Appears in the Proceedings of the 2009 Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures

Hierarchical Control Prediction: Support for Aggressive Predication

Hadi Esmaeilzadeh
Department of Computer Sciences
The University of Texas at Austin
hadi@cs.utexas.edu

Doug Burger
Microsoft Research
One Microsoft Way, Redmond, WA 98052
dburger@microsoft.com

Abstract

Predication of control edges has the potential advantages of improving fetch bandwidth and reducing branch mispredictions. However, heavily predicated code in out-of-order processors can lose significant performance by deferring resolution of the predicates until they are executed, whereas in non-predicated code those control arcs would have remained as branches, and would be resolved immediately in the fetch stage when they are predicted. Although predicate prediction can address this problem, three problems arise when trying to predict aggressively predicated code that contains multi-path hyperblocks: (1) How to maintain a high bandwidth of branch prediction to keep the instruction window full without having the predicate predictions interfere and without increasing the branch mispredictions, (2) how to determine which predicates in the multi-path hyperblocks should be predicted, and (3) how to achieve high predicate prediction accuracies without centralizing all prediction information in a single location. To solve these problems, this paper proposes a speculation architecture called hierarchical control prediction (HCP). In HCP, the control flow speculation is partitioned into two levels. In parallel with the branch predictor, which identifies the coarse-grain execution path by predicting the next hyperblock entry, HPC identifies and predicts the chain of predicate instructions along the predicted path of execution within each hyperblock, using encoded static path approximations in each branch instruction and local per-predicate histories to achieve high accuracies. Using a 16-core composable EDGE processor as the evaluation platform, this study shows that hierarchical control prediction can address these issues comprehensively, accelerating single-threaded execution by 19% compared to no predicate prediction, and thus achieving half of the 38% performance gain that ideal predicate prediction would attain.

1. Introduction

Predication offers several benefits, including linearized control flow for high-bandwidth instruction fetch and potentially reduced branch mispredictions. However, predicates are typically evaluated at execution time, which can cause large performance losses compared to correctly predicted branches. Even though previous research has shown that predication can improve performance significantly [1]–[4], superscalar architectures have applied only limited hammock predication due to the complexity and negative performance effects of combining predication with dynamic scheduling in an out-of-order environment [1], [5]. Predicate prediction can mitigate the performance losses associated with execution-time evaluation of control flow arcs, and has been evaluted for conventional superscalar architectures. However, previous predicate prediction research [6], [7] typically assumes that the predication is only applied to a small number of statically identified hammock branches that are anticipated to be difficult to predict.

Due to power constraints, multiple researchers are exploring execution models that accelerate single threads across multiple lightweight processor cores [8], [12]–[14], with the prediction and fetch functions distributed across those cores. One example is composable processor architectures [8], which use EDGE ISAs [13] to support in-flight execution of multiple predicated hyperblocks. Each block produces one branch result which determines the next block to execute. Within a block, all control is determined by predicates produced by test instructions. Since these microarchitectures are designed to support large windows of execution, they require both good prediction accuracy and high fetch bandwidth. These requirements make heavily predicated code a challenge: if no predicate prediction occurs, large performance losses due to inhibited parallelism result. If predicates are predicted and all predictions are serialized, distributed and high-bandwidth fetching becomes throttled. Non-serialized predictions, however, have the potential to produce high rates of mispredictions in a distributed microarchitecture, since much correlation information is lost. Finally, in heavliy predicated blocks, there are many potential paths through the block, and so determining which predicates to predict is important for predictor bandwidth and performance.

This paper studies a number of strategies for addressing these predicate prediction challenges. For brevity, we call the overall approach *Hierarchical Control Prediction* (HCP). HCP predicts branches independently from predicates, allowing multiple instruction blocks to be quickly predicted and launched in flight. Once in flight, each block's predicates are predicted. Thus, many in-flight blocks may be predicting their predicates in parallel. Within each block, HCP addresses the issue of which predicates to predict by first predicting all non-predicated test instructions, and then following the predication chain for each, predicting each predicated test instruction until the end of each chain is reached. This approach avoids predicting predicates down paths that are not predicated to be valid. We show that by tagging each branch with a three-bit exit code in the compiler, and choosing the codes for all of a block's branches to approximate the predicate path through the block to each branch, the overall branch (next-block) prediction accuracy can be kept high without requiring a serialization of all predicate predictions. Finally we show that by coupling an OGEHL predicate predictor with a local

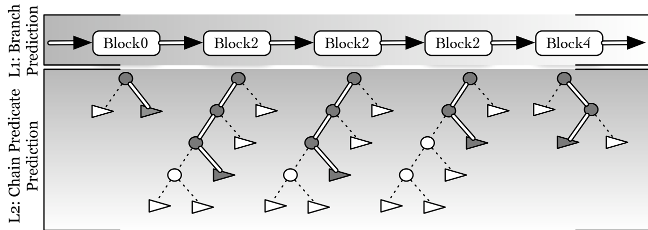


Figure 1: Program execution with HCP. In parallel with the branch predictor which identifies the coarse-grain execution path, the predicate predictor determines the fine-grain execution path within the fetched hyperblocks. The gray circles indicate the predicate instruction predicted as the ones on the correct path. The gray triangles indicate the taken exit branch from the hyperblocks.

table to track per-predicate history, high predicate prediction accuracies can be achieved within each block.

The results indicate that HCP is able to provide significant performance improvements despite the high degree of predication and microarchitectural distribution. Figure 8 shows the relative performance on a 16-core composable EDGE processor [8] (in single-threaded mode) of no predication (basic blocks), heavy predication (hyperblocks), and predication with perfect predicate prediction. Aggressive predication of basic blocks yields a 22% speedup, but much higher performance is possible; ideal predicate prediction would result in an additional 38% performance boost. Figure 10(a) shows mispredictions (load-store dependence, branch, and predicate) incurred per thousand instructions for no predication (basic blocks), heavy predication with predicate prediction (hyperblocks), and predicate prediction with confidence throttling. Even with predicate prediction, total mispredictions can be reduced compared to pure branch prediction. With these accurate predicate predictions and reduced total mispredictions, HCP attains a 19% performance increase over no predicate prediction, fully half of the ideal 38% speedup, using only 2KB of predicate predictor state per core.

