Dynamically Configurable Heterogeneous Multi-Core Architecture

Seminar Monthly Report - September

Submitted in partial fulfillment of the requirements

for the course

EE 694

by

Piyush Palav

(Roll No. 213079017)

Under the guidance of

Prof. Virendra Singh

Department of Electrical Engineering
Indian Institute of Technology Bombay

September 2022

Acknowledgement

I express my gratitude to my guide Prof. Virendra Singh for providing me the opportunity to work on this topic.

Piyush Palav
Electrical Engineering
IIT Bombay

Contents

List of Figures

Chapter 1

Introduction

In the past, increasing transistor counts and frequency resulted in direct translation into performance improvement of single processor cores. But the breakdown of Dennard scaling caused architects to turn to Chip Multiprocessors (CMPs) (multiple cores on same die) to provide increased performance within dictated power and thermal limits.

Homogeneous multicore chip-cores with same microarchitecture:*in-order(InO)* or *out-of-order(OoO)* can exploit Thread level parallelism (TLP) speeding up parallel code fragments to achieve high parallel performance. However, for applications with substantial sequential code fragments, Amdahl's law [1] states that the speedup will still be limited by the performance of the serial code. Only complex OoO execution engines can exploit Instruction level parallelism (ILP) to accelerate sequential code execution.

This led to the design of Asymmetric chip multiprocessors (ACMPs) which incorporated a few high-performance complex cores to speedup sequential code and many low-performance simple cores ensuring good parallel performance. However, it has been found that the performance asymmetry in ACMPs adversely affects the predictability and scalability of a large number of software applications [2].

Modern applications therefore, demand a hardware/software model that can dynamically adapt to the continually changing program phases in the same run. This necessitates the need for a dynamic configuration of simple homogeneous (symmetric) cores that provides a large number of simple cores to exploit TLP and a small number of complex cores to exploit ILP.

This work explores some of the various proposals that exploit the heterogeneity in microarchitecture to achieve high performance and energy efficiency for both single & multi-threaded workloads.

Chapter 2

Literature Survey

Amdahl's law [1] states that if fraction 'f' of a program's execution time was infinitely parallelizable with no scheduling overhead, while the remaining fraction, $(1 - f)$ was totally sequential, then the speedup on 'n' processors is governed by

$$Speedup_{parallel}(f, n) = \frac{1}{(1 - f) + \frac{f}{n}}$$

$$\lim_{n \to \infty} Speedup_{parallel}(f, n) = \frac{1}{(1 - f)}$$

Hence, even if a large number of cores can decrease parallel execution time but the total execution time is still governed by the microarchitecture of a single core that can exploit ILP in sequential fraction with manageable levels of power and complexity.

Asymmetric chip multiprocessor (ACMP) comprises of cores with different sizes and performance designed to adapt muti-cores to speedup both sequential and parallel applications. However, they are based on a strong underlying assumption that all computational cores provide equal performance and performance asymmetry breaks this assumption. Impact of performance asymmetry on predictability and scalability can be reduced by making the operating system kernel or/and application asymmetry aware [2]. This facilitated the need for design space exploration into utilizing many simple homogeneous cores for multi-threaded programs and merging them at run-time for single-threaded program execution.

Core Fusion [3] is a fully hardware based solution built on a homogeneous substrate that "fuses" many cores into a large CPU on demand and also, provides fine grain parallelism with the help of remaining cores. It mitigates the need for change in execution model, customized ISAs or specialised compiler support at the expense of hardware complexity.

Bahurupi [4] is a hardware-software based solution that accelerates the execution of sequential code by forming a dynamic "coalition" of two or more simple homogeneous cores that can execute basic blocks in parallel offering increased ILP. It requires minimal additional hardware resources and compiler support for coalition but the improvements in performance and energy consumption achieved are better than traditional complex OoO superscalar processors.

