A Unified View of Non-monotonic Core Selection and Application Steering in Heterogeneous Chip Multiprocessors

Sandeep Navada*, Niket K. Choudhary*,
Salil V. Wadhavkar*
CPU Design Center
Qualcomm
Raleigh, NC, USA
{snavada, niketc, salilw}@qti.qualcomm.com

Eric Rotenberg
Department of Electrical & Computer Engineering
North Carolina State University
Raleigh, NC, USA
ericro@ncsu.edu

***Abstract*—A single-ISA heterogeneous chip multiprocessor (HCMP) is an attractive substrate to improve single-thread performance and energy efficiency in the dark silicon era. We consider HCMPs comprised of non-monotonic core types where each core type is performance-optimized to different instruction-level behavior and hence cannot be ranked – different program phases achieve their highest performance on different cores. Although non-monotonic heterogeneous designs offer higher performance potential than either monotonic heterogeneous designs or homogeneous designs, steering applications to the best-performing core is challenging due to performance ambiguity of core types.**

In this paper, we present a unified view of selecting non-monotonic core types at design-time and steering program phases to cores at run-time. After comprehensive evaluation, we found that with N core types, the optimal HCMP for single-thread performance is comprised of an "average core" type coupled with N-1 "accelerator core" types that relieve distinct resource bottlenecks in the average core. This inspires a complementary steering algorithm in which a running program is continuously diagnosed for bottlenecks on the current core. If any are observed, the program is migrated to an accelerator core that relieves any of the bottlenecks and does not worsen any of them. If no accelerator core satisfies this condition, then the average core is selected.

In our evaluation, we show that a 4-core-type HCMP improves single-thread performance up to 76% and 15% on average over a homogeneous chip multiprocessor, and our steering algorithm is able to capture most of this performance gain. Further, we show that our steering algorithm on a 4-core-type HCMP is, on average, 33% more power-efficient ($BIPS^3$/watt) than a homogeneous chip multiprocessor.

Keywords - heterogeneous multi-core processor, adaptive processor, superscalar, core customization, thread migration, instruction-level parallelism, single-thread performance

*Sandeep Navada, Niket K. Choudhary and Salil V. Wadhavkar contributed to this paper when they were graduate students in the Department of Electrical and Computer Engineering at North Carolina State University.

I. INTRODUCTION

Industry has fully embraced multi-core chip design for platforms ranging from mobile phones to desktops to supercomputers. However, as we enter the "dark silicon" era [10][12], the number of cores that can be active is limited by the chip's power budget and is less than the number of cores that can be fit on the chip. Specialization is one way to profit from dark silicon. While adding more of the same core design is of little use in this context, introducing differently-designed cores enables run-time adaptation: migrating a thread among a dedicated ensemble of different core types as the thread's instruction-level behavior changes. Matching varying instruction-level behavior to cores optimized for different behaviors, can lead to faster and more energy-efficient execution. A multi-core architecture with multiple core types, which are functionally-equivalent but microarchitecturally-diverse, is called a single-ISA heterogeneous chip multiprocessor (HCMP).

Early HCMP designs were limited to *monotonic core types*: they could be clearly ranked from high-performance/high-power to low-performance/low-power independent of the application, providing an unambiguous performance and power spectrum [18][19][39]. A more powerful class of HCMP employs *non-monotonic core types* [20]: each core type is performance-optimized to different instruction-level behavior and hence cannot be ranked – different applications achieve their highest performance on different cores [6][27][28]. Although non-monotonic heterogeneous designs offer higher performance potential than either monotonic heterogeneous designs or homogeneous designs, steering applications to the best-performing core is challenging due to performance ambiguity of core types. Moreover, with non-monotonic cores, accurate steering is mandatory. Without it, performance can actually worsen since cores are tuned. Steering completes what was started by architecting non-monotonic cores in the first place.

In this paper, we demonstrate that *selecting* non-monotonic core types at design-time and *steering* programs to cores at run-time, are actually "two sides of the same coin".

We performed an exhaustive RTL-based [6] design space exploration of HCMPs with one through four superscalar core types that achieve the highest performance on 39 phases from SPEC. The addition of more core types yields more performance due to tailoring cores to diverse instruction-level behaviors (control-flow, data-flow, cache misses, etc.). With only one core type, the best core in the design space is a rather "average" core that strikes a reasonable balance between instruction-level parallelism (ILP) and frequency. If more core types are allowed, it turns out that the average core (*i.e.*, best homogeneous core) is *still* selected and additional core types are selected that relieve distinct resource bottlenecks in the average core, that constrain the performance of "outlier" program phases: a core type with larger window size (window bottleneck); one with higher issue width (width bottleneck); one with higher frequency (frequency bottleneck); and so forth. In summary, with N core types, the optimal HCMP for single-thread performance is comprised of an "average core" type coupled with N-1 "accelerator core" types that relieve distinct resource bottlenecks in the average core.

The fact that the selection of non-monotonic cores is empirically driven by average resource provisioning (average core) coupled with distinct resource bottleneck relief (accelerator cores), inspires a complementary steering algorithm. The running program is continuously diagnosed for bottlenecks on the current core. If any are observed, the program is migrated to an accelerator core that relieves any of the bottlenecks and does not worsen any of them. If no accelerator core satisfies this condition, then the average core is selected.

The average core / accelerator core phenomenon is arguably a result of our objective function: maximize the performance of one thread by providing it a private ensemble of diverse cores, essentially a single adaptive processor or *core-selectable processor* [26] whereby adaptivity is achieved by thread migration. We also take the paper in several other directions.

- *Power-constrained HCMP*: We constrain the core-selectable processor to a power budget. While the constituent cores change for different budgets, we empirically observe that the best homogeneous core type for this power budget always shows up as the average core type and the other core types relieve distinct bottlenecks in it. Thus, our steering approach is robust from less-constrained to more-constrained designs.
- *Multi-programmed workload*: Our steering algorithm is adapted to support multiple threads within the core-selectable processor. Threads decide locally on their preferred core and contention is resolved globally by conflict resolving mechanisms like ranking threads' bottleneck intensities, etc.
- *Arbitrary HCMP design*: We show how to apply bottleneck-driven steering to arbitrary HCMP designs, regardless of which criteria are used to guide the selection of cores at design-time. The idea is that one of the core types is the best on average among all core types *available* (even if it is not the best homogeneous core type), and bottleneck relief (*i.e.*, strengths) of other core types can always be gauged relative to it.

In our evaluation, we show that the best 4-core-type HCMP improves single-thread performance up to 76% and 15% on average over the best homogeneous chip multiprocessor, and our steering algorithm is able to capture most of this performance gain. We also show that our steering algorithm on the 4-core-type HCMP is, on average, 33% more power-efficient ($BIPS^3$/watt) than the homogeneous chip multiprocessor.

The significance and contribution of our work can be summarized as follows:

1) *First complete proposal to accelerate a single thread using a core-selectable processor*: This paper, and its precursors [6][26][27], comprise the only work focused on architecting and operating multiple diverse cores as one logical core to *minimize latency* of a single thread. Most of the related work in this area pair one big OOO core type with one little in-order core type (what ARM calls big.LITTLE [14]), or other similar fast/slow hybrid, where the objective is to either minimize energy consumption of a single thread while minimizing latency impact [18][22] or maximize throughput/watt/area of multiprogrammed, multi-threaded workloads [3][8][17][19][32][35][37]. Our work is unique in minimizing latency under low contention; in turn, this requires co-designing the ensemble of non-monotonic core types and the steering algorithm, rather than assuming a *de facto* monotonic HCMP. What little prior work there is on design exploration of core types, is also focused on throughput/watt/area of multiprogrammed workloads, does not factor-in frequency, and does not provide a companion steering algorithm [20].
2) *Average core / accelerator core phenomenon*: We are the first to empirically discover the average core/accelerator cores phenomenon of HCMPs architected for single-thread performance. This discovery was critical to coming upon the idea of using very simple distinctive bottleneck analysis for steering.
3) *Bottleneck-based steering approach:* We propose a simple and intuitive, bottleneck-based steering algorithm for HCMPs. It captures most of the performance gain in an HCMP and outperforms the current state-of-art steering algorithm (sampling algorithm).
4) *Sensitivity studies:* (i) We show that the average core / accelerator core phenomenon still holds for power-constrained core-selectable processors. (ii) We show how to adapt the steering algorithm for multiple threads sharing the core-selectable processor. (iii) We show that bottleneck-based steering can be applied to arbitrary HCMP designs because the designer can always identify a best-on-average core type and gauge other core types against it.

