H3 (Heterogeneity in 3D): A Logic-on-logic 3D-stacked Heterogeneous Multi-core Processor

Vinesh Srinivasan*, Rangeen Basu Roy Chowdhury†, Elliott Forbes‡, Randy Widialaksono†,
Zhenqain Zhang§, Joshua Schabel*, Sungkwan Ku†, Steve Lipa*, Eric Rotenberg*, W. Rhett Davis*, Paul D. Franzon*

*North Carolina State University, Raleigh, NC, USA
†Intel, Santa Clara, CA, USA
‡University of Wisconsin-La Crosse, La-Crosse, WI, USA
§Nvidia, Santa Clara, CA, USA

Email: {vsriniv3, jcledfo3, slipa, ericro, wdavis, paulf}@ncsu.edu, eforbes@uwlax.edu, zhenqianz@nvidia.com, {rangeen.basu.roy.chowdhury, randy.widialaksono, sungkwan.ku}@intel.com

***Abstract*—A single-ISA heterogeneous multi-core processor(HMP) [2], [7] is comprised of multiple core types that all implement the same instruction-set architecture (ISA) but have different microarchitectures. Performance and energy is optimized by migrating a thread's execution among core types as its characteristics change. Simulation-based studies with two core types, one simple (low power) and the other complex (high performance), has shown that being able to switch cores as frequently as once every 1,000 instructions increases energy savings by 50% compared to switching cores once every 10,000 instructions, for the same target performance [10]. These promising results rely on extremely low latencies for thread migration.**

Here we present the H3 chip that uses 3D die stacking and novel microarchitecture to implement a heterogeneous multi-core processor (HMP) with low-latency fast thread migration capabilities. We discuss details of the H3 design and present power and performance results from running various benchmarks on the chip. The H3 prototype can reduce power consumption of benchmarks by up to 26%.

I. Introduction

Single-ISA heterogeneous multi-core processors [7], [2] (HMP) have become commonplace in the last few years as architects aim to improve performance-per-watt of processors. In the past, computer architects have been able to deliver significant improvements in CPU performance by leveraging advances in process technology. But as speed improvements from technology scaling slow down with the end of Dennard scaling, processor architects are trying to eke out as much performance as possible from microarchitectural techniques and architectural improvements. Historically, architects have built a single processor design that had overprovisioned resources to try and improve performance for most applications. However this is not energy efficient since the applications that do not benefit from the overprovisioned logic, have to pay the energy tax. Not only do different applications have different characteristics, a single application can have distinct execution phases that behave differently. Processor specialization solves this energy efficiency wall by matching processor designs to application behavior. An HMP contains a few different cores with distinct designs. All core types can execute the same programs, but they do so at different performance and energy points. Each application phase is run on the core that is most appropriate for it, migrating between cores as the application transitions between phases. Although quite beneficial, the migration process is subject to large overheads due to transfer of a large amount of architectural state between the cores.

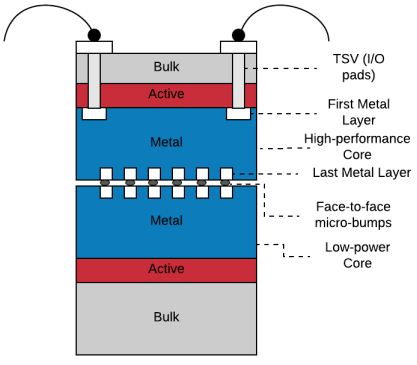
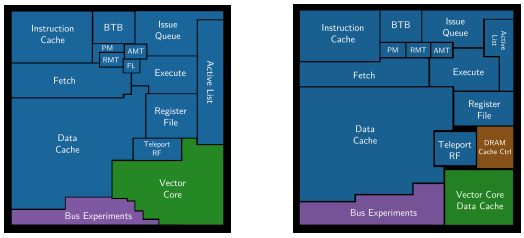


Figure 1: The two core dies are stacked using micro-bumps. Pads are connected to contacts using TSVs.



(a) Top Die (b) Bottom Die

Figure 2: Floorplan of the H3 Chip

A. Benefits of Fast Thread Migration

Previous work in the area of HMPs have noted the importance of reducing the thread migration latency in

Table I: Quantitative and qualitative comparison with the state of the art.

