Fast Register Consolidation and Migration for Heterogeneous Multi-core Processors

Elliott Forbes†
Department of Computer Science
University of Wisconsin – La Crosse
eforbes@uwlax.edu

Eric Rotenberg
Department of Electrical and Computer Engineering
North Carolina State University
ericro@ncsu.edu

***Abstract*—**

Single-ISA heterogeneous multi-core processors have been demonstrated to improve the performance and efficiency of general-purpose workloads. However, these designs leave some performance on the table due to the common assumption that the cost of migrating a program from one core to another is high. This high cost is due to the reliance on the operating system for a migration via a context switch. Many programs exhibit very fine-grained changes in behavior. A high-cost thread migration requires infrequent migrations, as the migration penalty must be amortized. In this paper, we investigate the impact that thread migrations impose on single-ISA heterogeneous systems.

To realize these performance and efficiency gains, we consider a design space of possible, realistic hardware thread migration schemes. The schemes implement a system in which the operating system is allowed to assign a thread at the granularity of pairs of cores, but then a hardware mechanism can freely move the thread between the cores at-will without operating system involvement. The focus of this work is on migrating program register state. The migration of register values is complicated by the physical register file, which may store logical register values in non-contiguous entries. Additionally, we assume that the participating core pairs operate at independent clock frequencies, further complicating the exchange of register values.

We identify three sources of overhead when implementing hardware thread migration: the power consumed by the additional hardware even when no migration is being performed, the latency of the actual migration, and the energy consumed during a migration. We evaluate several hardware alternatives in synthesizable Verilog, then use static timing analysis of the synthesized netlist to accurately measure these overheads in a 45nm process technology. Depending on the implementation, the power overhead is as low as 0.4% of the total core power. The latency can be pushed to as little as 33 cycles with a migration energy cost of less than 77nJ.

To further demonstrate the feasibility of low cost thread migrations, we fabricated one of the hardware thread migration schemes in a prototype processor. The processor consists of a pair of heterogeneous out-of-order cores, with hardware thread migration between the pair. We show that the design has a migration latency of between 50 and 103 cycles, closely matching our simulation results.

I. Introduction

As transistor scaling has slowed and we can no longer rely on smaller transistors equating to lower power, we continue to seek architectural innovations to improve the performance and energy efficiency of processors. One promising direction is to specialize processor cores to programs by employing multiple cores that implement the same instruction set, but with different internal microarchitectures. These single-ISA heterogeneous multi-core processors were proposed by Kumar et al. [8] [10] and have proven effective at achieving improvements in performance and efficiency by using the most suitable core to execute a given program. Our work (as well as others [14]) shows that additional benefit in heterogeneous multi-core architectures can manifest if the program can move between the cores as rapidly as possible. This recognizes the fact that programs often change their behavior frequently, and the program should move to the most suitable core to match this frequency.

Prior work has shown the benefits of decreasing the penalties associated with cache misses [2] [13] when migrating from one core to another. This paper analyzes the impact of migrating register values. Typically, a thread migration is handled by the operating system via a context switch. This involves saving the program register values with stores to a process control block, then scheduling the program on another core, using loads from the process control block to restore the program register values on the new core. This process can take on the order of thousands of cycles [7] [12]. Even if the operating system skips the scheduling step, the store/load pairs will likely cache miss, and thus still require hundreds of cycles to migrate.

Similar to [15], our approach is to provide an abstracted view of the cores to the operating system, where the operating system schedules programs to core pairs, but then once assigned, programs can freely move between the constituent cores. This allows us to streamline the movement of register values to be as low-latency and energy-efficient as possible.

This work proposes register migration schemes that are part of processor cores that implement an out-of-order execution model using physical register files (PRFs) that contain both committed and speculative register values. This style of architecture is regularly used in modern processors, and presents a challenge to quickly accessing register values. Since the PRFs hold both committed and speculative values, an intermediate step is required to consolidate only the committed values for migration to another core. While other styles of architectures exist that may avoid the need for this consolidation, we focus on these large PRF architectures specifically because they are commonly used in modern processors. We also require our migration hardware to allow migration between cores

†Author contributed to this paper while at North Carolina State University.

978-1-5090-5142-7/16/$31.00

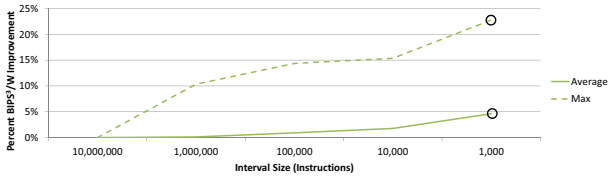


Fig. 1. Impact of interval size on benchmark efficiency.

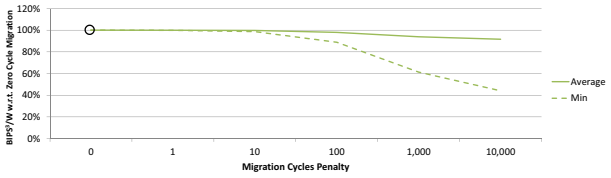


Fig. 2. Impact of migration cycles on benchmark efficiency.

