Architectural Contesting

Hashem H. Najaf-abadi Eric Rotenberg
Department of Electrical and Computer Engineering
North Carolina State University
{hhashem, ericro}@ece.ncsu.edu

Abstract

This paper presents results showing that workload behavior tends to vary considerably at granularities of less than a thousand instructions. If it were possible to adjust the microarchitecture to suit the workload behavior at such rates, significant single-thread performance enhancement would be achievable. However, previous techniques are too sluggish to be able to effectively respond to such fine-grain change.

An approach is proposed that exploits the multi-core trend to enable swift adjustment in the employed microarchitecture upon variation in workload behavior. A number of cores that are each custom-designed for optimum performance under a class of workloads concurrently execute code in a leader-follower arrangement. In this manner, effective execution automatically and fluidly transfers to the most suitable microarchitecture as the workload behavior varies. We refer to this approach as architectural contesting.

Two-way contesting yields an average speedup of 15% (maximum speedup of 25%) over a benchmark's own customized core. The paper also explores the interplay between contesting and the number of core types available in the heterogeneous multi-core. This exposes the broader issue of constrained heterogeneous multi-core design and how it influences, and may be influenced by, contesting.

1. Introduction

A major impediment to the effectiveness of microarchitectural techniques is their high dependence on the workload behavior. It is for this reason that previous studies have proposed techniques that enable the employed microarchitecture to dynamically change and become more suitable for the immediate workload behavior. Such techniques can be broadly categorized as either *adaptational* or *migrational* approaches. Adaptational approaches are based on a single processor design with adjustable design features (e.g., [9]). Migrational approaches are based on a number of differently designed processing cores, *i.e.*, a heterogeneous multi-core (e.g., [14]).

Regardless of the approach, the rate at which the employed architecture can be effectively changed depends on the rate at which 1) change in workload behavior can be *detected*, 2) the most suitable architecture for the new code region can be *determined*, and 3) the change can be *performed*. The challenge with adaptational techniques is in determining when and how the architecture should change. Similarly, the challenge with migrational techniques is in determining when and to which core execution should be transferred.

In this paper, we show that the speed of adjusting to change in workload behavior that is essential for high performance enhancement, is too fine-grain to be achieved with prior approaches. However, the availability of multiple cores can be exploited to enable the speedy transfer of execution to the most suitable architecture. In the proposed approach, code is simultaneously executed on a number of cores, each architected for optimum performance under a different class of workload behavior. With each core broadcasting its instruction results to the other cores, completion of instructions can be expedited in cores that are not suitable for the immediate code region. Thus, upon change in the workload behavior, the core that is most suitable for the new workload behavior will be able to automatically take the lead. In other words, detecting changes in workload behavior, determining the best architectural configuration, and transferring execution to that configuration, all take place automatically and fluidly with minimal latency. We refer to this technique as *architectural contesting* (or simply *contesting*).

Contesting is orthogonal to other sources of single-thread performance enhancement, as it exploits a unique source of performance enhancement, namely, fine-grain customization. Moreover, like other redundant threading architectures, it can be employed on a need-to-have basis, providing robustness in how resources are employed (throughput or single-thread performance) and how performance and power are balanced.

In the next section, the benefit of being able to change the processor configuration at different rates is examined. In Section 3, prior related work is outlined and relevant issues discussed. Section 4 discusses an implementation of architectural contesting. Section 5.1 describes the simulator, benchmarks, and the methodology for finding application-level customized cores for the SPEC2000 integer benchmarks. These benchmark-customized cores form the palette of core types for designing the various heterogeneous CMPs used in this paper. Section 5.2 presents results and analysis of 2-way contesting (contesting between two cores) assuming all core types are available in the CMP. Section 6 evaluates contesting in the context of more constrained heterogeneous CMP designs that have fewer core types. This requires an in-depth analysis of different figures of merit for guiding the selection of core types to be included in the CMP. Continuing where Section 6 leaves off, Section 7 discusses the subtle yet important interplay between contesting and the broader issue of designing constrained heterogeneous CMPs. Section 8 concludes the paper.

Below are some highlights from the results and analysis presented in Sections 5, 6, and 7:

- 2-way contesting yields an average speedup of 15% (maximum speedup of 25%) over a benchmark's own customized core.
- For most benchmarks, most of the performance enhancement of contesting comes from heterogeneity in the microarchitecture, although the benefit of heterogeneity in the L2 caches is noticeable in a few cases. For both sources, it is contesting that enables this heterogeneity to be exploited at a fine granularity.
- The speedup of contesting is even more pronounced in constrained heterogeneous CMPs: yielding an average speedup of 22% compared to executing the benchmark on the most suitable available core. The availability of fewer core types reduces the benefit of application-level heterogeneity. Contesting can compensates for this deficit.
- Compared to the best homogeneous CMP design, a constrained heterogeneous CMP design achieves an average speedup of 11% without contesting and 34% with contesting. In other words, contesting triples the single-thread performance advantage of heterogeneity in this system.
- Compared to the best homogeneous CMP design, contesting between only two core types yields the same or higher single-thread performance enhancement as executing on the best of three core types.

2. Motivation: The speed of change

For each SPEC2000 integer benchmark we evaluate the ability to switch execution between two microarchitectural configurations. The configurations are chosen from among eleven configurations, each customized for one of the benchmarks. They were extracted through a simulated annealing exploration process for 70nm technology (see section 5 for further details).

The execution of each benchmark's 100-million instruction Simpoint [16] was simulated on the customized configuration of each benchmark and the number of cycles to retire every 20 dynamic instructions was logged. Then, for each benchmark and every combination of two configurations, every 20-instruction region was considered to be retired at the rate of the faster of the two for that region – while factoring in the clock periods. The time spent in each region was then aggregated to determine the total execution time, and from that the best two configurations for each benchmark. The same process was repeated for 40-instruction regions, by summing the execution time of neighboring 20-instruction regions. The whole process was repeated for regions of up to 83 million instructions.

Figure 1 illustrates the speedup attained for each benchmark over the performance of its own customized architecture by switching execution between two core configurations at different rates. Also indicated in these graphs are the two configurations that provided the best speedup at each granularity. The different data-point symbols indicate different two-core combinations. While the best pair of cores for switching execution between rarely varies across different granularities for benchmarks such as *bzip*, it is highly dependent on the granularity of switching in benchmarks such as *perl*. At the coarsest granularity (i.e. the whole Simpoint), each benchmark achieves its best performance on its own customized configuration and attains no speedup.

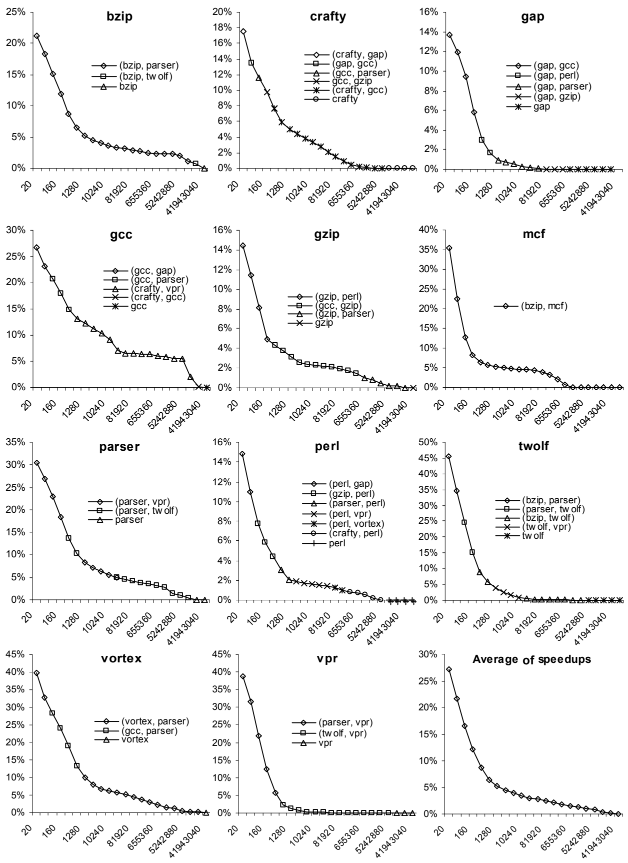

Figure 1. Percentage speedup of switching execution between two different configurations at different granularities, over the performance of the benchmark's own customized configuration.