Design	Thread migration latency	Distinct (separate) cores	Asynchronous (GALS)	Register-based ISA	Evaluation Methodology
ARM big.LITTLE [6]	20,000 cycles	Yes	Yes	Yes	Real system
Composite cores [9]	$<$ 32 cycles	No (shared frontend and data cache)	No	Yes	C++ Simulator
Execution Migration Machine [8]	$<$ 100 cycles	Yes	No	No (stack-based ISA)	RTL
Our approach, FRM in H3	$<$ 10 cycles	Yes	Yes	Yes	Chip and RTL

order to improve performance and energy efficiency of HMPs [10], [3]. The rationale behind occasionally running the workloads on a low-power, low-performance core is to reduce energy consumption with a negligible or no performance loss compared to always running on the high-performance core. The utilization of the low-power core increases significantly when the thread migration interval is small. This is due to the ability to exploit short phases that do not benefit from the high-performance core. Such fine-grained thread migrations are more sensitive to migration overheads as the opportunity to exploit the short phases is lost if the overhead is high. Current HMP designs rely solely on system software for thread migration, incurring kernel overheads and large time penalties of moving register values through the deep hierarchy of caches. The architectural state of a thread comprises of register state and memory state. Migration overheads mainly come from the following two sources:

- *Transferring architectural register state:* Architectural register state includes the data registers, control registers, performance counters, etc. In the traditional save-restore mechanism, these registers are saved to the main memory before migrating the thread to another core, where these registers are restored from the main memory. This is very expensive both in terms of time and energy.
- *Migration induced misses and mispredictions:* After migration, if the core misses a cache block that existed in the previous cache but not in the current cache, it is called a migration-induced miss. This problem is exacerbated when multiple threads are active on the processor. Other microarchitectural structures like the branch predictor, take time to get trained as well, leading to bad speculation and slowdown of program execution.

B. Reducing Thread Migration Cost by 3D-Stacking

We investigate microarchitectural techniques to reduce the thread migration latency in HMPs. We leverage 3D-die stacking for reducing thread migration cost by using two key features:

- *Fast Register Migration (FRM):* This feature allows instantaneous swap of architectural register state (data registers, control registers, performance counters, etc.) between the two stacked cores. Every storage cell of every architectural register is directly connected to its counterpart in the other core through a face-to-face via (Figure 4), allowing instantaneous swap of cell contents.
- *Cache-Core Decoupling (CCD):* This feature allows one CPU core to directly access the L1 caches from the other core, a remote cache access (Figure 5), thus avoiding cache flushes and eliminating migration-induced misses. This relies on the low latency of the inter-die connectivity to achieve the strict latency requirements of L1 cache accesses.

The H3 chip makes two very important contributions. *Firstly*, it demonstrates the feasibility of logic-on-logic die stacked processor designs. Designing for 3D die stacking is challenging but we effectively solve these challenges to create this prototype chip. *Secondly*, it explores microarchitectural techniques to utilize the massive amount of inter-die connectivity provided by 3D stacking, to improve performance and efficiency of processors. Section III provides some key details of these techniques.

The key contributions made by this paper are:

- Explain in detail the microarchitecture of a 3D-stacked HMP.
- Measure the performance and power of various microbenchmarks on the chip.
- Discuss potential use-cases of a processor with fast thread migration capabilities.

The rest of the paper is organized as follows. We discuss related work in Section II. Section III describes the microarchitecture of the H3 prototype chip. Section IV describes our validation methodology and presents various power and performance measurements from the chip. Section V discusses some potential use-cases of H3. Finally, we conclude the paper in Section VI.

II. Related Work

Table I compares latency and features of competing approaches. ARM big.LITTLE is too slow for fine-grain adaptation. The other three approaches are all sub-100-cycle. Composite cores does not use truly distinct cores, uses one clock domain, and is demonstrated with C++ simulation. Execution Migration Machine impressively features a 110-core chip, but uses a stack-based ISA to streamline the amount of state transferred and migration

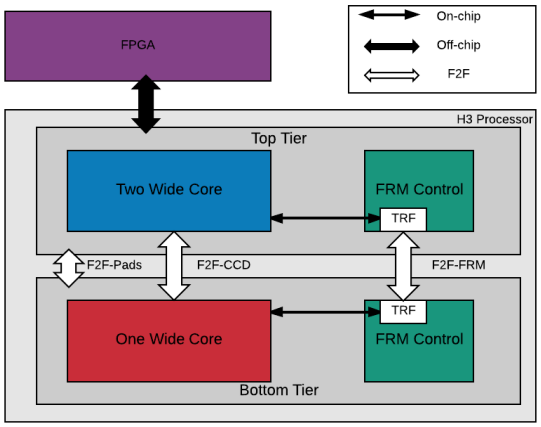


Figure 3: A block diagram of the H3 chip showing the two dies, cores, various interfaces, and the FPGA testbench.

latency depends on network-on-chip congestion. Reported results do not appear to be from the chip itself, yet. This paper, and the chip measurements contained herein, are the culmination of the H3 project, the evolution of which is chronicled in a series of works. The rationale for H3 is discussed at length in Rotenberg et al. [10]. That paper also describes a 2D version of H3, a dry run without 3D stacking to flush out the microarchitecture and physical design flow. The 2D prototype is described after tapeout but prior to receiving silicon. Measurements and lessons from the 2D prototype are presented in Forbes et al. [4] [5]. Alternative strategies for FRM, independent of 2D versus 3D considerations yet cognizant of GALS HMPs, are explored in Forbes et al. [3]. In his MS thesis, Srinivasan [11] discusses at length the microarchitecture, RTL design, and RTL/gate-level verification of H3 Phase II (the 3D version presented herein) after tapeout and before silicon. The physical design effort of the 3D chip are described in Widialaksono et al. [12]. To the best of our knowledge, the H3 chip is the first prototype of a 3D-stacked HMP in academia.