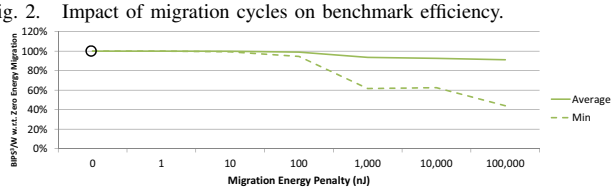


Fig. 3. Impact of migration energy on benchmark efficiency.

that operate at different frequencies, since frequency is a key differentiating factor in heterogeneous cores.

To show the potential of hardware thread migration and to understand the impact that energy and latency overheads have on program efficiency, we perform a series of experiments. We augmented our detailed processor simulator to report performance and energy statistics at regular instruction intervals (details of the simulator appear in Section IV). This way, performance for each interval can be compared across different core configurations. This also allows statistics from consecutive intervals to be aggregated to form larger intervals and allows for adding arbitrary cycle and energy penalties to reflect migrations.

Figure 1 shows the impact of changing the granularity of interval. The coarsest granularity appears on the left of the graph, and represents the baseline efficiency when assuming a dual heterogeneous core pair wherein each benchmark is pinned to the better of the two cores, determined *a priori*. This data point is roughly 15% to 20% above the efficiency of a single homogeneous core. Looking at the right side of the graph, we can see that as we consider smaller interval sizes, we gain more efficiency. The line labeled "max" plots the efficiency of the benchmark that has the highest improvement at that granularity. We circle the finest granularity data point of this graph to signify that this data point is the most ideal in our experiments and serves as a new baseline for the remaining graphs in this section. We can see from this graph that we can expect an average improvement of about 5% over that of coarse grain heterogeneous (no migration) cores.

Splitting benchmarks into intervals allows us to add arbitrary cycle and energy penalties for performing a migration. Figure 2 applies a range of cycle penalties for a migration assuming the finest interval size and no migration energy overhead. Likewise, Figure 3 shows the impact of adding various migration energy penalties. Both of these graphs show the efficiency relative to the ideal case with no migration overheads (circled). For each interval of each benchmark, we decide to migrate from the current core to the opposite core if the efficiency of the interval on the opposite core plus the cost of a migration is better than the efficiency of staying on the current core. The curves labeled "min" show the efficiency of the benchmark that was impacted the most. There is a knee in the cycle penalty curve between a 10 and 100 cycle migration penalty. A knee also exists in the energy penalty curve at about 100nJ. If a particular hardware migration mechanism is to retain as much benefit as possible, then the migration penalties should stay within these bounds.

Prior work [8] [9] [10] [11] [21] has explored the benefits of heterogeneous multi-core processors in depth. Thus, the focus of this work is on evaluating hardware register consolidation and migration alternatives rather than an exhaustive evaluation of heterogeneous multi-cores. Section II highlights the prior related work. We propose a design space of alternative implementations in Section III. Our evaluation methodology is detailed in Section IV, and results are reported in Section V. Finally, the design alternative with highest overall benefit has been fabricated in a prototype chip, and we report the results of testing the migration hardware in Section VI.

II. Related Work

Prior work has shown the benefits, and have explored ways of reducing the penalties, of thread migrations. Constantinou et al. [5] introduces the notion that migrations can induce penalties such as cache misses that would not have occurred if the migration had not taken place. They suggest that migrations should be infrequent because of these migration-induced overheads.

The Execution Migration Machine [13] project proposes a many-core paradigm where programs migrate to a processor node that holds the data on which the program will operate instead of moving data needed by a program to the processor node on which it is executing. The key insight is that program register state is smaller than the cache working set so it is more efficient to move the program than it is to move the data. They recognize that moving register values is not an insignificant overhead, and propose using a stack-based instruction set to reduce the number of registers that must be transferred between nodes. However, this impacts the implementation of the cores, and limits the applicability of their approach for register migration.

Composite Cores [14] identifies the importance of fast program state migration for use in heterogeneous cores. Their approach implements heterogeneity within a single core by having two backends that share a single frontend. One backend is an in-order execution model, and the other is out-of-order. They do not suffer from migration penalties by virtue of the shared frontend. We differ by retaining two distinct cores, which have the advantage that both cores can be used simultaneously and both cores can operate at independent frequencies.

ARM has developed an architecture referred to as big.LITTLE [7] to demonstrate the viability of heterogeneous multi-core processors. Their architecture consists of "big" out-of-order cores and a "little" in-order core. When a program has modest performance requirements, the little core can be used

to save power and energy. They cite a thread migration penalty of roughly 20 thousand cycles, which requires infrequent migrations and long program phases to amortize the penalty.

With the In Kernel Switcher (IKS), Linero [15] takes a similar approach to abstracting the visibility of cores to the operating system. IKS presents a pair of big.LITTLE heterogeneous cores to the OS as a single core. Frequency scaling is then used to differentiate between cores – the big core is used for the high frequency modes, and the little core for the low frequency modes. This scheme requires that only a single thread occupies the core pair.

Work by Sawalha et al. [17] has also proposed hardware solutions to accelerate thread migrations. They allow for hardware migrations between more than two cores by using a crossbar to switch between cores and dedicated thread context storage. It is unclear what the impact of their architecture might have on overall performance and the overhead imposed by their circuits as they do not report an evaluation of their architecture.

Brown et al. [2] have worked extensively on minimizing the impact of cache misses when performing thread migrations. They suggest that when a migration should occur, the program working set should be identified, and only that working set should move between cores instead of moving all cache state. Their work assumes that register state will be stored to memory before moving between cores.

