *jpeg_dct* and *jpeg_huff*, energy increases because there are not enough loop iterations to overcome the overheads of filling multiple L0 caches.

When mapping data-level parallelism across multiple cores, much work is duplicated. This includes repeated computation or access of data, and repeated fetching of identical instructions. We explore using one core as a *helper core* to provide common data required by all other cores. This reduces the work done by the data-parallel cores and contention at L1 banks, at the cost of reducing the number of cores processing the input data by one. This process of extracting redundant work is known as scalarisation [20].

The impact of such helper cores is shown in Figure 8. *dijkstra* and *stringsearch* are excluded as they are too control-intensive to benefit from a helper core. *adpcmc* is excluded because it makes use of DOACROSS parallelism, so the cores require more decoupling than the helper core allows. In most cases, energy consumption decreases from the plain DLP implementations because less work is being done in total. For *bitcount_inner*, *bitcount_outer* and *jpeg_color*, total energy consumption reduces as the number of cores increases because the helper core is able to provide data to more cores at once, so needs to do so fewer times. The performance impact depends on the amount of work which can be offloaded onto the helper core and the number of cores being used, and ranges from a 20% decline for *jpeg_dct* to a 16% improvement for *jpeg_color*. Energy consumption for 8 cores is an average of 11% lower than without the helper core.

In practice, the helper core could take a variety of forms, i.e. it could itself be a virtual processor composed of multiple cores to take advantage of further parallelism. Alternatively, the helper core could be used to allocate work to the other cores, forming a *worker farm*. A worker farm allows load balancing between cores, and is most useful when there are many independent tasks to be performed, with a high variance in execution time. Instead of being allocated a static subset of tasks, worker cores request new tasks from a master core when necessary. This allows cores which complete their work quicker to continue being productive, while slower cores do not hold up the others.

Loki's worker farm implementation therefore offers a tradeoff: plain DLP offers higher potential throughput by using more cores, but is vulnerable to variance in task length; worker farms spend a core to offer better load balancing. This is a similar tradeoff to that of scalarisation, where one core is removed from the group to fetch data and perform computations common to the rest of the cores. With the current implementation, a worker farm is limited to five workers, as each needs a separate communication channel

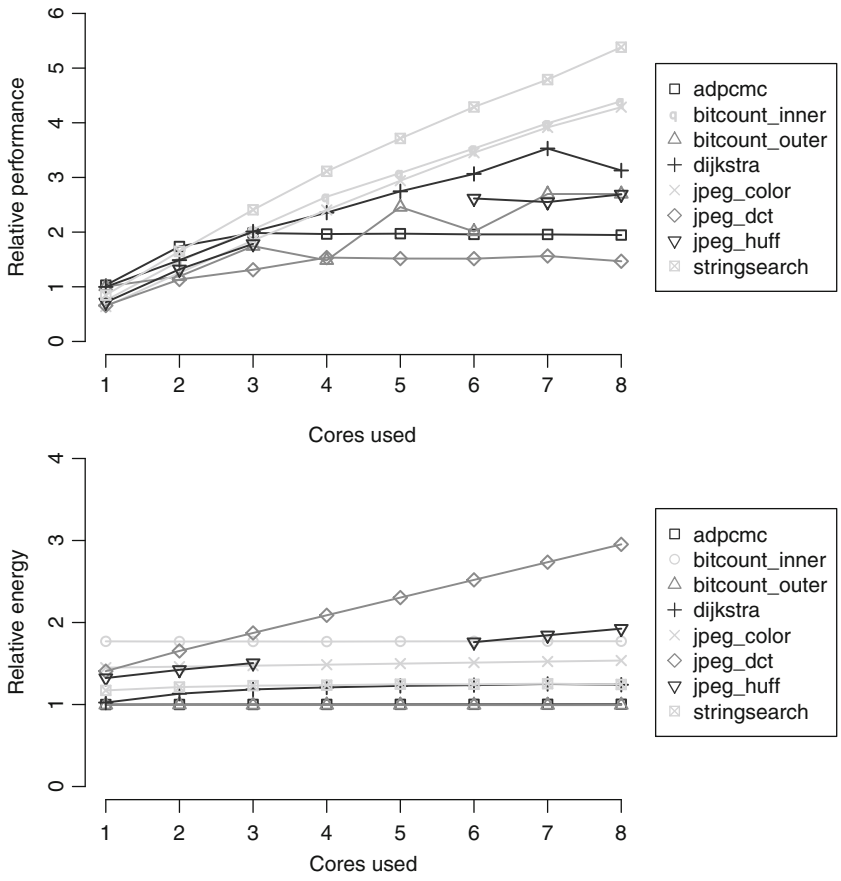


Figure 7 Performance and energy consumption as the number of data-parallel cores changes, relative to the baseline sequential implementation.

with the master core; this could be improved by using a hierarchy, or by allowing worker cores to share channels.

Energy for a worker farm is almost always going to be higher than for normal DLP; all of the same work is being performed, plus the overheads of the master core issuing tasks. This technique is targeted at loops with variable iteration execution times, and may be able to improve performance in these cases.

Figure 9 shows how the largest worker farm (5 workers and 1 master) compares with a DLP group of the same size, and with the largest DLP group tested. Kernels which contain no control flow are excluded as these will have very little variance in execution time, and so will not be helped by the worker farm execution pattern.

bitcount_inner and *stringsearch* show no improvement in performance when moving from DLP to a worker farm. Their loops are tight, so the overheads of communicating with the master core are not worthwhile. Performance losses are less than the $\frac{1}{6}$ which might be expected by removing one of the cores from the data-parallel computation, indicating that load balancing helps slightly. *bitcount_outer* makes more effective use of its six cores than the plain DLP implementation, but is not able to match the total throughput of the 8 core version. *dijkstra* shows a large performance improvement of 30% over 6-core DLP, and also outperforms the 8 core implementation comfortably. This indicates that there is a high variability in the execution times of *dijkstra*'s loop iterations, and that in some cases, intelligent management of computation can result in lower execution times, even when far less execution resources are available.

