Exploiting Tightly-Coupled Cores

Daniel Bates, Alex Bradbury, Andreas Koltes and Robert Mullins
Computer Laboratory, University of Cambridge, UK
Email: {Daniel.Bates, Alex.Bradbury, Andreas.Koltes, Robert.Mullins}@cl.cam.ac.uk

***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. 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.**

I. 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 general-purpose 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.

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. Through 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 is possible on a traditional multi-core architecture. We suggest the programmer and compiler specify an application-specific virtual architecture or *overlay* for their target application. This is a network of the best processors, helper engines, accelerators, memories and routers for that application. 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 work was supported by EPSRC grant EP/G033110/1.

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 and at low-cost over a single (logical) on-chip network. This design must allow cores and memories to be composed to form larger computation structures, and must also allow 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 must additionally support 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

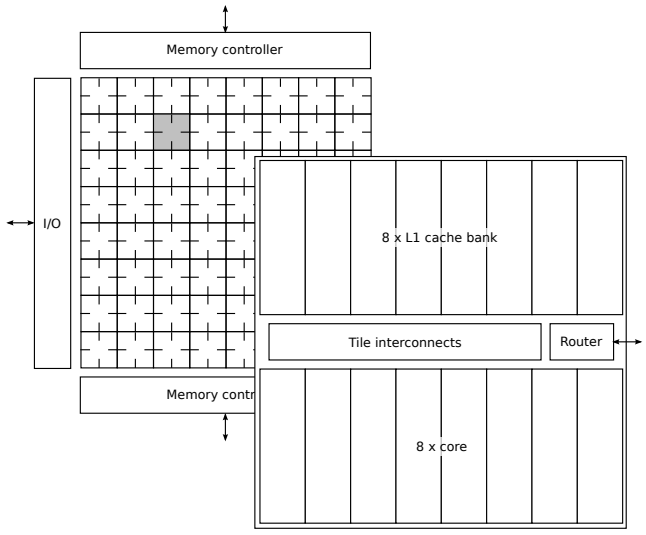


Fig. 1. Loki's tiled architecture. Left: chip with one tile highlighted. Right: tile block diagram.

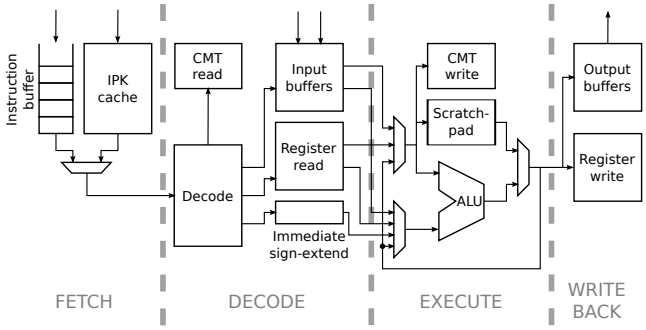


Fig. 2. Loki core microarchitecture block diagram. IPK: Instruction Packet – atomic group of instructions similar to a basic block. CMT: Channel Map Table – mapping between logical and physical network addresses.

they can be configured. This is achieved by exposing many hardware elements through the instruction set and performing all specialisation in software. This main contributions of this paper are:

- An overview of the Loki architecture; one instance of the class of communication-centric architectures we describe (Section II);
- A framework for high-level energy modelling, and a detailed performance, energy and area characterisation for the Loki architecture (Section III);
- Demonstration that tightly-coupled cores, through the provision of compiler-controlled interconnect, allow a broad range of parallelisation techniques (Section IV).

II. Loki architecture

Loki is a homogeneous, tiled architecture, composed of cores and memories connected through an on-chip network (Figure 1). 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.

A. 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 with the aid of an explicit configuration. The aim is to tailor the execution and communication patterns to each program or phase of 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.

B. 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. The local network is implemented as a collection of networks each optimised for a particular communication pattern. Cores and memories communicate with each other over two fast crossbars (one in each direction) with half-cycle latencies to minimise memory latency. Each 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. For the global interconnect we use a simple placeholder packet-switched mesh network, with a router in each tile providing single-cycle hops to neighbouring tiles.

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. Instruction packets may also 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

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

(a) C code

```
setchmapi 1, r11
[...]
fetch     r10
xor       r11, r14, r13
lli       r12, %lo(crc_32_tab)
lui       r12, %hi(crc_32_tab)
andi      r11, r11, 255
slli      r11, r11, 2
addu      r11, r12, r11
ldw       0(r11) -> 1
srli      r12, r14, 8
xor.eop   r11, r2, r12
```

(b) Loki assembly code

Fig. 3. CRC code example showing features of Loki's instruction set.