Intel has explored hardware mechanisms that are intended to improve the cost of a context switch by introducing the Task State Segment (TSS) [1]. The TSS holds program context information to be accessed by the operating system kernel. However, the TSS still relies on storage in the memory hierarchy, which will still incur significant overheads when migrating threads between cores.

III. Register Migration Alternatives

Many possibilities exist for register consolidation and migration. We introduce a design space that categorizes several of these alternatives. Figure 4 depicts this design space. The design space has two dimensions, one for the method used to consolidate register values, and the other for what type of hardware is used to exchange register values from one core to the other. When viewing Figure 4 in color, the green designs are those which we evaluated in detail, and red indicates designs whose results can likely be inferred from the other designs, but full evaluation has been left for future work.

In our design space, register consolidation can either be "demand" or "continuous". Demand consolidation (DC) implies that registers are only consolidated when newly introduced instructions are executed via the normal execution pipeline. Continuous consolidation (CC) keeps a consolidated version of the register file up-to-date with respect to the most recent committed results. The hardware mechanism used to move register values between cores comprises the other dimension, and can either be with a Teleport Register File [16] [22] (TRF) or with an asynchronous FIFO (AFIFO) plus an Architectural Register File (ARF). TRFs are always used in pairs (one for each core) and allow for the single cycle bulk copy of all bitcells of one TRF to be exchanged with the corresponding bitcells of the other TRF. The alternative

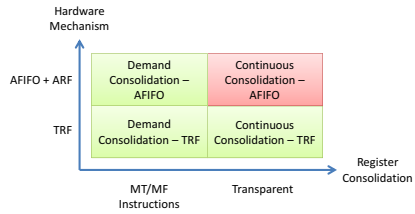


Fig. 4. Design space of hardware register migration alternatives used in this work.

is to use an AFIFO plus ARF for each core, such that the AFIFO corrects for asynchronous clock domain crossings, and the ARF holds values until both cores are ready to exchange instruction streams.

The following sections outline each of these designs, including a baseline core pair with no migration hardware.

A. No Migration Hardware

Our work assumes a baseline heterogeneous core pair with no migration hardware. Figure 5 shows a block diagram of these baseline cores and the actions performed during a thread migration (only a migration from Core0 to Core1 is shown). The following events occur to complete a thread migration:

1) An interrupt signal is raised to both cores to indicate a migration should occur.
2) The instruction at the head of the Active List is examined, and if it is valid and completed, the *next* PC is copied to the Exception PC (EPC) register.
3) The results of the instruction at the head of the Active List are committed, and the remaining instructions in the Active List are flushed.
4) Fetch is redirected to a "suspend" interrupt handler that contains a sequence of store instructions, one for each logical register.
5) After all registers are saved to the memory hierarchy, a barrier (a trap) instruction is executed to signal that the core has finished writing all register values and is waiting for the opposite core to do the same.
6) Once both cores have reached their barriers, the values of the EPCs are exchanged.
7) An interrupt is sent to both cores.
8) Fetch is redirected to a "resume" interrupt handler consisting of a sequence of load instructions, one for each register.
9) After setting each register value with a load instruction, the resume handler executes an Exception Return instruction that reads from the EPC.
10) The user program now continues on the opposite core.

This design has several interesting implications. First, the policy that requires the instruction to not only be valid, but also completed corrects for a corner case in which the instruction at the head of the Active List happens to be a branch instruction that has mispredicted. If that branch had not completed, then it would be possible for the EPC to be written with a predicted *next* PC that is incorrect. Second, that the remaining contents of the Active List will be discarded may mean that good results will be lost and re-executed on the opposite core. Finally, if the instruction at the head of the Active List is not valid or completed, then processing the suspend interrupt is deferred.

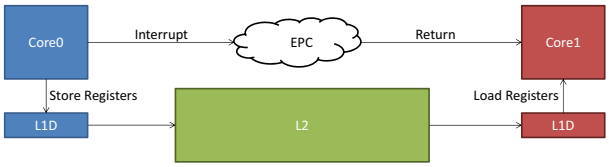


Fig. 5. Block diagram of baseline core pair with no migration hardware.

All of these policy decisions simplified the implementation, at the expense of possibly increasing migration latency.

Another performance consideration is that storing register values in the suspend handler on one core will guarantee that the corresponding load in the resume handler on the opposite core will cache miss due to a coherence invalidation (if it wouldn't have otherwise been a cold miss).

By using store instructions in the suspend handler, the problem of register consolidation is solved by virtue of the normal renaming mechanism. The store will have its source register renamed no matter where the current register value resides in the PRF.

One final note is that the PISA instruction set [3], implemented by the FabScalar toolchain, does not support all of the necessary features required by this migration mechanism. This necessitated adding hardware that would not have otherwise existed if PISA defined operating system-level support. This was the case for the EPC exchange hardware (shown in a cloud in Figure 5), as well as for the barrier and exception return instructions. Of the results reported in Section V, performance results (number of cycles) include this hardware, but physical design results (clock period, power, and energy) do not. Evaluating this way gives the performance benefit to the baseline without any of the physical design penalties.