2. Hierarchical Control Flow Prediction

With hierarchical control flow prediction scheme, the compiler applies if-conversion [9] more aggressively while relying on the underlying dynamic control flow speculation scheme to alleviate the overheads of predication. The compiler produces large hyperblocks[1] comprising more than two execution paths guarded by multiple predicates. Ideally, hard-to-predict branches as well as branches that enable sufficiently linearized control flow are predicated, and the remaining control points are left as branches. With this partitioning, the correct path of execution is identified with two levels of hierarchy as illustrated in Figure 1. As depicted, the branch predictor, which predicts all branches, only identifies the coarse-grain path of execution, while the predicate predictor determines the fine-grain path of execution within the fetched hyperblocks. By selectively predicting the predicates, the predicate predictor identifies the correct path of execution among the different paths constructing the fetched hyperblock. In addition to leveraging the benefits of predicate prediction without incurring the cost of extra data dependences, this scheme improves the total control flow speculation accuracy and allows the processor to better utilize its fetch bandwidth (see Figure 10(a)).

[1]A hyperblock is a set of predicated basic blocks in which control may only enter from the top, but may exit from one or more locations [9].

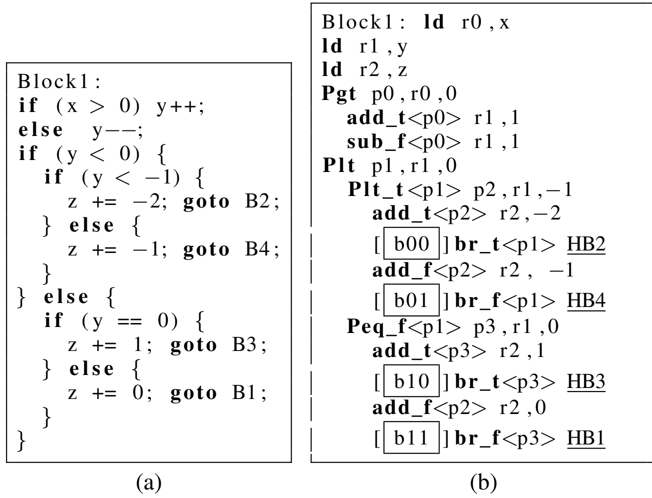
Figure 2(a) shows a sample C code containing a number of basic blocks. The compiler if-converts all the conditional statements and forms a large hyperblock comprising six different paths guarded by four predicates. Figure 2(b) shows the resulting hyperblock in the intermediate representation. In the figure, `add_t<p1>` means that the `add` operation is predicated on the `p1` predicate with the true polarity. Similarly, `sub_f<p0>` indicates that the `sub` operation is predicated on `p0` with the false polarity. The six paths in the hyperblock and the dpendences between the predicates guarding these paths are illustrated in Figure 2(c). To correctly speculate the execution path in the hyperblock, the predicate predictor needs to always speculate the output of the `p0` and `p1`. Nonetheless, strictly one of `p2` or `p3` needs to be predicted to avoid the execution of two exclusive paths at the same time. We propose the chain prediction strategy to address this issue.

2.1. Chain Predicate Prediction

In chain predicate prediction, all the predicates are predicted while being dispatched to the reservation stations. The reservation station is augmented with one bit which stores the speculative value of the predicate instructions. If the predicate instruction is not guarded by another predicate, is marked ready-to-issue after being predicted. Predicates `p0` and `p1` in Figure 2 are unguarded predicates that will be ready-to-issue speculatively once dispatched. On the other hand, the guarded predicates are predicted at dispatch time yet not marked as ready. The guarded predicates will wait for a matching predicate to enable them. In the example, predicates `p2` and `p3` are both predicted but are not marked ready. Given `p1` is predicted as `true`, only `p2` will be issued speculatively. With the proposed chain predicate prediction scheme, only *one* of the exclusive paths originating from `p1` is executed speculatively. The speculatively issued predicate instructions send the predicted values to the dependent instructions, enabling those with the matching polarity.

After sending the speculative value to the dependent instructions, the predicted predicate instruction is preserved in the reservation station waiting for its operands to arrive. Once the operands arrive, the predicate instruction is marked ready for the second time. The predicate instruction is executed for the second time with the correct non-speculative values and the result is compared to the speculative value stored in the reservation station. If the result is different from the speculative value, a misprediction signal is raised to flush the pipeline[2].

[2]It is possible to perform a selective replay in the case of predicate misprediction. However, due to its complexity in our design, the pipeline is flushed in the case of predicate misprediction.



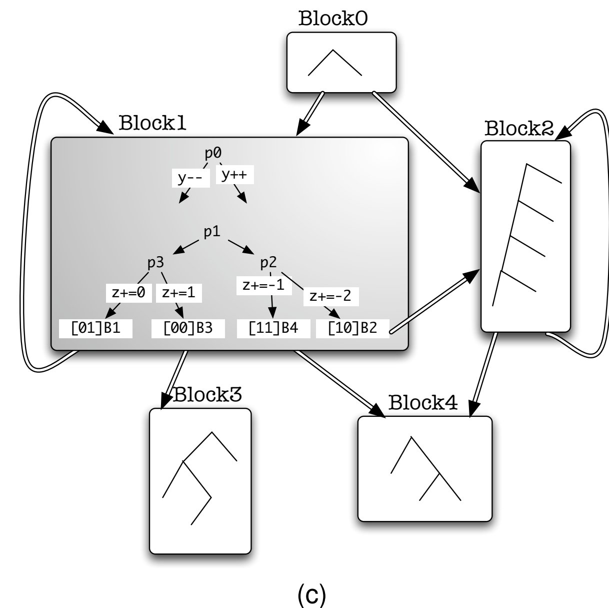


Figure 2: (a) The C code consisting of a number of basic blocks. (b) The equivalent multi-path hyperblock in the intermediate representation. (c) The six different paths constructing the hyperblock.