We also perform a limit study on the possibility of each instruction being cached by only a single core, and distributed to all others when necessary (Figure 10). Instructions are distributed before being decoded: Loki's decode logic is very simple, and existing buses can be used, rather than requiring a wider bus for decoded instructions. In the limit case (*Lower bound*), this will cut instruction supply costs (including memory accesses and network activity) by the number of cores. More realistic implementations are also presented: *Multicast direct* includes the cost of communicating the instructions directly to other cores' pipeline registers, and *Multicast to buffers* uses the existing core-to-core network to send instructions to cores' instruction buffers.

With no duplicate instructions in the cores' L0 caches, the L0 cache capacity of the group scales up by the number of cores. Access costs remain constant, however, since only a single cache is accessed at a time. *Larger cache* shows the energy impact of L0 caches which are 8 times larger

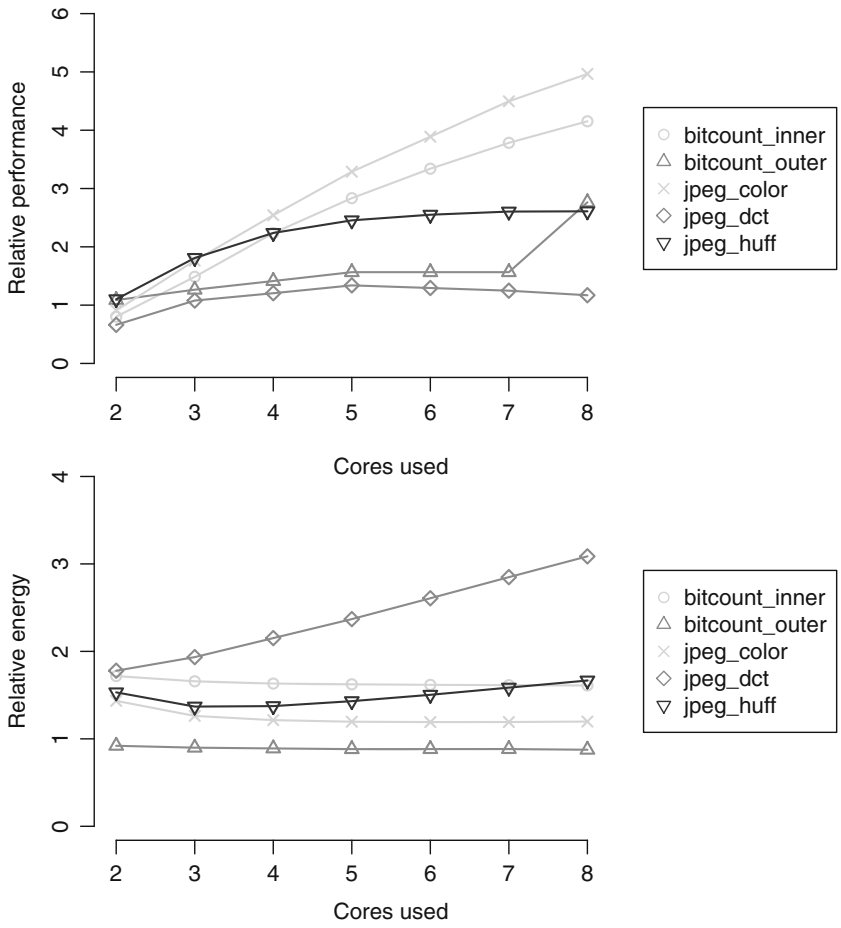


Figure 8 Performance and energy consumption as the width of the number of data-parallel changes, when making use of a helper core, relative to the baseline sequential implementation.

but have the same access costs. Techniques for switching between different cores' instruction caches have been demonstrated previously by the Elm architecture [4]. The extra cache capacity improves performance by an average of 14% for 8 cores.

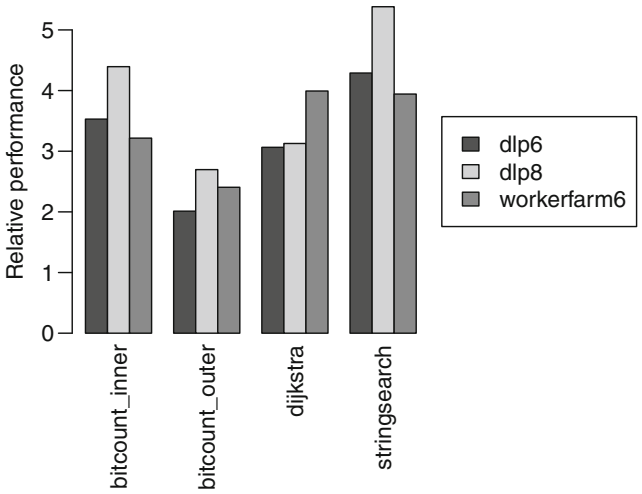


Figure 9 Performance comparison between worker farm and DLP for a selection of benchmark kernels. All figures are relative to the single core base case.

The technique is only suitable for DOALL parallelism, since the cores all execute the same instruction at (roughly) the same time. We assume that it is possible for data to be arranged in memory such that the effects of additional contention at the L1 banks are negligible.

Modern embedded processors often have SIMD extensions to their instruction sets to improve performance and reduce power consumption. We believe that Loki's flexibility allows us to increase coverage and accelerate a higher fraction of code. In addition, this optimisation can be applied in combination with scalarisation to further reduce energy consumption.

4.3 Dataflow

Dataflow is an execution paradigm where a change in the value of a variable automatically forces recomputation of any variables which depend on it.