B. Demand Consolidation – TRF

The Demand Consolidation – TRF (DC-T) alternative behaves similarly to the baseline core pair on a migration, but removes the costly copies through the memory hierarchy. In this alternative, the stores in the suspend handler are replaced with new move-to instructions. Likewise the loads in the resume handler are replaced with new move-from instructions. Since this design uses a TRF pair to exchange register values, the new instructions are referred to as move-to TRF (`MTTRF`) and move-from TRF (`MFTRF`), respectively. The exchange of register values is through two TRFs, one for each core. Both TRFs are the same size as the number of logical registers (including EPC). Figure 6 shows a simplified block diagram of the DC-T alternative.

The `MTTRF` and `MFTRF` instructions flow through the normal execution pipeline. The `MTTRF` has a single source register operand. That operand is renamed with the usual register renaming hardware. The PRF is read during the Register Read Stage as usual, but during the Execute Stage, the value is written to the TRF entry that corresponds to the source operand logical register identifier. Using the normal register renamer solves the consolidation problem. The `MFTRF` has a single destination operand, also renamed using the usual renaming hardware. During the Execute Stage, the TRF entry that corresponds to the destination operand logical register identifier is read. The value is then written to the PRF during the Writeback Stage as usual. The EPC location of the TRF is written when the migration interrupt is processed.

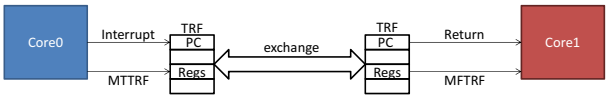


Fig. 6. Block diagram of hardware migration using demand consolidation and a TRF for core-to-core value transfer (DC-T).

The TRF pair provides a single cycle exchange of register values. TRFs were carefully designed to allow the exchange to safely occur between different clock domains. During a migration, the cores independently write their respective TRFs using their own core clocks. Once both cores have finished writing all of their values, they signal to the TRFs that the exchange can occur. The TRFs will then switch to a third clock that is common to both TRFs, exchange values, then switch the clocks of the TRFs back to the core clocks.

While the `MTTRF` and `MFTRF` instructions obviate the need for costly copies through the memory hierarchy, they do not eliminate all of the migration latency. The `MTTRF` and `MFTRF` instructions must traverse the pipeline as usual, incurring the execution time needed to execute these instructions. One design decision that is also a factor in this regard is that all of these instructions traverse a single lane of the processor backend (the Issue Stage onward). This serializes the execution of any `MTTRF` or `MFTRF` instructions in the pipeline.

A final observation is that the DC-T alternative only reads and writes the TRFs during a migration. When the user program is running, the migration hardware is left idle. This allows for a variant of DC-T where the inputs and clocks of the TRFs and migration hardware are gated, to reduce spurious switching activity. This variant is evaluated in Section V. Power gating could also potentially be explored, however our standard cell library does not include power gating cells.

C. Demand Consolidation – Asynchronous FIFO

The Demand Consolidation – Asynchronous FIFO (DC-A) alternative also avoids register copies through the memory hierarchy by introducing new move-to and move-from instructions, similar to the DC-T alternative. But instead of relying on a TRF pair to exchange values, the exchange is done by coupling an AFIFO with an ARF in each direction (a total of two AFIFOs and two ARFs for a core pair). Thus, the new instructions are referred to as move-to-AFIFO (`MTFIFO`) and move-from-ARF (`MFARF`). Figure 7 shows a block diagram of the DC-A alternative (only one direction shown, for clarity).

The steps to complete a thread migration are the same as the DC-T alternative. The semantics of the `MTFIFO` are the same as `MTTRF`, except that the register value (tagged with the register identifier) is pushed to an AFIFO instead of a TRF. Similarly, the `MFARF` instruction semantics match the `MFTRF` instruction, except the register value is read from an ARF instead of a TRF. The AFIFOs are written using the clock of the core executing the `MTFIFO` instruction. A small state machine monitors the "empty" bit of the AFIFO, and when the FIFO is not empty, it pops the FIFO and writes the value to the appropriate entry in the corresponding ARF. The state machine and ARF are clocked using the receiving core's clock.

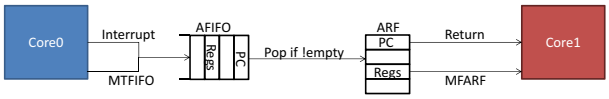


Fig. 7. Block diagram of hardware migration using demand consolidation and an Asynchronous FIFO and ARF for core-to-core value transfer (DC-A).

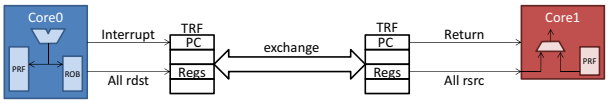


Fig. 8. Block diagram of hardware migration using continuous consolidation and a TRF for core-to-core value transfer (CC-T).

This strategy handles crossing the asynchronous clock domains of the cores.

The DC-A design has very similar migration latency implications as the DC-T alternative. The DC-A alternative can also have the same clock and input gated variant as DC-T.

D. Continuous Consolidation – TRF

The Continuous Consolidation – TRF (CC-T) alternative differs from the DC-T design by always keeping the TRFs up-to-date with the most recent committed register values without the need for explicit move-to or move-from instructions. Thus, the TRF is updated transparently from the instruction set perspective. Figure 8 shows a simplified block diagram of the CC-T alternative (only showing the Core0 to Core1 migration). The CC-T alternative achieves a much lower thread migration latency since the TRF exchange can occur very soon after a migration interrupt because it does not need to wait for move-to or move-from instructions to execute. However, this comes at the expense of increased complexity in the TRFs and core pipelines.