2.2. First Level of Hierarchy: Branch Prediction

2.2.1. Exit Prediction. In parallel with the predicate predictor, which speculatively identifies the correct execution path among the predicated paths forming the fetched hyperblocks, the branch predictor determines the coarse-grain execution path of the program. Unlike conventional branch predictors, which predict the taken/non-taken direction of each branch, the branch predictor in HCP predicts the exit of hyperblocks. The exit of the hyperblock is the ID of the branch that will be taken after the resolution of all the predicates in the hyperblock. In Figure 2, the branch IDs are binary values in the square brackets surrounded by a box. Each branch ID identifies a branch instruction in the hyperblock and is encoded in the opcode of the branch instruction. In the example, given that `p1` and `p2` are both resolve with `true` value, the taken branch will be the branch with the ID equal to `b10`. The exit predictor predicts the ID of the branch which will be taken

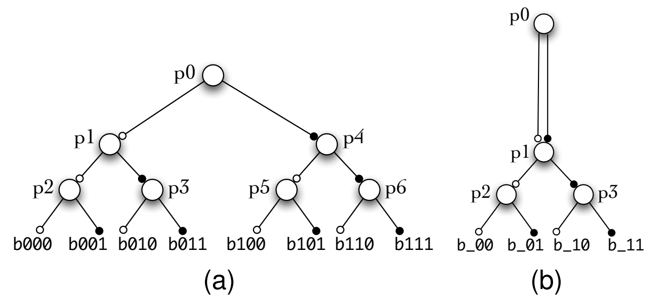


Figure 4: Predicate dependence tree. The nodes are predicate instructions and the edges indicate dependences between the predicates. The binary codes represent the branch IDs assigned to each exit. The hollow-headed arrows represent the true polarity and the solid-headed arrows represent the false polarity.

after the resolution of the predicates in the hyperblock. The branch predictor determines the branch target by using the predicted branch ID to access the branch target buffer. The predicted target is the entry of another hyperblock, which will be fetched next. In HCP, the branch/exit predictor can run ahead without waiting even for the branch instruction to be fetched or the predicates to resolve. This approach decouples the branch prediction from predicate prediction and makes it possible for them to run in parallel without interfering with or waiting for each other (see Figure 1).

Design and implementation of a next block predictor is presented in [8], which comprises two main components: the exit predictor, which predicts the ID of the branch that will be taken out of the hyperblock and the target predictor, which predicts the address of the next hyperblock based on the predicted branch ID. The exit predictor is a hybrid Alpha 21264-like tournament predictor composed of one two-level local, one global, and one choice predictor. The exit predictor uses local and global exit histories built from branch IDs assigned statically to each branch in the hyperblock. The branch IDs are used to construct the local and the global histories in the exit predictor instead of taken/non-taken bits as used in conventional branch predictors. In this implementation, the branch ID is a three-bit value encoded in the branch instruction opcode and assigned based on program order. This simple branch ID assignment does not encode any correlation information in the branch IDs that are used to construct the global and local history information. We propose path-based branch ID assignment to enhance the quality of information captured in the history registers.

2.2.2. Path-Based Branch ID Assignment. We describe the path-based branch ID assignment procedure by two examples. Figure 4 illustrates two predicate trees in which the nodes are predicate instructions and the edges represent the dependences between the predicate instructions. Hollow-headed arrows represent the false polarity, whereas solid-headed arrows represent the true polarity. For example, in Figure 4(a), node `p2` is guarded on false polarity by node `p1`, while node `p6` is predicated (on true polarity) on node `p4`, which in turn is guarded (on true polarity) by node `p0`. The leaves of the tree and the numbers beneath the leaves represent the

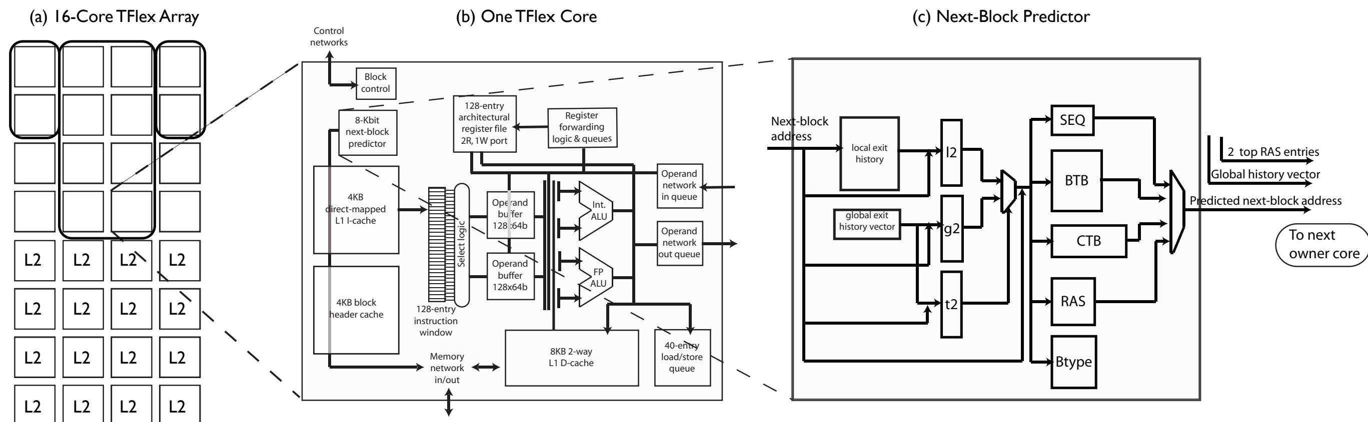


Figure 3: (a) 16-core TFlex processor. (b) Components and structure of a single TFlex core. (c) Internal organization of the next-block predictor.

hyperblock exits and the IDs assigned by the compiler to the corresponding branches. The branch ID assignment in Figure 4(a) encodes the entire predicate path that leads to the corresponding hyperblock exits. For example, the `b101` branch/exit will be taken if `p0`, `p4`, and `p5` resolve to `true`, `false`, `true`, respectively. Figure 4(b) shows the case where the branch ID assignment can not encode the entire predicate path because of the presence of a control-independent point in the tree. In this case, the branch IDs partially encode the predicate path leading to the hyperblock exits. Even though the encoding is partial it still contains some of the correlation information. With path-based branch ID assignment, the compiler assigns the branch IDs as described above, enhancing the quality of information collected in the global history registers of the branch and predicate predictors. The results presented in Section 3 shows that compile-time path-based branch ID assignment considerably improves the accuracy of the run-time predicate prediction.