The rest of the paper is organized as follows. Background on processor cores is presented in Section II. Section III explains our process of architecting the non-monotonic

HCMP. Section IV details our technique of using bottleneck information to steer applications in a non-monotonic HCMP. Experimental methodology and results are presented in Sections V and VI, respectively. Section VII discusses related work. Finally, Section VIII concludes the paper.

II. Background on Processor Cores

A single-ISA heterogeneous chip multiprocessor (HCMP) consists of multiple differently-designed core types. A core type can be varied along three dimensions: structure sizes (issue queue, load and store queues, physical register file / reorder buffer, caches, and predictors), pipeline stage widths and clock period.

In a core, there is a tradeoff between exploiting more instruction-level parallelism (ILP) and achieving a higher clock frequency, and this tradeoff works out differently for different program phases. Exploiting more ILP requires increasing the sizes of ILP-extracting structures and/or increasing the widths of pipeline stages. The additional circuit complexity may increase cycle time, however. Different program phases have different amounts and distributions of ILP (as well as memory-level parallelism), causing them to be characterized by different optimal core types.

III. Core Selection in HCMP

Selecting the core-types, which make up a good HCMP design, is not straightforward because each application phase prefers a different core-type. As we can only have a limited number of core-types in a given HCMP, this results in a compromise. To choose the core-types, which make up the optimal HCMP design, we perform a rigorous design space exploration using genetic algorithms [11] for the SPEC benchmark suite.

Section IIIA discusses our methodology for architecting an N-core-type HCMP. Section IIIB describes the optimal unconstrained HCMP design with 2, 3 and 4 core-types. Section IIIC discusses the optimal power-constrained HCMP design.

A. Methodology

1) Basis of Core Design Space: FabScalar Toolset

The core design space is created using FabScalar's Canonical Superscalar Template and Canonical Pipeline Stage Library (CPSL) [6]. The template has the following canonical pipeline stages: Fetch, Decode, Rename, Dispatch, Issue, Register Read, Execute, Writeback, and Retire. The CPSL provides different register-transfer-level (RTL) designs for each canonical pipeline stage, that differ in their superscalar width and depth of sub-pipelining (pipelining within a canonical stage). Sizes of microarchitectural structures are parameterized.

Instructions-per-cycle (IPC), clock period, and power, are measured as follows.

IPC is obtained using FabScalar's cycle-accurate C++ simulator, which has been validated against FabScalar's RTL designs [6].

To get a reliable estimate of the clock period, we make use of FabScalar's Performance-Power-Area (PPA) tool [7]. PPA provides propagation delays for each and every pipeline stage design in the entire CPSL, and these delays are from logic synthesis (for logic) and FabMem (for highly-ported RAMs/CAMs). (For caches, PPA uses CACTI [40], with its device and wire parameters adjusted to match those of the standard cell library used for synthesis [6].) The CPSL allows composing thousands of cores and their cycle times are known from the detailed, comprehensive CPSL characterization just described.

For power modeling, we again use FabScalar's cycle-accurate C++ simulator, which, like the Wattch method [4], combines per-unit energy numbers with their activity counts. Unlike Wattch, the per-unit dynamic/static energy numbers come from the FabScalar PPA tool, which provides per-unit energy numbers from logic synthesis and FabMem, for hundreds of fine-grain components in the core. Moreover, per-unit energies for all superscalar widths, depths, and structure sizes are fully represented within PPA.

2) Core Design Space

The core design space is spanned by the independent design parameters enumerated in Table I. The core design space consists of design points created by varying the clock period and the complexity (structure sizes and superscalar widths) of different pipeline stages.

Depths of pipeline stages are dependent design parameters. For a given design point, the depth of a pipeline stage is the degree of sub-pipelining required so that its complexity can fit in the clock period.

TABLE I: MICROARCHITECTURAL DESIGN PARAMETERS.

Parameter	Value Range	Number
Front end width	2, 3, 4, 5, 6, 7, 8	7
Issue width	2, 3, 4, 5, 6, 7, 8	7
Phys. Reg. File size	64, 128, 192, 256, 384, 512	6
Issue Queue size	16, 24, 32, 48, 64, 96, 128	7
Load Queue size/ Store Queue size	8/8, 16/16, 24/24, 32/32, 40/40, 48/48, 56/56, 64/64	8
L1 Instr. Cache size (KB)	8, 16, 32, 64, 128	5
L1 Data Cache size (KB)	8, 16, 32, 64, 128	5
L2 Cache size	2MB	1
Clock Period	0.5→1.2 ns (delta=0.1ns)	8

The cartesian product of all parameter values in Table I, gives *3.3 million design points*. Not all of these design points are valid, however. Firstly, as we are using FabScalar's CPSL, a pipeline stage cannot be made arbitrarily deep. Thus, some design points cannot meet their clock periods. Secondly, for a given pipeline stage, only those design points are considered that have the largest structure(s) for a given width, depth, and frequency of that pipeline stage. By performing this design space pruning, we are able to restrict it to *13,966 design points*.

We assume private L1 instruction and data caches, and a shared L2 cache. Block size is 64 bytes, L1 associativity is 4, and L2 associativity is 8. The memory access latency is fixed at 100 ns. We use a gshare branch predictor with 64K entries, a branch target buffer with 4K entries and a return address stack with 16 entries.

3) Core Selection Algorithm

An HCMP consists of N core-types selected from the core design space. Hence, the HCMP design space consists of every combination of N core-types selected from the core design space. Clearly, finding the optimal HCMP design is a big search endeavor. To give an idea of the size of the HCMP design space, let us consider the 4-core-type HCMP. The design space of the 4-core-type HCMP has every 4-core combination selected from 13,966 design points in the core design space. This amounts to $1.59*10^{15}$ HCMP design points. This is a huge design space and it is impossible to exhaustively search it.

Hence, we use a genetic algorithm [11] to find the best N-core-type HCMP design. Please note that every core-type in the entire core design space (13,966 design points) is first evaluated on 39 SimPoint phases [36] (10 million instructions each) obtained from the SPEC benchmark suite. The fully-characterized core design space serves as input to the genetic algorithm for finding the best N-core-type HCMP. For the genetic algorithm, an individual is a given N-core-type HCMP design selected from the HCMP design space and the objective function is the harmonic mean of the performance values (billions-of-instructions-per-second (BIPS)) of the 39 phases, assuming each phase is steered to its best core-type out of the N core-types. We use the standard mutation, crossover and selection operators in the genetic algorithm. The population size is 100 individuals and we keep running until the best HCMP design does not change for 10,000 generations.

We use this methodology to find the core configurations of 2-core-type, 3-core-type and 4-core-type HCMPs. We validated the genetic algorithm results with an exhaustive search for 1-core-type, 2-core-type and 3-core-type HCMPs (as exhaustive search is feasible for them). The next section discusses the core configurations.