As instructions execute, their results must be written to the TRF in addition to the PRF. Since the size of the TRF is the same as the number of logical registers, it is best to write only committed instruction values. This change requires adding new write ports to the TRF when the commit width is greater than one. However, in a typical out-of-order pipeline, the result values of instructions are not available at commit time – their values were long ago written to the PRF and not buffered elsewhere. To fix this problem, we buffer the instruction results in a separate RAM (an extension of the Active List) until instructions commit. Also, it is possible that more than one instruction commits the same logical register in a given cycle. Thus, new write-after-write hazard checking logic must also be included to write only the latest value of a register to the TRF. These changes eliminate the need for any move-to instructions, however the suspend interrupt handler still contains a single instruction, the barrier.

To eliminate the need for move-from instructions after a migration has occured necessitates several changes to the register rename logic and rename map table (RMT) to allow registers to be renamed to either the PRF or the TRF. This requires adding a bit to each entry of the RMT, used to distinguish whether a logical register value can be found in the PRF or in the TRF. After a migration, all register values are in the TRF, so all of these new bits in the RMT will be set to point to the TRF. As instructions write new values, the corresponding bits in the RMT must be cleared to indicate that the register value can now be found in the PRF. At any given time, the RMT could have some entries point to registers in the PRF and others point to registers in the TRF. This means that the Architectural Map Table (AMT) must also be extended with these new bits because the AMT is used to repair entries in the RMT in the event of a pipeline recovery. These changes allow the resume interrupt handler to also only consist of a single instruction, this time a single Exception Return instruction.

One final observation with the CC-T alternative is that the TRFs must always be kept up-to-date, which implies that the TRFs and associated migration logic cannot be safely clock or input gated to save power, as was the case with DC-T and DC-A.

E. Continuous Consolidation – Asynchronous FIFO

Continuous Consolidation – Asynchronous FIFO (CC-A) is identical to the CC-T alternative, with the exception that the TRFs are replaced with asynchronous FIFO and ARF pairs just like the changes from DC-T to DC-A. While we do not fully evaluate the CC-A alternative, it is reasonable to assume that any trends when comparing DC-T to DC-A can also be applied to comparing CC-T to CC-A.

IV. Methodology

This section details the methods and tools used to evaluate the hardware thread migration alternatives. The two main components of our evaluation infrastructure are 1) instantiations of full cores written in Verilog RTL that were generated by FabScalar [4], and 2) an in-house C++ performance simulator. The Verilog RTL was augmented with synthesizable implementations for each of the migration alternatives, allowing for full netlist simulation to verify correctness and for detailed timing and power analysis. The C++ simulator overcomes some of the limitations of RTL simulation while retaining the timing and power statistics derived from the RTL implementations. These tools are described in Section IV-A and IV-B, respectively. The workloads and metrics used in this work are briefly described in Section IV-C.

A. RTL Model

Evaluating the impact of register value consolidation and thread migration demands carefully considering the power and timing overheads with the highest possible fidelity. To meet this need, we turn to a Verilog RTL model of our proposed migration alternatives. These alternatives can then be carried through industry-standard electronic design automation (EDA) tools to derive highly accurate estimates for area, power, and timing.

The FabScalar [4] toolchain provides synthesizable RTL for the core pipeline for a wide variety of configurations. This serves as a starting point for our dual heterogeneous core model. We generated two “reference cores”, whose core parameters are shown in Table II in Section VI. FabScalar cores do not include caches or off-chip I/O, so the reference cores include in-house developed L1 instruction and data caches as well as DesignWare SERDES for off-chip I/O. The unified L2 cache is assumed to be off-chip. With these modifications, the top-level module is pin-accurate for a dual

TABLE I. EDA TOOLS USED IN THIS WORK.

Name and Version
Cadence NC-Verilog, vers. 09.20-s019
Synopsys Design Compiler, vers. H-2013.03-SP2
Synopsys PrimeTime, vers. H-2013.06-SP1
Cadence SoC Encounter, vers. 9.1
FreePDK 45nm process technology library [20], vers. 1.3

heterogeneous multi-core processor and serves as the baseline processor with no register consolidation or migration hardware.

The RTL for these reference cores is then replicated and each copy is augmented with the register consolidation and migration hardware according to the alternatives described in Section III. The designs include hand-written TRF implementations and DesignWare asynchronous FIFOs, where applicable. These designs are then synthesized to obtain gate-level netlists. The netlists are then simulated using several hand-written microbenchmarks as well as a few SPEC CPU2000 benchmark regions. These simulations are performed to gain confidence that the implementations are functionally correct and also to collect the gate-level switching activity. The switching activity is then used for accurate power analysis. The EDA tools used for this flow are listed in Table I.

B. Performance Simulator

While the RTL models provide highly accurate area, timing and power estimates, they are somewhat inconvenient for use in performance estimation. The C++ simulator developed for this work is a cycle-level, execute-at-execute model. To model power, the simulator is augmented with a database of characterization data of individual pipeline structures taken from FabScalar modules. The RTL model in the previous section assumes that caches are synthesized to D flip-flops. To estimate caches with full custom memories, we populate the database with characterization data derived from CACTI [19] whose technology files have been modified to support our FreePDK45 process technology. This database is queried during simulation to retrieve energy data for each pipeline stage and structure.