3. Second Level of Hierarchy: Speculating Predicated Paths

Before diving into the design of the predicate predictor, the second level of prediction hierarchy in HCP, we present a general distributed/clustered execution model. The proposed predicate predictor is designed and implemented considering the challenges and issues of decentralized prediction, while having partial correlation information available at the prediction sites (cores/clusters). One of the contributions of this work is constructing global history information by using compile-time generated ISA tags (branch IDs) as well as the local information available at each prediction site. The design objective is to achieve a high degree of accuracy, while minimizing the communication among the prediction sites. The design tradeoff is between the prediction accuracy and the amount of correlation information communicated.

It is important to notice that the mechanisms and approaches presented in this paper (HCP hierarchical speculation

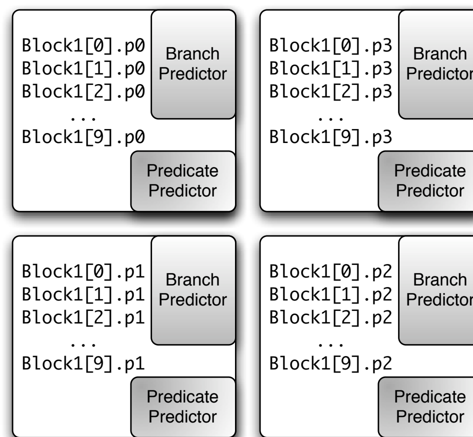


Figure 5: Distributed execution of a hyperblock. In all the iterations, the predicate instructions are mapped to the same core. Each core is augmented with a predicate predictor.

scheme, chain predicate prediction, and path-based branch ID assignment) are not specific to any architecture. However, the high-bandwidth fetch offered by predication can be exploited more effectively in clustered or distributed architectures [8], [10]–[15]. These architectures provide enough fetch-dispatch-execution resources to mitigate resource contention caused by false-path instructions. This section describes the speculation of predicated paths in a larger context that includes both distributed and conventional architectures.

Based on the chain prediction approach, predicate prediction is performed in the dispatch stage. As depicted in Figure 5, the proposed predictor design assumes that each core/cluster dispatches the instructions mapped to that core/cluster. Thus, each core/cluster is augmented with an independent predicate predictor. At the beginning of each fetch cycle, the core/cluster that predicts the exit branch ID broadcasts the predicted branch ID to all the cores/clusters that will execute the instructions in the hyperblock to be fetched. In addition, it is assumed that each instruction in the hyperblock is always mapped to the same core/cluster on which was mapped in previous iterations

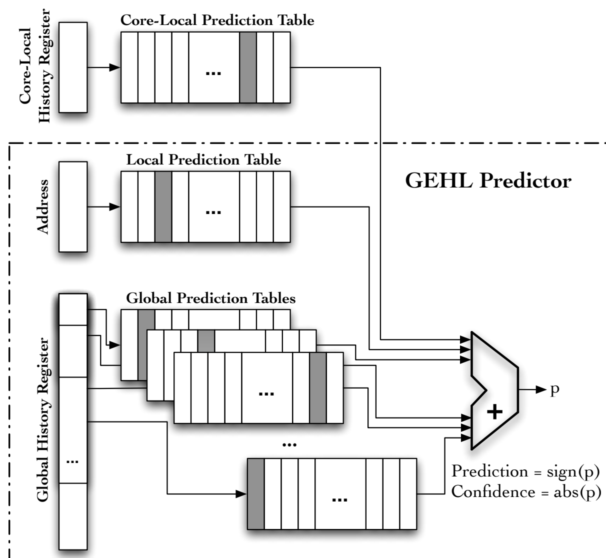


Figure 6: Architecture of the predicate predictor. The base predictor is GEHL surrounded by a dashed box in the drawing. A different number of bits from the global history registers are used to compute the index to the prediction tables. Each entry in the prediction table is a signed saturating counter. The core-local prediction table is the extra table indexed by CLPHR, which augments the GEHL predictor to improve the prediction accuracy.

(see Figure 5). With this general assumptions, this section discusses the design and implementation of a distributed predicate predictor that speculates on the correct path of execution in the fetched hyperblocks and functions as the second level of hierarchy in the HCP scheme.

3.1. Base Predictor

We use a GEometric History Length (GEHL) [16] branch predictor as the base prediction algorithm because, compared to the other state-of-the-art predictors, GEHL delivers high accuracy, requires less state, and has lower complexity. As depicted in Figure 6, the GEHL predictor uses multiple prediction tables to generate a prediction. Each entry in the tables is a signed saturating counter. The tables are indexed by hashing contents of the global history register (GHR) with predicate instruction addresses. The hash function of each table uses a different number of bits from the GHR. The number of bits used to compute the index to the prediction tables form a geometric series. Because in the GEHL predictor some tables are indexed using recent bits of the GHR and other tables are indexed using old bits of the GHR, GEHL can capture correlation between recent and old predicates. The prediction is the sign of sum of all the values retrieved from the prediction tables; positive corresponds to a `true` predicate value and negative corresponds to a `false` predicate value. The absolute value of the sum is the estimated confidence level of the prediction. When the confidence estimate is less than a pre-sepcified threshold, the predictor chooses not to predict. The confidence threshold is used to throttle the speculation when the predictor is not confident about the prediction.