B. Unconstrained HCMP Design

The core-types in the optimal 2-core-type, 3-core-type and 4-core-type HCMPs are given in Table II. The microarchitecture configurations of these core-types are shown in Table III. Note that the performance of the 4-core-type HCMP is within 98% of the optimal performance bound of having customized core-types for each of the SPEC phases used – which is why we stopped at 4-core-types.

TABLE II: CORE-TYPES IN THE OPTIMAL 2-CORE-TYPE, 3-CORE-TYPE AND 4-CORE-TYPE HCMPS. ALSO SHOWN IS THE BEST SINGLE CORE-TYPE FOR A HOMOGENEOUS DESIGN.

Number of core-types in HCMP	Core combination
1 (homogeneous design)	A
2	A, LW
3	A, LW, N
4	A, L, N, W

We see that, irrespective of the number of core-types, the A core-type – which is also the best single core-type for a homogeneous design (called the "average" core-type) – is always present in the optimal heterogeneous design. We further see that the other core-types in the optimal HCMP (called "accelerator" core-types) aim to remove distinct bottlenecks. In the optimal 2-core-type HCMP, in addition to core-type A, we have core-type LW, which has both a larger window (L) and a wider issue width (W). Core-type LW targets application phases with a window bottleneck or a width bottleneck. In addition to core-types A and LW, the optimal 3-core-type HCMP has core-type N (narrow core) that focuses on application phases with a severe ILP bottleneck. In the optimal 4-core-type HCMP, in addition to retaining core-types A and N, two others emerge: core-type L (large-window core) which targets application phases with a window (reorder buffer, issue queue) bottleneck, and core-type W (wide core) which focuses on application phases with a width (fetch width, issue width) bottleneck. In other words, in the optimal 4-core-type HCMP, core-type L targets application phases with distant ILP, core-type W targets application phases with nearby ILP, core-type N targets application phases with low ILP, and the average core-type targets all other application phases.

TABLE III: MICROARCHITECTURE CONFIGURATIONS OF DIFFERENT CORE-TYPES IN THE UNCONSTRAINED HCMP DESIGNS.

Core Type	Clock Period (ns)	ILP-extracting buffers (IQ,LSQ,ROB)	Widths (fetch, issue)	Caches (KB) (I$, D$)
A	**0.6**	**32, 128, 128**	**3, 4**	**64, 64**
N	**0.5**	**32, 64, 64**	**2, 2**	**16, 16**
L	**0.7**	**48, 128, 384**	**4, 4**	**128, 128**
W	**0.7**	**32, 128, 128**	**6, 6**	**128, 32**
LW	**0.7**	**48, 128, 192**	**4, 5**	**128, 32**

The change between the 3-core-type and 4-core-type HCMPs is quite interesting: the provisioning of one more core-type causes core-type LW, present in the 3-core-type HCMP, to be "split" into two, more specialized core-types L and W that each concentrate on only one of the two bottlenecks. Consequently, L is freed to be even larger than LW (but same issue width as A) and W is freed to be even wider than LW (but same size as A), so that each targets their respective bottlenecks more effectively than the in-between LW core-type.

In summary, the optimal HCMP consists of (1) an average core-type that is comparable to current commercial high-performance superscalar processors (4-issue, 128 in-flight instructions, etc.) and is also the best single core-type in a homogeneous design, (2) accelerator core-types that match distinct and intuitive ILP behaviors. The next section uncovers if this average core/accelerator core phenomenon is also applicable for an HCMP design with a given single-thread power budget.

C. Power-constrained HCMP Design

To understand the effect of imposing a power budget on the optimal HCMP design, we vary the power budget from 1.5W to 5.5W at 1W intervals.

The optimal HCMP designs for peak power constraints of 5.5W, 4.5W and 3.5W are found to be the same as that with the unconstrained power budget. This shows that the optimal, unconstrained HCMP design is still a good, power-efficient design. *This is due to the fact that frequency, like power, provides a disincentive for making cores arbitrarily complex.*

The core-types obtained in the optimal 4-core-type HCMP design with a 2.5W peak power constraint are given in Table IV. In Table IV, the A1 core-type is the best single core-type for a homogeneous design (average core-type) for the 2.5W power budget. Core-type W1 has a bigger width and hence concentrates on application phases with a width bottleneck. Core-type L1 has a bigger window and hence focuses on application phases with a window bottleneck. Core-type N1 has a higher clock frequency and hence targets application phases with an ILP bottleneck.

TABLE IV: MICROARCHITECTURE CONFIGURATIONS OF DIFFERENT CORE-TYPES IN THE 4-CORE-TYPE HCMP WITH 2.5W POWER BUDGET.

Core Type	Clock Period (ns)	ILP-extracting buffers (IQ,LSQ,ROB)	Widths (fetch, issue)	Caches (KB) (I$, D$)
A1	**0.6**	**32, 128, 128**	**3, 4**	**64, 64**
N1	**0.5**	**32, 64, 64**	**2, 2**	**16, 16**
W1	**0.7**	**48, 128, 128**	**4, 5**	**128, 32**
L1	**0.8**	**64, 128, 256**	**4, 5**	**32, 64**

At a 1.5W power budget, we find that there is no performance improvement if we have more than 3 core-types. This is because at the 1.5W power budget, the HCMP design is so constrained that there are only 202 feasible core-types (compared to 13,966 core-types in the original design space). The core-types obtained in the optimal 3-core-type HCMP design with a 1.5W peak power constraint are given in Table V.

TABLE V: MICROARCHITECTURE CONFIGURATIONS OF DIFFERENT CORE-TYPES IN THE 3-CORE-TYPE HCMP WITH 1.5W POWER BUDGET.

Core Type	Clock Period (ns)	ILP-extracting buffers (IQ,LSQ,ROB)	Widths (fetch, issue)	Caches (KB) (I$, D$)
A2	**0.6**	**48, 32, 64**	**3, 2**	**64, 8**
L2	**1.0**	**16, 128, 192**	**3, 4**	**128, 128**
W2	**0.8**	**16, 64, 64**	**3, 6**	**128, 16**

We see that irrespective of the number of core-types, the average core-type always shows up in the optimal design for both 2.5W and 1.5W power budgets.

IV. APPLICATION STEERING IN HCMP

Our steering algorithm works as follows. The program is continuously monitored using performance counters, a counter for each potential bottleneck. Every 10K instructions, the counters are compared against threshold values (and reset for monitoring in the next interval). If a given counter exceeds its threshold, the application is deemed to suffer the corresponding bottleneck in the measured interval on the current core. If there is an accelerator core-type that relieves any of the bottlenecks and does not worsen any of the bottlenecks, the application will migrate to that core-type, otherwise it will remain on or migrate to the average core-type. Each accelerator core-type has its own bottleneck signature which indicates its "bottleneck strengths", *i.e.*, bottlenecks that it relieves, and "bottleneck weaknesses", *i.e.*, bottlenecks that it worsens. The average core-type does not have a bottleneck signature because it is the default and all other core-types are gauged with respect to it.

Section IVA describes the performance counters and their thresholds, for diagnosing the application's bottlenecks. Section IVB describes a simple methodology for designers to derive bottleneck signatures for core-types in any HCMP design, *i.e.*, characterize bottleneck strengths and bottleneck weaknesses for each core-type relative to the best core-type, on average, in the design.

A. Diagnosing Bottlenecks

Table VI shows the performance counters used for diagnosing bottlenecks [25] of the application in the current interval on the current core-type.

TABLE VI: COUNTERS USED FOR DIAGNOSING BOTTLENECKS.