III. H3 Chip: An experimental 3D-stacked HMP

The H3 processor is a logic-on-logic 3D stacked single-ISA heterogeneous multicore architecture. Our prototype contains two OOO cores with varying capabilities, "Two-wide" core and "One-wide" core, named based on their peak instruction fetch rates. The cores are stacked on top of each other using micro-bump based face-to-face (F2F) die stacking technology (Figure 1), from Ziptronics™. The floorplan of the top and bottom dies are shown in Figure 2. An overview of the H3 architecture is shown in Figure 3. The chip was designed using Global Foundries 130nm process and targets a maximum operating frequency of 65 MHz. Although the older process is not ideal for performance, this is the only 3D process we could get access to. Table II shows that various physical design metrics of the H3 chip.

Table II: Physical design metrics of the 3D processor stack.

Parameter	Top Die	Bottom Die
Die Dimensions (sq mm)	3.92 x 3.92	3.92 x 3.92
Standard Cells	886,361	678,854
Memory macros	17	17
Nets	482,479	328,535
Inter-tier F2F signal nets	6,077	
Inter-tier power vias	30,796	

The H3 cores were designed using the FabScalar [2] toolset and uses the PISA instruction set architecture (ISA) [1]. The two-wide core, on the top die, is a high performance core that has bigger OOO structure sizes, to extract as much ILP as possible from the instruction stream. The bottom die has the one-wide core, a low power core with smaller structure sizes more suitable for phases that have low ILP and hard-to-predict branches, which typically don't benefit much from aggressive speculation. Table III lists the various microarchitecture details of the two cores.

Table III: Core types in the 3D processor stack.

Parameter	High-Performance (Top Die)	Low-Power (Bottom Die)
Frontend Width	2	1
Issue Width	3	3
Pipeline Depth	9	9
Issue Queue Size	32	16
Phy Reg. File Size	96	64
Ld/Ste Queue Size	16/16	16/16
Reorder Buffer Size	64	32
L1 I-Cache	private, 4 KB, 1-way, 8 B block, 1 cycle	
L1 D-Cache	private, 2 KB, 1-way, 16 B block, 1 cycle	

Besides the cores, the chip also contains the Fast Register Migration (FRM) controller logic. As shown in Figure 3, there are two types of interfaces in the H3 processor.

- *The off-chip interfaces (I/O pads)* that connect the H3 processor to the FPGA-based test harness and are used for interfacing with the L2 cache and memory controller implemented in the FPGA. These interfaces include clock, reset, and other important control signals. Since the two dies are connected to each other by stacking the top metal layers (Figure 1), the H3 chip has pads only on the top-die. The pads are custom designed to use rectangles in the first metal layer (M1), which are connected to bondpoints through TSVs.
- *The F2F interfaces* that connect the tiers. The bottom tier has all of its off-chip buses and control signals routed to pads on the top tier via the F2F-Pads interface. The two FRM units use the F2F-FRM interface for swapping register contents. The F2F-CCD interface is used for the inter-cache connectivity for implementing Cache-Core Decoupling (CCD).

A. Fast Register Migration (FRM)

Fast Register Migration (FRM) is a mechanism for low-latency transfer of architectural register state between the

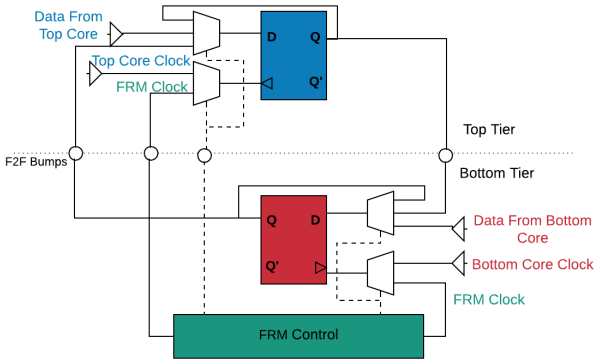


Figure 4: Microarchitecture of Fast Register Migration.