This simulation infrastructure allows for a very large design space of out-of-order cores. To pare down the number of cores to evaluate, we limit the design space by selecting 18 representative cores, similar to the G21 strategy used in [4]. In this paper, the designs were selected to span pipeline widths of 1-wide through 6-wide, with three cycle time targets each. This allows for small, medium and large relative structure sizes. With these core configurations, we ran an exhaustive design space exploration (DSE) to find the best single core *and* the best core pair for both our performance and efficiency metrics.

C. Workload and Metrics

For our work, performance is measured in billions-of-instructions-per-second (BIPS) to account for comparisons across both cycles and frequency. The efficiency metric uses both performance (in BIPS) and power (in Watts) for a combined metric $BIPS^3/W$.

We use benchmarks from both SPEC CPU2000 and CPU2006 for our workload. We use `gcc` version 4.5.2 with optimization level -O3 to compile as many of these benchmarks as possible. We extract benchmark regions using the SimPoint [18] tool, configured to find 10 million instruction regions with a maximum of ten regions per benchmark. This results in a total of 179 program phases from 32 benchmarks.

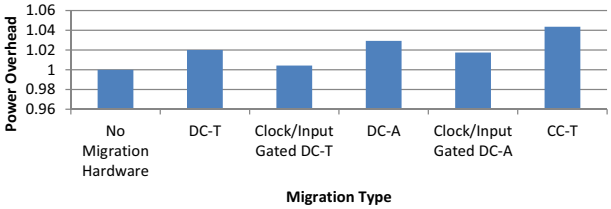


Fig. 9. Power overhead when including migration hardware. Note that the y-axis does not start at zero.

V. RESULTS

The focus of this section is to evaluate the quality of the different consolidation and migration schemes introduced in Section III. We also determine which alternative is likely to best meet the performance and efficiency needs when used in a heterogeneous multi-core. Some alternatives in Section III can have their migration hardware clock and input gated to save power between migrations, the following results only plot these variants when applicable.

The additional power used when including the migration hardware, even when migrations are not occuring (i.e. power overhead during user program execution), is shown in Figure 9. This overhead should be kept as low as possible since it is a tax imposed by including migration hardware. Notice from this graph that clock and input gating for the DC-T and DC-A alternatives is clearly beneficial. Also, the use of a TRF has a benefit over using an asynchronous FIFO and ARF. This is because the TRF and ARF are similarly sized RAMs (and thus roughly equal in power), but the asynchronous FIFO adds static power consumption relative to the TRF scheme, even when clock and input gating. These estimates were derived by running a microbenchmark with no migrations on the synthesized netlist of each alternative and using power analysis to capture the average power.

To estimate the migration latency and migration energy, the netlists of each alternative was simulated while executing a microbenchmark that repeatedly migrates from one core to the other then back for thousands of migrations. The results are shown in Figure 10 and Figure 11, respectively. For these experiments, several data points are collected for each alternative, varying the number registers transferred by altering the number of instructions used to consolidate registers. The migration latency of DC-T and DC-A are nearly identical, since the only difference is in the register exchange mechanisms, which have similar latencies. These latency results show that substantial savings are realized no matter which alternative is employed, although the CC-T has constant migration latency no matter how many registers are transferred since the TRF is always kept up-to-date. We can also see that migration energy is very low for each migration alternative.

Figure 12 shows the average performance and efficiency for our benchmarks with respect to coarse-grain thread migrations, repeating the simulation experiments from Section I. We allow migrations at the smallest interval size. But for these estimates, we use the measured power overhead, migration latency, and

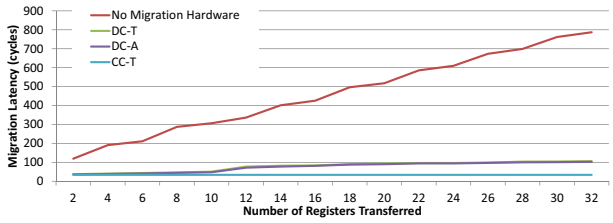


Fig. 10. Average number of cycles to complete a full thread migration.

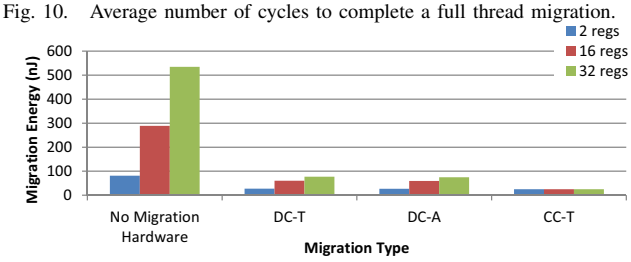


Fig. 11. Average amount of energy required to complete a full thread migration.

migration energy to appropriately penalize the benchmark for including migration hardware and for performing migrations. We can see from these estimates that the benefit of adding migration hardware does not always overcome their overheads. This is especially true for efficiency evidenced by the negative improvement for many alternatives.