These results illustrate that the greatest potential of being able to dynamically adjust the microarchitecture to the workload is attainable at granularities of less than a thousand instructions. While the benchmarks *gcc* and *gzip* attain a modest portion of their maximum speedup in coarser granularities, most benchmarks display little or no performance enhancement with coarser switching of the microarchitecture. The knee in the curve in most of these benchmarks occurs near the 1280-instruction granularity. For instance the graph for average speedup displays a mere 5% speedup for granularities in this range, while displaying up to ~25% speedup for finer granularities. Previously proposed approaches to dynamically adjusting the architecture to the workload are unable to exploit such fine-grain change in workload behavior.

In most cases the customized microarchitecture of a benchmark is among the best two cores to switch execution between. However, for *twolf,* the benchmark that attains the largest fine-grain speedup, the customized architectures of *vortex* and *parser* are the best two. This is notable as an application-level customized architecture is forced to compromise performance in fine-grain regions in order to attain good overall performance. This infers that switching execution between architectures that are custom designed for applications may not necessarily provide the best performance enhancement from fine-grain switching. Nevertheless, these architectures are good candidates for improving application-level performance and multi-programming throughput – issues of general importance in a CMP design.

3. Related work and discussion

The *slipstream* paradigm [3] employs two simultaneous execution streams of the same code that interact to improve overall single-thread performance. One execution stream is expedited through speculatively skipping ineffectual work, but needs to be checked by a redundant stream. However, the redundant stream itself is also expedited as the speculative stream passes it highly accurate branch and value predictions. More recent related work is the *paceline* leader-checker microachitecture [15]. In this approach a leader-core runs the thread at a higher-than-rated frequency, while passing execution hints and prefetches to a safely-clocked checker core.

In both these techniques, the leading core is fixed (paceline occasionally swaps cores' roles for temperature control) and is expedited in a manner that needs to be checked for correctness, thus the need for forwarding instruction results to a checker. In contesting however, the leading core varies depending on the workload behavior, and gains lead purely because it is more suitable for the immediate region of code. Thus, there is no need for it to be checked. Instruction results are forwarded to the other cores not for checking, but rather to keep them from falling behind so they can take lead as swiftly as possible when the workload behavior changes.

The *datascalar* paradigm [12] enables single thread performance enhancement through enabling the distribution of the program data-set across the local memory of multiple cores. Frequency scaling techniques [21] are also related work in that they provide variability in the performance-power tradeoff.

This paper culminates our precursor proposal [20], which cursorily evaluated contesting with little architectural diversity (two processor widths).

3.1. Changing the microarchitecture

One approach to enabling change in the employed microarchitecture is to have an adaptable microarchitecture. However, it is generally infeasible to maintain a balanced pipeline as individual architectural units are scaled. This imbalance is inevitable due to the fact that all microarchitectural units are tied to a common pipeline structure.

Reconfigurable computing [23] exploits field-programmable technology to build processors that can fundamentally transform their architecture. Such approaches can provide abundant architectural diversity. However, the implementation of a specific architecture in reconfigurable technology is prone to severe sub-optimal fixed-configuration performance.

A different approach is to employ multiple processing cores with different designs. Kumar et al. investigate the use of heterogeneous multi-core architectures [1]. They show that the incorporation of processors that have a range of high to low complexity (and performance) in a constrained die area can result in greater throughput for multi-threaded workloads. In more recent work, they find "non-monotonic" architectural diversity to result in better throughput enhancement [14].

3.2. Determining the best microarchitecture

A large range of prior work has studied different approaches to enable aspects of a processor microarchitecture to change and become more suitable for the immediate workload behavior in order to achieve better power efficiency. Ponomarev et al. [5] and Folengnani et al. [6] propose approaches to learning the optimum issue queue size, and Yang et al. [7] propose cache miss-rate as a metric for determining when to downsize or upsize an adaptable I-cache.

In Complexity Adaptive Processing (CAPs) [4], a single processor is architected such that the tradeoff between IPC and clock-rate can be dynamically altered. An essential component of the CAP architecture is the "configuration control" unit which selects the optimal configuration for the immediate workload through a heuristic learning technique or profiling information. Dhodapkar and Smith [2] and Balasubramonian and Albonesi [8] propose tuning processes for identifying the best-suited configuration for the immediate code. *Temporal* approaches, such as the Rochester algorithm [8] or signature based approaches [2], require lengthy tuning processes. *Positional* approaches [10] enable faster adaptation, and have been found to be more effective, yet they are unsuitable for fine-grain switching due to the drastic increase in storage requirement.

Dropsho et al. [11] propose the separation of microarchitectural units, in what is referred to as the Globally-Asynchronous Locally-Synchronous (GALS) design. In this approach, different units are asynchronous to each other, thus allowing each to be scaled independently. This allows for more predictability in the effect of independently scaled units on overall performance. Thus, it relieves the system of the need for an exhaustive tuning process that tries out different configurations.

Chen et al. [13] investigate the potential of employing pipelines of different widths and dynamically directing work to them based on local ILP. They use the parallelism metrics gathered from a dynamic Data Dependence Tracking mechanism to steer windows of instructions to suitable pipelines. In order to avoid most of the inter-cluster penalty, they limit switching between clusters to coarse granularities and continuously forward values to the disabled pipeline. However, data dependence tracking takes a two-prong view of the microarchitectural design space, consisting of either simple pipelines that can be clocked at high frequencies or wide superscalars that can be clocked at lower frequencies. In reality, a large range of microarchitectural parameters affect overall performance. After all, even a wide superscalar processor can be clocked at a high frequency if it is pipelined deeply.

4. Implementation

The implementation of a contesting system resembles other redundant-execution leader-follower architectures, such as Slipstream [3], SRT [18], AR-SMT [17], DCE [22], Paceline [15] and DataScalar [12]. The main novelty of contesting is not in the employed mechanisms, but rather the purpose for which they are employed. This section describes the implementation of a contesting system. While the description is generalized for N-way contesting, the subsequent results section is for a 2-way contesting system.

Figure 2 illustrates an architectural contesting multi-core system. The four cores, A, B, C, and D, concurrently attempt to execute the same code. We use Core C in the figure to explain the mechanics of contesting.

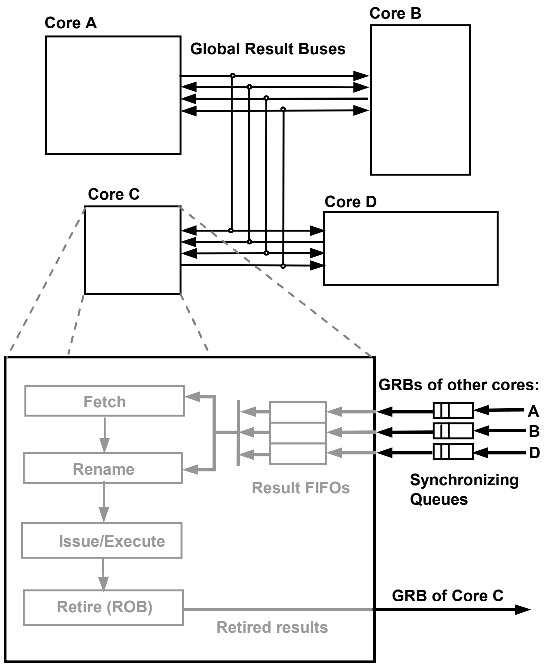

Figure 2. A generalized architectural contesting multi-core system.