Dataflow can be implemented on Loki by placing a small number of instructions on each core and setting up

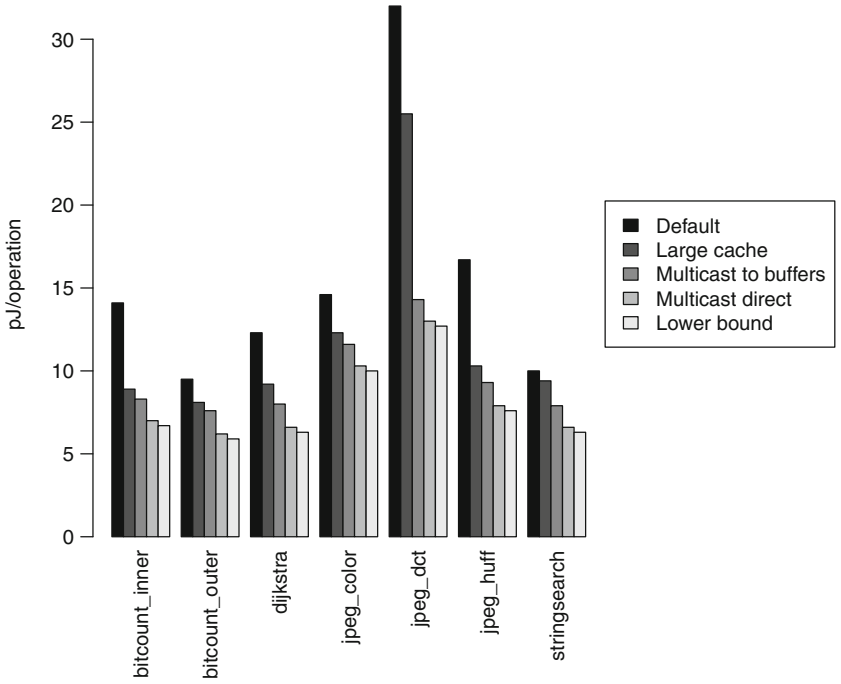


Figure 10 Energy consumption of various instruction sharing strategies. Results are for 8 cores. *Default*: DLP with no instruction sharing; *Large cache*: each core has 8× the L0 cache capacity; *Multicast to buffers*: instruction is read from one L0 cache and distributed to instruction buffer of all other cores; *Multicast direct*: instruction is read from one L0 cache and distributed directly to decode stage of all other cores; *Lower bound*: instruction distribution is instant and consumes no energy.

the required communication paths between cores to satisfy their data dependencies. The instructions on each core are executed repeatedly until computation has finished. We call these *persistent* instruction packets, and they are executed by using a special version of the *fetch* instruction which specifies that the packet should execute repeatedly until a *next instruction packet* command is received or a new packet is fetched. Loki's blocking network accesses mean that cores wait to receive new data before processing it. One of the advantages of the dataflow execution pattern is that it reduces switching activity in each pipeline. If the persistent instruction packet contains a single instruction, it can remain in the execute stage and much of the pipeline can be power gated after the first access: the entire fetch pipeline stage (including pipeline register); decoder; channel map table; and register file (if nothing is written to it).

Although much of the pipeline is superfluous when one instruction is executed repeatedly, network buffers, arbiters and interconnect see increased activity. Dataflow execution is only beneficial if these costs are outweighed by the savings in reduced pipeline activity.

Coarse-grained reconfigurable architectures (CGRAs) are composed of a mesh of functional units and are designed to execute dataflow graphs with low overhead. Loki can be seen as similar to a CGRA, but with an entire processor instead of a simple functional unit. This increases computation overheads, but allows better performance in other cases, such as control-intensive code. It is possible to map multiple instructions to a single Loki core in cases where overheads of pure dataflow are too high.

Figures 11 and 12 show how behaviour changes for two benchmarks with tight loops which can make use of the dataflow execution pattern. For each benchmark, a baseline running on a single core is compared against a version where instructions are spread across as many cores as possible to mimic traditional dataflow (*spread*), and a version where all instructions on the critical path are placed on a single core (*perf*).

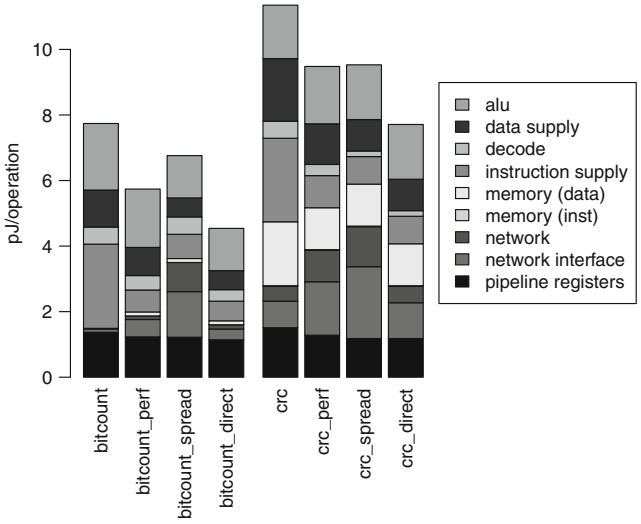


Figure 11 Energy distribution when using dataflow execution pattern.

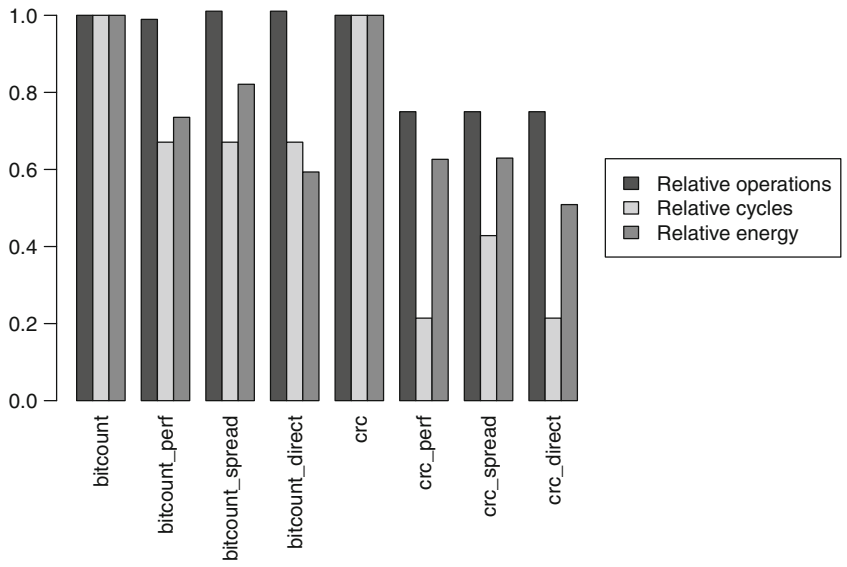


Figure 12 Relative behaviour when using dataflow execution pattern.

Figure 11 shows that energy spent on instruction and data supply decreases as the application is spread across more cores. This is because the average number of instructions on each core decreases, and it is possible to fit them into the more-efficient instruction buffer and bypass the L0 cache. Data supply energy is reduced due to fewer register accesses, but is replaced by increased network costs. Components such as the decoder and pipeline registers also show reduced activity.