two stacked cores. It is made possible by using a dedicated structure, called the **Teleport Register File (TRF)**, that is used to consolidate the architectural register state. Face-to-face buses connect each bitcell of the TRF to the other core's TRF as shown in Figure 4. When a thread migration is initiated, special instructions are used to transfer the register state of the thread to the TRF, which is followed by a bulk swap of the entire TRF. Our swap mechanism is a globally-asynchronous locally-synchronous (GALS) design. The *cores* and *FRM control* units operate in different clock domains. This obviates the need for 3D clock tree synthesis and allows the two heterogeneous cores to have different frequencies, increasing diversity. Before a swap can happen, the FRM control switches both TRFs to use the FRM clock to avoid asynchrony during the swap. This requires the clock network feeding the TRF flops to be equipped with a clock mux to choose between core clock and FRM clock. There is also an input mux as TRFs are written by the core before a migrate and by the other tier's TRF during the transfer. The select lines for these muxes come from the FRM control unit. Based on our measurements, our chip can swap architectural register state between the cores in less than 10 cycles, after the cores have quiesced and architectural state has been consolidated into the TRF.

As shown in Figure 3, both tiers have an FRM control unit. There can be only one master control unit, however, to select the clock and data inputs to TRF flops, as shown in Figure 4. The top tier's FRM control unit is the master.

A migration is initiated by either an external interrupt or a migration instruction (more about these types are discussed later). The external interrupt is received by the master FRM control unit. A migration instruction may execute in either core: if the top core, it signals the master directly; if the bottom core, it signals the slave FRM control unit which signals the master FRM control unit. All signaling between cores and their adjacent FRM control units, and between master and slave FRM control units, is implemented with an asynchronous handshaking protocol.

The steps to perform a migration are as follows, for the case of an external interrupt. The master FRM control unit receives the interrupt and handshakes with the slave FRM control unit. Then, the FRM control units send suspend interrupts to their respective cores. This invokes a suspend interrupt service routine (ISR). The suspend ISR is responsible for consolidating all architectural register state into the TRF using special move-to-TRF (MTTRF) instructions. The suspend ISR ends with a special barrier instruction, which, when retired from the core, signals the adjacent FRM control unit that the core is suspended and the TRF is ready. Note that suspension of the cores happens on both tiers in parallel and asynchronously. Once both barrier signals have reached the master FRM control unit (one by way of the slave FRM control unit), the master switches the TRFs' clocks to its own clock and swaps the two TRFs in a single cycle via the wide F2F-FRM interface. Then, the master FRM control unit signals the slave FRM control unit to resume. Both master and slave send resume interrupts to their respective cores, which invokes the resume ISR on both cores. The resume ISR uses special move-from-TRF (MFTRF) instructions to copy the architectural register state to the core's physical registers. The suspend and resume ISRs are shown in Listing 1.

Listing 1: Suspend and Resume ISRs

Suspend ISR	**Resume ISR**
Execution suspended	**mftrf** $r1
in source core	**mftrf** $r2
mttrf $r1	.
mttrf $r2	.
.	**mftrf** $r31
.	**mftrf** $r32
mttrf $r31	**eret**
mttrf $r32	*Execution resumes*
barrier	*in destination core*

There are two scenarios that can occur in thread migration: (a) a single thread migrates from an active core to an idle core, and (b) two threads swap cores. The master FRM control unit can distinguish the first scenario and suppress resuming operation on the newly idle core.

H3 supports two ways to initiate thread migrations that enable a wide range of scheduling mechanisms:

- *Hardware-triggered migrations (hard migrations)* These are caused by an external interrupt, and are initiated by an external scheduler based on the information provided by performance monitors within the cores. These migrations can occur between two active cores or one inactive and other active core. When two threads are running, a hard migration will suspend both threads, swap the architectural states, and simultaneously resume the threads on the new cores.
- *Software-triggered migrations (soft migrations)* These

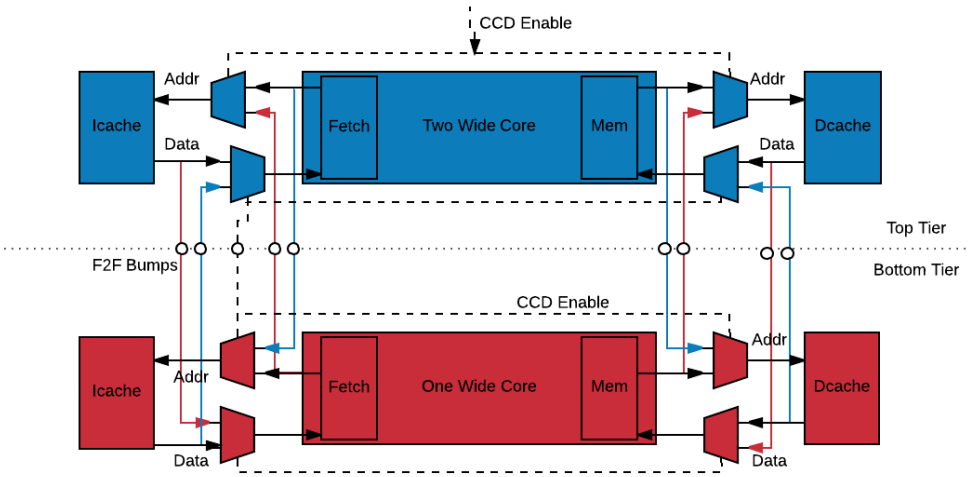


Figure 5: Microarchitecture of Cache Core Decoupling.