4.1. Leveraging results from other cores

4.1.1. Global result buses. A core broadcasts the results of its retired instructions to the other cores via its own *global result bus* (GRB). For example, Core C has its own outgoing GRB. Its GRB has three sinks, at Cores A, B, and D.

Conversely, Core C receives results from three incoming GRBs, the GRBs of Cores A, B, and D. Since Core C may have a different clock frequency from the other cores, synchronizing queues, borrowed from recent GALS proposals [11], are used to interface Core C with its three incoming GRBs. Results from the three incoming GRBs are then transferred to three *result FIFOs* within Core C.

4.1.2. Pop counters and fetch counter. Core C maintains a “pop counter” for each of its three result FIFOs, as shown in Figure 3. The pop counter of a result FIFO is incremented each time Core C pops a result from it. For example, the pop counter for result FIFO A is 106, meaning that the results of 106 retired instructions from Core A have been popped. The implication is that the head-entry of result FIFO A contains (or will contain) the result of retired instruction #107 from Core A. Effectively, from the values of the pop counters A, B, and D, we can infer the logical positions in the dynamic instruction stream of the head-entries of the result FIFOs A, B, and D. Their logical positions are explicitly shown in Figure 3 by their horizontal placement along the retired dynamic instruction stream (which is shown at the top of the diagram). As shown, the head-entry of result FIFO A is currently at retired instruction #107 and the head-entries of result FIFOs B and D are both at retired instruction #102.

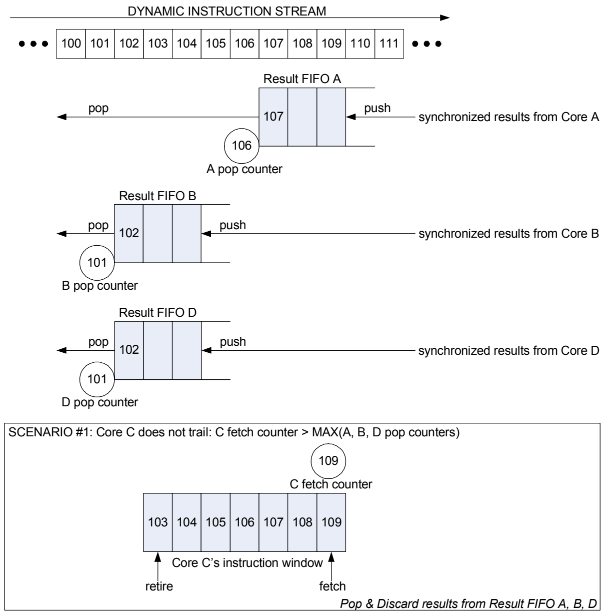
DYNAMIC INSTRUCTION STREAM
100 101 102 103 104 105 106 107 108 109 110 111
Result FIFO A
pop
push
107
synchronized results from Core A
106
A pop counter
Result FIFO B
102
synchronized results from Core B
101
B pop counter
Result FIFO D
102
synchronized results from Core D
101
D pop counter
SCENARIO #1: Core C does not trail: C fetch counter > MAX(A, B, D pop counters)
109
C fetch counter
103 104 105 106 107 108 109
Core C's instruction window
retire
fetch
Pop & Discard results from Result FIFO A, B, D

Figure 3. Example where Core C does not trail.

Core C also maintains a single “fetch counter” that indicates how many correct instructions (those that will ultimately be retired) it has fetched. (The fetch counter may be temporarily incorrect due to a mispredicted branch in Core C. This issue is handled at the end of this subsection.) Effectively, from the value of the fetch counter, we can infer where the most recently fetched instruction, at the tail-entry of Core C’s instruction window, is logically positioned within the dynamic instruction stream. As before, the logical position of Core C’s instruction window (all instructions that have been fetched but not yet retired) is explicitly shown in Figure 3 by its horizontal placement along the retired dynamic instruction stream. In the example Scenario #1, the fetch counter contains 109, therefore, the newest instruction at the tail-entry of Core C’s instruction window is to-be-retired instruction #109 (assuming the fetch unit is on the correct path).

By comparing its fetch counter to the maximum of its three pop counters, Core C can determine whether or not it is trailing the most advanced result FIFO. There are only two possible scenarios:

1. *Scenario #1*: In Figure 3, result FIFO A leads the other two result FIFOs because it has the highest pop counter, 106. The fetch counter, 109, is higher still. This means Core C is not trailing and cannot be accelerated by any of the other cores' results. When Core C fetches the next instruction, #110, none of the result FIFOs is advanced enough in the dynamic instruction stream to provide a result for it. If this next instruction is a branch, it must be predicted and executed. If it is a register-producing instruction, it must execute to produce its value. As long as the fetch counter is greater than the maximum pop counter, late results are popped and discarded from all result FIFOs as soon as these results are received from the GRBs.
2. *Scenario #2*: Core C's lead over the most advanced result FIFO may erode. This erosion reaches a turning point when the fetch counter *equals* the maximum pop counter, as shown in Figure 4 (the trailing result FIFOs B and D are not shown). At this turning point, Core C's fetch unit and the head-entry of the most advanced result FIFO are logically at the same instruction in the dynamic instruction stream. In Figure 4, the next instruction to be fetched by Core C is #127, which happens to be the instruction for which the head-entry of result FIFO A contains a result. This is no coincidence: it is because the A pop counter (number of instructions popped from result FIFO A) and the fetch counter (number of correct instructions fetched by Core C) match. A communication channel is established from result FIFO A to Core C's fetch unit. Now, instead of popping and discarding late results from result FIFO A as soon as they arrive, the FIFO is popped when Core C's fetch unit fetches the next instruction (causing both the A pop counter and the fetch counter to increment, therefore, they remain equal). The popped result is paired with the newly fetched instruction. If the result FIFO A is empty when the next instruction is fetched, however, it simply means that Core C is no longer trailing and the tables turn again to Scenario #1 above.

The fetch counter may be speculative due to branches. Core C's fetch counter is guaranteed to be correct when it is trailing (Scenario #2) because known branch outcomes are available from the result FIFO A (no mispredictions). On the other hand, it is not guaranteed to be correct when Core C is not trailing because the fetch unit must predict branches as usual. If a branch is mispredicted, the fetch counter is temporarily incorrect because it counts incorrect instructions that are not ultimately retired. This is dealt with simply by checkpointing the fetch counter at every branch. When a mispredicted branch executes, the fetch counter is restored to its correct value representing instructions up to and including the branch.

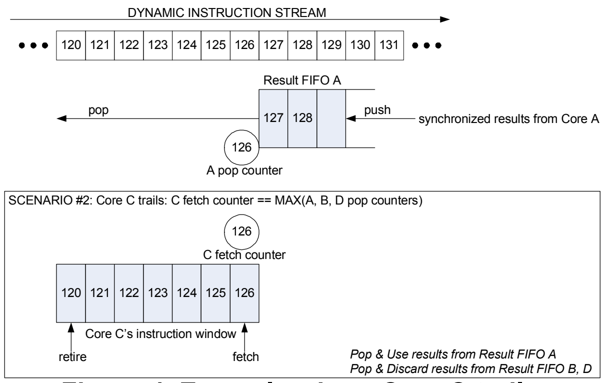
DYNAMIC INSTRUCTION STREAM
120 121 122 123 124 125 126 127 128 129 130 131
Result FIFO A
pop
127 128
push
synchronized results from Core A
126
A pop counter
SCENARIO #2: Core C trails: C fetch counter == MAX(A, B, D pop counters)
126
C fetch counter
120 121 122 123 124 125 126
Core C's instruction window
retire
fetch
Pop & Use results from Result FIFO A
Pop & Discard results from Result FIFO B, D

Figure 4. Example where Core C trails.