Figure 12 shows that execution time and performance both improve over the baseline in all cases. *crc* sees a reduction in the number of operations due to the increased number of available registers. Performance does not improve when the application is spread across more cores because network latency is introduced to the critical path, slowing execution. Energy consumption doesn't see any improvement in these cases either. It was found that keeping a value in a local register file was 3.7pJ cheaper than sending the value to another core (assuming 50% of bits toggle). This is greater than the 2.7pJ saved when a core repeatedly executes a single instruction and is able to bypass many components in the pipeline.

Both latency and energy consumption can be improved by taking inspiration from CGRAs and providing direct links between functional units of neighbouring cores. This would bypass much of the network, and reduce latency to zero cycles, at a cost of larger multiplexers at ALU inputs. Since each core can consume two inputs and produce one output but has only two neighbours, the worst case is that two-thirds of dataflow communication can use these direct links. In practice, the fraction is often much higher because of operations with fewer inputs or outputs and instructions which use multicast instead. 75% of *bitcount*'s communication was between neighbouring cores, and 88% for *crc*.

The results of using this technique are shown in the *direct* entries in Figures 11 and 12. Network latency no longer adds to the critical path, so performance matches the *perf* case, but more cores are able to enter a low-energy state. *bitcount*'s energy reduces by 28% compared with the *spread* case to 4.5pJ/operation, and *crc*'s energy reduces by 19% to 7.7pJ/operation.

4.4 Pipeline-level Parallelism

Pipeline (or streaming) parallelism involves each core independently processing data, and passing the result onto the next core. This pattern has similarities to dataflow, but is coarser-grained, generally has a linear communication structure, and has a greater focus on data locality. Locality is improved by having each core working on a small section of the program and usually on a subsection of the input data. At the same time, parallelism is exploited by executing multiple pipeline stages simultaneously.

This can be implemented on traditional multi-core architectures, but we have more flexibility on Loki: each pipeline stage can be made parallel (useful for eliminating bottlenecks) and cheaper communication allows finer-grained stages. We also explore the use of pipelining for reasons other than improving performance: energy consumption can be reduced by making use of the increased cache and register capacity of multiple cores.

Each pipeline stage can be mapped to a virtual processor on the Loki fabric. The virtual processor can be a single core, or it could be a group of cores, specialised for the particular workload. The virtual processor can exploit any

form of parallelism, or could be optimised to reduce energy consumption (or both).

Pipeline parallelism was manually extracted from applicable MiBench applications by creating a function for each pipeline stage whose result was the input for the next stage. Figure 13 presents the performance and energy impact of this transformation. For *stringsearch*, performance improved by 4.2× with 6 cores, and for *jpeg_color*, performance improved by 1.8× with 3 cores. In both cases, energy consumption rose at first, due to the overheads of the wrapper function used to implement pipelining, but then fell as cores' tasks became small enough to fit in the L0 cache. We expect that these overheads can be reduced with compiler optimisation, improving the profitability of this execution pattern in the process.

We further explored the effects of pipelining for improved cache behaviour with the *adpcmc* benchmark; its main loop body does not have an obvious point at which it can be split, and there are dependencies between loop iterations which prevent traditional pipeline parallelism. The loop body was naively split at the basic block boundary closest to the halfway point such that each section fit in an L0 cache, and register contents were communicated across the network as necessary. This transformation effectively creates a *virtual processor* which is tailored to the application by providing sufficient instruction cache space. A positive side effect is that the number of registers and functional units also increase, allowing for parallelism and reduced register pressure. After applying the basic optimisations described in Section 3.2, performance improved by 153% and energy reduced by 69% to 9.6pJ/operation. These superlinear improvements were helped by improved caching, ILP extraction, and a 15% reduction in instruction count due to the extra registers. Mapping the code across two cores outperforms a single core with twice as much cache by 2× and improves energy consumption by 20%.

4.5 Summary

We have shown that it is possible to use tightly-coupled cores to profitably exploit multiple forms of parallelism: DLP, dataflow and task-level pipelines. This allows a broader coverage of parallelism, as each application can only usefully be parallelised using a subset of execution patterns. We also suggest small modifications to the hardware, such as allowing more direct communication between neighbouring cores, which improve performance and energy consumption further. SIMD execution with instruction sharing achieves an average of 3.6× speedup with 8 cores with only 2% more energy consumed. *bitcount_inner* sees a 6.4× speedup and *bitcount_outer* sees a 20% energy reduction over the single core baseline. Dataflow execution was able to improve performance of *crc* by 4.7× using 5 cores, with a 35% drop in energy consumption. Task-level pipelining

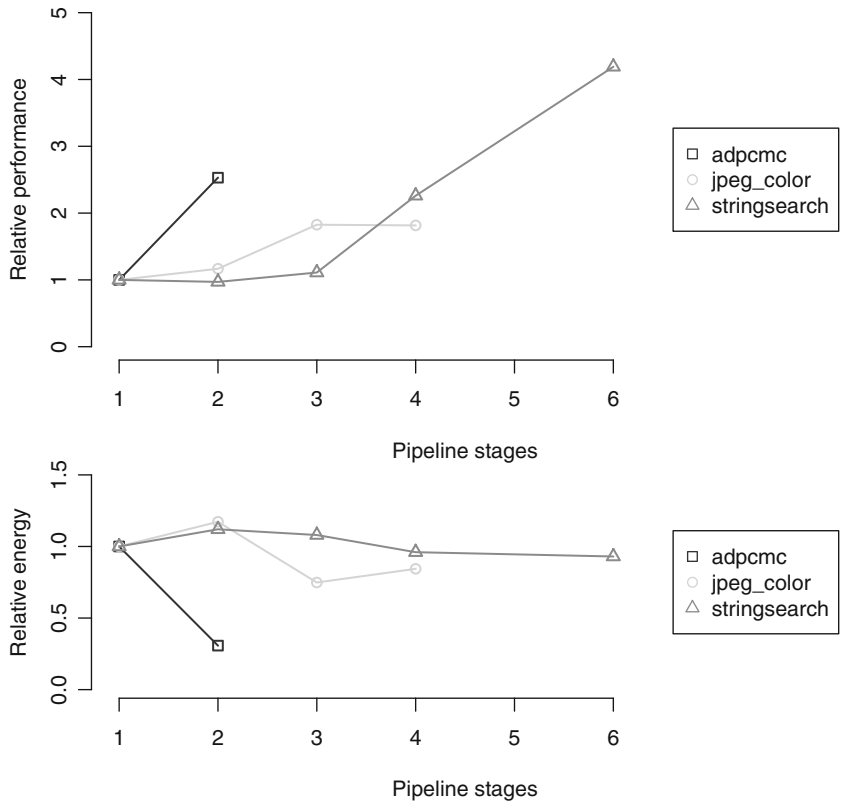


Figure 13 Relative performance and energy consumption when using software pipelines of different lengths, relative to the baseline sequential implementation.