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 [13]. 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). 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, and hence threads if necessary, to be remapped transparently at run-time.

Figure 3 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*: 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.

C. Instruction and data supply

Instruction packets can be stored in each core's 64-entry 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 [1], [2]. 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.

Each core also has a 16-entry 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.

D. 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. This could be useful for implementing virtual memory, memory protection, and transactional memory, for example. It also makes the memory banks easily accessible to multiple cores. 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.

III. Methodology

A. 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.

B. Benchmarks

Our experiments are performed using the MiBench benchmark suite [3]. 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 [4] C standard library implementation.

We use a custom LLVM-based [5] 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.

C. 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 [6] 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 for a commercial 40nm low-V_t process. We select only low-V_t cells and leakage is subsequently low and is not reported here. Timing is closed using a multi-corner PVT analysis where 0.99V and -40°C is 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 simultaneous constraints from the instruction packet cache, register file, and memory bank. The design is conservatively margined at the WC corner including foundry recommendations for OCV 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 synthesizing and modelling each datapath component separately will likely overestimate costs slightly.

The floorplan of a single tile is shown in Figure 4. 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, while 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-entry instruction cache (with 16 cache tags); 32-entry register-file; 7 network buffers of 4 entries each; 256-entry scratchpad memory; 16-entry channel map table.

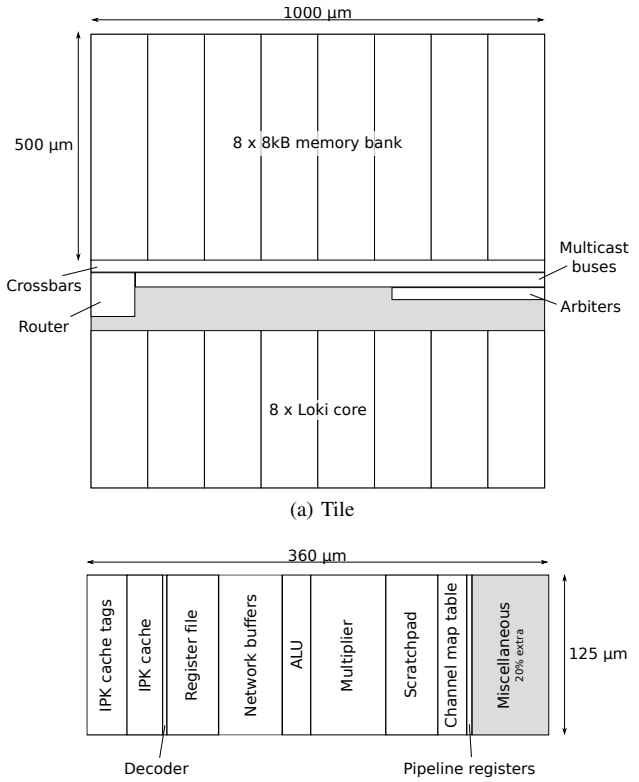
IV. Evaluation

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.

A. Baseline

Energy consumption for each benchmark running on a single Loki core is shown in Figure 5 – 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* 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



(a) Tile

(b) Core

Fig. 4. 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.

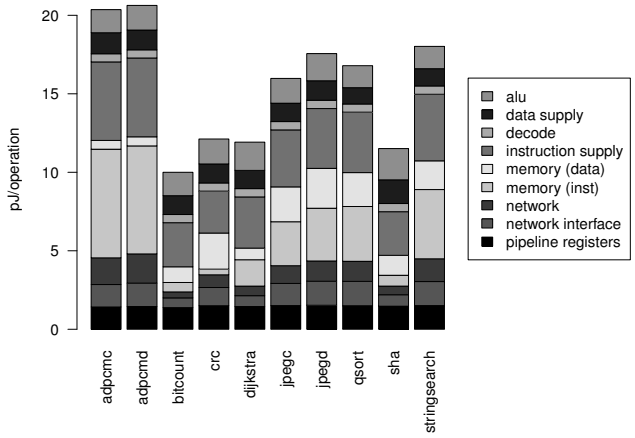


Fig. 5. MiBench baseline energy distribution.