A subtle corner case arises when Core C is not trailing, its fetch unit mispredicts a branch, and then a retired instance of the branch is received in the most advanced result FIFO before the branch is resolved by Core C itself. Earlier, we explained that when Core C is not trailing (Scenario #1), all received results are immediately popped and discarded from the result FIFOs. To handle the corner case, however, received branches are not summarily discarded. Instead, after popping a branch and incrementing the pop counter, the value of the pop counter is compared against the checkpointed fetch counter of the oldest unresolved branch in Core C. If they match and the branch is found to be mispredicted (by checking its prediction against the popped branch outcome), then the mispredicted branch is resolved early. Note that the fetch counter will be restored to its checkpointed value which naturally matches the pop counter. This means Core C is now perceived to be trailing (fetch counter = maximum pop counter), and the table has turned from Scenario #1 to Scenario #2. This corner case is depicted in Figure 5.

4.1.3. Injecting results. As explained in the previous subsection, when Core C is trailing, it pairs popped results from result FIFO A with its fetched instructions. A result is used in lieu of executing the instruction. The instruction is completed early in the fetch stage, if it is a branch, or in the rename stage, if it is a register-producing instruction. Early completion in the fetch stage is implemented by overriding the branch prediction logic. Early completion in the rename stage is implemented by directly writing a value into the destination physical register. This requires stealing register file write ports from the execution core. The transfer of ownership of write ports, from the writeback stage to the rename stage, is gradual. Any already-issued instructions will be able to write their values in the writeback stage as promised by the scheduler. Over the span of several cycles, fewer write ports are allocated to the scheduler and more write ports are allocated to the rename stage. This is consistent with the fact that the issue queue is gradually emptied as no new instructions are dispatched into it. When it is completely drained, all write ports are allocated to the rename stage.

A more straightforward alternative to this port reallocation scheme is to continue dispatching instructions into the issue queue but to mark them as immediately ready, since they already have their destination values with them: they will issue expeditiously (free of all data dependences) and write their values in the writeback stage like usual.

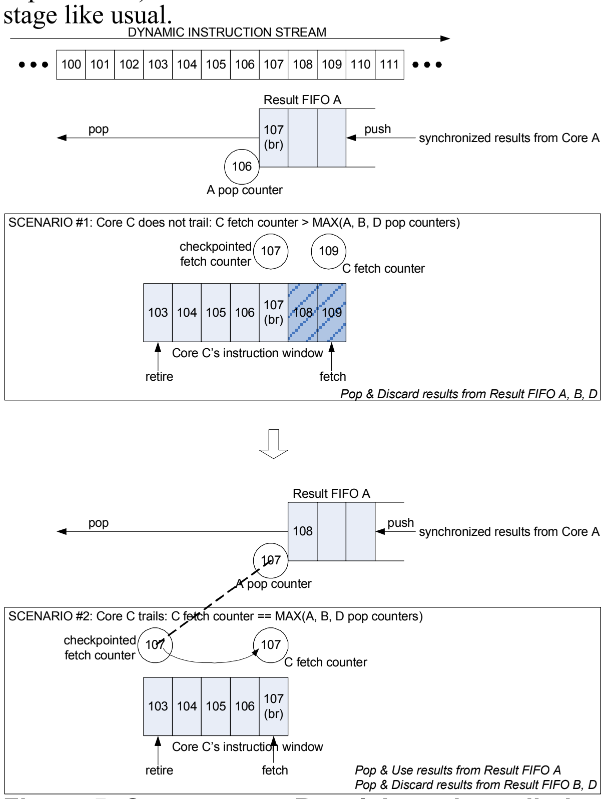
DYNAMIC INSTRUCTION STREAM
100 101 102 103 104 105 106 107 108 109 110 111
Result FIFO A
pop
push
synchronized results from Core A
107 (br)
106
A pop counter
SCENARIO #1: Core C does not trail: C fetch counter > MAX(A, B, D pop counters)
checkpointed fetch counter
107
109
C fetch counter
103 104 105 106 107 (br) 108 109
Core C's instruction window
retire
fetch
Pop & Discard results from Result FIFO A, B, D
108
107
SCENARIO #2: Core C trails: C fetch counter == MAX(A, B, D pop counters)
Pop & Use results from Result FIFO A
Pop & Discard results from Result FIFO B, D

Figure 5. Corner case: Resolving misprediction early causes transition from Scenario #1 to #2.

4.1.4. Lagging distance. The fetch counter and pop counters only need to be large enough to represent the maximum number of dynamic instructions that is allowed to separate the leading and lagging cores.

By leveraging the result FIFOs, execution in the lagging cores will never fall too far behind. Thus, when the code phase changes, all the cores will be contested fairly in the new phase without the need for actually detecting the change of phase, and the core that is best suited will automatically be able to *take lead*.

How far behind a lagging core is depends on the physical propagation delay between cores. When the characteristics of the code change, it is this *lagging distance* that a core needs to catch-up on before it can become the head of the pack and commence effective execution. Note that it is not necessary for all the lagging cores to receive the result of a retired instruction in the same cycle. This issue is of convenience, as different cores may be at differing distances from each other.

Although the frequencies and retirement widths of the cores may differ, the peak rate at which instruction results are retired by any core – in instructions-per-second (IPS) – must be sustainable by all other cores. That is, the peak retirement rate (in IPS) of any core must be less than or equal to the peak rate (in IPS) at which instruction results can be written to the register file and memory in any other core. Without this condition, a lagging core may unboundedly fall behind, resulting in excessive catch-up time and therefore defeating the purpose of architectural contesting. We refer to a lagging core that cannot keep-up with the leading core as a *saturated lagger*. This scenario can be dealt with simply by disabling contesting mode for the saturated lagger.

4.2. Handling stores

Stores are redundantly performed in the private cache levels of the cores. The private cache levels are configured to use the write-through policy to simplify contesting (this does not preclude using the write-back policy in non-contesting modes). To prevent lagging cores from incorrectly observing future stores of less-lagging cores, stores stop short of writing through to the shared cache level. Here, we employ a synchronizing store queue similar to SRT's store queue [18].

In SRT, the store queue waits for both instances of a store (the leading and trailing threads' instances), before performing a single merged instance to the L1 cache, and loads from the leading thread search the store queue in addition to the L1 cache. Similarly, the synchronizing store queue employed for contesting buffers stores and keeps track of which cores have privately performed each store. When the oldest store has been privately performed by all cores, a single merged instance is performed to the shared cache level.

4.3. Handling exceptions

A synchronous exception (e.g., error, TLB miss, system call) will be detected by all the contesting cores, although not at the same time. We could take the same approach as previous work [17][3] that designates one core to handle the exception (terminate threads in the non-designated cores, service the exception in the designated core, and refork threads in the non-designated cores including preloading TLB entries). We advocate a new approach: a redundant-thread-aware parallelized exception handler. An explicitly-parallel software handler can perform the necessary coordination to achieve correct results for all of the cores, avoiding the overhead of terminating and reforking threads. A core calls the exception handler when it reaches the exception. The handler increments a semaphore and checks its value to determine whether or not all contesting cores have reached the exception. If no, the handler on this core goes to sleep. If yes, the handler wakes up all the other sleeping handlers and they may coordinate handling the exception on all the cores.

For asynchronous exceptions caused by external interrupts, one of the cores is designated to listen for external interrupts. In this case, it is difficult to stop all redundant threads at the same point without resorting to an elaborate hardware handshaking protocol, and so the first approach is used.

5. Measuring up to the very best

5.1. Methodology

The *sim-mase* simulator from the Simplescalar V4.0 toolset [24] has been modified to model the contesting implementation described above. The simulator was modified to enable time-synchronous execution of multiple simulator instances that model different proportional clock periods. In order to model time-synchronous execution, the simulator instances perform handshaking in a round-robin arrangement. Receiving a handshaking signal signifies the passing of a base time-unit (specifically 0.01ns). Each simulator instance executes an iteration of its top-level simulation loop upon the passing of as many time-units as there are in the clock period it is modeling. For example, a simulator instance modeling a 3GHz core will execute one iteration of its top-level simulation loop every 33 time-units.

