Architectural *Contesting*: Exposing and Exploiting Temperamental Behavior

Hashem H. Najaf-abadi Eric Rotenberg

Electrical and Computer Engineering Department
North Carolina State University
{hhashem, ericro} @ ece.ncsu.edu

Abstract

Previous studies have proposed techniques to dynamically change the architecture of a processor to better suit the characteristics of the workload at hand. However, all such approaches are prone to a fundamental trade-off between the architectural diversity they can provide and the latency of architectural change, their fixed-configuration performance and the complexity of finding the best architectural configuration for the workload at hand.

In this study we argue that the full potential of dynamic architectural customization can only be achieved by diminishing the effect of the degree of available architectural diversity on the aforementioned performance factors.

The performance of a statically designed processing core in a heterogeneous multi-core system is independent of the architectural diversity available. In addition, it is apparent that concurrent execution of code on differently architected cores automatically reveals which architecture is more suitable for the characteristics of a particular workload.

We therefore propose architectural contesting; the redundant execution of code on a number of differently architected processors (each customized for a different set of workload characteristics) in a leader follower arrangement, such that the leader and follower cores continuously shift roles as one core or the other becomes more favorable for new code phases. In this manner effective execution is naturally transferred from one static architecture to the other with little latency.

In this study, we show that the contesting of only processor width can yield an average speedup of 7.5% and up to 12.5% in integer SPEC benchmarks.

1. Introduction

The effectiveness of a major portion of the architectural design aspects of the superscalar processor are highly dependent on the characteristics of the workload being executed. This is while workload characteristics may vary considerably between applications and workload phases. Therefore, in an attempt to enhance execution performance, previous studies have proposed approaches to enable *architectural changeability*. However, there is a fundamental trade-off between the architectural diversity that can be provided through changeability and three important performance factors; 1) the latency for which it takes the architecture to change, 2) the performance of each fixed-configuration, and 3) the complexity of finding the most suitable configuration.

On the other hand, greater architectural diversity increases system-perceived workload *temperamentality*; the rate at which such a degree of change occurs in the workload characteristics that *significantly* better local performance is attainable through the continuation of execution being transferred to a different architecture. Previously proposed approaches either fall short in providing much architectural diversity (thus limiting the perceived temperamentality), or are prone to large reconfiguration latencies and suboptimal fixed-configuration performance (rendering them incapable of exploiting potentially exposed temperamentality).

The purpose of this study is to find an approach that can provide broad architectural diversity, yet be capable of transferring execution of code to the most suitable configuration with minimal latency and not be prone to sub-optimal fix-configuration performance. In other words, we seek an approach that can expose temperamentality while maintaining the ability to exploit it.

As a solution, we propose *architectural contesting*. In this approach, instead of changing a single processor, the same program is simultaneously executed on multiple cores, each statically designed for optimum performance under certain workload characteristics. In order to prevent any configuration from falling behind, execution proceeds in a leader-follower manner, with the leading core in each region of code expediting the completion of instructions in lagging cores by forwarding retired instruction results to them. When the workload characteristics change, the configuration that is more suitable for the new characteristics will naturally take lead of other configurations. Thus, the detection of which configuration is more suitable for the code at hand is implemented implicitly with no adaptation latency. There is only a slight *catch-up* time for the new leading core in a phase to surpass the

leading core of the previous phase. Moreover, each core is designed statically and therefore the system is prone to no fixed-configuration performance inefficiency.

From an energy-consumption standpoint, this approach, in its basic form, is far from optimal. However, one must consider that when a core is lagging it will be receiving a steady stream of instruction results that it can directly retire without the power-hungry operations of instruction wakeup, execution and bypassing. In addition, total deactivation of processing architectures that are grossly unsuitable for a region of code at a coarse granularity (e.g. the application level) may be a viable solution to improving power efficiency.