allows core resources to be used more efficiently, resulting in a 2.5× speedup and 70% energy reduction with two cores for the *adpcmc* benchmark – far better than when exploiting DOACROSS parallelism in the same benchmark.

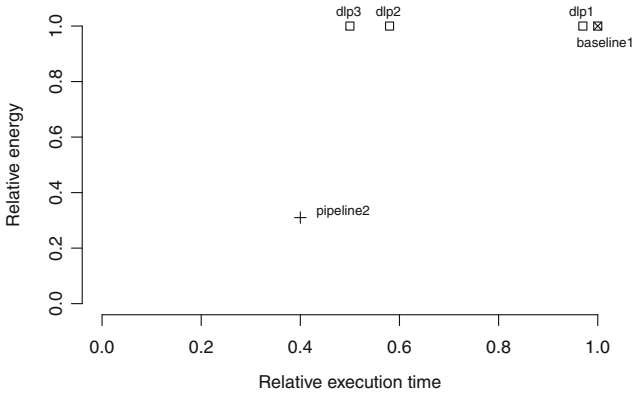
Also possible, though beyond the scope of this paper, is the ability to exploit instruction-level parallelism across multiple cores. This can be performed in a VLIW-like way, using the low-latency network for data forwarding, or by assigning decoupled instruction strands to each core.

The ability to use multiple cores to increase the resources available to an application suggests that it may be sensible to deliberately under-provision each core, with the expectation that the appropriate number will be grouped together for the task at hand. This would mean lower-power building blocks for virtual architectures, and the ability to provide resources at a finer granularity.

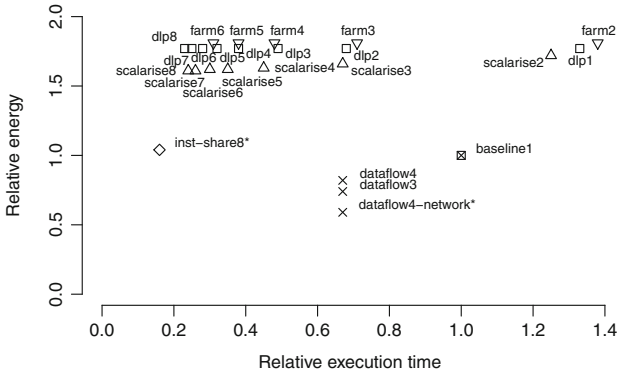
Figure 14 presents a comparison of execution patterns for each of the nine benchmark kernels used to demonstrate execution patterns in this paper. It can be seen that there is no execution pattern which is always the best: different patterns are useful in different situations. Indeed, there is often a selection of configurations for a particular benchmark which lie on the Pareto front, allowing for different energy-performance tradeoffs. This is the case even with optimistic limit study data included (marked with asterisks). Interestingly, the Pareto curve for a particular benchmark often contains configurations from multiple different execution patterns. This is because different execution patterns scale differently: task-level pipelines improve cache performance, but only up to the point where all code is held locally, while data-level parallelism's performance potential is limited instead by dependencies between loop iterations, for example. Since benchmarks have different affinities to each execution pattern, the overall configuration space is complex, and unique to each benchmark.

For example, the inner loop of *bitcount* can be executed in the shortest time by using a data-parallel SIMD structure to execute several loop iterations simultaneously, but energy is lowest when taking advantage of power gating with the dataflow execution pattern. This is because all iterations are independent, allowing an arbitrary number of them to be executed at the same time. The loop is also very tight, making dataflow a good match, as many cores can be specialised so much that they repeatedly execute a single instruction, and are able to power gate large sections of their pipelines. Loop unrolling would be required to increase the number of operations per clock cycle performed by the dataflow pattern.

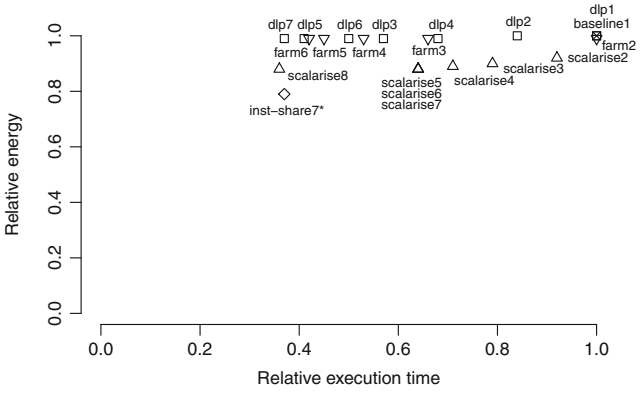
Conversely, JPEG colour conversion is fastest when many cores are used to exploit DLP, with one core reserved to provide common data, while energy is lowest when a pipeline structure is used. Again, all loop iterations are independent, but this time, a significant fraction of the work is