The benchmarks used throughout are the 100-million instruction SimPoints [16] of the SPEC2000 integer benchmarks (except for *eon*, which we were unable to compile with the Simplescalar compiler).

In this study, we consider the pool of prospective cores in the heterogeneous CMP to consist of cores that are customized for individual SPEC2000 integer benchmarks in 70nm technology. We used the XpScalar design-space exploration framework [19], which employs a simulated-annealing exploration process, to arrive at the benchmark-customized cores. XpScalar varies multiple design parameters, including superscalar width, register-file/ROB size, issue-queue size, load-store queue size, L1 and L2 cache configurations, and clock frequency. The depth of pipelining of various architectural units/stages is consistent with the processor's frequency and the complexity of these units/stages. The customized core of each benchmark and its performance with respect to all benchmarks is reproduced in Appendix A.

5.2. Results

We limit our evaluation to 2-way architectural contesting (contesting between two cores). An issue of importance is the considered core-to-core latency, or the time it takes for an instruction result to travel from one core to another. In this part of the study, a one nanosecond (three cycles of a 3 Ghz processor) core-to-core latency is considered. The effect of scaling this latency is measured in Subsection 5.2.2.

Figure 6 shows the performance (instructions per time, IPT) of contesting, for each benchmark. For each benchmark, the two cores that are contested (from among all benchmark-customized cores) are those two which give the highest performance when contested; the pair of contesting cores used by a given benchmark is labeled above its bar in the graph. For comparison, the IPT of each benchmark on its own customized core is also shown. Contesting yields an average speedup of 15% over a benchmark's own customized core. The largest speedup is attained for the benchmark *gcc* at 25%. Four out of the eleven studied benchmarks attain more than 18% speedup.

The averaged results in Figure 1 (of Section 2) show that achieving speedups in the range of 15% over a benchmark's own customized configuration requires the ability to switch execution between configurations at a rate of around 100 instructions. This number of instructions is proportional to the number of instructions in the pipeline of an average configuration at any instance. However, using previously proposed techniques to detect changes in workload behavior, determine a suitable configuration, and transfer execution to it, can most probably be achieved at a rate of a few thousand instructions at the very best – which drastically diminishes the benefit of being able to adjust the microarchitecture.

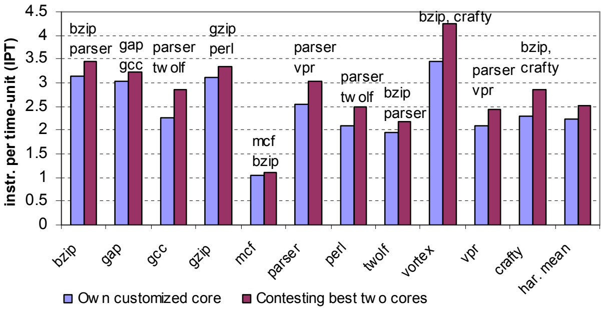

Figure 6. IPT of each benchmark for 1) execution on its own customized core and 2) contesting between two cores that maximize contested-execution performance (the two contested cores are shown above the bars).

5.2.1. The source of performance enhancement. A question that may arise is how integral heterogeneity in the microarchitecture of the cores is to this performance enhancement, and whether the origin is heterogeneity in the cache configurations. Differentiating the heterogeneity in the caches from that in the microachitecture of the cores can provide insight into the origin of the performance enhancement. However, note that the best cache configuration for a workload is not independent of other microarchitectural design factors.

In order to address this question, each benchmark is executed with contesting between two cores that differ only in their L2 caches. One of the cores is one of the best two cores for contesting. The other is the same core, but with its L2 cache (configuration and access latency) replaced with that of the other best core for contesting. For example, *bzip* was originally contested between the customized cores of *bzip* and *parser* (contesting these two cores yielded the highest performance). For the modified experiment, *bzip* is contested between two *bzip* cores, except that one of these otherwise identical cores has the L2 cache of the *parser* core. This experiment is repeated with two *parser* cores, one of which has the L2 cache of *bzip*. The higher performing trial of these two trials is used.

Figure 7 shows the speedup of contesting. The total height of each bar represents the speedup of contesting in the original experiment (heterogeneity in both the microarchitecture and L2 cache). The bottom fraction of each bar represents the speedup of contesting in the modified experiment, isolating the performance enhancement due to heterogeneous L2 caches. These results show that, for most of the benchmarks (other than *gcc* and *parser*), only a minor portion of the performance enhancement can be attributed to only heterogeneity in the L2 cache. All the same, it is contesting that enables this heterogeneity to be exploited at a fine granularity.

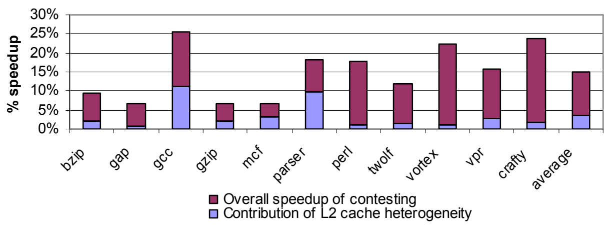

Figure 7. Isolating the contribution of L2 cache heterogeneity to the performance enhancement of contesting.

5.2.2. The effect of core-to-core latency. Figure 8 shows the effect that the core-to-core latency has on the average speedup of contesting between the best two cores for each benchmark over the performance of the benchmark on its own customized core. These results show a decrease in the performance enhancement of contesting as this latency increases. At a latency of 100ns the average performance benefit reduces to 6%. These results show the importance of the propagation delay of the GRB.

Moreover, these results also show that different workloads are affected differently by the core-to-core latency. For instance, while the speedup of a benchmark such as *bzip* degrades by less than 1% when the latency increases from 1ns to 2ns, that of *gzip* decreases by more than 35% percent for the same increase in latency.

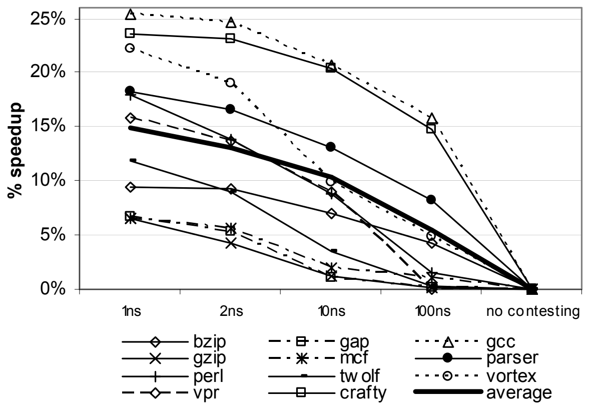

Figure 8. Speedup of contesting for different core-to-core latencies over customized cores.

6. Evaluation with limited core types

Section 5 evaluates the performance enhancement attainable from contesting between the best two core types for fine-grain switching, for each benchmark. Since the best pair of contesting cores differs from one benchmark to the next, the previous evaluation implies that the customized core types of all benchmarks are available in the heterogeneous CMP. However, there may be fewer core types in a realistic heterogeneous CMP. In Section 6.1, we first address general heterogeneous CMP design and what influences the best set of core types to employ when the number of core types is limited. In Section 6.2, we apply these principles to design heterogeneous CMPs with only two core types. Finally, in Section 6.3, we evaluate the performance enhancement of contesting between the cores of the systematically-designed dual-core-type heterogeneous CMPs from Section 6.2.

6.1. The design goal

The best combination of microarchitectural configurations to employ in a heterogeneous system depends on the design goal.

If the design goal is to minimize the total execution time of a set of benchmarks when submitted to the system one-by-one – as is customary in single-core microarchitecture evaluation – a representative figure of merit is the harmonic-mean of the performance (instructions per time unit, IPT) of all benchmarks when each is executed on the most suitable core available. This figure of merit is improved if the benchmarks are weighted by the frequency with which they occur in the system. Without these weights, benchmarks that run infrequently but have long run-times may have disproportionate influence on the perceived-best core types.