We use the best reported GEHL predictor in [16] with eight tables and a 125-bit global history register in each core/cluster. With this organization, the total state of each predictor is 11 Kbytes. This large configuration is used to evaluate various schemes of constructing global history without incurring accuracy loss due to the small storage budget. After identifying the best strategy to construct the global history, the predictors are sized down such that they deliver about the same accuracy, but with a smaller state.

3.2. Constructing Global History Information

In this section, different approaches of constructing the global predicate history are presented. The goal is to achieve a high degree of accuracy by exploiting the correlation between the predicates, while minimizing the communication among the cores/clusters.

3.2.1. Core-Local Predicate History Register (CLPHR). The first approach, Core-Local Predicate History Register, implements one extreme of the design space. CLPHR restricts each predictor to only use the local information available at the prediction site. With this approach, the predictors do not communicate any information to one another. Each core/cluster owns its exclusive global history register, which only tracks the predicate instructions mapped to that core/cluster. The MPKI results for this approach are presented in the second column of Table 1. The CLPHR approach is fairly accurate because the dependent instruction are usually mapped to the same core/cluster. Therefore, dependent predicate instructions are also mapped to the same core/cluster. CLPHR exploits the correlation between these instructions. In addition, the base GEHL predictor uses long histories, which to some extent makes the predictor robust to information loss. In fact this approach trades prediction accuracy for no communication. The advantage of the CLPHR approach is that it requires no communication.

3.2.2. Global ID History Register (GIHR). The next approach, Global ID History Register (GIHR), implements another extreme of the design space. This approach uses the branch IDs predicted by the block exit predictor to construct the global history register. As discussed before, the block exit predictor uses the same approach to construct its own global history register. In this approach, the branch IDs are assigned statically at compile-time based on program order without encoding any predicate path information. To examine this approach, we only used three bits in the branch instruction opcode. Using three bits implies that each hyperblock can only have eight exit points or branch instructions. To construct the GIHR global history register at each predicate prediction site (dispatch stage of each core/cluster), the exit predictor broadcasts the predicted branch ID to all of the cores/clusters that will execute the hyperblock. The broadcast of the predicted branch ID can be combined with the fetch command that is sent regardless to all the cores/clusters to initiate the

Table 1: Mispredictions per 1000 instructions (MPKI) for different approaches of constructing global history register[3].

	CLPHR	GIHR	GIHR $\oplus$ CLPHR$_6$	GIHR $\oplus$ CLPHR$_{10}$	GIHR $\oplus$ CLPHR$_{36}$	GPHR.all	GPHR.all $\oplus$ CLPHR$_{10}$	GPHR.exit	**GPHR.exit $\oplus$ CLPHR$_{10}$**
bzip2	2.08	2.28	1.97	1.99	2.24	1.55	1.47	1.42	**1.37**
crafty	1.76	7.55	3.06	2.85	3.90	4.34	1.90	1.69	**1.36**
gcc	2.31	2.23	1.94	1.88	2.07	1.89	1.69	1.50	**1.44**
gzip	5.98	6.00	5.99	5.76	5.96	5.96	5.74	5.88	**5.75**
parser	1.41	1.63	1.53	1.44	1.38	2.01	1.67	1.27	**1.19**
perlbmk	0.04	0.05	0.04	0.04	0.04	0.04	0.03	0.03	**0.03**
twolf	11.41	11.99	10.98	10.92	11.82	10.86	10.15	9.87	**9.08**
vortex	0.66	0.67	0.57	0.58	0.62	0.51	0.46	0.51	**0.46**
average	3.21	4.05	3.26	3.18	3.50	3.39	2.89	2.77	**2.59**

fetch/dispatch process. The GIHR trades limited broadcast communication for better accuracy. However, as the results in the third column of Table 1 show, on average compared to CLPHR, GIHR incurs 0.84 more mispredictions per thousand instructions. The low accuracy is due to the low quality of the information captured in the global history register. The branch IDs do not encode any predicate path information, which severely reduces the quality of the information in GIHR.

3.2.3. GIHR $\oplus$ CLPHR. To take advantage of both approaches, we have combined the GIHR and CLPHR approaches by augmenting the GEHL predictor with and extra prediction table. As illustrated in Figure 6, the GEHL predictor is augmented with another table indexed by the CLPHR instead of the main global history register, which is GIHR in this case. To generate the prediction, the value retrieved from this extra table is added to the values retrieved from the other tables. As presented in Table 1 columns four through six, this compound approach is examined using various sizes for CLPHR. The results suggest that by adding the extra table indexed by a 10-bit CLPHR, the augmented GIHR method incurs 0.05 fewer MPKI than the CLPHR method. This approach trades extra space for better prediction accuracy. The core-local predicate history register (CLPHR) captures the correlation between the predicates and compensates for the lack of correlation information in the global ID history register (GIHR) resulting in improved prediction accuracy.

3.2.4. GPHR.all $\oplus$ CLPHR. The GPHR.all approach implements another point in the design space. In GPHR.all, all the prediction sites (cores/clusters) use the same global history register, which contains all the predicate outputs. In this approach when a core makes a prediction, it broadcasts the value to all the predictors to update their global history register. By using this approach, every prediction site can use all the information available about the predicates. Because of the large number of broadcast messages and the complexity of sequencing the predicate values, this approach is not practical in a distributed architecture, however, it can be adopted for conventional architectures. This approach trades broadcast communication for better correlation information in the global history registers. The results in the seventh column of Table 1 show that without CLPHR augmentation, GPHR.all approach performs worse than both CLPHR and compound GIHR. The lower accuracy is due to the pollution of the global history register. As discussed earlier, it is often the case that only one of the predicated path is executed. That is, only a subset of the predicates that determine the correct path should be included in the global history. Augmentation with a core-local prediction table compensates for the pollution by trading space for better accuracy (column eight).