migrations are initiated by the thread itself using a special migrate instruction. The suspend and resume ISRs are not invoked by the hardware, rather they are instructions inlined in the user-level program. Soft migrations can only occur between one active and inactive core. It offers more control to the executing program to migrate at specific phase transition points, determined by the programmer or by the compiler. It also allows the compiler to optimize the suspend and resume instructions to move only live registers, further reducing the thread migration latency.

B. Cache-Core Decoupling (CCD)

Cache-Core Decoupling (CCD) reduces the overhead caused by migration-induced cache misses. CCD enables each core to access the other core's L1 instruction and data caches. Figure 5 shows the CCD logic located at the instruction fetch and load-store units of the cores, that connect each core to the other core's instruction and data caches, respectively. Two muxes, one for address and one for data, at each core's instruction and data caches, select between local core access and F2F wires from the remote core. CCD can be enabled or disabled with an external CCD-enable pin. When disabled, only the local core has access to the caches of a tier, so a thread experiences migration-induced misses when it switches cores. When enabled, both the local and remote cores have access to the caches of a tier, but not at the same time: access to a tier's caches changes between the local and remote cores as the thread, which is pinned to the tier's caches, migrates.

If enabled, CCD allows the thread to access the same cache before and after migration. This allows the threads to freely migrate from one core to another without having to switch caches. This feature is also supported when two threads are running and migrating between the two cores. The threads continue to access the same cache irrespective of which core it is running on throughout its execution. CCD also enables using the L1 cache best suited for the program's working set if the cores have different L1 cache parameters. A thread can be mapped to the best core-cache pair throughout its execution.

The benefit of CCD – mitigating migration-induced misses – must be weighed against its potential drawback, which is adding delays of two muxes (address and data) in the hit path of the local core. Our analysis from H3 synthesis shows that the two sub-paths, (1) address to D-cache and (2) D-cache data to pipeline, increase by 17% and 7%, respectively. While this increase does not include the D-cache access itself, the hit path increase is not trivial. In a commercial high-frequency design, CCD would have to be evaluated with all the facts about the division of logic among pipeline stages of the cache path, opportunities to leverage existing muxes, cycle-stealing opportunities, etc.

CCD also presents a timing challenge for a remote core accessing the cache. The remote core is in a different clock domain, hence, synchronizing queues are required at both tier crossings: one for address received from the remote core and one for data sent to the remote core. This introduces three timing challenges. First, synchronizing queues may add one or several cycles to the remote core's hit path. Second, now the core must have a reconfigurable hit latency with respect to waking up dependents. Third, the remote latency may vary owing to non-determinism of synchronizing domain crossings. The last two challenges may be addressed by always treating the remote cache access as a "fast miss path" instead of as a "hit path", but the first challenge remains. In short, the extra cycles experienced for *every* access by the remote core, must be weighed against the migration-induced misses that are avoided.

Note, in the H3 prototype, we did not implement synchronizing queues for CCD. Instead, a precondition for enabling H3's CCD is that the same clock source must feed both tiers' clock pins, for synchronous operation only.

Without CCD, there are two types of migration-induced misses. First, compulsory, capacity, and conflict misses that occur in one tier's cache, may *recur* in the other tier's cache, owing to applying the same working set redundantly to two caches. Second, a write to a block in one cache invalidates the copy of that block in the other cache [1]. If the thread migrates back and re-references the invalidated block, it experiences a migration-induced invalidation miss.

IV. H3 Validation and Results

Our chip is packaged in a 100-pin ceramic QFP package. The packaged H3 chip is housed in a compatible 100-pin socket on a custom-designed 4-layer PCB. The PCB attaches to a Xilinx ML605 FPGA board via its FMC LPC connector. Our FPGA testbench implements the L2-cache controller, instruction and data memory and associated logic. A RHEL-6 Linux PC acts as the user interface for our test setup. A custom Command Line Interface (CLI) client on the Linux machine, called the Console, provides an interactive shell to the user. Since the block RAMs in our FPGA are limited in capacity, we can only fit microbenchmarks with small memory footprint in the FPGA, hence microbenchmarks were used for performance and power measurements on the H3 chip. Other results presented in this section using SPEC 2000 integer benchmarks were from RTL simulation of the chip along with the test environment.