Benchmark weights may not be available, however. In this situation, it may be desirable to use the average (arithmetic-mean) of IPTs as the figure of merit. Average IPT focuses on raw throughput instead of total time, which may lead to more performance-robust core types in the face of uncertain benchmark frequencies.

Neither of these metrics (harmonic-mean IPT and average IPT) accounts for core-contention between jobs and may thus bring about imbalance in the number of benchmarks better suited for the different core types. For example, if the design goal is simply to maximize either the harmonic-mean IPT or average IPT for ten benchmarks on two core types, the ultimate CMP design may be guided toward one core type that is favored by nine benchmarks and one core type that is favored by one benchmark. While this CMP design is optimum when there is no contention between jobs, it is not necessarily optimum when there is contention.

If contention between jobs is a concern, the best combination of core types to employ in the system is influenced by 1) the rate and distribution of job submissions and 2) how jobs are scheduled. Below, we develop a figure of merit that accounts for contention, which is based on two simplifying assumptions regarding job distribution and scheduling, respectively. First, we consider a uniform distribution of job submissions. That is, jobs have an equal probability of belonging to one of the considered workload types (*i.e.*, one of the benchmarks). Unevenness in the distribution can be modeled by assigning importance weights that are proportional to the probability of a workload type being submitted to the system. Burstiness in the arrival of jobs of the same workload type decreases the value of heterogeneity. Second, we consider a scheduling policy that directs a job to the core type for which it is best suited, even if all cores of that type are currently busy and the job must be queued, instead of directing it to the best *available* core. This policy is reasonable if all cores are heavily loaded.

In this setting, the arrival rate of jobs at the job-queue of a specific core will be proportional to the number of job types that prefer that core. Therefore, according to Little's law, the average number of jobs in a job-queue will also be proportional to the number of job types that prefer the corresponding core. Therefore, a representative figure of merit can be attained by dividing the performance (IPT) of each benchmark when executed on the most suitable core type in the CMP design by the number of benchmarks that share the core type, and then taking the harmonic mean. We refer to this figure of merit as the *contention-weighted harmonic-mean IPT.*

6.2. Exploring combinations of cores

The best combination of core types to employ in a heterogeneous CMP is determined by searching all the possible combinations of core types for one that maximizes the considered figure of merit. Once again we limit the pool of prospective core types to the customized cores for individual SPEC2000 integer benchmarks. In addition, the number of core types in the heterogeneous CMP is limited to only two. This does not limit the total number of cores, that is, conceivably there could be multiple instances of each core type.

Since we separately consider three different figures of merit – average IPT (*avg*), harmonic-mean IPT (*har*), and contention-weighted harmonic-mean IPT (*cw-har*), as discussed in the previous section – we arrive at three different heterogeneous CMP designs, HET-A, HET-B, and HET-C, respectively. These designs are displayed in the first three rows of Table 1. The table shows which two core types comprise each of HET-A, HET-B, and HET-C. A core type is identified by the name of the benchmark for which it is customized, *e.g.*, the customized core type for the *gcc* benchmark is named "*gcc*". HET-A is comprised of the *parser* and *twolf* core types, HET-B is comprised of the *gcc* and *mcf* core types, and HET-C is comprised of the *bzip* and *crafty* core types.

Since this paper is ultimately concerned with single-thread performance (the benefit of contesting), regardless of the figure of merit used to arrive at a CMP design, the last column of Table 1 shows the harmonic-mean of the IPTs of all benchmarks when each is run on the most suitable core of a given design. Since HET-B was designed using this figure of merit to begin with, as expected, it has the highest harmonic-mean IPT among HET-A, HET-B, and HET-C.

Two other designs are included in the last two rows of Table 1. The first of these, HOM, is a homogeneous CMP design with only one core type, namely, the core type that gives the best performance on average for all benchmarks. Of all the customized core types, the *gcc* core type is the best overall (it maximizes both average IPT and harmonic-mean IPT). The last design, HET-ALL, is a heterogeneous CMP comprised of all the customized core types, so that each benchmark runs on its customized core.

While the *gcc* core is the best individual performer among the benchmark-customized cores, it is possible for there to be an even better core that is explicitly customized for the benchmark suite as a whole. We conducted an exploration of the design space for the aggregate performance across all benchmarks using the XpScalar exploration process [19]. The customized core that was attained provided negligible overall performance enhancement over that of the *gcc* core.

Table 1. Five CMP designs and their performance.

CMP design	Designed based on which figure of merit?	Constituent core types	Harmonic-mean of IPT
HET-A	avg	parser & twolf cores	1.76
HET-B	har	gcc & mcf cores	1.88
HET-C	cw-har	bzip & crafty cores	1.87
HOM	avg or har	gcc core	1.57
HET-ALL	Not applicable	customized cores of all benchmarks	2.1

The performance results in the last column of Table 1 show that, when exploited at the application level, unconstrained heterogeneity can provide up to a 34% increase in harmonic-mean IPT (HET-ALL compared to HOM). With only two core types, HET-C provides a 19% increase in harmonic-mean IPT (HET-C compared to HOM). In other experiments, not shown here, we determined that the overall performance of a heterogeneous CMP with four core types is within 2% of the performance of HET-ALL.

Figure 9 displays the IPTs of individual benchmarks on the five CMP designs of Table 1. For a given CMP design, a benchmark is run on the most suitable core type available in that design. These results show how the choice of available core types impacts individual benchmark performance.

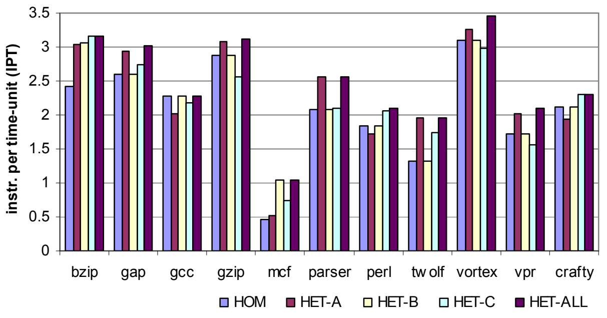

Figure 9. IPT for each benchmark when executed on the most suitable core type available in the CMP, for five CMP designs.

In this section, we evaluate the single-thread performance enhancement of contesting on top of the HET-A, HET-B, and HET-C CMP designs from the previous section. The chief purpose of this exercise is to study contesting in the context of a heterogeneous CMP that was systematically designed to exploit application-level heterogeneity with a limited number of core types, and not explicitly for the purpose of fine-grain switching within an application. This is the expected setting in which contesting might be deployed.

Figure 10 shows the performance (instructions per time unit, IPT) of each benchmark, for three scenarios: 1) execution on HOM ("HOM"), 2) execution on the most suitable core type of HET-A ("HET-A, no contesting"), and 3) contested execution between the two core types of HET-A ("HET-A, contesting").

Contesting on HET-A yields an average speedup of 16% and a maximum speedup of 41% (for *gcc*) compared to not contesting. An interesting result is that, while the benchmarks *gcc, perl,* and *crafty* observe lower performance on the better of the two cores of

HET-A compared to the overall best core provided by HOM, their contested execution with them more than compensates for the deficit.

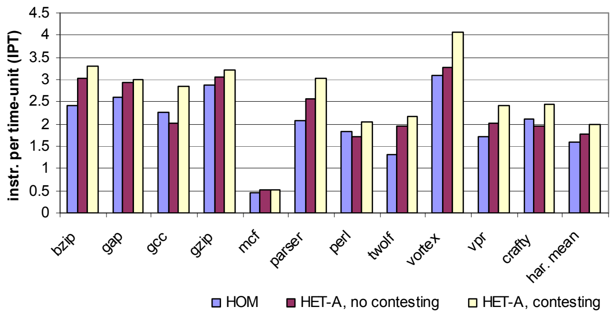

Figure 10. Performance of each benchmark on HOM, HET-A without contesting, and HET-A with contesting.