We find that through the contesting of a limited form of architectural diversification, such as processor width, only modest speedup is attainable. However, the ability to switch between statically customized processors brings about a paradigm which motivates the development of broader architectural diversity; workload-customized architectures that perform well under certain workload characteristics at the cost of poor overall performance (innovation that are of no value in a static general purpose environment).

In the next subsection we present an outline of related work. Section 2 discusses different aspects of the implementation of architectural contesting. Section 3 provides empirical results prior to conclusion.

1.1. Background and Related Work

Many approaches have been proposed to enhance the IPC of single-threaded execution through speculative expedition of the retrieval of independent work that would otherwise be delayed due to false control-flow dependencies [24, 25, 9]. In contrast, in the approach we focus on in this study resources are invested on enabling more efficient execution of instructions as they become available in traditional von-neumann order.

Among such prior proposals, the *slipstream* architecture [9] is related to the approach proposed here in that two different simultaneous execution streams of the same code interact to improve overall performance. In the case of slipstream however, the interaction enhances performance by means of expedition of *delinquent* instructions.

A plethora of prior work has studied different approaches to design processors that can change aspects of their architecture to better suit the workload at hand. Many studies have focused on *temporal* and *positional* learning techniques for adaptation of the processor configuration for power efficiency with minimal performance degradation. For instance, Ponomarev et al [11] and Folengnani and Gonzalez [12] propose approaches to learning the optimum issue queue size, and Yang et al [13] propose cache miss-rate as a metric for determining when to downsize or upsize I-cache. A major feature of all such approaches is that they are based on the common knowledge that downsizing architectural units (disabling hardware) inevitably results in a reduction in energy consumption. This is while the overall effect on performance is much more difficult to predictable.

In Complexity Adaptive Processing (CAPs) [10], a single processor is architected such that the tradeoff between IPC and clock-rate can dynamically be altered. An essential component of the CAP architecture is the "configuration control" unit. This unit selects the optimal configuration for the code at hand through heuristic learning techniques or profiling information. Dhodapkar and Smith [13] and Balasubramonian and Albonesi [14] propose general *tuning* processes for identifying the best-suited configuration for a phase of code from a performance standpoint. However, fine-grain phase resolution requires low-latency adaptation of the architecture to the workload. *Temporal* approaches, such as the *Rochester algorithm* [16] or signature based approaches [5], require lengthy tuning processes. *Positional* approaches [17] enable faster adaptation, and have been found to be more effective than temporal approaches, but are generally table based (e.g. the signature table in [5]). This itself limits the phase granularity that can be effectively implemented. Moreover, all such approaches are greatly limited in the architectural diversity they can provide due to major portions of the architecture being implemented statically.

In addition, the architectural inflexibility of such designs is itself a source of imbalance and sub-optimal fixed-configuration performance. For instance, it is generally infeasible to maintain a balanced pipeline as one architectural unit is upsized and the clock frequency scaled proportionally. This imbalance is inevitable due to the fact that all architectural units function at the same clock frequency (in conventional synchronous design). Thus, in order to maintain correct functionality, the clock period must be scaled to meet the latency of the slowest pipeline stage, resulting in downsized units to observe slack.

The globally synchronous-clock design is also the cause of unpredictability in the overall performance effect of scaling individual units. Reconfiguration of the memory hierarchy [14] is an exception in this respect, as memory access latency is decoupled from the functionality of the processing core.

Dropsho et al [20] propose the separation of architectural units in what is referred to as a Globally-Asynchronous Locally-Synchronous (GALS) design, in order to allow for each unit to be scaled independently, with more predictability in the effect on overall performance. This relieves the system of the need for an

exhaustive tuning process that *tries-out* different configurations. Nevertheless this approach is limited in the degree of architectural diversity it can provide. Moreover, all the pipeline stages of individual units need to be designed to meet the shortest clock period possible, resulting in timing slack within upsized units. In addition, asynchronous buffering between *variable-frequency* clock domains requires stringent setup/hold times to be met, further degrading fixed-configuration performance.