data at 65nm [7] 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

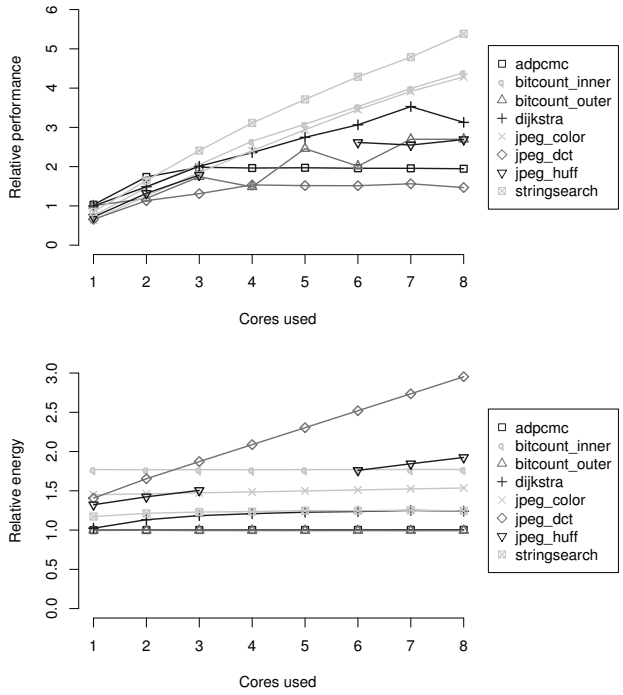


Fig. 6. Performance and energy consumption as the number of data-parallel cores changes, relative to the baseline sequential implementation.

instructions in place of each ARM instruction to improve performance, while still consuming relatively little power.

B. 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 6 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

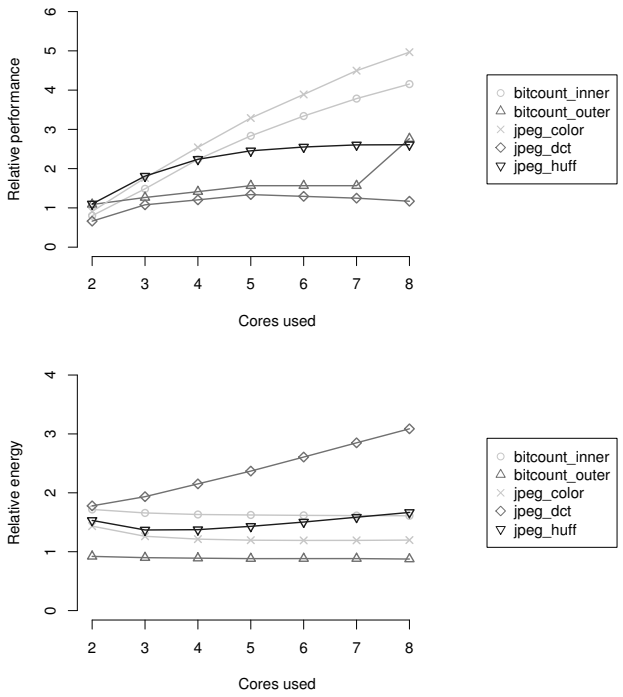


Fig. 7. Performance and energy consumption as the width of the number of data-parallel changes, when making use of a helper core, relative to the baseline sequential implementation.

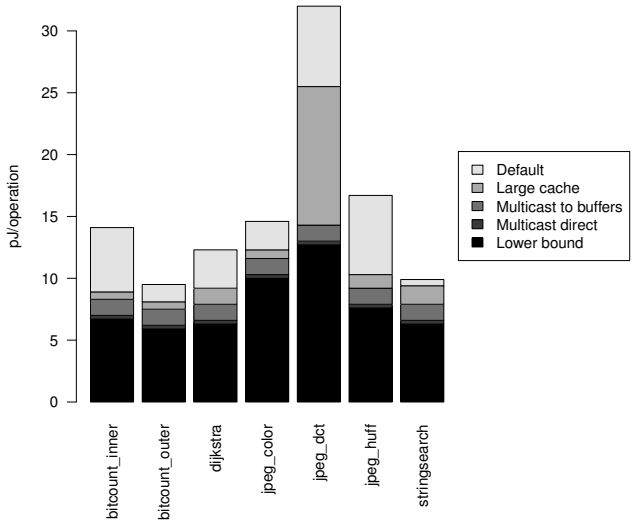


Fig. 8. Energy consumption of various instruction sharing strategies. Results are for 8 cores.

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 [8].

The impact of such helper cores is shown in Figure 7. *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.

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 8). 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 but have the same access costs. Techniques for switching between different cores' instruction caches have been demonstrated previously by the Elm architecture [9]. The extra cache capacity improves performance by an average of 14% for 8 cores.

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.

The two optimisations described in this section—the helper core and instruction sharing—can be applied in combination to further reduce energy consumption.