Figure 11 shows the IPT of each benchmark under the same three scenarios, except that the HET-B CMP design is used instead of the HET-A CMP design. From Table 1, HET-B is comprised of the *gcc* and *mcf* core types. Due to the long clock period of the *mcf* core, it tends to become a saturated lagger in the contested execution of half of the benchmarks, resulting in little benefit for these. Nevertheless, contesting on HET-B yields an average speedup of 13% and a maximum speedup of 39% (for *twolf*) compared to not contesting.

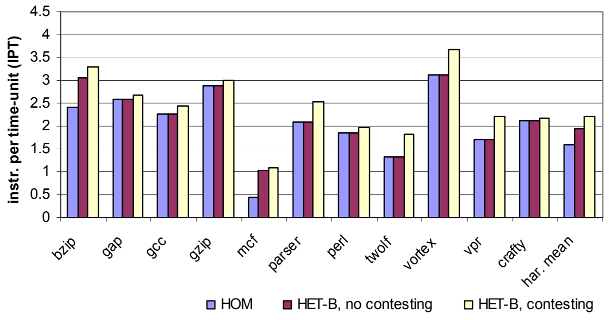

Figure 11. Performance of each benchmark on HOM, HET-B without contesting, and HET-B with contesting.

The highest overall speedup is attained for contesting between the two core types of the HET-C CMP design (featuring the *bzip* and *crafty* core types). Figure 12 shows the results for HET-C. Contesting on HET-C yields an average speedup of 22% and a maximum speedup of 50% (for *vpr*) compared to not contesting. For the benchmarks *gzip*, *vortex,* and *vpr*, contesting prevents performance from dipping below that of the overall best core provided by HOM, and even boosts performance significantly above it. These speedups are attained through active participation of both cores in the effective computation.

With contesting, HET-C achieves an average speedup of 34% over HOM. In contrast, without contesting, HET-C achieves an average speedup of 11% over HOM. Therefore, contesting has roughly tripled the single-thread performance advantage of heterogeneity in this system. HET-C was primarily designed with heavy-loading of the system in mind. Thus, contesting can be viewed as a technique that provides robustness to heterogeneity – allowing a system to be primarily designed for heavy loading, yet not compromise single-thread performance when the system is lightly loaded (this issue is further addressed in the next section).

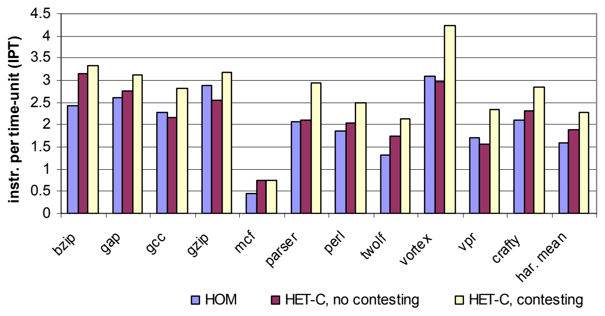

Figure 12. Performance of each benchmark on HOM, HET-C without contesting, and HET-C with contesting.

7. Discussion

7.1. Combination of core types for contesting

Section 6 considered clear-cut figures of merit for the design of a heterogeneous CMP with a constrained number of core types. However, the truly best design goal may be more involved. For instance, an issue that may be of concern is certain benchmarks performing worse than they would have in a homogeneous system, despite heterogeneity improving single-thread performance on the whole. This can be reflected in the figure of merit by penalizing it when that is the case. Another issue may be the robustness of the design to perform well both when the system is loaded (throughput) and unloaded (single-thread performance). Combining simpler figures of merit can reflect such hybrid design goals.

The design of a constrained heterogeneous CMP involves compromises, by virtue of limiting the number of core types and optimizing for one figure of merit or the other. The results in Section 6 showed that contesting provides a degree of performance robustness that compensates for certain side-effects of these compromises. Here we discuss two examples:

- Section 6 showed that a constrained heterogeneous CMP design improves single-thread performance for the benchmark suite as a whole compared to a homogeneous CMP design, but that specific benchmarks may nonetheless perform worse. The results showed that contested-execution of these benchmarks makes up for this performance deficit.
- The *cw-har* figure of merit, like the *avg* and *har* figures of merit, tries to steer the design of a constrained heterogeneous CMP towards higher single-thread performance, but it balances this goal with the need to distribute job types (benchmarks) evenly among the core types in anticipation of a heavily loaded system. Thus, while this figure of merit does exploit application-level heterogeneity for higher single-thread performance, it may not do so to the same extent as the other narrower

figures of merit. The results in Section 6 showed that contesting boosts single-thread performance of the heterogeneous CMP that was designed taking into account heavy loading (HET-C), compensating for any deficit with respect to the other designs (HET-A, HET-B). Thus, for systems that observe periods of heavy loading, HET-C with contesting as an available (but optional) mode of execution is a better design point than the other considered CMPs, because it is more robust, handling both periods of heavy and light loading well.

7.2. Customizing cores for contesting

Cores that are customized for application-level performance are not necessarily suitable for fine-grain regions of code. Architectural contesting will provide its greatest advantage when the cores are customized not for applications, but for fine-grain regions of code. In other words, the true single-thread performance potential of contesting can only be achieved when the cores are customized with contesting in mind. On the other hand, cores that are customized for fine-grain regions of code-will not necessarily be the best for application-level performance. Thus, a heterogeneous CMP consisting of such cores may hamper the benefit of "lower hanging fruit": throughput improvement.

Determining the best core designs for contesting is much more complex than determining the best core design for an application, as the different core designs need to be explored together in contesting pairs (or contesting trios, etc.) – resulting in an explosion in an already vast overall design space. In addition, conducting design exploration across this design-space is a slower process, as measuring the performance of design points involves simulation of contested execution (which is more time-consuming than simulation of conventional execution).

7.3. Contesting vs. more core types

We introduce a fourth constrained heterogeneous CMP design, HET-D, which is comprised of three core types instead of just two. The *har* figure of merit was used to select the three core types (maximizes harmonic-mean IPT of the benchmarks). HET-D is comprised of the customized cores of *twolf*, *crafty*, and *mcf*.

For each benchmark, Figure 13 compares the performance of contesting between the two core types of HET-C ("HET-C, contesting") to the performance of executing the benchmark on the most suitable core type of HET-D ("HET-D, no contesting"). In addition, the performance of executing the benchmark on its own customized core ("HET-ALL, no contesting") is shown for comparison. These results show that, on average, contesting between only two core types can yield as much single-thread performance enhancement as a heterogeneous system with all eleven core types (and typically more for a majority of benchmarks), and slightly more than a system with three core types.

Therefore, in terms of maximizing single-thread performance, contesting may be a more cost-effective approach than increasing the number of core types in the system.

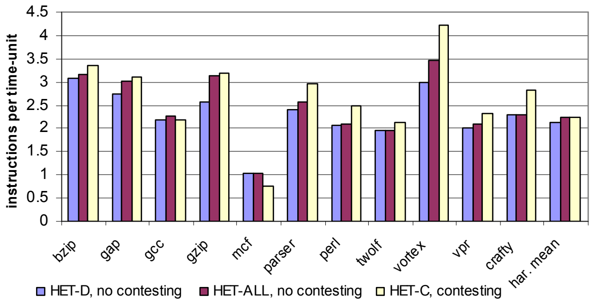

Figure 13. Comparison of contesting between two core types vs. exploiting more core types.

8. Summary

This paper showed that workload behavior tends to vary considerably at fine granularities. Architectural contesting leverages the differently-designed cores in a heterogeneous multi-core to automatically and fluidly transfer effective execution to the most suitable core, exploiting workload variations that are too fine-grain to be handled by previous adaptational and migrational approaches.