Reconfigurable computing [18] exploits advances in field-programmable VLSI technology in order to build processors that can fundamentally transform their architecture. Such approaches allow abundant architectural diversity to be available to the system, exposing considerable workload temperamentality. However, reconfiguration can not take place at a high rate, rendering the system incapable of exploiting the exposed temperamentality. In addition, the implementation of a specific architecture in reconfigurable technology is prone to severe sub-optimal fixed-configuration performance.

Other related work is a study by Kumar et al [2] which investigates the use of *heterogeneous* multi-core architectures. 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 [23]. Chen et al [22] also investigate the potential of employing cores of different widths and directing work to cores based on local ILP.

The Datascalar paradigm proposed by Burger et al [21], is an intriguing variant form of architectural contesting in which the processing cores differ only in the subset of the program data-set assigned to them.

2. Implementation of Architectural Contesting

Figure 1 illustrates a possible overall organization for a multi-core architecture that *contests* the performance of four processor architectures. The different cores concurrently attempt to execute the same code (possibly retrieved from the same front-end fetch engine). Each core broadcasts the instructions it retires along with their destination values (if applicable) to the other cores. The inter-core connection over which a core broadcasts its retired instruction results can be considered an extension of the *result bus*. We refer to this bus as the *global result bus* (GRB), and to the conventional intra-core result bus over which *completed* instruction results are broadcasted to the instruction queue entries and the register file as the *local result bus*. This broadcasting bus is pipelined in order for it to be able to sustain the maximum throughput of retired instructions. There will inevitably be a number of cycles delay between when an instruction retires in one core and when the result reaches others cores.

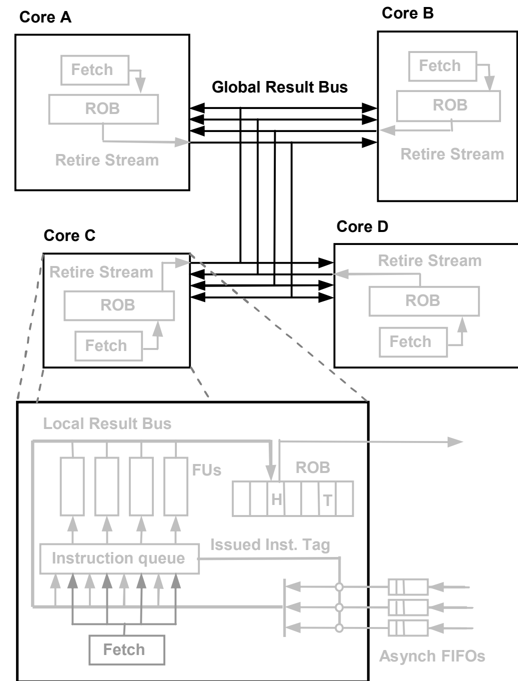

Figure 1: The general layout of an architectural contesting multi-core system.

When a core receives the result of an instruction that it has not yet retired, it abandons the execution of that instruction and retires the received result of the instruction to the register file or memory. This is performed by broadcasting the result on the *local* result bus. In this manner, execution in the lagging processing cores will never fall too far behind, so that 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 for the new code phase will naturally be able to *take lead*. The leading core from the previous phase will nevertheless have a slight head-start. How far behind a lagging core is depends on the physical distance between cores and the characteristics of the process technology. 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. However, in order to simplify parameterization of architectural configurations in this study we consider the result of a retired instruction to go through the same

number of pipeline stages of the global result bus before reaching any other core.

Since a core may be generating retired instructions at a frequency different from the functional frequency of a lagging recipient core, the retired instruction stream entering a core will need to go through an asynchronous buffer. Although the working frequencies of the cores may vary, the rate at which instruction results are retired by any core must be sustainable by all other cores. That is, the maximum retirement throughput of any core must be less than or equal to the maximum throughput at which instruction results can be written-back to the register file or memory in any other core. Without this condition, the lagging distance of a lagging core may unboundedly increase, resulting in excessive catch-up time, and therefore defeating the purpose of architectural contesting. We refer to a lagging core that can not keep-up with the leading core as a *saturated lagger*.