Performance counter	Comments
Branch misprediction stall counter	Number of cycles stalled due to branch misprediction
L2-cache stall counter	Number of cycles stalled due to L2 cache miss
I-cache stall counter	Number of cycles instruction fetch is stalled because of instruction cache miss
D-cache stall counter	Number of cycles a load is stalled because of data cache miss
Issue-width stall counter	Number of cycles in which ready instructions stall because of lack of issue width
Window stall counter	Number of cycles in which instruction dispatch is stalled because of a blocked issue queue, reorder buffer, load queue or store queue.
Cycle counter	Total cycles to execute the profiled segment

Only seven counters are used, hence, area and power overheads for bottleneck diagnosis are negligible. Each of these counters, except the cycle counter, denotes the cycles stalled due to the corresponding resource or factor. Once the measurement interval (10K instructions) is complete, the counters are normalized with respect to the total cycle count of the measurement interval. If a normalized performance counter is above a certain threshold, then the corresponding resource or factor is considered a bottleneck.

Thresholds are determined empirically using genetic algorithms for each resource/factor and for each core-type. We use a set of threshold values in the bottleneck signature table as the input to the genetic algorithm. For training, we select four benchmarks having different application characteristics: bzip, mcf, swim and mgrid. The objective function for the genetic algorithm is to maximize the performance value obtained by bottleneck steering using the given bottleneck signature table.

B. Deriving Bottleneck Signatures of Core-types

Bottleneck signatures of accelerator core-types are derived by the designers of the HCMP at design-time. This section describes a methodology for doing so.

The basic idea is to identify individual strengths and weaknesses of an accelerator core-type relative to the average core-type. For example, the strength of core-type L (large core) is its larger window and its weakness is its lower frequency. Then we identify the bottlenecks corresponding to its strengths and weaknesses. For example, the bottleneck corresponding to the strength of core-type L is the window bottleneck. Similarly, the bottlenecks corresponding to the weakness of core-type L are the L2 cache bottleneck and branch misprediction bottleneck, because these ILP-degrading

factors serialize execution, and serial regions favor high frequency over large window or width.

To generalize, let the bottlenecks corresponding to strengths be S1, S2, S3,... and those corresponding to weaknesses be W1, W2, W3,… Then, the signature of the core-type is {S1 || S2 || S3...} && !{W1 || W2 || W3...}. In other words, application phases having S1 or S2 or S3 bottlenecks and at the same time not having W1 or W2 or W3 bottlenecks, will prefer this core-type.

To demonstrate, we derive the bottleneck signatures of core-types in the unconstrained 4-core-type HCMP. It has core-types L, W, N and A. Core-type L has a bigger window (reorder buffer, issue queue and load/store queues) than the other core-types. Hence, application phases having a window bottleneck will prefer core-type L. In addition, if the application phase is to take advantage of a large window, it should not have branch misprediction or L2 cache bottlenecks. Similarly, core-type W has a wider width than all the other core-types, hence, application phases having a width bottleneck, no branch misprediction bottleneck, and no L2 cache bottleneck, will prefer core-type W. Core-type N has a narrower width and a smaller window. However, it has a higher clock frequency. Hence, application phases having low ILP will prefer core-type N. That is, application phases having branch misprediction or L2 cache bottlenecks will prefer core-type N. The above discussion is summarized in Table VII(a). If the diagnosed bottlenecks of the application do not match any of the above signatures, then it is steered to core-type A.

TABLE VII(A): BOTTLENECK SIGNATURE TABLE FOR THE UNCONSTRAINED 4-CORE-TYPE HCMP. (CORE-TYPE A DOES NOT HAVE A SIGNATURE.)

Bottleneck Signature	Core type
Window && !(CTRL || L2)	L
Width && !(CTRL || L2)	W
(CTRL || L2) && !(Width)	N

We can fine-tune the bottleneck signature table by taking into account the I-cache and D-cache behaviors. We see that core-type L has bigger caches. Hence, application phases having I-cache or D-cache bottlenecks will prefer core-type L. Similarly, core-type W has a bigger I-cache but a smaller D-cache. Hence, application phases having an I-cache bottleneck and no D-cache bottleneck will prefer core-type W. The updated bottleneck signature table is presented in Table VII(b).

Because the N core-type has a smaller window, one might conclude that this factor should be included in the signature's weaknesses term, *i.e.*, … !(Width || Window). However, if we have a L2 bottleneck, then L2-missed load instructions will definitely clog the window, causing there to be a window bottleneck, too. In other words, there is a window bottleneck whenever there is a L2 bottleneck. This will cause the naively prescribed signature of the N core-type ((… || L2) && !(… || Window)), to not be set when there is a L2 bottleneck. Hence, the signature of the N core-type is changed to (CTRL || L2) && !(Width). Similarly, when there is a L2 cache bottleneck, there will be I-cache and D-cache bottlenecks, too. Accordingly, we also exclude I$ and D$ from the weaknesses term of N's signature. Please note that this is the only exception to the algorithm for deriving signatures.

TABLE VII(B): UPDATED BOTTLENECK SIGNATURE TABLE FOR THE UNCONSTRAINED 4-CORE-TYPE HCMP.

Bottleneck Signature	Core type
(Window || I$ || D$) && !(CTRL || L2)	L
(Width || I$) && !(CTRL \|\| L2 \|\| D$)	W
(CTRL || L2) && !(Width)	N

C. Discussions

1) Power-constrained HCMPs

Bottleneck signatures are derived for the power-constrained HCMPs using the same method. Due to limited space, we do not walk through the derivation and only show the end results in Tables VIII and IX.

TABLE VIII: BOTTLENECK SIGNATURE TABLE FOR 4-CORE-TYPE HCMP UNDER 2.5W POWER BUDGET. (CORE-TYPE A1 DOES NOT HAVE A SIGNATURE.)

Bottleneck Signature	Core type
(Window) && !(CTRL || L2 || I$)	L1
(Width || I$) && !(CTRL \|\| L2 \|\| D$)	W1
(CTRL || L2) && !(Width)	N1

TABLE IX: BOTTLENECK SIGNATURE TABLE FOR 3-CORE-TYPE HCMP UNDER 1.5W POWER BUDGET. (CORE-TYPE A2 DOES NOT HAVE A SIGNATURE.)

Bottleneck Signature	Core type
(Window || D$) && !(CTRL || L2)	L2
(Width || I$) && !(CTRL || L2)	W2

2) Multi-programmed Workloads

For a multi-programmed workload, the bottleneck-driven steering algorithm works similar to a single-threaded workload. The algorithm determines the best core-type for each benchmark. In case more than one benchmark selects the same core-type, we randomly keep one of them in this core-type and assign other benchmark(s) to unassigned core-types.

3) Arbitrary HCMP Designs

For an arbitrary HCMP design, which does not follow the average/accelerator core phenomenon, we first find the best homogeneous core out of the cores in the heterogeneous design. We treat this core as the default core. Bottleneck signatures of other cores are obtained by following the algorithm in Section IVB.

4) Interaction with Operating System

We advocate the following approach for coordinating the operating system (O/S) and the hardware. First, the O/S is responsible for macro-scheduling – using nice levels and other factors to decide which threads are running versus sleeping in the next O/S quantum. Second, the O/S creates two logical partitions of the processor. For one partition, the O/S is heterogeneity-aware and is in sole control of thread-to-core mappings. A configuration register disables hardware steering for this partition. For the other partition, the O/S is agnostic of thread-to-core mappings, permitting single-thread acceleration via bottleneck-driven steering. When the current O/S quantum expires, the O/S needs to swap-out selected threads based on

macro-scheduling. Since threads in the second partition freely migrate possibly many times during the quantum, a status register is consulted by the O/S scheduler to locate threads to be swapped out and only their cores are interrupted.

The O/S is free to configure the two partitions as it desires, from no control at one extreme to full control at the other. The first partition (software control) allows not only O/S-driven optimization, but also privileged users can request particular core-types to run on.