In addition to evaluating two-way contesting in a relatively unconstrained heterogeneous multi-core (as many core types as benchmarks), the paper explored the interplay between contesting and the number of core types in more constrained heterogeneous multi-core designs. This exposed the broader issue of constrained heterogeneous multi-core design and how it influences, and may be influenced by, contesting.

9. Acknowledgments

We thank Dean Tullsen and the anonymous reviewers for their valuable feedback on improving this paper. This research was supported by NSF grants No. CCF-0429843 and CCF-0811707, and generous funding 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.

10. References

[1] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, K. I. Farkas, "Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance," *31st Int'l Symposium on Computer Architecture*, June 2004.
[2] A. Dhodapkar, J. E. Smith, "Managing Multi-Configuration Hardware via Dynamic Working Set Analysis," *29th Int'l Symposium on Computer Architecture*, May 2002.
[3] K. Sundaramoorthy, Z. Purser, E. Rotenberg, "Slipstream Processors: Improving both Performance and Fault Tolerance," *9th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.
[4] D. Albonesi. "Dynamic IPC/clock rate optimization," *25th Int'l Symposium on Computer Architecture*, July 1998.
[5] D. Ponomarev, G. Kucuk, O. Ergin, K. Ghose, P. Kogge, "Energy-Efficient Issue Queue Design," *IEEE Transactions on Very Large Scale Integration Systems*, 11(5), Oct. 2003.

[6] D. Folegnani, A. Gonzalez, “Energy-Effective Issue Logic,” *28th Int’l Symposium on Computer Architecture,* July 2001.
[7] S.-H. Yang, M. D. Powell, B. Falsafi, K. Roy, T. Vijaykumar, “An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-submicron High-performance Caches,” *7th Int’l Symposium on High-Performance Computer Architecture*, Jan. 2001.
[8] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, S. Dwarkadas, “Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures,” *33rd Int’l Symposium on Microarchitecture*, Dec. 2000.
[9] D. H. Albonesi et al. “Dynamically Tuning Processor Resources with Adaptive Processing,” *IEEE Computer*, 36(12), Dec. 2003.
[10] M. Huang, J. Renau, J. Torrellas, “Positional Adaptation of Processors: Application to Energy Reduction,” *30th Int’l Symposium on Computer Architecture*, June 2003.
[11] S. G. Dropsho, G. Semeraro, D. H. Albonesi, G. Magklis, M. L. Scott, “Dynamically Trading Frequency for Complexity in a GALS Microprocessor,” *37th Int’l Symposium on Microarchitecture*, Dec. 2004.
[12] D. Burger, S. Kaxiras, J. R. Goodman, “DataScalar Architectures,” *24th Int’l Symposium on Computer Architecture*, June 1997.
[13] L. Chen, D.H. Albonesi, S. Dropsho, “Dynamically Matching ILP Characteristics via a Heterogeneous Clustered Microarchitecture,” *IBM Watson Conf. on the Interaction Between Architecture, Circuits, and Compilers*, Oct. 2004.
[14] R. Kumar, D. M. Tullsen, N. P. Jouppi, “Core Architecture Optimization for Heterogeneous Chip Multiprocessors,” *Int’l Conferenceon Parallel Architectures and Compilation Techniques*, Sep. 2006.
[15] B. Greskamp, J. Torrellas, “Paceline: Safely Overclocking to Improve CMP Performance under Parameter Variation,” *6th Int’l Conference on Parallel Architectures and Compilation Techniques*, Sep. 2007.
[16] T. Sherwood, E. Perelman, G. Hamerly, B. Calder, “Automatically Characterizing Large Scale Program Behavior,” *10th Int’l Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.
[17] E. Rotenberg, “AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors,” *29th Int’l Symposium on Fault-Tolerant Computing*, June 1999.
[18] S. K. Reinhardt, S. S. Mukherjee, “Transient Fault Detection via Simultaneous Multithreading,” *27th Int’l Symposium on Computer Architecture*, June 2000.
[19] H. H. Najaf-abadi, E Rotenberg, “Configurational Workload Characterization,” *Int’l Symposium on Performance Analysis of Systems and Software*, April 2008.
[20] H. H. Najaf-abadi, E. Rotenberg, “Architectural Contesting: Exposing and Exploiting Temperamental Behavior,” *1st Reconfigurable and Adaptive Architecture Workshop*, Dec. 2006.
[21] T. Pering, R. Broderson, “Energy Efficient Voltage Scaling for Real-Time Operating Systems,” *4th Real-Time Technology and Applications Symposium*, June 1998.
[22] H. Zhou, “Dual-Core Execution: Building a Highly Scalable Single-Thread Instruction Window,” *14th Int’l Conference on Parallel Architectures and Compilation Techniques*, Sep. 2005.
[23] S. Hauck, A. DeHon, “Reconfigurable Computing: The Theory and Practice of FPGA-Based Computing,” *Morgan Kaufman*, 2008.
[24] E. Larson, S. Chatterjee, T. Austin, “The MASE Microarchitecture Simulation Environment,” *Int’l Symposium on Performance Analysis of Systems and Software*, June 2001.

Appendix A:
The microarchitecture configurations of core types customized for individual SPEC2000 integer benchmarks, and the performance of each benchmark when executed on each core type [19]. Each column represents the customized configuration of a benchmark.

	bzip	crafty	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
bzip	3.15	2.02	1.73	2.41	2.11	2.56	2.09	2.03	3.05	2.24	2.95
crafty	0.78	2.31	1.15	2.11	1.91	0.48	1.97	2.06	1.29	2.12	1.30
gap	1.39	2.75	3.02	2.60	2.92	0.89	2.89	2.79	2.00	2.47	2.05
gcc	1.17	2.17	1.42	2.27	2.03	0.75	2.02	1.63	1.79	2.06	1.80
gzip	1.78	2.56	2.02	2.88	3.13	1.28	3.01	2.14	2.39	2.57	2.37
mcf	0.74	0.40	0.30	0.45	0.29	0.93	0.32	0.41	0.52	0.42	0.52
parser	1.86	2.11	2.19	2.08	2.47	1.32	2.62	1.86	2.39	2.15	2.30
perl	0.85	2.02	0.90	1.81	1.67	0.54	1.65	2.07	1.32	1.81	1.30
twolf	1.65	0.98	0.81	1.26	0.88	1.18	1.10	0.91	1.83	1.16	1.77
vortex	1.68	2.98	2.55	3.09	2.91	1.07	3.41	2.78	2.61	3.43	2.54
vpr	1.56	1.33	1.13	1.72	1.09	1.05	1.36	1.29	2.00	1.51	2.09
No. of cycles for memory access	112	321	173	186	198	120	198	321	172	213	172
No. of pipeline stage of the front-end	4	12	6	7	7	4	7	12	6	8	6
Dispatch, issue, and commit width	5	8	4	4	4	3	4	5	5	7	5
ROB size	512	64	128	256	64	1024	512	256	512	512	256
Issue queue size	64	32	32	32	32	64	32	32	64	32	64
Min. lat. for awakening of dep. Instr.	0	3	1	1	1	0	1	3	1	2	1
Pipeline depth of Scheduler/Reg-file	1	3	1	2	1	1	2	4	2	4	2
Clock period	0.49	0.19	0.33	0.31	0.29	0.45	0.29	0.19	0.33	0.27	0.3
L1D associativity	2	1	1	1	1	2	1	1	8	4	2
L1D block-size	32	8	8	8	128	128	64	8	64	32	32
L1D no. of sets	1k	16k	2k	32k	256	1k	2k	2k	128	1k	128
L1D access latency	2	5	2	4	3	5	3	3	3	5	2
L2D associativity	4	16	4	8	1	4	8	16	4	16	8
L2D block-size	64	64	256	64	128	128	512	64	128	128	128
L2d no. of sets	8k	128	128	1k	4k	8k	32	128	2k	128	1k
L2D access latency	15	7	4	6	5	27	12	7	12	6	12
LS-queue size	128	64	256	256	128	64	256	128	256	256	64