Each instruction is assigned an ID according to a global numeration across all the different cores. This instruction ID is also broadcasted along with the corresponding retired instruction. As instructions retire in-order, each core can determine the execution of which of its instructions it can drop simply by comparing instruction IDs with that of incoming retired instruction streams. In section 2.3 we look into different implementation criterions for writing-back an instruction result received from the GRB.

The following subsections explore different aspects of the major components of a contesting system.

2.1. The Front-end Fetch Unit

A major issue regarding each core in a contesting system is whether the different cores should share the front-end fetch unit (they are after all executing the same code). Figure 2 illustrates two basic approaches for the configuration of the fetch unit. If different fetch-unit configurations are to be contested, the cores will obviously have separate fetch units.

Cores that employ the same fetch mechanism can share the same fetch unit. However, the fetch unit is not an autonomous architectural unit and requires feedback from the processing core for verification of speculations. This can be performed by connecting the global result bus to the fetch unit. In this manner the first core to retire a conditional control transfer instruction will be providing feedback to the fetch unit. This form of design will however add to the complexity of the fetch unit and increase the latency between speculation and resolution of control-transfer instruction. It has been shown that the latency for which it takes a branch to resolve can have severe effect on overall performance [18].

An alternative approach is to employ separate fetch units per core even when the fetch mechanisms are the same. In this manner, the leading core will be able to feedback results to its own fetch unit with minimal delay, resulting in minimal latency between speculation and resolving of speculations. Since the performance of the leading core determines the performance of the overall system, this issue is of great importance. The downside of this approach is redundancy in fetch units that performing identically.

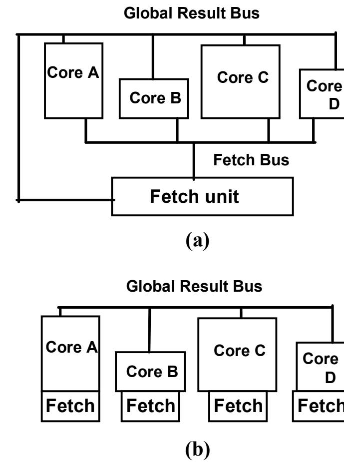

Figure 2: Two configurations for the front-end fetch unit; a) A common fetch unit connected to the global result bus, and b) Separate fetch units that obtain feedback information directly from the corresponding processing core.

Contesting different fetch mechanisms appears as an intriguing approach to dealing with branch mispredictions. However, divergence of control-flow paths in the different cores introduces the issue of dealing with results received from the GRB that have not been locally dispatched yet. Such a scenario will most probably not occur when the fetch unit is shared and the system is not saturated (i.e. the latency of an instruction to retire in core A, be transmitted over the GRB and reach core B, will most probably be greater than the latency of the same instruction to be dispatched in core B). However, with differing fetch mechanisms, a global result may arrive that has not been locally dispatched due to control-flow misspeculation. Therefore, there may be need for adjusting ROB entries based on results received from the GRB.

2.2. The Core Layout & Global Result Bus

Physical core layout and the structure of the global result bus, the main artery of a contesting system, determine the point-to-point latency between cores, and therefore the lagging distance of different architectures – depending on the leader.

This introduces a heuristically-exploitable dimension to the design space. An issue that may influence design options is the tendency of workload phases for which core A is more suitable to be followed by a phase of code for which core B is more suitable. If such a tendency is observed, it may be more beneficial for core B to have a short lagging distance when core A is leading, than it is for other cores. Placing cores A and B in close physical proximity will allow effective execution to be transferred to core B with less delay, enabling better exploitation of temperamentality.