(a) ADPCM encode

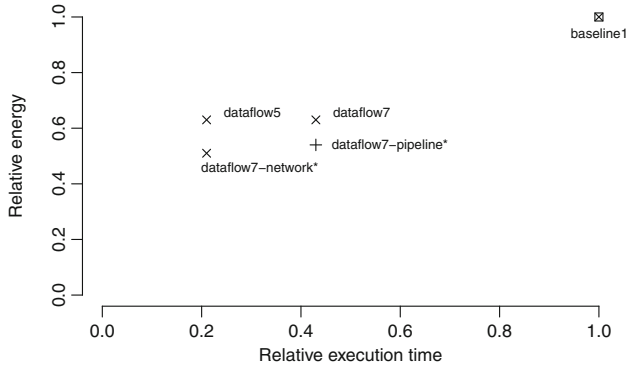


(b) Bit count inner loop

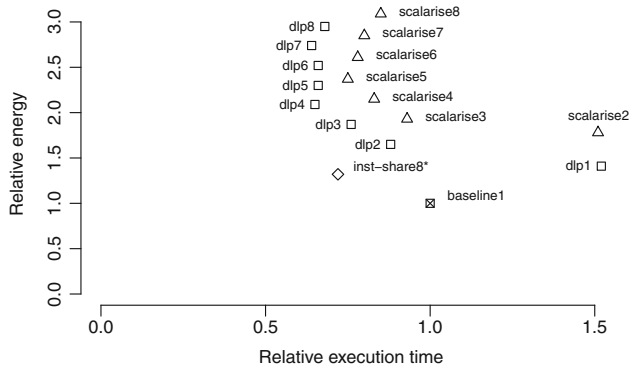


(c) Bit count outer loop

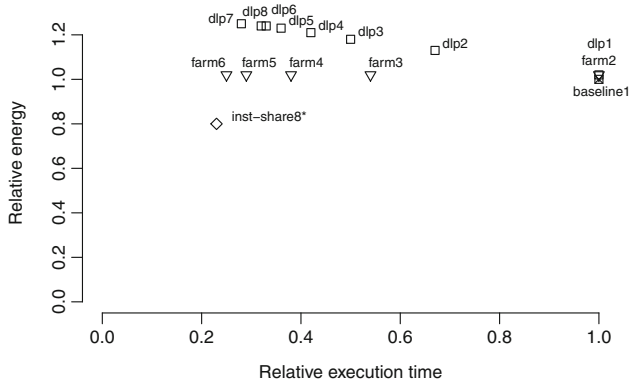
Figure 14 Summary of execution patterns. All results are relative to a single-core baseline. *dlp*: data-level parallelism pattern; *scalarise*: DLP with common work extracted to a helper core; *farm*: DLP using a worker farm for load balance; *inst-share*: DLP with instruction sharing; *pipeline*: task-level pipeline execution pattern; *dataflow*: dataflow execution pattern; *dataflow-network*: dataflow with direct links between neighbouring cores; *dataflow-pipeline*: dataflow with two ALUs per core; <execution-pattern>*N*: pattern used a total of *N* cores; <execution-pattern>*: results are estimated from a limit study, rather than a simulation.



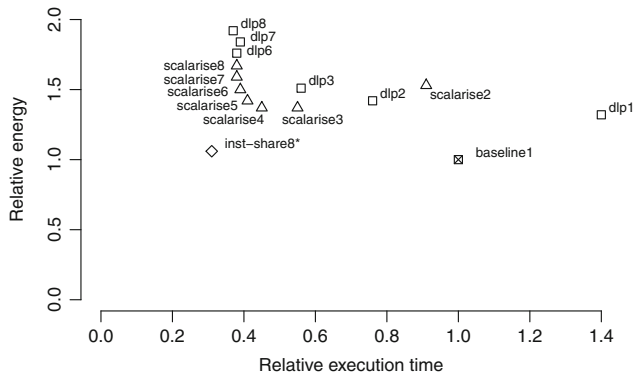
(d) CRC



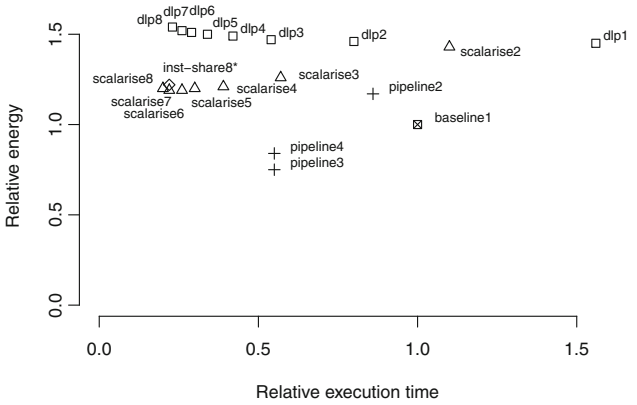
(g) JPEG DCT



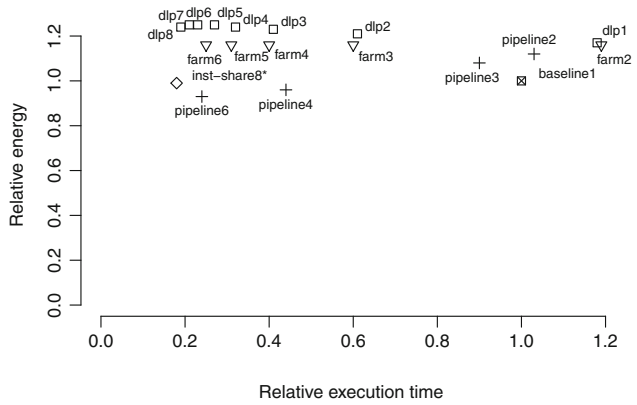
(e) Dijkstra's algorithm



(h) JPEG Huffman encoding initialisation



(f) JPEG colour conversion



(i) String search

Figure 14 (continued)

repeated in all iterations, so a performance (and energy efficiency) boost is seen when this work is extracted to a helper core. The task-level pipeline reduces energy consumption by reducing the amount of code executed by each core so that it fits entirely in the low-energy L0 instruction cache. Energy is lowest when each core works on a separate colour channel.

This confirms that flexibility is an important feature of the Loki architecture, as it allows acceleration or energy reduction for a wider range of applications than an architecture limited to any subset of execution patterns.

5 Related Work

The Raw processor [25] also provides tightly coupled on-chip networks. Raw's static networks provide low-latency communication between cores. Access to them is provided by register mapped input and output FIFOs. The static

routers themselves execute programs that dictate the how the network is configured on a cycle-by-cycle basis. In contrast, Loki exploits statically allocated channel buffers and end-to-end flow control when required. Loki also places a number of cores within a single tile supported by local point-to-point and multicast networks.