We use frequency and voltage independent metrics for reporting power and performance since the prototype is built using an older process and runs at a relatively low frequency. We use Instructions Per Cycle (IPC) for performance and Dynamic Capacitance (Cdyn) for power. The H3 cores have basic performance counters in them, that can be read from the FPGA testbench. We used these performance counters to measure IPC of benchmarks. In order to measure the power consumed by a benchmark, we used a Data Acquisition System to sample the current supplied to the H3 chip while the benchmark is running. We then calculate an average current across the entire benchmark run. We use the measured average current, the supply voltage, and operating frequency to calculate the Dynamic Capacitance (Cdyn) for each benchmark using the formula:

$$Cdyn = Current/(Voltage * Frequency)$$

Cdyn is a voltage and frequency independent metric and measures the inherent switching capacitance of the core for a benchmark.

[1]H3 implements write-through L1 caches to a shared L2 cache in the FPGA testbench. The L2 cache tracks blocks residing in the L1 caches, and a write-through to an L1 cache causes the L2 to invalidate the copy in the other L1 cache.

A. Fast Register Migration Latency

This experiment shows the variation in fast register migration latency with different number of architectural registers transferred during migration. We used a microbenchmark that repeatedly migrates the thread between the cores. The microbenchmark consists of only a soft migrate loop which uses **suspend ISR** for consolidation of architectural state, a **migrate** instruction to migrate the thread after consolidation, and **resume ISR** to resume the thread. Thus this test measures just the FRM latency, which includes execution latencies of the suspend and resume ISRs. For this study, the FRM latency for 1 migration is identified by taking an average over 100 migrations. The number of registers to be transferred during each migration is varied manually from 1 to 34 (architectural registers in PISA ISA) and the test repeated until we have all the data points.

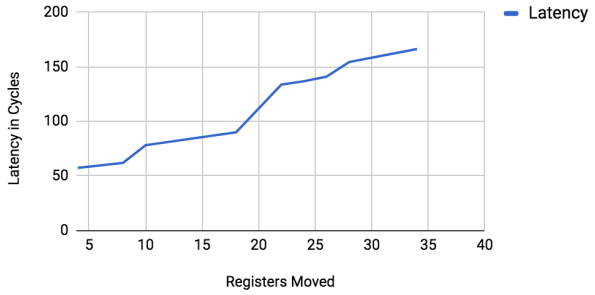


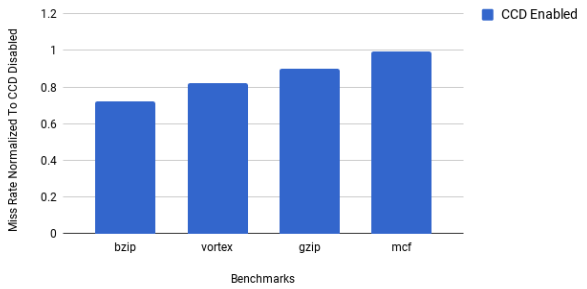
Figure 6: Register migration latencies for different number of architectural registers transferred.

As seen in Figure 6, maximum latency for moving all 34 integer registers is around 160 cycles. On an average it takes around 100 cycles for transfer of architectural state from source to destination core. The latency mainly comes from startup delay for filling the pipeline stages after a migrate, and executing the suspend and resume ISRs. The swap takes less than 10 cycles, which includes clock switching and register swap, irrespective of the number of registers moved. This experiment was also successfully done with the cores running at different clock frequencies thus validating the GALS design of FRM.

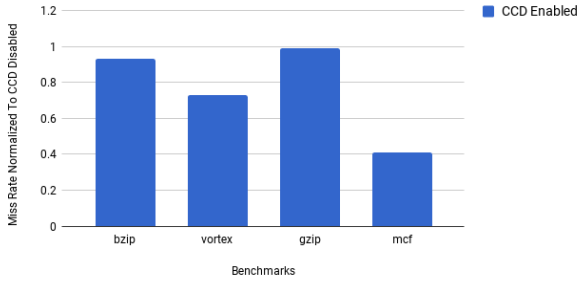
B. Cache-Core Decoupling

This section measures the impact of CCD on cache miss rates when there are migrations. The SPEC benchmarks used in this study were run for 10M instructions with an external migration interrupt every 16K cycles. Thus, the migrations are at arbitrary points in the program execution instead of being phase-driven. The cache miss rate with CCD, normalized to that without CCD, is shown in Figure 7.