3.2.5. GPHR.exit $\oplus$ CLPHR. The branch IDs, which are used to construct the global history, do not encode any correlation information. By using the proposed compile-time path-based branch ID assignment, the global history constructed from these branch IDs contains the correlation information between predicates. We refer to this method that uses the path-based branch IDs to construct the global history information as GPHR.exit. This approach trades limited communication for better accuracy. As the results in Table 1 column nine and column ten show, GPHR.exit outperforms all of the previous approaches. Augmenting GPHR.exit with the core-local prediction table also improves the prediction accuracy by 0.18 MPKI. This approach takes advantage of compile-time statically assigned path-based branch IDs to improve the run-time dynamic predicate prediction accuracy.

The best prediction accuracy is achieved when the global history register is constructed from branch IDs assigned statically based on the predicate path leading to the branches. Using path-based branch IDs outperforms the GPHR.all approach, which stores all the predicate values in the global history register. Since hyperblocks consist of multiple paths of execution from which only one or two gets executed, including all the predicates in the global history information decreases prediction accuracy.

3.3. Sizing the Predictor

Starting with the 11K predictor, we alter the number and width of the table entries to find a predictor with a smaller size but comparable accuracy. The tradeoff is between the storage budget allocated to the predictor and its accuracy. Sizing down the tables increases the chance of destructive aliasing and reduces the accuracy of the predictor. By examining different sizes and widths in an ad hoc manner, a predictor with comparable accuracy is found. The chosen predictor, which is referred to by the 2K predictor, comprises 17.5 Kbits or

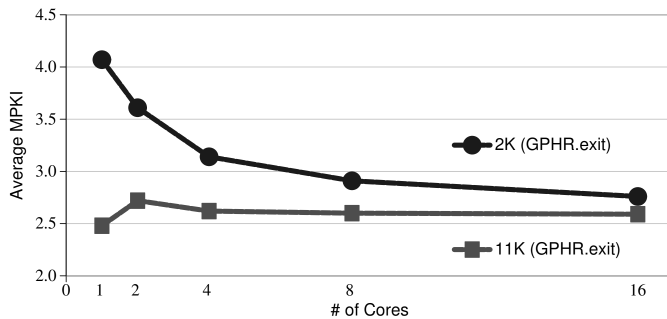


Figure 7: MPKI achieved with an 11K and a 2K predictor at each prediction site (core/cluster).

2.1875 KBytes of state. The 2K predictor is only 0.17 MPKI less accurate than the 11K predictor when used in the 16-core configuration, the common case for a distributed architecture (see Figure 7). It also delivers acceptable accuracy with other configurations. The results in Figure 7 show that running with a smaller number of cores results in less storage, more aliasing, and reduced accuracy when employing the 2K predictor. On the contrary, the accuracy of the 11K predictor does not significantly change by reducing the number of cores. It even performs slightly better with the 1-core configuration. In this case, the CLPHR captures all the correlation information while constructive aliasing improves the quality of the information stored in the tables. Furthermore, the 11K was sized and optimized for a single core configuration, while the 2K predictor is designed for a 16-core configuration.

4. Experimental Results

4.1. TFlex Composable Lightweight Processor

This section evaluates the hierarchical control flow prediction scheme using the TFlex composable lightweight processor [8], which implements an EDGE ISA [13]. EDGE ISAs are low-overhead, fully predicated instruction set architectures [17]. By supporting block-based execution model, EDGE ISAs eliminate the need for the reorder buffer in the micro-architecture. The register renaming stage is also eliminated through supporting direct instruction communication. These characteristics alleviate two major overheads of predication, pipeline stalls because of fetching and allocating false-path instructions in the reorder buffer and multiple register definitions along multiple if-converted control paths. Furthermore, distributed architectures similar to TFlex provide a high fetch bandwidth and large number of micro-architectural resources that considerably reduce resource contention caused by false-path instructions.

The TFlex processor is composed of distributed lightweight yet full-fledged processors that each can run threads individually or form a more powerful logical processor by aggregating with other cores. The operating system can run 16 threads each on one core, allocate all of the 16 cores to one thread, or assign a various number of cores to different threads. To provide this capability, all micro-architectural structures are distributed across the chip, which includes instruction and

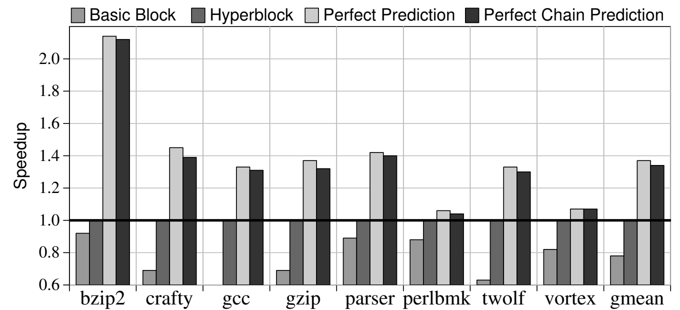


Figure 8: Achievable speedup with perfect chain predicate prediction. The baseline is hyperblock execution without predicate prediction.

data caches, branch predictor, register file, instruction window, and execution units (see Figure 3). Distributed fetch, execute, commit, flush, and misspeculation recovery protocols allow these structures to perform as a concrete logical processor.

To evaluate HCP, the TFlex cycle accurate simulator [8] is augmented with the compound GPHR.exit predicate prediction approach[4] The results are reported for eight integer SPEC2000 benchmarks simulated with the reference (large) data set using single simpoints [18]. The baseline is heavily predicated code, which is compiled with the `-Omax` flag using the compilation techniques presented in [19].

4.2. Performance Impact of HCP

4.2.1. Chain Predicate Prediction. Figure 8 shows achievable speedup limit with perfect chain prediction. As shown in Figure 8 perfect prediction without delaying the prediction of the guarded predicates is only slightly better than the perfect chain prediction. This slight difference results from the fact that unguarded predicates are predicted and passed down very early to the dependent instructions. On the other hand, dependent guarded predicates are also usually mapped close to or on the same core as the guarding predicates.