Raw was later commercialised by Tilera and now provides up to 72 cores on a single package [26]. Adapteva's Epiphany [2] is a similar commercial processor in this area, with a large number of cores connected by a mesh network. Epiphany's design is focused on high-throughput and low-power floating-point computation, with less of a focus on cooperation between cores. Like Loki, KALRAY's many-core architecture [15] can combine its cores in a variety of different ways to exploit different forms of parallelism, but its base computation unit is much more complex, and shared memory is used instead of direct message passing.

ACRES [1] explored the compilation issues and opportunities when programs are to be mapped spatially across a homogeneous fabric. Loki is able to emulate many of the capabilities of the ACRES proposal.

PPA [23] and Smart Memories [21] both allow an architecture to reconfigure itself in software to adapt to an application's needs. Smart Memories is able to partition its physical memory into virtual memories, each with different capacities, line sizes, replacement policies, and so on. PPA is able to dynamically adjust the number of functional units being used, depending on the available parallelism in the program. SCALE [18] and TRIPS [7] both introduce new ways of executing programs. SCALE is an instantiation of the vector-thread paradigm, which allows execution to move between SIMD and MIMD depending on the type of parallelism available. TRIPS makes use of the EDGE ISA to efficiently exploit dataflow parallelism on a homogeneous fabric and blur the boundaries between cores. Loki uses tightly-coupled cores to provide further flexibility: as well as changing the number of cores being used, it is also possible to change the type of parallelism they exploit. Loki is able to efficiently emulate the SIMD parallelism exploited by PPA, the master-slave and independent execution patterns of SCALE and the dataflow execution of TRIPS.

The Elm architecture [4] explores a number of techniques that permits software to better control the movement of instructions and data in order to improve energy efficiency. Both communication resources and the movement of instructions and data through the storage hierarchies can be managed by the compiler. Groups of four processors are grouped within an Ensemble and local interconnects permit register-mapped single-cycle communication (blocking and non-blocking) between the cores. SIMD execution is supported by allowing a single core to broadcast instructions to others in the group. Elm's contributions are mostly focused on improving efficiency within a single core, so can be seen as generally orthogonal to this work.

There have also been a number of recent architectures described that are able to dynamically compose a small number of cores to create more powerful multiple-issue cores [6, 12]. A related approach starts with a complex superscalar core and makes modifications to allow it to switch between single-thread-high-performance and multiple-thread-high-throughput modes [16]. These architectures are limited in the types of parallelism they are able to exploit, so in some cases will have to settle for a sub-optimal configuration.

In this paper, parallelism was extracted manually. There has been a lot of recent work on automatic parallelisation, however, and much of this could be applied to Loki. It is possible to extract DOALL parallelism [10], DOACROSS parallelism [8], and pipeline parallelism [22]. Dataflow graphs are standard intermediate representations within compilers, and can be mapped to cores automatically. There also exist transformations to increase the amount of time that an execution pattern can be used; Zhong et al. use speculation to extract more DOALL parallelism [27], and pipeline parallelism can become an enabling transformation for other forms of parallelism [11].

6 Conclusion

The addition of low-latency and low-cost point-to-point and multi-cast interconnect between cores provides an opportunity to exploit a variety of parallel execution patterns on a relatively simple low-power homogeneous platform. This flexibility allows better application performance and/or energy consumption than any fixed subset of parallelism types because the underlying structures of applications can be mapped to hardware in a more direct way. Streamlined processor pipelines permit energy per operation to be reduced to around 10pJ, more than an order of magnitude lower than typical mobile application processors. Furthermore, many of the execution patterns explored are able to simultaneously improve performance and energy as more cores are employed.

Modern mobile systems are required to provide 1000GOPS at around 1pJ per operation for specialised tasks such as 4G signal processing. This work is a step towards many-core systems with more than 1000 cores which will, we predict, be able to reach and exceed this target without the need for complex heterogeneous architectures. We suggest that design and verification effort is better spent on optimising a regular, all-purpose architecture, rather than a wide range of programmable processors and fixed-function accelerators.

7 Future Work

The work in this paper is currently being extended to explore different ways in which cores across multiple tiles can be used. As a group, we are also extending our compiler's support for exploiting multiple cores given modest amounts of ILP; exploring dynamic reconfiguration, and the ability to reconfigure more of the design, such as network protocols; and investigating the possibility of providing access to configurable accelerators.

Longer term plans include the fabrication of a test chip (scheduled for 2015) and the exploration of improved support for security and operating systems.

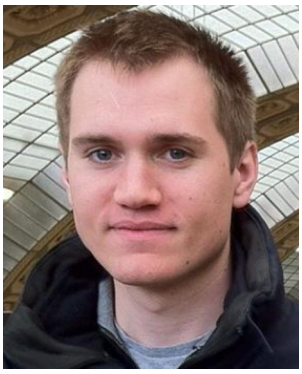
Acknowledgements The authors would like to thank the anonymous reviewers for their insightful comments and guidance.