Chapter 3

Review

3.1 Core Fusion

3.1.1 Summary

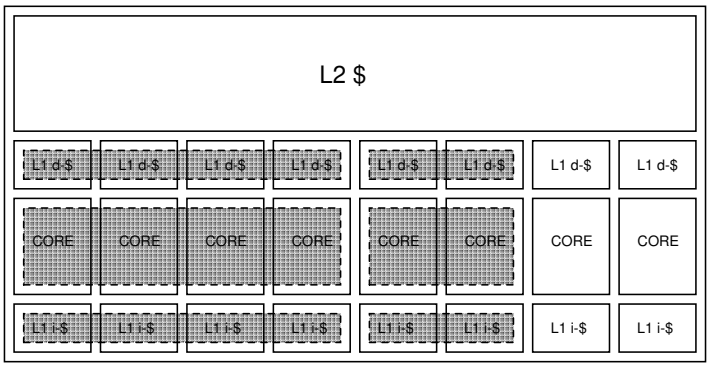

Figure 3.1: Eight core CMP with core fusion capability

In normal execution mode, Core fusion technique provides a large number of homogeneous cores that can be utilized for parallel execution and it can "fuse" many such cores dynamically to present a single large fused group that can accelerate sequential code region execution.

Fetch Management Unit (FMU) facilitates collective instruction fetch (aligned with core zero) with the help of re-configurable, distributed L1 i-cache organisation. In parallel mode, "n" subblocks and one tag within each i-cache constitute a cache block while in fused mode, a cache block spans all "n" i-caches, with each i-cache holding one subblock and a tag replica where "n" is the number of cores in the fused group (assuming each core has one private L1 i-cache). In case of a i-TLB miss in fused mode, FMU refills i-TLB of all cores in the fused group. FMU is also used to gang-invalidate i-TLB entries.

Since collective fetch is aligned with core zero, each branch instruction always accesses a particular core's branch predictor and BTB and hence, the effective capacity is increased by the number of cores present in the fused group. PC redirection in case of predict branch taken and branch misdirection correction is also handled by the FMU. In such cases, mispredicted instructions are flushed. To allow a core's branch predictor to learn correlation with branches handled by other cores, Global History Register (GHR) is replicated across all cores and its updates synchronized through FMU.
All RAS operations are processed by core zero.

On a fetch stall by one core in a fused group, to preserve fetch alignment, all remaining core fetches are stalled and overfetched instructions flushed by communication via FMU.

Each core decodes an instruction and sends it to a Steering management unit (SMU) which handles renaming and steering with the help of a global steering table that tracks the architectural registers' mapping, free lists, rename maps and renaming/steering logic. Operand communication across cores is handled by maintaining copy-in and copy-out queue per core.

If ROB of any core is blocked, to ensure in-order commit of instructions, all other cores are also not allowed to commit and the stall/resume signals are exchanged across ROBs.

Load-stores are handled by implementing a banked-by-address Load-Store Queue (LSQ).

Dynamic configuration of cores to adapt to software diversity is enabled through a pair of FUSE and SPLIT ISA instructions.

3.1.2 Strengths

1. No need for special execution model or custom ISAs as compared to TRIPS [5] and Multiscalar [6].

2. Additional programming or specialized compiler support mitigated.

3. Isolation across threads in parallel mode of execution achieved since all cores and their structures are fundamentally independent.

3.1.3 Weakness

1. Latency of the operand crossbar used for effective operand communication across cores affects performance tremendously.

2. Inserting NOP instruction in ROB and dummy entries in LSQ to allow in-order commit deteriorates performance.

3.1.4 Opportunity

Runtime utilization heuristic that tracks periodically the utilization of individual cores and can signal remaining cores to FUSE in case of high workload or existing cores in fused group to SPLIT in case of under-utilization or if the performance benefit obtained is not enough to amortize the cost of power consumption.