V. Experimental Methodology

A. Benchmarks

We used the SPEC 2000 benchmark suite with the given reference inputs. (FabScalar uses the PISA ISA [1] which constrains us to an older version of gcc, hence, many SPEC 2006 benchmarks could not be compiled.) We simulate the first four billion instructions of each benchmark. Please note that it is important to simulate for a longer time to capture the different phases of the program and to see the effects of thread migration.

B. Metrics

The performance metric is billions-of-instructions-per-second (BIPS). To quantify the combined performance and power efficiency of our steering algorithm, we use $BIPS^3$/watt. $BIPS^3$/watt is a voltage-independent metric, hence, $BIPS^3$/watt is preferred over energy-delay-product [23]. Please note that *watt* here refers to average power and not peak power. (Peak power was only used in the selection of core-types at design-time, and only for the power-constrained HCMPs.)

C. Evaluation Methodology

The default thread migration overhead, for copying registers from old core to new core, is 100 cycles. We also show results for overheads of 1K and 10K cycles, and the differences are negligible because bottleneck-steering avoids unnecessary migrations. We chose 100 cycles as the default in order to maximize performance of the state-of-art sampling algorithm to which we compare. It is more sensitive to the overhead because it does trials on all the core-types during its sampling phase, causing more frequent migrations. Thread migration incurs extra L1 cache misses and these are accounted for in the results.

All graphs in the result section are normalized with respect to running solely on the average core-type of the particular HCMP (A, A1, or A2, depending on which HCMP), *i.e.*, the best homogeneous design.

We compare bottleneck-driven steering with the following steering algorithms: sampling algorithm, optimal steering algorithm and oracle steering algorithm. Note that the optimal and oracle steering algorithms cannot be implemented practically and are presented as upper bounds. However, we do include the thread migration overhead in all the algorithms. These algorithms are explained in detail below.

Sampling algorithm: To model the sampling algorithm [3][19], we introduce two parameters: switching interval and sampling interval. After every switching interval, the application is run on each core-type for the given sampling interval. After sampling, the application is run on the best core-type found during sampling, for the rest of the switching interval. We evaluated different switching and sampling intervals and found that a switching interval of 1M instructions and a sampling interval of 10K instructions works best.

Optimal steering algorithm: The best core for the current 10K segment is the core used for running the next 10K segment of the application.

Oracle steering algorithm: For every 10K segment, the best core is used for running the application.

VI. Results

A. Comparison of Different Steering Algorithms

In the unconstrained 4-core-type HCMP, the bottleneck-driven steering algorithm performs 12% better than the average core-type for the SPEC benchmark suite (Figure 1). We further see that the performance of the oracle steering algorithm and the optimal steering algorithm are 15% and 13% better than the average core-type, respectively. This shows that the bottleneck-driven steering algorithm captures most of the performance improvement available.

To make sure that our thresholds and bottleneck signatures are applicable in general, we also tested our algorithm with SPEC on a different input set (test) and MiBench. In both SPEC (test) and MiBench, we found that bottleneck steering is able to tap 99% of the performance yielded by optimal steering.

For a deeper understanding, we find the misprediction rate and switching rate of the sampling algorithm and the bottleneck steering algorithm. The misprediction rate is defined as the ratio between the number of 10K segments in which the best core is not predicted correctly and the total number of 10K segments. Similarly, the switching rate is defined as the ratio between the number of times the program switches cores and the total number of 10K segments. The extent to which a given misprediction impacts performance depends on how suboptimal the predicted core-type is for the 10K segment, whether or not it underperforms the average core-type (potential slow-down), and whether or not a core switch was undertaken (overhead). A core switch always incurs a time penalty.

In Figure 2, we see that the switching rate for the bottleneck steering algorithm (switch_bot) is less than or comparable to the sampling algorithm (switch_samp). Similarly, for all benchmarks except gcc, ammp and equake, the misprediction rate for bottleneck steering (misp_bot) is less than or comparable to sampling (misp_samp). Hence, for most benchmarks, the bottleneck steering algorithm performs better than or comparable to the sampling algorithm. For gcc and ammp, because of higher misprediction rates, bottleneck steering performs worse than sampling. For equake, even though bottleneck steering has a higher misprediction rate, bottleneck steering performs slightly better than sampling because of a lower switching rate and potentially mild suboptimality of the mispredicted core-type.

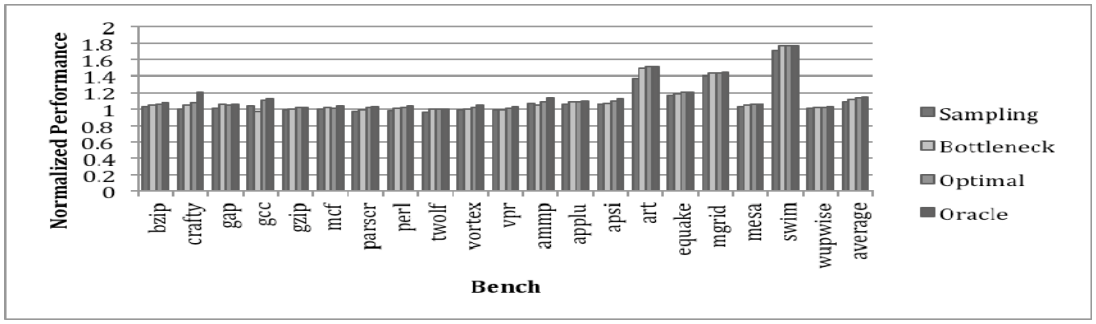

Figure 1: Performance of bottleneck-driven and other steering algorithms for 4-core-type HCMP.

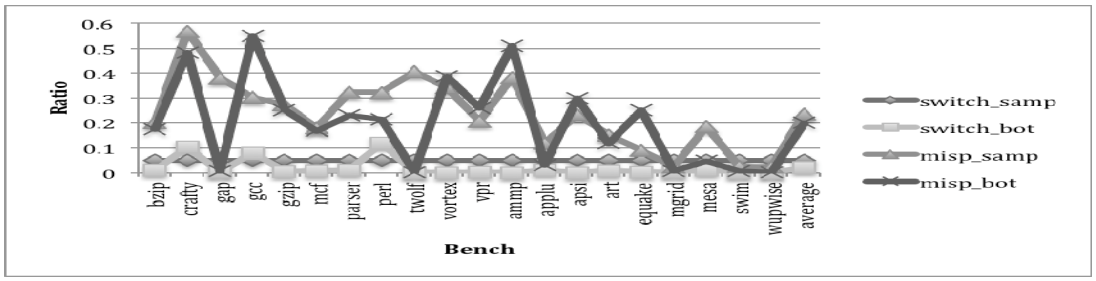

Figure 2: Comparison of switching rate and misprediction rate of sampling algorithm with that of bottleneck-driven steering algorithm.

Figure 3 shows the occupancy of different core-types in the unconstrained 4-core-type HCMP assuming oracle steering. It is interesting to see that there are significant differences in the occupancies of different core-types, across benchmarks.

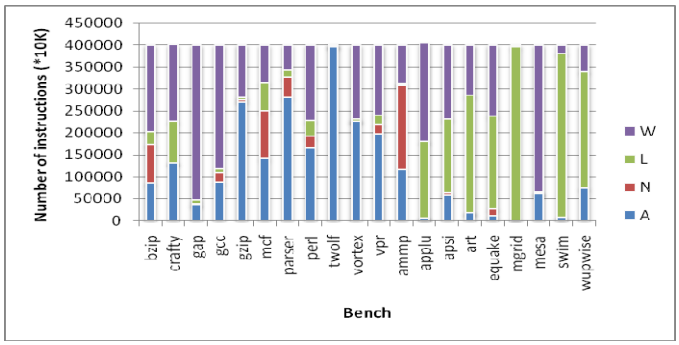

Figure 3: Number of instructions spent in each core-type in the unconstrained 4-core-type HCMP.