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.

C. 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 setting up the required communication network, and placing a small number of instructions on each core for repeated execution. 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 clock 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 9 and 10 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 9 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.

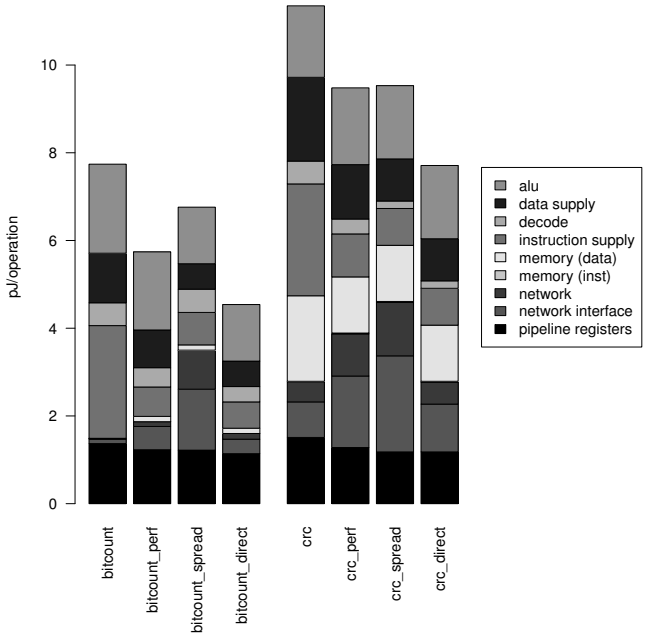


Fig. 9. Energy distribution when using dataflow execution pattern.

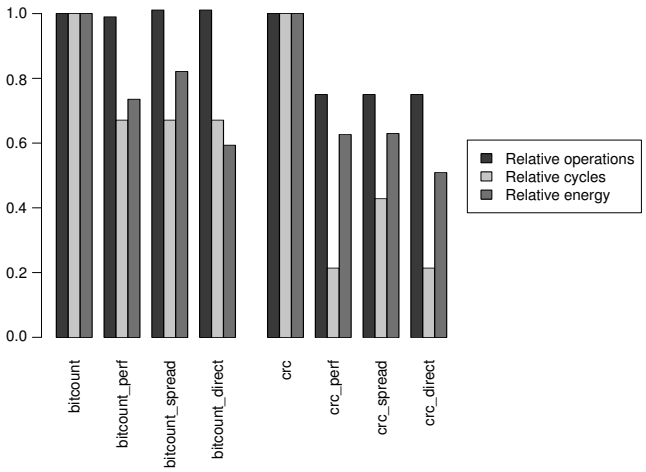


Fig. 10. Relative performance when using dataflow execution pattern.

Figure 10 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 9 and 10. 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% over the *spread* case to 4.5pJ/operation, and *crc*'s energy reduces by 19% to 7.7pJ/operation.

D. Pipeline-level parallelism

Pipeline (or streaming) parallelism involves each core independently processing data, and passing the result onto the next core. Locality is improved by having each core working on a smaller section of the program, and 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 11 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.

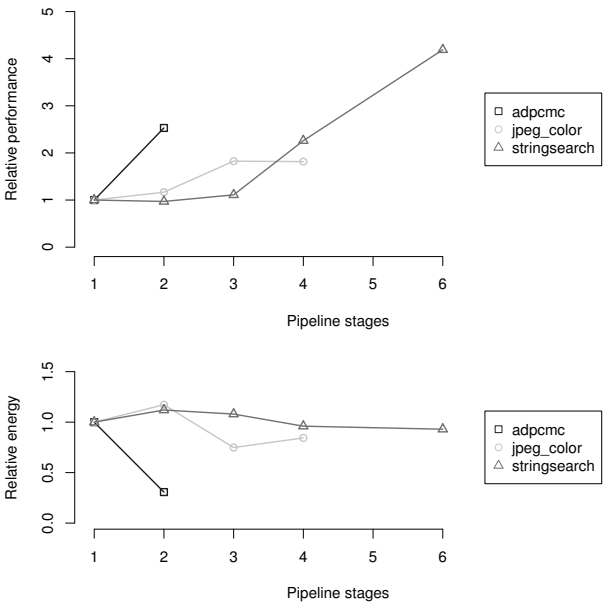


Fig. 11. Relative performance and energy consumption when using software pipelines of different lengths, relative to the baseline sequential implementation.