The best alternative for performance is CC-T. This is because CC-T does not need to wait for register copying instructions to execute – the suspension, register exchange, and resumption can occur very quickly after a migration interrupt. The DC-T with clock and input gating is the best alternative for efficiency. This is because it has the best balance between short migration latency (from Figure 10) while minimizing the power overhead (Figure 9) and energy for each migration (Figure 11).

Figure 12 shows an *average* of all programs, which can hide the impact (both positively and negatively) of migration hardware. Figure 13 shows the efficiency of the clock and input gated DC-T alternative for each benchmark. This graph shows that about 45% of benchmarks have improved efficiency, about 30% of benchmarks have no impact from migration hardware, and the remaining benchmarks have slightly worse efficiency due to the power overhead penalty.

As a final note, adding migration hardware also has the potential to increase the pipeline clock period, as well as consume additional die area. These characteristics for each

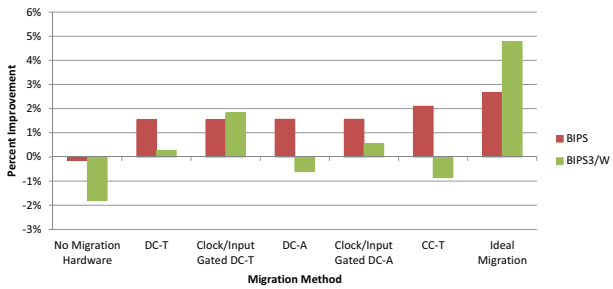


Fig. 12. Average speed-up for hardware migration schemes compared to heterogeneous core pair with coarse-grain migrations.

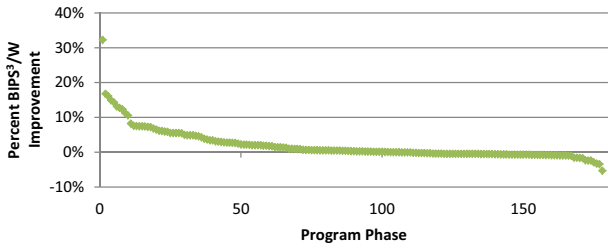


Fig. 13. Efficiency of each program phase for DC-T compared to heterogeneous core pair with coarse-grain migrations.

TABLE II. PROTOTYPE CHIP CORES.

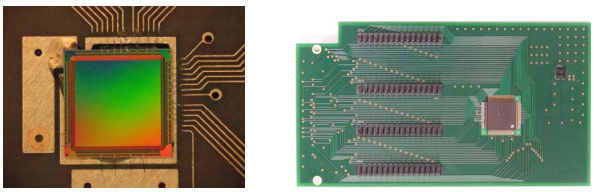
	Core0	Core1
Frontend Width	2	1
Issue Width	3	3
Depth	9	9
IQ Size	32	16
PRF Size	96	64
LQ/SQ Size	16/16	16/16
Active List Size	64	32
L1 I-Cache	private, 4kB, 1-way, 8B blocks, 1 cycle, no prefetch	
L1 D-Cache	private, 8kB, 4-way, 16B blocks, 2 cycle, no prefetch	

migration alternative were measured, but are not shown here as neither were substantially impacted.

VI. PROTOTYPE

As a proof of concept of our hardware thread migration architecture, we fabricated a prototype chip. The chip includes a heterogeneous core pair of out-of-order cores – the "reference cores" introduced in Section IV-A. The core parameters are listed in Table II. These cores utilize the DC-T hardware migration scheme. Thus, they implement the move-to and move-from instructions for register consolidation and use a TRF for transferring register values from one core to the other. We used a 130nm process technology.

A photo of the die, wire-bonded to a test PCB, is shown in Figure 14a. Figure 14b shows a fully packaged and assembled PCB. This PCB has a mezzanine connector (on the back of the PCB) that mates the PCB to a Xilinx ML-605 development FPGA board. The FPGA serves to generate the signals necessary to exercise the chip, the main tasks of which are to generate the independent clocks for each core, as well as to host the L2 caches and memory controller to service memory requests. More details of this prototype appear in [6].



(a) Die. (b) Package and PCB.

Fig. 14. Photos of fabricated chip and test PCB.

Figure 15 shows the average latency of a thread migration, taken from actual measurements of the prototype chip. Notice that these measurements almost exactly match our simulation results shown in Figure 10 in Section V. To obtain these measurements, a simple test microbenchmark was written which repeatedly migrates a thread between the cores, for one

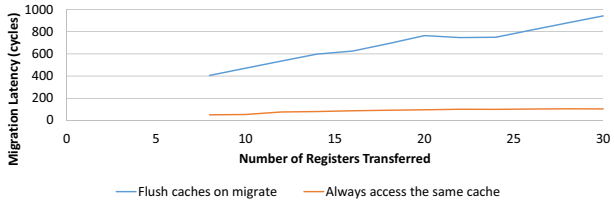


Fig. 15. Measured migration latency from prototype chip.

million migrations. The total number of cycles are recorded to execute all migrations, then the total cycles is divided by the number of migrations. This minimizes the impact of any warm-up overheads.

To verify functionality, the cores were tested using many different relative frequencies. However, for this experiment, the cores are operated at the same frequency to ensure that cycles can be directly compared. Also, when operating at the same frequency, a mode is available where cache values are pinned to a single cache, and a migrating thread can access the cached values without incurring migration-induced cache misses. This side-steps an early design decision that flushes caches on a migration. Figure 15 shows the latency in both modes. Our most recent version of this design introduces a cache coherence policy to eliminate cache flushes on a migration, while still allowing the cores to operate at independent frequencies. This new version is currently being fabricated.