B. Robustness Analysis

1) Efficiency Metric

For the robustness analysis, we first study the change in metric from performance (BIPS) to efficiency ($BIPS^3$/watt). To evaluate the efficiency of our steering algorithm, we compare $BIPS^3$/watt for our steering algorithm with that of the average core-type. In Figure 4, we see that our steering algorithm outperforms the average core-type by 33% in terms of efficiency. In contrast, the sampling algorithm only performs 25% better than the average core-type.

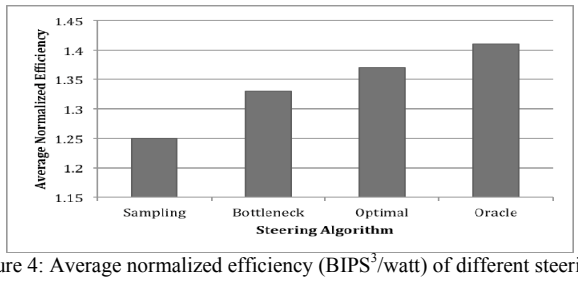

Figure 4: Average normalized efficiency ($BIPS^3$/watt) of different steering algorithms.

2) Changing Thread Migration Overhead

Figure 5 shows sensitivity to the thread migration overhead. With a thread migration overhead of 1,000 cycles, the average performance gain is 11%. If we further increase the thread migration overhead to 10,000 cycles, the average performance gain reduces to 10%. This shows that even with one or two orders of magnitude increase in thread migration overhead, the performance does not reduce drastically.

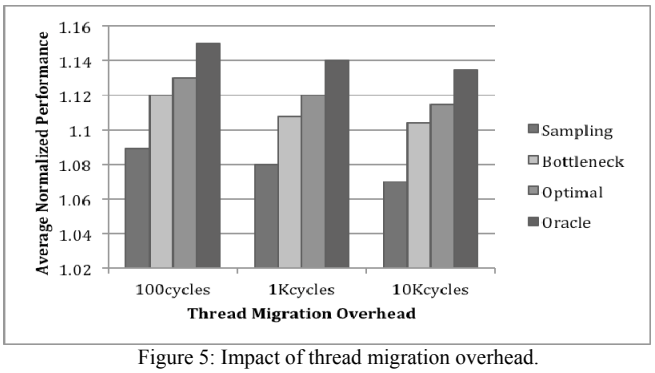

Figure 5: Impact of thread migration overhead.

C. Power-constrained HCMPs

The 2.5W and 1.5W HCMP results are shown in Figure 6. (Please see Section IIIC for constrained HCMP design configurations.) At the 2.5W power budget, the bottleneck-driven steering algorithm in the 4-core-type HCMP performs 8% better than the average core-type (at 2.5W power budget), as opposed to the 11.5% performance gain achieved by oracle steering. Note that the performance value obtained at 2.5 W is normalized to the best homogeneous core-type at 2.5W. At the 1.5W power budget, the bottleneck-driven steering algorithm in the 3-core-type HCMP performs 11% better than the average core-type (at 1.5W power budget), as opposed to the 15% performance gain achieved by oracle steering.

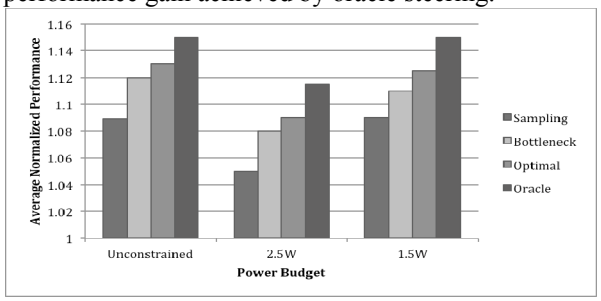

Figure 6: Performance of bottleneck-driven and other steering algorithms for HCMPs at different power budgets.

D. Multi-programmed Workloads

For creating multi-programmed workloads from SPEC benchmarks, we make use of the clusters obtained by subsetting of SPEC benchmarks by Phansalkar et al. [30]. We create four classes of 4-benchmark workloads for the 4-core-type HCMP by selecting benchmarks from various clusters. In the first class, we select four benchmarks all from different clusters. In the second, third and fourth classes, we get benchmarks from three clusters, two clusters and one cluster, respectively. Figure 7 shows the performance results for the 4-core-type HCMP for multi-programmed workloads. Label N (A, B, C, D) denotes a workload created from N clusters, specifically, the A^{th}, B^{th}, C^{th} and D^{th} cluster in Table 2 in [30].

In Figure 7, for oracle steering, for every 10K instruction interval of the multi-programmed workload, we choose the best application-to-core mapping out of 4! (4 factorial) possible mappings. Sampling algorithm is not shown because of massive overhead for multi-programmed workload. We see that average normalized performance of bottleneck-driven steering algorithm of 4-core-type HCMP is 9% better than the 4-core-type homogeneous CMP. In contrast, the oracle steering algorithm performs 18% better than the 4-core-type homogeneous CMP. This shows that there is room for improvement in the bottleneck-driven steering algorithm for multi-programmed workloads by using a more sophisticated conflict resolving mechanism (instead of just random), such as ranking the severity of bottlenecks. We leave this for future work.

E. Arbitrary HCMP Design

In Figure 8, we evaluate our bottleneck-driven steering algorithm on a 3-core-type HCMP which does not follow the average/accelerator core phenomenon. The 3-core-type HCMP does not have an average core-type and has core-types N, L and W. We see that bottleneck-driven steering is able to obtain most of the performance of optimal steering. Note that this graph has been normalized to the performance of oracle steering.

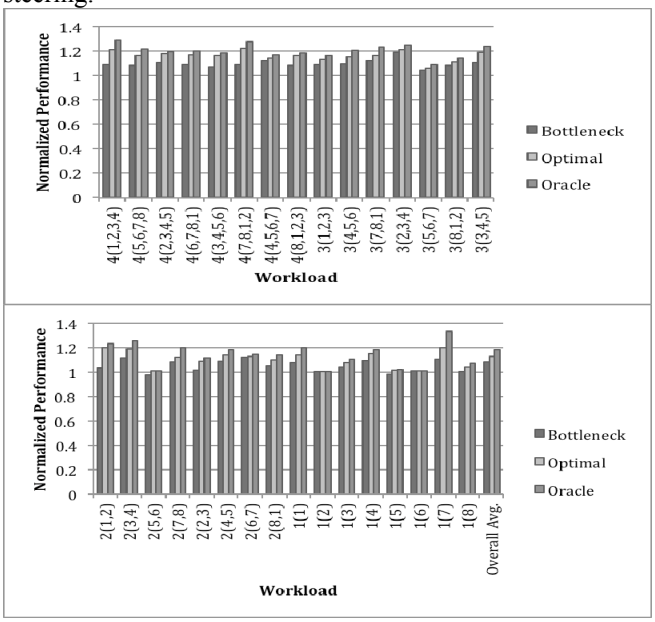

Figure 7: Performace of bottleneck-driven steering algorithm for 4-core-type HCMP for multi-programmed workloads.

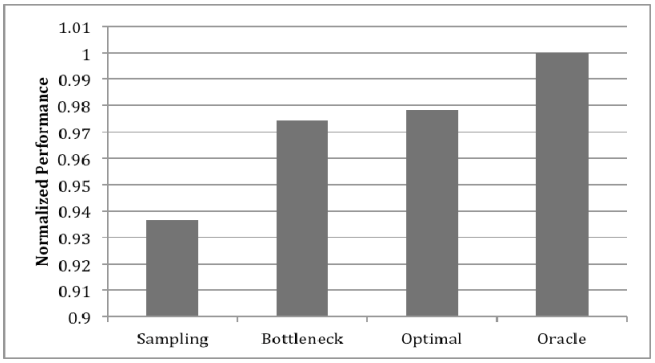

Figure 8: Performace of bottleneck-driven steering algorithm for a 3-core-type HCMP having core-types N, L and W.