3.2 Bahurupi

3.2.1 Summary

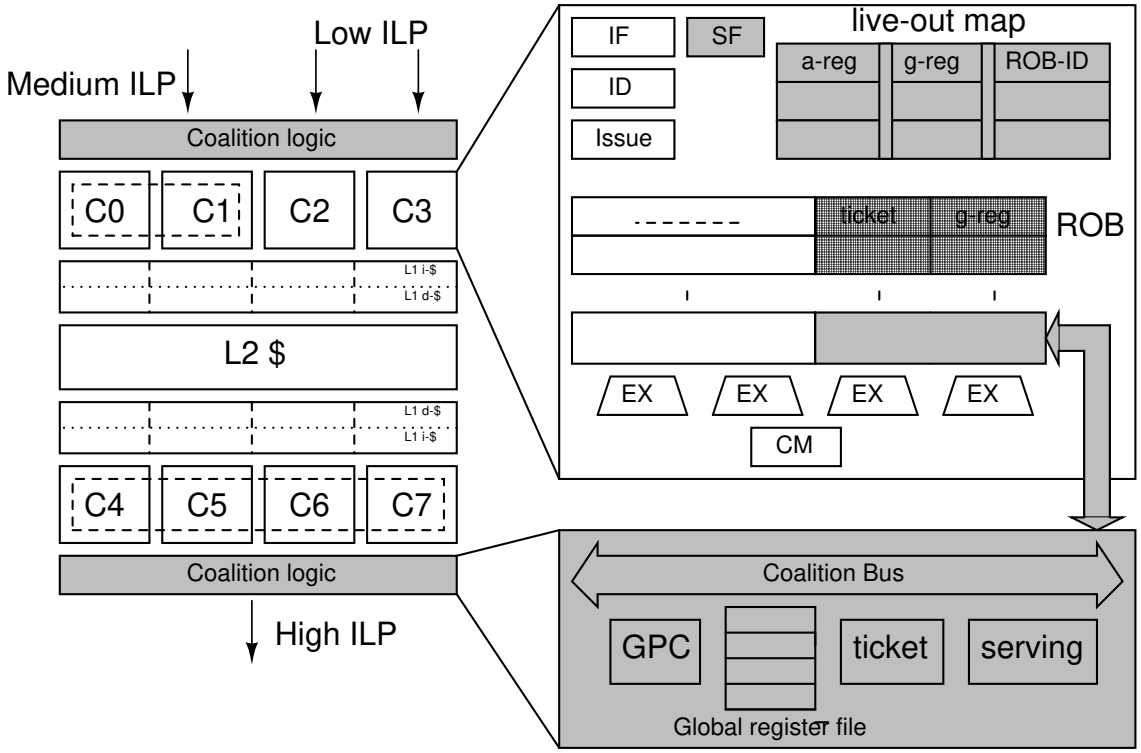

Figure 3.2: Bahurupi architectue with additional hardware for coalition highlighted

In normal execution mode, Bahurupi multi-core architecture can efficiently support multi-threaded execution and to improve single threaded program execution time, it can dynamically form a "coalition" of 2-4 cores in a cluster. A special instruction is used to request a core to join or leave from the coalition. The unit of execution for a core is a basic block-a sequence of instructions with single entry and single exit point. A core fetches one basic block of instructions at a time but the main sequential speedup of Bahurupi design comes from out-of-order parallel execution of basic blocks on cores that form coalition. Hence, it needs compiler support to identify basic blocks and inter-block register dependencies if any. This dependency information is explicitly conveyed by placing and fetching (SF) *"sentinel"* instructions at the start of each basic block. Sentinel instruction format is as follows:

Opcode	BB_SIZE	BB_TYPE	LI_0	LI_1	LI_M	LO_0	LO_1	LO_N

- BB_SIZE indicates the number of instructions in basic block
- BB_TYPE specifies if the basic block ends with a branch instruction
- LI_0—LI_M register fields indicate M *live-in* registers where a live-in register is a register that is alive at the entry of a basic block and used inside the basic block.
- LO_0—LO_N register fields indicate N *live-out* registers where a live-out register is a register that is alive at the exit of a basic block and updated inside the basic block.

Live-in and *live-out* register values are communicated among cores by using a shared global register file that contains a global map table for renaming and a global physical register file.

Global program counter (GPC) is maintained to allow renaming of global registers and execution of sentinel instructions in program order and access and update to GPC by various cores is maintained by a locking mechanism.