4.2.2. Speedup. Figure 9(a) shows the speedup over the baseline (hyperblock execution without predicate prediction) along with the achievable performance with perfect prediction. As shown in Figure 8, employing high-degree of predication with no predicate prediction improves the performance by 22%; however, because of the extra data dependences introduced by predication, it does not benefit from the 38% performance improvement achievable by perfect predicate prediction. The 2K predictor performs almost the same as the 11K predictor in the 16-core configuration and achieves a 19% speedup, half of the potential performance improvement.

[4]TFlex architectural parameters: **Instruction Supply:** Partitioned 8KB I-cache (1-cycle hit); Local/Gshare Tournament predictor (8K+256 bits, 3 cycle latency) with speculative updates; Local: 64(L1) + 128(L2), Global: 512, Choice: 512, RAS: 16, CTB: 16, BTB: 128, Btype: 256. **Execution** Out-of-order execution, RAM structured 128-entry issue window, dual-issue (up to two INT and one FP). **Data Supply** Partitioned 8KB D-cache (2-cycle hit, 2-way set-associative, 1-read port and 1-write port); 64-entry LSQ bank; 4MB decoupled S-NUCA L2 cache (8-way set-associative, LRU-replacement); L2-hit latency varies from 5 cycles to 27 cycles depending on memory address; average (unloaded) main memory latency is 150 cycles.

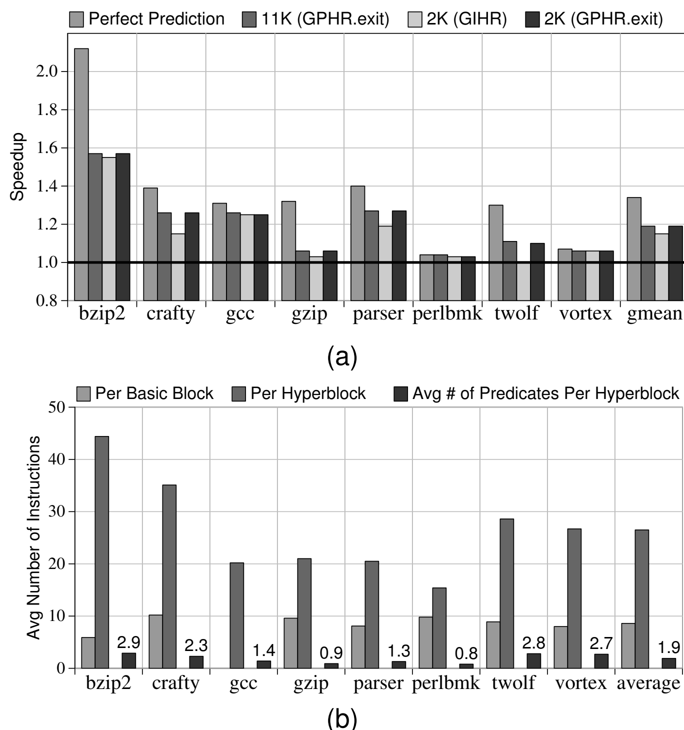


Figure 9: (a) Achieved speedup using the 11K and the 2K predictors. (b) Average number of instructions per basic block and hyperblock as well as the average number of predicates per hyperblock.

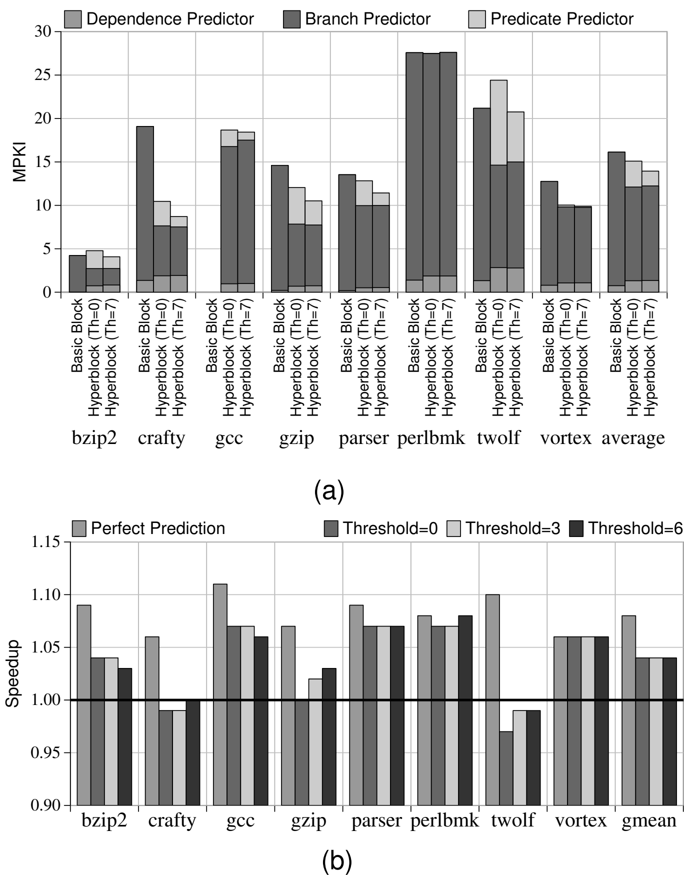


Figure 10: (a) MPKI variance across predicate prediction schemes (16-core configuration). (b) Speedup with the 2-core configuration.

By comparing the graphs in Figure 9(a) and Figure 9(b), we see a direct correlation between the size of hyperblocks and achievable speedup. For instance, bzip, which has the largest hyperblocks, shows the highest potential for speedup, whereas perlbmk, which has the smallest hyper blocks, has little potential for performance improvement. This suggests that the more aggressive the predication, the more effective the hierarchical control prediction. It is also noticeable that path-based branch ID assignment improves the speedup from 15% to 19% as shown by the two rightmost bars in Figure 9(a).

Among the benchmarks, bzip, crafty, gcc, and parser achieve a speedup that is more than half of the potential speedup. This improvement is the result of the high prediction accuracy achieved by the interplay between the compiler and the micro-architecture (path-based branch ID assignment). Even though perlbmk and vortex do not provide a high potential speedup from predicate prediction, the achieved performance is very close to the limit. The relatively low accuracy of the predictor for the gzip and twolf benchmarks results in a low performance benefit compared to the achievable limit. Nevertheless, both of the benchmarks benefit from predicate prediction.