As the results show, the overall miss rate for both instruction and data cache goes down significantly with CCD enabled when compared with baseline where CCD is disabled. This comes from the benefit of accessing the same cache throughout the program execution. Apart from the



(a) Data cache miss rates



(b) Instruction cache miss rates

Figure 7: Cache miss rates for SPEC2K INT simpoints with CCD enabled.

misses caused by limits of our cache capacity, there are no additional migration induced misses when CCD is enabled. The reduced I-cache miss rate with CCD is particularly crucial as in the absence of CCD, the cold I-cache misses after migration usually erode the performance/power benefits of a migration. The D-cache miss rate is high when CCD is disabled. This is attributed to cold cache misses during the initial few migrations and misses caused by invalidations as a result of stores to the cache block when the program is running on the other core. CCD eliminates both these causes showing significant reduction in data cache miss rate. The I-cache miss rate with CCD disabled is not as bad as D-cache miss rate. This is because of the absence of invalidation triggered misses in case of I-cache, and also since these programs tend to execute the same loop body, they tend to hit in I-cache even when regularly migrating between cores. This can be seen in case of bzip and gzip, as CCD disabled performs almost as same as CCD enabled. On the other hand, mcf and vortex benefits from CCD as it eliminates initial cold cache misses and also migration induced misses.

Figure 8 shows IPC results for the same migration experiment. The first and second bars are IPCs when running the benchmark only on the 2-wide core or only on the 1-wide core (no migrations), respectively. The next two bars show IPC with migrations, for two different cache configurations: CCD enabled, CCD disabled. As seen from the Figure 8, the performance for CCD enabled is close to running on the cores without migration. The CCD disabled configuration shows performance degradation because of migration-induced cache misses during migrations.

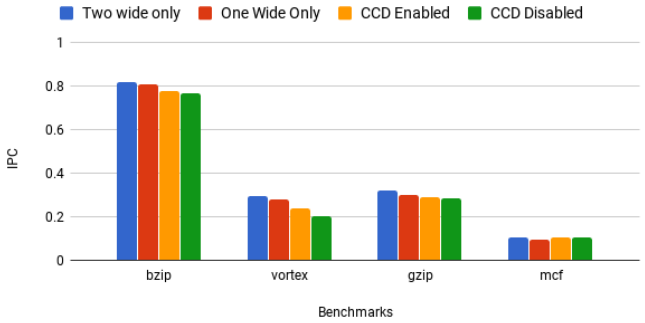


Figure 8: IPC for different H3 configurations.

C. Performance and Power of H3

We used a several microbenchmarks to measure the power and performance of the H3 chip. Table IV shows the IPC, Cdyn and Power of the various microbenchmarks on the two cores in the H3 chip.

As a general trend, the Cdyn of the one-wide core is 30% lower than that of the two-wide core for almost all benchmarks. However, the IPC for most benchmarks is within 10% of the two-wide core. Some benchmarks such as *binary search* (IPC difference of only 3%) have very similar performance on both cores. These benchmarks have many data-dependent branches and loads and tend to not gain much performance from aggressive speculation. Such benchmarks are poster children for heterogeneous multi-core processors such as H3. A real workload will typically use many of the kernels together and will have distinct phases, thus benefiting heavily from H3-style architectures. H3 will save close to 40% power with negligible performance loss for such workloads.

To demonstrate the benefit of fast thread migration in a heterogeneous multi-core processor, we craft a mini workload that combines two kernels, load-accumulate and conditional array reduction. The load-accumulate kernel continuously loads from the same location and does an arithmetic operation on the value. This behaves similar to an application that is polling a memory location and doing some operation on it. The second kernel conditionally reduces an array to a single value. It calculates the difference between the sum of odd values and the sum of even values in an array. The condition to determine

Table IV: IPC, Cdyn, and Power (at frequency of 10MHz) for various microbenchmarks

	IPC		**Cdyn (nF)**		**Power (mW)**	
Benchmark	**2W**	**1W**	**2W**	**1W**	**2W**	**1W**
Bubble Sort	0.97	0.87	1.04	0.72	16.25	11.25
Binary Search	0.63	0.61	0.96	0.64	15.00	10.00
LFSR	1.04	1.00	1.12	0.80	17.50	12.50
Prime Numbers	0.97	0.87	1.20	0.80	18.75	12.50
Reduce an array	1.27	0.97	1.20	0.88	18.75	13.75

odd or even, is dependent on the actual value contained in the array location and nearly impossible to predict. Both kernels retire nearly equal number of instructions.

Figure 9 plots the IPC and Cdyn of the mini workload in three scenarios: Running on two-wide core only, one-wide core only, and running in heterogeneous mode. In the heterogeneous mode, we execute the first kernel on the two-wide core, migrate, and execute the second kernel on the one-wide core. The first kernel has higher IPC hence can benefit from the two-wide core's higher fetch rate. The second kernel has unpredictable data-dependent branches and lots of cache accesses, leading to low IPC. Two-wide core has slightly higher IPC than one-wide in this phase, but very high Cdyn. Hence the one-wide is more suitable for the second kernel. As seen from the figure, the workload exhibits 26% lower Cdyn and loses 16% IPC when running in heterogeneous mode compared to always running on two-wide core. The Cdyn observed is around 8% greater than always running on one-wide core but shows 20% higher performance.