An issue of importance in the design of the global result bus is that it is interconnecting potentially different clock domains and therefore there is need for asynchronous buffering at some point in order to synchronize communication. However, this asynchronous buffering, unlike that in a GALS architecture [20], will not affect fixed configuration performance, and will be placed between fixed-frequency clock domains (allowing better timing characteristics).

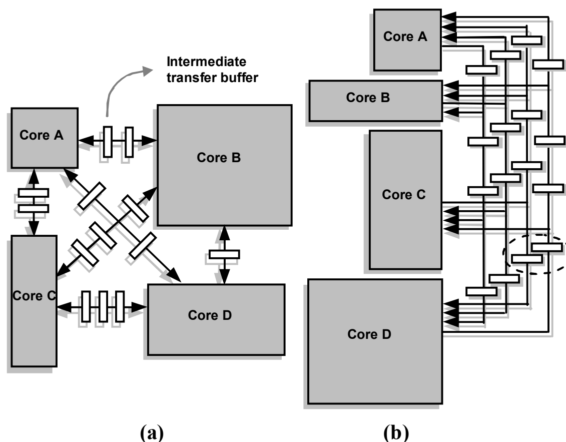

Figure 3: Example configurations for the layout of the processing cores and the partitioning of the global result buses.

Other heuristic design aspects of the GRB may also prove worthy of exploration. For example, figure 3 displays two potential layouts for the cores and the GRB buffering of a 4-way contesting system. In layout (b), the result of the same instruction being broadcasted by core D may be found to also have been broadcasted by core C. If this issue is detected at the circled neighboring GRB buffers, the broadcasting of core D's version of the result can be discontinued, resulting in better GRB power management.

2.3. The Memory Hierarchy

The memory accesses of each core in the contesting system are independent of that of other cores. Therefore, if every core were to have a complete independent memory hierarchy the system will function correctly. However, in implementation, such an approach will result in extremely high cost.

Since all the concurrently running cores in a contesting system are effectively executing one program, a more rational approach is for them all to share the same memory state. However, as in any multi-core architecture, it is of importance for the design of the memory hierarchy to be such that memory consistency is preserved.

An approach to enforcing consistency is to have all the cores snoop a *global data bus*, while requests put on this bus are augmented with the ID of the corresponding instruction. If a core observes that a particular memory request has already been submitted on the global data bus, it will not submit its own request. If the instruction is a *load* its value can be obtained from the GRB (see section 2.4) or a buffer of recently snooped values. If it is a *store* (considering write-through cache), it has already been performed anyway. Correctly dealing with *line-evictions* (considering write-back cache) is a more complicated issue and deeper analysis of this issue is deferred for future work.

In general, a single point of access is necessary to enforce memory consistency, and as the number of cores in the system increases a directory-based approach may even prove more effective. Therefore, an issue of concern is that an increase in architectural diversity will inevitably affect fixed-configuration performance adversely through complicating memory accesses. Nevertheless, cache misses are expected to be infrequent. Moreover, augmenting architectural contesting with the Datascalar [21] approach of distributing subsets of the program dataset among cores may result in better cache utilization. We leave analysis of such issues for future work, and consider completely separate memory hierarchies in this study.

2.4. The Resolution of Uncompleted Instructions

The condition for writing-back results received from the GRB is a design issue of importance, as it determines the efficiency in which the bypass bandwidth and register-file ports are utilized. The optimal option from an IPC standpoint is to write-back the GRB result of any instruction that has not locally *completed* as soon as it becomes available. However, due to the out-of-order nature of instruction completion, implementation of such an approach will result in considerable design complexity, i.e. there will be need for an associative lookup in order to determine whether the instruction has completed or not. This approach requires accurate *resolution of uncompleted instructions* (visibility of which instructions have not completed). Any approach that employs a lower resolution will increase the catch-up latency observed by lagging cores.