ACKNOWLEDGMENTS

Thanks to the Heterogeneity in 3D (H3) research and design team at NC State: Rangeen Basu Roy Chowdhury, Brandon Dwiel, Vinesh Srinivasan, Randy Widialaksono, Zhenqian Zhang, Steve Lipa, Rhett Davis, Paul Franzon. This project is supported by a grant from Intel.

REFERENCES

[1] Intel 64 and IA-32 Architectures Software Developer's Manual. September 2015.

[2] Jeffery A. Brown, Leo Porter, and Dean M. Tullsen. Fast Thread Migration via Cache Working Set Prediction. In *Proceedings of the 17th Annual International Symposium on High Performance Computer Architecture*, HPCA-17, pages 193–204, February 2011.

[3] Doug Burger and Todd M. Austin. The Simplescalar Tool Set, Version 2.0. Technical Report TR1342, Computer Sciences Department, University of Wisconsin - Madison, 1997.

[4] Niket K. Choudhary, Salil V. Wadhavkar, Tanmay A. Shah, Hiran Mayukh, Jayneel Gandhi, Brandon H. Dwiel, Sandeep Navada, Hashem H. Najaf-abadi, and Eric 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*, ISCA-38, pages 11–22, June 2011.

[5] Theofanis Constantinou, Yiannakis Sazeides, Pierre Michaud, Damien Fetis, and Andre Seznec. Performance Implications of Single Thread Migration on a Chip Multi-core. In *Workshop on Design, Architecture, and Simulation of Chip Multiprocessors*, November 2005.

[6] Elliott Forbes, Rangeen Basu Roy Chowdhury, Brandon Dwiel, Anil Kannepalli, Vinesh Srinivasan, Zhenqian Zhang, Randy Widialaksono, Thomas Belanger, Steve Lipa, Eric Rotenberg, W. Rhett Davis, and Paul D. Franzon. Experiences with Two FabScalar-based Chips. In *Proceedings of the 6th Workshop on Architectural Research Prototyping, held in conjunction with ISCA-42*, WARP 2015, June 2015.

[7] Peter Greenhalgh. Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. September 2011.

[8] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-36, December 2003.

[9] Rakesh Kumar, Dean M. Tullsen, and Norman P. Jouppi. Core Architecture Optimization for Heterogeneous Chip Multiprocessors. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT-15, pages 23–32, September 2006.

[10] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA-31, June 2004.

[11] Benjamin C. Lee and David M. Brooks. Illustrative Design Space Studies with Microarchitectural Regression Models. In *IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA-13, pages 340–351, 2007.

[12] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the Cost of Context Switch. In *Proceedings of the 2007 ACM Workshop on Experimental Computer Science*, ExpCS '07, June 2007.

[13] Mieszko Lis, Keun Sup Shim, Brandon Cho, Ilia Lebedev, and Srinivas Devadas. Hardware-level Thread Migration in a 110-core Shared-Memory Processor. In *Proceedings of the 25th Annual Hot Chips: A Symposium on High Performance Chips*, HC25, August 2013.

[14] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M. Sleiman, Ronald Dreslinski, Thomas F. Wenisch, and Scott Mahlke. Composite Cores: Pushing Heterogeneity into a Core. In *Proceedings of the 45th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO-45, pages 317–328, December 2012.

[15] Mathieu Poirier. In Kernel Switcher: A Solution to Support ARM's New big.LITTLE Technology. In *Proceedings of the Embedded Linux Conference*, ELC'13, February 2013.

[16] Eric Rotenberg, Brandon Dwiel, Elliott Forbes, Zhenqian Zhang, Randy Widialaksono, Rangeen Basu Roy Chowdhury, Nyunyi Tshibangu, Steve Lipa, W. Rhett Davis, and Paul D. Franzon. Rationale for a 3D Heterogeneous Multi-core Processor. In *Proceedings of the 31st International Conference on Computer Design*, ICCD-31, pages 154–168, October 2013.

[17] Lina Sawalha, Monte Tull, and Ronald Barnes. Hardware Thread-contxt Switching. *Electronics Letters*, 49(6), March 2013.

[18] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-X, pages 45–57, October 2002.

[19] Shyamkumar Thoziyoor and Naveen Muralimanohar and Jung Ho Ahn and Norman P. Jouppi. CACTI 5.1. *Tech. Report HPL-2008-20, HP Labs*, 2008.

[20] James E. Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W. Rhett Davis, Paul D. Franzon, Michael Bucher, Sunil Basavarajaiah, Julie Oh, and Ravi Jenkal. FreePDK: An Open-Source Variation-Aware Design Kit. In *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, MSE '07, pages 173–174, June 2007.

[21] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating Critical Section Execution with Asymmetric Multi-core Architectures. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-XIV, pages 253–264, March 2009.

[22] Zhenqian Zhang, Brandon Noia, Krishnendu Chakrabarty, and Paul D. Franzon. Face-to-Face Bus Design with Built-in Self-Test in 3D ICs. In *Proceedings of the International IEEE 3D Systems Integration Conference*, 3DIC, October 2013.