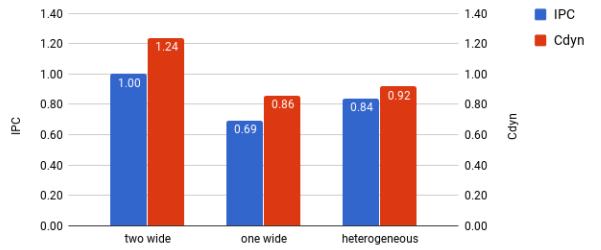


Figure 9: IPC and Cdyn of mini workload in one-wide, two-wide, and heterogeneous modes.

V. Future Work

H3 provides an low-latency fast thread migration capability that can be used for various techniques which were not possible before. One such use case would be to use FTM for fast phase detection using a couple of techniques. In one technique, a thread can be spawned on both cores to identify the best core for a discrete quantum of time. The slower running thread can then be stopped and later the procedure is repeated for next quantum. The latency for forking threads is greatly minimized by FTM in H3. Alternatively, one can choose not to fork threads but instead migrate threads and sample performance on the two cores, then schedule phases in future based on sampled data. Using FTM as a fast sampling, forking methodology will be explored in a future work.

VI. Conclusion

In this paper, we presented the H3 chip, a novel 3D logic-on-logic stacked HMP. It uses novel microarchitectural techniques, such as Fast Register Migration (FRM) and Cache Core Decoupling (CCD), that utilize the inter-die connectivity to improve performance. The performance benefits from these techniques have been explored with experiments using the prototype chip. FRM latency data shows that architectural register state can be transferred between cores in as low as 100 cycles. CCD has been shown to reduce migration-induced misses significantly, enabling frequent migrations to save power. H3 significantly benefits benchmarks with distinct phases, decreasing power by about 26%.

VII. Acknowledgements

The H3 project is supported by a grant from Intel.

References

[1] D. Burger, T. Austin, and S. Bennett, "Evaluating future microprocessors: The simplescalar toolset," University of Wisconsin-Madison, Tech. Rep. CS-TR-1308, 1996.

[2] N. K. Choudhary, S. V. Wadhavkar, T. A. Shah, H. Mayukh, J. Gandhi, B. H. Dwiel, S. Navada, H. H. Najaf-abadi, and E. Rotenberg, "Fabscalar: Composing synthesizable rtl designs of arbitrary cores within a canonical superscalar template," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA-38, June 2011.

[3] E. Forbes and E. Rotenberg, "Fast register consolidation and migration for heterogeneous multi-core processors," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, Oct 2016.

[4] E. Forbes, Z. Zhang, R. Widialaksono, B. Dwiel, R. B. R. Chowdhury, V. Srinivasan, S. Lipa, E. Rotenberg, W. R. Davis, and P. D. Franzon, "Under 100-cycle thread migration latency in a single-isa heterogeneous multi-core processor," in *Poster session of Hot Chips*, 2015.

[5] E. Forbes, R. B. R. Chowdhury, B. Dwiel, A. Kannepalli, V. Srinivasan, Z. Zhang, R. Widialaksono, T. Belanger, S. Lipa, E. Rotenberg *et al.*, "Experiences with two fabscalar-based chips," in *6th Workshop on Architectural Research Prototyping (WARP-6)*, 2015.

[6] P. Greenhalgh, "big.little processing with arm cortex-a15 & cortex-a7," 2011. [Online]. Available: https://web.archive.org/web/20131017064722/http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf

[7] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-isa heterogeneous multi-core architectures: the potential for processor power reduction," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, 2003.

[8] M. Lis, K. S. Shim, B. Cho, I. Lebedev, and S. Devadas, "Hardware-level thread migration in a 110-core shared-memory multiprocessor," in *2013 IEEE Hot Chips 25 Symposium (HCS)*, Aug 2013.

[9] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke, "Composite cores: Pushing heterogeneity into a core," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45, 2012.

[10] E. Rotenberg, B. H. Dwiel, E. Forbes, Z. Zhang, R. Widialaksono, R. B. R. Chowdhury, N. Tshibangu, S. Lipa, W. R. Davis, and P. D. Franzon, "Rationale for a 3d heterogeneous multi-core processor," in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, 2013.

[11] V. Srinivasan, "Phase ii implementation and verification of the h3 processor," 2015. [Online]. Available: https://repository.lib.ncsu.edu/handle/1840.16/10828

[12] R. Widialaksono, R. B. R. Chowdhury, Z. Zhang, J. Schabel, S. Lipa, E. Rotenberg, W. R. Davis, and P. Franzon., "Physical design of a 3d-stacked heterogeneous multi-core processor," in *2016 IEEE International 3D Systems Integration Conference (3DIC)*, Nov 2016.