An alternative criterion for dropping an instruction and using the GRB result is the instruction to not have been *dispatched* locally. Since instructions are dispatched in-order, determining whether an instruction has been locally dispatched only requires a single value comparison. The downside of this approach is lower resolution of uncompleted instructions; i.e. newly issued instructions are left to perform an operation for which the result is already available.
Another alternative is to use a result from the GRB only if the instruction has not been *issued* locally. A relaxed implementation of this approach can allow greater resolution while also requiring only a single value comparison. In this approach, instructions are assigned tags according to a global enumeration and the tags of instructions received from the GRB are compared with the highest tag-number of issued instructions.
The resolution of long-latency instructions is of greater importance, and it may be beneficial to employ extra logic to identify such in-flight instructions. Depending on how the memory hierarchy is shared between cores load instructions may also require special treatment in order to efficiently and appropriately deal with cache misses.
In brief, there is a tradeoff between the effect of the resolution of uncompleted instructions on the lagging distance of cores and the added design complexity of greater resolution.

3. Empirical Observations

3.1. The Simulation Environment

We have developed a simulator capable of modeling the architectural contesting of two differently configured processors that run at different clock frequencies. The simulation of each core is based on the Simplescalar [3] instruction set. The differently configured cores place their retired instructions in a global buffer. A specified number of time units following the retirement of an instruction in one core, the result of the instruction can be retrieved by the other core if it has not yet issued that instruction. The local version of the instruction is removed from the issue queue of the corresponding core and the instruction is retired within the available retirement bandwidth.
Each core in the simulator has a completely separate memory hierarchy in order to simplify implementation of the simulator. In a real system however, levels of the memory hierarchy that are above the L1-cache would more likely be shared. Load instructions that are retrieved from the global buffer are retired without utilizing any cache access bandwidth. Store instructions are excluded from this process and execute as usual (due to the complete separation of the memory spaces).

3.2. Contesting Processor Width

In this study, we present results for the contesting of processor width only. Contesting greater architectural diversity will yield greater performance enhancement. However, due to the vast processor design space, providing an example of a contesting system that will not be deemed a mere special-case is virtually impossible.
Figure 4 displays the overall speedup of integer benchmarks when executed in a system that contests a 2-wide with an 8-wide superscalar processor. The results display the speedup of contesting over the performance of each separate core. Using the superscalar microarchitecture model by Palacharla et al [1] for 0.18 micron technology, the critical path of a 2-wide processor with a 32-entry issue-queue is the wakeup-select logic. That of an 8-wide processor with a 128-entry issue-queue is the bypass logic. Thus the clock period of the 2-wide processor is approximately 0.55 that of the 8-wide processor. In order to observe the effect of a broader range of process characteristics and microarchitectural features, we consider clock period ratios 0.5 and 0.6 in addition to 0.55.
The L1 data cache of the 2-wide processor is 2-ported 64KB 4-way set-associative with a 64-byte line size. For the 8-wide processor we choose 4-ported 96KB 6-way associative cache, which according to the CACTI-4.0 cache model [26] has approximately double the access latency. The cache miss-latency in both core configurations is equal to 10 clock cycles of the 8-wide processor.
The benchmarks *bzip* and *gap* observe an exceptionally high amount of ILP. Consequently, the 8-wide processor retires instructions at a rate that is unsustainable by the 2-wide processor, resulting in the saturation of the 2-wide processor, and no speedup is attained through contesting. Since saturation can be made detectable and the saturated lagger deactivated, we exclude these two benchmarks from our analysis.
With the core-to-core latency equal to 2 clock cycles of the 2-wide processor, and the ratio of clock periods equal to 0.55, architectural contesting gains an average minimum speedup of ~7.5% over any of the two fixed-configurations, with *perl* gaining ~12.5% speedup. With the clock-period ratio equal to 0.5, the 2-wide core is of better performance across all benchmarks (other than *bzip* and *gap*), and it is the speedup over this core that determines the ‘minimum’ speedup. However, as the clock-period ratio increases, the 2-wide processor looses its edge over the 8-wide and is outperformed under *vortex*, *vpr* and *gcc*.
Also displayed in figure 4 is the minimum speedup attainable though contesting these two cores when the core-to-core latency is equal to 7 clock cycles of the 2-wide processor. These results show the impact that

core-to-core latency may have on performance. This performance degradation highlights the importance of the layout of cores relative to each other as an important dimension of the design space of a contesting system.
Figure 5 displays the percentage of instructions dropped in each core when the ratio of clock periods is equal to 0.55. These results show that an average of ~10% of instructions in either core are dropped before being issued.