VII. Related Work

A. HCMP Design

Early work on HCMPs assumed that the cores consisted of different generations of the Alpha processor family or frequency-scaled versions of x86 processors [18][19]. Thus, the constituent cores are monotonic.

Subsequently, Kumar et al. [20] demonstrated significant performance benefits for multi-programmed workloads when the HCMP is designed from the ground-up. The resulting cores are non-monotonic. Their non-monotonic designs differ from ours – and are contrary to average core / accelerator core – for at least two reasons. First, they do not consider the effect of superscalar complexity on core frequency. All cores in their design space have the same pipeline depth and frequency, from the smallest/narrowest core to the biggest/widest core.

This is evident in the fact that the best CMP design at their largest area and power budget is a homogeneous design comprised of only one core type: the most complex core type in the design space. In this paper, even when we do not constrain area or power, the optimal CMP does not contain our most complex core and it is heterogeneous. The ILP / frequency trade-off is a sufficient forcing function for both restraint and core diversity. Second, the focus of their HCMP design is on maximizing throughput and ours is on minimizing latency of a single thread (core-selectable processor), although we did present a global aspect of the steering algorithm for multiple contending threads. Together, these two differences explain why Kumar et al. did not observe the average core / accelerator core phenomenon. We believe the two studies are complementary and provide unique perspectives for the architecture community. Finally, our work includes a steering algorithm whereas Kumar et al. used oracle steering.

Azizi et al. [2] studied the energy-performance tradeoffs in processor architecture and performed a marginal-cost analysis. We find the optimal HCMP designs for maximizing single-thread performance for both unconstrained and constrained power budgets. Further, our analysis is based on a detailed high-fidelity RTL model [6].

Choudhary et al. [6] studied a workload-agnostic palette of 21 core types. Lee et al. [21] found the optimal HCMP for efficiency ($BIPS^3$/watt) using K-means clustering. Neither work uncovers the average/accelerator core insight or leverages the unified view of core selection and application steering.

Dynamic voltage and frequency scaling (DVFS) is orthogonal to heterogeneity. Further, Grochowsky et al. [13] found HCMP provided more benefit compared to DVFS. In addition, recent research has shown that DVFS is showing diminishing returns on newer platforms [34]. Non-monotonic HCMP provides a compelling alternative.

B. Application Steering in HCMP

Current techniques for steering fall into the following categories: sampling-based approaches, heuristics-based approaches, static approaches and model-based approaches.

1) Sampling-based Approaches

Sampling-based approaches run the application on every core type after a switching interval to determine the best core type for the application for the next interval.

Kumar et al. [19] proposed the sampling algorithm for steering in a monotonic HCMP. Becchi et al. [3] proposed a steering algorithm, which relied on the speedup factor (performance improvement on a fast core relative to a slow core) for a monotonic HCMP. The speedup factor was computed by running a thread on each core separately. Winter et al. [41] explored thread scheduling and global power management techniques in HCMP. They compared different algorithms like brute force, greedy and local search for thread scheduling. All examined schemes required sampling to determine the best thread-to-core assignment. Sawalha et al. [33] proposed recording IPCs of the different program phases in a table and using it for scheduling when the phase recurs. However, it still required sampling the performance of the threads on each core type. Sondag et al. [37] also records past phase behavior, but uses a static analysis tool to mark phases in the binary *a priori* and also insert performance measurement code for each phase.

Each of the above sampling techniques suffers from the overhead of successively migrating the application on all the core types after every switching interval. Further, this overhead would increase with the number of core types [8]. In our technique, the application directly migrates from the current core type to the best core type without any need for sampling.

2) Heuristics-based Approaches

Saez et al. [32] proposed a dynamic algorithm for steering program phases among homogeneous cores running at two different frequencies (emulating the big core/little core class of monotonic HCMP). They measured the L2 miss rate to find the optimal core mapping. Koufaty et al. [17] used proprietary tools to emulate an asymmetric system where the cores differed in the number of micro-ops that could be retired per cycle. They assumed cores of two types: a big core capable of retiring four micro-ops per cycle and a small core capable of retiring a single micro-op per cycle. Still, these cores are monotonic in nature and do not exploit the full performance advantage of HCMP design. They correlated the application behavior with off-chip and on-chip stalls. They scheduled the applications suffering from memory stalls and other resource stalls on the smaller core and the rest on the bigger core. Patsilaras et al. [29] dynamically scheduled threads based on the amount of MLP which is estimated by the number of L2 misses.

It has been shown that the above approaches may cause suboptimal scheduling as memory intensity alone is not enough for determining the optimal workload-to-core mapping [8]. Further, these techniques were conceived and evaluated for the big core/little core class of monotonic HCMPs, and it is not clear how they can be extended to non-monotonic HCMPs.

Rodrigues et al. [31] proposed a tiled architecture wherein each tile has two core types, one with strong integer resources and weak floating-point resources and *vice versa* for the other. Hence, the cores are non-monotonic, but in a less subtle way than our HCMP: thread swapping is based on percentages of integer and floating-point instructions.

3) Static Approaches

Static approaches use offline techniques to determine the optimal core mapping for the code fragment.

Chen and John [5] proposed a scheduling algorithm that matches programs and cores. Programs and cores are projected onto a common multi-dimensional space: programs, based on their resource demands, and cores, based on their configurations. The scheduler then assigns programs to cores based on Euclidean distances between them. The approach exploits only inter-program diversity and does not adapt to phase changes within programs. Shelepov et al. [35] proposed a static algorithm for steering program phases among frequency-scaled versions of the same processor. They embedded reuse distance profile signatures into the binary,

which enable the core to quickly estimate the L2 cache miss rate. Using the L2 cache miss rate, they were able to find the optimal core mapping.

These static approaches require that workloads be profiled beforehand. Further, the workload behavior may be drastically impacted by the input data [8]. On the other hand, our technique is a dynamic technique.

4) Model-based Approaches

Model-based approaches use a dynamic model to determine the best core for the application.

Craeynest et al. [8] collected MLP and ILP information to predict the performance on the other core in a big core/little core style monotonic HCMP. Dubach et al. [9] used machine learning to dynamically predict the best hardware configuration for a program phase in a reconfigurable processor. Lukefahr et al. [22] designed a predictive-based feedback controller for switching in a closely-coupled big core/little core monotonic HCMP.

It is unclear how these techniques can be adapted to a non-monotonic HCMP. Our technique uses a bottleneck model to predict the optimal workload-to-core mapping in a non-monotonic HCMP.

VIII. Summary and Future Work

HCMPs are an attractive substrate for improving single-thread performance and energy efficiency. More powerful classes of HCMPs employ *non-monotonic core types* where each core type is performance-optimized to different instruction-level behavior and hence cannot be ranked – different applications achieve their highest performance on different cores. Although non-monotonic heterogeneous designs offer higher performance potential than either monotonic heterogeneous designs or homogeneous designs, steering applications to the best-performing core is challenging due to performance ambiguity of core types.

In this paper, we present a unified view of *selecting* non-monotonic core types at design-time and *steering* programs to cores at run-time. After comprehensive evaluation, we found that with N core types, the optimal HCMP for single-thread performance is comprised of an "average" core-type coupled with N-1 "accelerator" core-types that relieve distinct resource bottlenecks in the average core-type. This inspires a complementary steering algorithm in which the application is continuously monitored for bottlenecks. The application is migrated to a core-type that relieves the bottlenecks.