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 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 III-B, performance improved by 153% and energy reduced by 69% to 9.6pJ/operation. These super-linear improvements were helped by a 15% reduction in instruction count due to the extra registers, ILP extraction, and improved caching. Mapping the code across two cores outperforms a single core with twice as much cache by 2× and improves energy consumption by 20%.

E. 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 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. 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 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.

V. Related work

The Raw processor [12] 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.

ACRES [10] 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 [11] and Smart Memories [14] 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 [13] and TRIPS [15] 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 [9] 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.

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 [16], [17]. A related approach starts with a complex superscalar core and make modifications to allow it to switch between single-thread-high-performance and multiple-thread-high-throughput modes [18]. 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 [19], DOACROSS parallelism [20], and pipeline parallelism [21]. 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 [22], and pipeline parallelism can become an enabling transformation for other forms of parallelism [23].

VI. 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. 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. Current work is on 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.

Future mobile systems will be required to provide 1000GOPS at ∼1pJ/operation. This work is a step towards many-core systems with more than 1000 cores which will, we predict, be able to achieve 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.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their feedback and guidance.

REFERENCES

[1] J. Kin, M. Gupta, and W. H. Mangione-Smith, "The filter cache: an energy efficient memory structure," in *Proc. of MICRO 30*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 184–193.

[2] J. Park, J. Balfour, and W. J. Dally, "Maximizing the filter rate of L0 compiler-managed instruction stores by pinning," Stanford University, Tech. Rep. 126, 2009.

[3] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14.

[4] J. Johnston and T. Fitzsimmons, "The newlib homepage," http://sourceware.org/newlib/, 2011.

[5] C. Lattner, "The LLVM compiler infrastructure," http://llvm.org/.

[6] 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 *Proc. of DATE '09*. 3001 Leuven, Belgium: European Design and Automation Association, 2009, pp. 423–428.

[7] ARM Ltd., "ARM1176 processor," http://www.arm.com/products/processors/classic/arm11/arm1176.php, 2013.

[8] Y. Lee, R. Krashinsky, V. Grover, S. W. Keckler, and K. Asanovic, "Convergence and scalarization for data-parallel architectures," in *Proc. of CGO 2013*, 2013.

[9] J. Balfour, "Efficient embedded computing," Ph.D. dissertation, Stanford University, May 2010.

[10] B. S. Ang and M. Schlansker, "ACRES architecture and compilation," Hewlett-Packard, Tech. Rep., April 2004.

[11] H. Park, Y. Park, and S. Mahlke, "Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications," in *Proc. of MICRO 42*. New York, NY, USA: ACM, 2009, pp. 370–380.

[12] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The Raw microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, pp. 25–35, March 2002.

[13] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, "The vector-thread architecture," in *Proc. of ISCA '04*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 52–.

[14] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart Memories: a modular reconfigurable architecture," in *Proc. of ISCA '00*. New York, NY, USA: ACM, 2000, pp. 161–171.

[15] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS Team, "Scaling to the end of silicon with EDGE architectures," *Computer*, vol. 37, no. 7, pp. 44–55, Jul. 2004.

[16] M. Boyer, D. Tarjan, and K. Skadron, "Federation: Boosting per-thread performance of throughput-oriented manycore architectures," *ACM Trans. Archit. Code Optim.*, vol. 7, pp. 19:1–19:38, December 2010.

[17] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, "Core fusion: accommodating software diversity in chip multiprocessors," in *Proc. of ISCA '07*. New York, NY, USA: ACM, 2007, pp. 186–197.

[18] Khubaib, M. A. Suleman, M. Hashemi, C. Wilkerson, and Y. N. Patt, "Morphcore: An energy-efficient microarchitecture for high performance ILP and high throughput TLP," in *Proc. of MICRO45*, 2012.

[19] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam, "Maximizing multiprocessor performance with the SUIF compiler," *Computer*, vol. 29, no. 12, pp. 84–89, Dec. 1996.

[20] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks, "HELIX: automatic parallelization of irregular programs for chip multiprocessing," in *Proc. of CGO '12*. New York, NY, USA: ACM, 2012, pp. 84–93.

[21] G. Ottoni, R. Rangan, A. Stoler, and D. August, "Automatic thread extraction with decoupled software pipelining," in *Proc. of MICRO-38*, nov. 2005, p. 12 pp.

[22] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke, "Uncovering hidden loop level parallelism in sequential applications," in *Proc. of HPCA14*, 2008.

[23] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T.-H. Hung, and D. I. August, "Decoupled software pipelining creates parallelization opportunities," in *Proc. of CGO '10*. New York, NY, USA: ACM, 2010, pp. 121–130.