In-order commit of instructions to ensure speculative execution and precise exception is handled through a shared ticket lock mechanism that contains two registers:*serving* and *ticket*. When a core locks the GPC, it reads and increments the current value of *ticket* register and tags the ROB entries of all instructions in its basic block with this *ticket* value. After all instructions in its basic block are committed, it increments the value of *serving* register. Thus, instructions from the next block are committed when they are ready to commit and their *ticket* value matches the value in the *serving* register.

To correct branch mispredictions and exceptions, all cores are signaled to flush their fetch queues and ROB entries with *ticket* values greater than the *ticket* value of mispredicted branch instruction. GPC is restored to point to the sentinel instruction preceding the corrected branch direction and global *ticket* value is restored to one plus the *ticket* value of mispredicted branch instruction.

In normal mode, each core is allocated a L1 cache bank and all the 'N' cache banks behave as a 'N'-way set associative shared cache in coalition mode.

WAR (write-after-read) and RAW(read-after-write) memory hazards are removed through memory disambiguation at commit stage.

3.2.2 Strengths

1. No need for special execution model or custom ISAs as compared to TRIPS [5] and Multiscalar [6].

2. Minimal modifications to internal micro-architecture of cores, additional hardware resources and compiler support.

3. Dedicated powerful core to accelerate slow thread execution not needed since a bigger coalition can speedup the slow thread.

4. Achieves significant speedup and power reduction as compared to traditional complex OoO processors since it can exploit ILP from independent instructions across parallel basic blocks compared to restricted serial fetch and execution of basic blocks in OoO processors.

3.2.3 Weakness

Since at most two coalitions can occur at any point in time, cores present in a coalition can be overutilized or underutilized.

3.2.4 Opportunity

Runtime utilization heuristic that tracks periodically the utilization of individual cores and can signal remaining cores to coalesce in case of high workload or existing cores in coalition to leave in case of under-utilization.

Chapter 4

Conclusion

Impact of performance asymmetry in ACMPs motivated architects to explore the ability of merging and splitting simple homogeneous cores at runtime to address the software diversity present in majority of modern workloads. Core fusion is a micro-architectural technique and Bahurupi is a hardware-software solution that exploits heterogeneity to support incremental parallelization.

Core Fusion fuses cores to exploit ILP from sequential phase and executes parallel code on many simple symmetric cores, thereby exploiting TLP. It provides an average speedup of 50% for the floating-point SPEC applications and 30% for the integer SPEC applications when using a quad-core fused configuration compared with fine-grain 2-way CMP. However, the fused core consumes high power and the single threaded performance provided is less than that of a large complex OoO core statically optimized for single thread due to additional latencies among its pipeline stages. Also, the reconfiguration overhead associated with switching between fused mode and simple mode reduces performance.

Bahurupi employs a hardware-software cooperative model that forms a "coalition" of 2-4 cores to speedup sequential mode execution and executes parallel threads on simple homogeneous cores. It requires minimal changes to underlying core micro-architecture and uses hints from the compiler for making dynamic reconfiguration decisions. As compared to Core Fusion, quad-core Bahurupi obtains an average speedup of 91% for SPEC integer applications and 210% for SPEC floating-point applications compared to baseline 2-way core. It can achieve the performance of complex OoO processor without paying the price of complex hardware and associated energy efficiency of the later.

However, both of the given proposals do not explicitly track the utilization of individual cores and hence, a utilization heuristic can be introduced to improve performance further while decreasing power consumption at the same time.

References

[1] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[2] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, "The impact of performance asymmetry in emerging multicore architectures," in *32nd International Symposium on Computer Architecture (ISCA'05)*, pp. 506–517, 2005.

[3] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, "Core fusion: accommodating software diversity in chip multiprocessors," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 186–197, 2007.

[4] M. Pricopi and T. Mitra, "Bahurupi: A polymorphic heterogeneous multi-core architecture," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, p. 22, 2012.

[5] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. Keckler, and C. Moore, "Exploiting ilp, tlp, and dlp with the polymorphous trips architecture," in *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, pp. 422–433, 2003.

[6] G. Sohi, S. Breach, and T. Vijaykumar, "Multiscalar processors," in *Proceedings 22nd Annual International Symposium on Computer Architecture*, pp. 414–425, 1995.