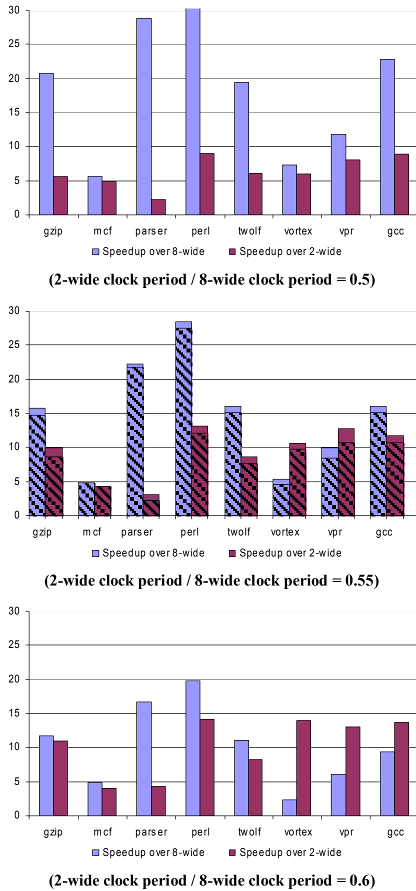

Figure 4: Percentage speedup of contesting 2-wide and 8-wide cores, over the performance of each core independently. Shaded columns represent speedup with a 7 cycle core-to-core latency.

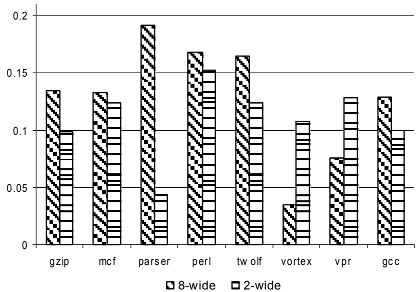

Figure 5: Ratio of dropped instructions in each of the contested cores.

When the fetch unit is shared between cores, the percentage of dropped instructions can be viewed as a measure indicative of the power efficiency of different implementations of architectural contesting.

4. Conclusions & Future Work

We propose architectural contesting as an approach that employs a proportional amount of resources as Thread-level Speculation and the Multiscalar paradigm, in order to enable faster execution of instructions that become available in normal von-neumann order.
We find only modest speedup to be attainable through contesting limited architectural diversity (i.e. processor width). However, this paradigm motivates the development of broader architectural diversity in the form of workload-customized architectures that perform well only under certain workload characteristics. Core-to-core latency is also found to be of impact on the speedup attainable through this approach.
Potential directions for future work consist of (1) analyzing the performance enhancement attainable through the contesting of broader architectural diversity, (2) approaches to dealing with saturated lagging cores, (3) studying the effect of more detailed implementation issues, such as the resolution of uncompleted instructions and the complexity of dealing with shared memory between contesting cores, (4) analyzing different topological structures and layouts for the global result bus, and (5) studying the effect of employing a shared fetch unit for contesting cores.