HCMPs open up a whole new direction of microarchitecture research. Many microarchitectural optimizations, which have been proposed before, have never been put into practice. One possible reason is that they do not provide universal benefit and may actually degrade performance in some circumstances. As each core-type targets a narrow workload space, HCMP provides a great platform to reconsider these optimizations. Techniques that harness far-flung ILP, like run-ahead execution [24], continual flow pipelines, checkpoint processing and recovery [38], etc., are worth exploring for the large (L) core-type in our 4-core-type HCMP. Similarly, optimizations that help in finding nearby ILP, like trace cache, clustered architecture, value prediction, etc. [16], are worth exploring for the wide (W) core-type.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This research was supported by NSF grants CCF-0811707 and CCF-1018517, and gifts from Intel and IBM. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] T. Austin, E. Larson and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling, IEEE Micro, Feb. 2002.

[2] O. Azizi, A. Mahesri, B. Lee, S. Patel and M. Horowitz. Energy-performance Tradeoffs in Processor Architecture and Circuit Design: a Marginal Cost Analysis, *Int'l Symposium on Computer Architecture*, 2010.

[3] M. Becchi and P. Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures, *Int'l Conference on Computing Frontiers*, 2006.

[4] D. Brooks, V. Tiwari and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations, *Int'l Symposium on Computer Architecture*, 2000.

[5] J. Chen and L. John. Efficient Program Scheduling for Heterogeneous Multi-core Processors, *Design Automation Conference*, 2009.

[6] N. Choudhary, S. Wadhavkar, T. Shah, H. Mayukh, J. Gandhi, B. Dwiel, S. Navada, H. Najaf-abadi and E. Rotenberg. FabScalar: Composing Synthesizable RTL Designs of Arbitrary Cores within a Canonical Superscalar Template, *Int'l Symposium on Computer Architecture,* 2011.

[7] N. Choudhary. FabScalar: Automating the Design of Superscalar Processors, Ph.D. Thesis, Department of Electrical and Computer Engineering, North Carolina State University, May 2012.

[8] K. Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez and J. Emer. Scheduling Heterogeneous Multi-Cores through Performance Impact Estimation (PIE), *Int'l Symposium on Computer Architecture*, 2012.

[9] C. Dubach, T. Jones, E. Bonilla and M. Boyle. A Predictive Model for Dynamic Microarchitectural Adaptivity Control, *Int'l Symposium on Microarchitecture,* 2010.

[10] H. Esmaeilzadeh, E. Blem, R. Amant, K. Sankaralingam and D. Burger. Dark Silicon and the End of Multicore Scaling, *Int'l Symposium on Computer Architecture,* 2011.

[11] D. Goldberg. Genetic Algorithms in Search, Optimization, and Machine Learning, Reading, Mass [u.a.]: Addison-Wesley.

[12] N. Goulding, J. Sampson, G. Venkatesh, S. Garcia, V. Bryskin, J. Martinez, S. Swanson and M. Taylor. GreenDroid: A Mobile Application Processor for a Future of Dark Silicon, *HotChips*, 2010.

[13] E. Grochowski, R. Ronen, J. Shen and H. Wang. Best of Both Latency and Throughput, *Int'l Conference on Computer Design*, 2004.

[14] P. Greenhalgh. Big.LITTLE processing, white paper, ARM, 2011.

[15] M. Hill and M. Marty. Amdahl's law in the multicore era, In IEEE Computer, 41(7), 2008.

[16] D. Kaeli and P. Yew. Speculative Execution in High Performance Computer Architectures, Eds. CRC Press, 2005.

[17] D. Koufaty, D. Reddy and S. Hahn. Bias Scheduling in Heterogeneous Multicore Architectures, *EuroSys,* 2010.

[18] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan and D. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction*, Int'l Symposium on Microarchitecture*, 2003.

[19] R. Kumar, D. Tullsen, P. Ranganathan N. Jouppi and K. Farkas. Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance, *Int'l Symposium on Computer Architecture*, 2004.

[20] R. Kumar, D. Tullsen and N. Jouppi. Core Architecture Optimization for Heterogeneous Chip Multiprocessors, *Int'l Conference on Parallel Architecture and Compilation Techniques,* 2006.

[21] B. Lee and D. Brooks. Illustrative Design Space Studies with Microarchitectural Regression Models, *Int'l Symposium on High Performance Computer Architecture*, 2007.

[22] A. Lukefahr, S. Padmanabha, R. Das, F. Sleiman, R. Dreslinski, T. Wenisch and S. Mahlke. Composite Cores: Pushing Heterogeneity into a Core, *Int'l Symposium on Microarchitecture*, 2012.

[23] A. Martin and M. Nystroem. ET2: A Metric for Time and Energy Efficiency of Computation, *Power-Aware Computing*, 2001.

[24] O. Mutlu, J. Stark, C. Wilkerson and Y. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors, *Int'l Symposium on High-Performance Computer Architecture*, 2003.

[25] S. Navada, N. Choudhary and E. Rotenberg. Criticality-driven Superscalar Design Space Exploration, *Int'l Conference on Parallel Architecture and Compilation Techniques*, 2010.

[26] H. Najaf-abadi, N. Choudhary and E. Rotenberg. Core-Selectability in Chip Multiprocessors, *Int'l Conference on Parallel Architecture and Compilation Techniques*, 2009.

[27] H. Najaf-abadi and E. Rotenberg. Architectural Contesting, *Int'l Symposium on High Performance Computer Architecture*, 2009.

[28] H. Najaf-abadi and E. Rotenberg. Configurational Workload Characterization, *Int'l Symposium on Performance Analysis of Systems and Software*, 2008.

[29] G. Patsilaras, N. Choudhary and J. Tuck. Efficiently Exploiting Memory Level Parallelism on Asymmetric Multicore Processors in the Dark Silicon Era, *ACM Transactions on Architecture and Code Optimization*, 2012.

[30] A. Phansalkar, A. Joshi, L. Eeckhout and L. John. Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites, *Int'l Symposium on Performance Analysis of Systems and Software*, 2005.

[31] R. Rodrigues, A. Annamalai, I. Koren, S. Kundu and O. Khan. Performance per Watt Benefits of Dynamic Core Morphing in Asymmetric Multicores, *Int'l Conference on Parallel Architecture and Compilation Techniques*, 2011.

[32] J. Saez, M. Prieto, A. Fedorova and S. Blagodurov. A Comprehensive Scheduler for Asymmetric Multicore Processors, *EuroSys*, 2010.

[33] L. Sawalha, S. Wolff, M. Tull and R. Barnes. Phase-guided Scheduling on Single-ISA Heterogeneous Multicore Processors, *Euromicro Conference on Digital System Design*, 2011.

[34] E. Le Sueur and G. Heiser. Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns, *Workshop on Power Aware Computing and Systems*, 2010.

[35] D. Shelepov, J. Saez, S. Jeffery, A. Fedorova, N. Perez, Z. Huang, S. Blagodurov and V. Kumar. HASS: a Scheduler for Heterogeneous Multicore Systems, *ACM Operating System Review* 43, 2 (2009), 66-75.

[36] T. Sherwood, E. Perelman, G. Hamerly and B. Calder. Automatically Characterizing Large Scale Program Behavior, *Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[37] T. Sondag and H. Rajan. Phase-based Tuning for Better Utilization of Performance-Asymmetric Multicore Processors, *Int'l Symposium on Code Generation and Optimization*, 2011.

[38] S. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, Continual Flow Pipelines, *Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[39] M. Suleman, O. Mutlu, M. Qureshi and Y. Patt. Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures, *Int'l Conference on Architectural Support for Programming Languages and Operating Systems,* 2009.

[40] S. Wilton and N. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model, *IEEE Journal of Solid State Circuits*, 1996.

[41] J. Winter, D. Albonesi and C. Shoemaker. Scalable Thread Scheduling and Global Power Management for Heterogeneous Many-Core Architectures, *Int'l Conference on Parallel Architecture and Compilation Techniques*, 2010.