References

1. Ang, B.S., & Schlansker, M. (2004). ACRES architecture and compilation. Tech. rep., Hewlett-Packard. URL www.hpl.hp.com/techreports/2003/HPL-2003-209R1.pdf.
2. Gwennap, L. (2011). Adapteva: More flops, less watts. *Microprocessor Report*, *6*(13), 11–02.
3. ARM Ltd. (2013). ARM1176 processor. URL http://www.arm.com/products/processors/classic/arm11/arm1176.php.
4. Balfour, J. (2010). Efficient embedded computing. PhD thesis, Stanford University. URL http://cva.stanford.edu/publications/2010/jbalfour-thesis.pdf.
5. Bates, D., Bradbury, A., Koltes, A., Mullins, R. (2013). Exploiting tightly-coupled cores. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII)*, (pp. 296–305). doi:10.1109/SAMOS.2013.6621138.
6. Boyer, M., Tarjan, D., Skadron, K. (2010). Federation: Boosting per-thread performance of throughput-oriented manycore architectures. *ACM Trans Archit Code Optim*, *7*, 19:1–19:38. URL http://doi.acm.org/10.1145/1880043.1880046.
7. Burger, D., Keckler, S.W., McKinley, K.S., Dahlin, M., John, L.K., Lin, C., Moore, C.R., Burrill, J., McDonald, R.G., Yoder, W., the TRIPS Team (2004). Scaling to the end of silicon with EDGE architectures. *Computer*, *37*(7), 44–55. doi:10.1109/MC.2004.65.
8. Campanoni, S., Jones, T., Holloway, G., Reddi, V.J., Wei, G.Y., Brooks, D. (2012). HELIX: automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. *CGO '12*, (pp. 84–93). New York: ACM. doi:http://doi.acm.org/10.1145/2259016.2259028.
9. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B. (2001). MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings 4th Annual Workshop on Workload Characterization*. *WWC-4*, (pp. 3–14). Washington: IEEE International Workshop, IEEE Computer Society. doi:10.1109/WWC.2001.15. http://portal.acm.org/citation.cfm?id=1128020.1128563.
10. Hall, M.W., Anderson, J.M., Amarasinghe, S.P., Murphy, B.R., Liao, S.W., Bugnion, E., Lam, M.S. (1996). Maximizing multiprocessor performance with the SUIF compiler. *Computer*, *29*(12), 84–89. doi:10.1109/2.546613.
11. Huang, J., Raman, A., Jablin, T.B., Zhang, Y., Hung, T.H., August, D.I. (2010). Decoupled software pipelining creates parallelization opportunities. In *Proceedings of the 8th annual IEEE/ACM International Symposium on Code Generation and Optimization*. *CGO '10*, (pp. 121–130). New York: ACM. doi:http://doi.acm.org/10.1145/1772954.1772973.
12. Ipek, E., Kirman, M., Kirman, N., Martinez, J.F. (2007). Core Fusion: accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th annual International Symposium on Computer Architecture*. *ISCA '07*, (pp. 186–197). New York: ACM. doi:http://doi.acm.org/10.1145/1250662.1250686.
13. Johnston, J., & Fitzsimmons, T. (2011). The newlib homepage. URL http://sourceware.org/newlib/.
14. Kahng, A.B., Li, B., Peh, L.S., Samadi, K. (2009). 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, European Design and Automation Association*. *DATE '09*, (pp. 423–428). Belgium: 3001 Leuven. URL http://dl.acm.org/citation.cfm?id=1874620.1874721.
15. KALRAY Corporation (2012). Many-core processors – KALRAY. URL http://www.kalray.eu/.
16. Khubaib, Suleman, M.A., Hashemi, M., Wilkerson, C., Patt, Y.N. (2012). MorphCore: An energy-efficient microarchitecture for high performance ILP and high throughput TLP. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. *MICRO-45*, (pp. 305–316). Washington: IEEE Computer Society. doi:10.1109/MICRO.2012.36.
17. Kin, J., Gupta, M., Mangione-Smith, W.H. (1997). The filter cache: An energy efficient memory structure. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*. *MICRO-30*, (pp. 184–193). Washington: IEEE Computer Society. URL http://dl.acm.org/citation.cfm?id=266800.266818.
18. Krashinsky, R., Batten, C., Hampton, M., Gerding, S., Pharris, B., Casper, J., Asanovic, K. (2004). The vector-thread architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*. *ISCA '04*, (pp. 52–63). Washington: IEEE Computer Society. URL http://dl.acm.org/citation.cfm?id=998680.1006736.
19. Lattner, C. (2010). The LLVM compiler infrastructure. URL http://llvm.org/.
20. Lee, Y., Krashinsky, R., Grover, V., Keckler, S.W., Asanovic, K. (2013). Convergence and scalarization for data-parallel architectures. In *Proceedings of the 2013 International Symposium on Code Generation and Optimization*. URL http://www.eecs.berkeley.edu/yunsup/papers/scalarization-cgo2013.pdf.
21. Mai, K., Paaske, T., Jayasena, N., Ho, R., Dally, W.J., Horowitz, M. (2000). Smart Memories: a modular reconfigurable architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. *ISCA '00*, (pp. 161–171). New York: ACM. doi:http://doi.acm.org/10.1145/339647.339673.
22. Ottoni, G., Rangan, R., Stoler, A., August, D. (2005). Automatic thread extraction with decoupled software pipelining. In *Proceedings of 38th Annual IEEE/ACM International Symposium on Microarchitecture*. *MICRO-38*, (pp. 105–108).
23. Park, H., Park, Y., Mahlke, S. (2009a). Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. *MICRO 42*, (pp. 370–380). New York: ACM. doi:http://doi.acm.org/10.1145/1669112.1669160.

24. Park, J., Balfour, J., Dally, W.J. (2009b). Maximizing the filter rate of L0 compiler-managed instruction stores by pinning. Tech. Rep. 126, Stanford University. URL http://cva.stanford.edu/projects/elm/pubs/tecrep_2009_pinning.pdf.
25. Taylor, M.B., Kim, J., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffman, H., Johnson, P., Lee, J.W., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S., Agarwal, A. (2002). The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, *22*, 25–35. doi:10.1109/MM.2002.997877. http://dl.acm.org/citation.cfm?id=623304.624515.
26. Tilera Corporation (2010). Tilera Processors Webpage. URL http://www.tilera.com/products/processors.php.
27. Zhong, H., Mehrara, M., Lieberman, S., Mahlke, S. (2008). Uncovering hidden loop level parallelism in sequential applications. In *IEEE 14th International Symposium on High Performance Computer Architecture. HPCA 2008*, (pp. 290–301).

Daniel Bates is a Research Associate in the Computer Laboratory at the University of Cambridge, where he also received his BA and PhD degrees. His research focus is on coarse-grained reconfigurable systems, hardware/software co-design, and novel ways of exploiting parallelism.



Alex Bradbury received his undergraduate degree from the University of Cambridge Computer Laboratory in 2009, where he is currently pursuing his Ph.D. His research interests include optimising compilers for many-core systems, novel processing fabrics, and compilation for non-traditional architectures.

Andreas Koltes is a Doctoral Researcher in the Computer Laboratory at the University of Cambridge. His main research interests lie in the areas of Reconfigurable Computing, Computer Architecture and System-on-Chip Integration. He holds a German Diplom degree in computer science from the University of Passau, Germany.

Robert D. Mullins is a Senior Lecturer in the Computer Laboratory at the University of Cambridge. His research is focused in the areas of computer architecture and VLSI design. He has a particular interest in on-chip interconnection networks, chip-multiprocessors and novel parallel processing fabrics. He has BEng, MSc and PhD degrees from the University of Edinburgh.