References

[1] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors", In *the 24th Inter. Symp. on Comp. Arch. (ISCA)*, June 2-4, 1997.
[2] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. "Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction", In *Inter. Symp. on Micro (MICRO)*, Dec. 2003
[3] J. E. Smith. "Instruction-level distributed processing", *IEEE Computer*, 34(4):59.65, April 2001.
[4] S. Balakrishnan, R. Rajwar, M. Upton, K. Lai, "The Impact of Performance Asymmetry in Emerging Multicore Architectures" *32nd Inter Symp on Comp Arch.*, June 2005.
[5] A. Dhodapkar and J. Smith. "Managing Multi-Configuration Hardware via Dynamic Working Set Analysis", In *Inter. Symp. on Comp. Arch. (ISCA),* pp 233-244, 2002.
[6] M. Huang, J. Renau, and J. Torrellas. "Profile-based energy reduction for high-performance processors". In *4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO4)*, Dec. 2001.
[7] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction", In *30th Inter. Symp. on Comp. Arch.*, 2003
[8] H. Akkary and M.A. Driscoll, "A Dynamic Multithreading Processor", in *Proc. 31st. Ann. Int. Symp. on Micro*, 1998.

[9] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. "Slipstream Processors: Improving both Performance and Fault Tolerance" *ASPLOS-IX*, Nov. 2000.
[10] D. Albonesi. "Dynamic IPC/clock rate optimization", *the 25th Inter. Symp. on Comp. Arch. (ISCA)*, pp. 282--292, June 1998.
[11] D. Ponomarev, G. Kucuk, O. Ergin, K. Ghose, P. Kogge, "Energy-Efficient Issue Queue Design", *IEEE Transactions on Very Large Scale Integration Systems*, 2003.
[12] D. Folegnani and A. Gonzlez. "Energy-Effective Issue Logic", *Inter. Symp. on Comp. Arch.* (ISCA) 2001.
[13] S.-H. Yang, M. D. Powell, B. Falsafi, K. Roy, and T. Vijaykumar, "An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance caches" *In Inter. Symp. on High-Perf. Comp. Arch.*, pp 147-157, 2001.
[14] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. "Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures" In *Inter. Symp. on Micro.*, pp 245-257, December 2000.
[15] D. H. Albonesi et al. "Dynamically Tuning Processor Resources with Adaptive Processing". In *IEEE Computer, December* 2003.
[16] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, S. Dwarkadas, "Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures ", *Inter. Symp. on Micro.*, 2000.
[17] M. Huang, J. Renau, J. Torrellas, "Positional Adaptation of Processors: Application to Energy Reduction", *Inter. Symp. on Comp. Arch.,* 2003.
[18] E. Hao, P. Chang, and Y. Patt, "The Effect of Speculatively Updating Branch History on Branch Prediction Accuracy, Revisited", *27th Inter. Symp. on Micro. (MICRO)*, pp. 228-232, 1994.
[19] K. Compton and S. Hauck. "Reconfigurable computing: A survey of systems and software", *ACM Computing Surveys, 34(2)*, 2002.
[20] S. G. Dropsho, G. Semeraro, D. H. Albonesi, G. Magklis, M. L. Scott, "Dynamically Trading Frequency for Complexity in a GALS Microprocessor", *Inter. Symp. on Microarch. (MICRO)*, pp. 157-168, 2004.
[21] D. Burger, S. Kaxiras, and J. R. Goodman, "DataScalar Architectures", *24th Inter. Symp. on Comp. Arch. (ISCA)*, June, 1997.
[22] L. Chen, D.H. Albonesi, and S. Dropsho, "Dynamically Matching ILP Characteristics Via a Heterogeneous Clustered Microarchitecture", *IBM Watson Conf. on the Intera. Between Arch., Circuits, and Compilers*, pp. 136-143, Oct. 2004.
[23] R. Kumar, D. M. Tullsen, N. P. Jouppi "Core architecture optimization for heterogeneous chip multiprocessors", *Parallel Arch. and Comp. Tech.* (*PACT),* pp. 23-32, *2006*.
[24] M. Cintra, J. F. Mart´ınez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Inter. Symp. on Comp. Arch. (ISCA)*, pp. 13–24, 2000
[25] M. Franklin, "Multiscalar Processors", Kluwer Academic Publishers, 2002.
[26] D. Tarjan, S. Thoziyoor; N. P. Jouppi, "CACTI 4.0", *Technical report HPL-2006-86*, 2006.