4.2.3. Throttling the Speculation. The confidence threshold can be used to throttle the speculation in the cases where the predictor is not confident about the prediction. In 16-core configuration, the only benchmark that benefits from confidence-based speculation throttling is twolf, which suffers from high misprediction rate. As depicted in Figure 10(a) in the 16-core configuration, despite the fact that confidence estimation reduces the number of predicate mispredictions, the total number of mispredictions is not reduced significantly. The predicate predictor only contributes to a relatively low fraction of overall mispredictions, and therefore the predicate misprediction reduction does not manifest as the reduction in the total number of mispredictions. Furthermore, speculation throttling suppresses a fraction of correct predictions, which exacerbates the results.

As shown in Figure 10(b), with 2-core configurartion the performance improvement potential is less than the potential level in the 16-core configuration. There are fewer resources available to take advantage of the parallelism exposed by predicate prediction, which eliminates the extra data dependences introduced by if-conversion. In this configuration, the misprediction penalty is more severe because of the reduced fetch and execution bandwidth. The increased misprediction rate (see Figure 7) due to the reduced amount of storage and increased degree of destructive aliasing exacerbates the performance loss. The crafty and twolf benchmarks, which already suffer from relatively low prediction accuracy, lose performance that can be mitigated by speculation throttling using a high confidence threshold. The gzip and perlbmk benchmarks achieve performance improvement with higher confidence levels. The other benchmarks do not benefit from confidence estimation for similar reasons described for the 16-

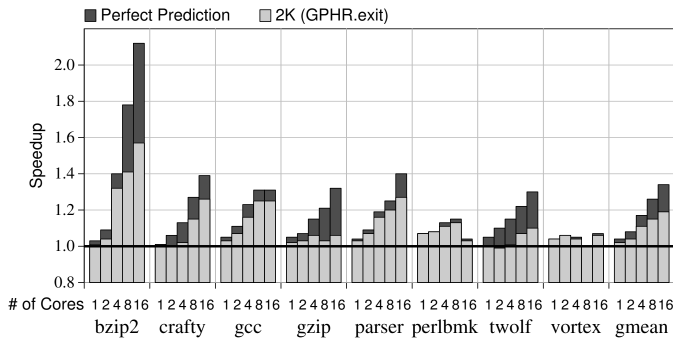


Figure 11: Speedup with different numbers of cores (issue widths).

core configuration. Nonetheless, in the 2-core configuration, the distributed predicate prediction scheme improves performance by 4%, which is half of the potential speedup.

4.2.4. Issue Width and HCP. Figure 11 shows the limit and the achieved speedup with different issue widths (number of aggregated cores). The larger the issue width (the amount of micro-architectural resources), the higher the speedup achieved by employing HCP. Larger window configurations with more micro-architectural resources can better exploit the parallelism exposed by predicate prediction. As Figure 11 shows, on average, HCP consistently achieves half of the potential speedup.

5. Related Work

Various studies show the effectiveness of predication in dynamically scheduled architectures. Pnevmatikatos and Sohi [1] evaluate the effect of predicated execution on instruction level parallelism in the presence of branch prediction. They show that predication can increase the effective size of basic blocks, which provides more opportunity for both the compiler and the dynamic hardware to extract fine-grained parallelism. The results also suggest that full predication can considerably increase the number of dynamic instructions executed between two branch mispredictions, which improves throughput. Chang et al. [3] employ predication to reduce the number of branch mispredictions by eliminating hard-to-predict branches. In this approach, profiling is used to identify the hard-to-predict hammock branches. In this approach, if-conversion is applied only in a limited form. This technique reduces the number of mispredictions; however, because of the limited application of if-conversion, this approach does not take full advantage of predication. Mahlke et al. [2] study full and partial predication and show that significant performance improvement is achievable by hyperblock formation or even partial predication in a relatively wide-issue machine (8-wide). To balance predication and control flow, August et al. [4] study where and when predication should be employed. They conclude that due to resource contention caused by the execution of false-path instructions, if-conversion should be applied selectively based on detailed analysis of the dynamic program behavior and the availability of micro-architectural resources. Even though the prior art suggests that predication can improve performance significantly, predication is only employed in a limited form (hammock predication) in out-of-order processors due to the lack of techniques that can alleviate the overheads of heavy predication in the dynamically scheduled architectures.

Different approaches have been proposed to cope with the overheads of predication [1], [4]–[7], [20], from which we focus on predicate prediction schemes [6], [20] and wish branches [20]. Chuang et al. [6] propose predicate prediction for out-of-order processors to alleviate the problem of multiple register definitions along the if-converted control paths. They reverse all the if-conversions by predicting the predicates, which reduces the effectiveness of predication penalty. To preserve the benefit of predication, this method utilizes a replay mechanism that makes the predicate misprediction penalty less than the branch misprediction. To preserve the benefit of predication on hard-to-predict branches, Quiñones et al. [7] propose selective predicate prediction that predicts predicates selectively based on the estimated confidence of prediction. This approach maintains 84% of the if-converted hard-to-predict branches (predicates) and outperforms the original predicate prediction approach [6]. Kim et al. [20] adopt a different approach to use predication for hard-to-predict branches. In their approach, the compiler preserves the branch instruction in the form of wish branch/jump in the binary. This way, the branch predictor, which is augmented with a confidence estimation mechanism, can dynamically decide whether to fetch the predicated code or predict the branch. It may be beneficial to combine predicate prediction with wish branches.

The previous predicate prediction research typically assumes that the predication is highly restrained and only applied to hammock branches that are difficult to predict. Within such hyperblocks, there are only two execution paths that are guarded by a single predicate. This type of limited hammock predication does not offer the full benefits of predication. The prior predicate prediction research does not provide accommodations for speculatively identifying the correct path within the hyperblocks comprising multiple predicated paths (more than two). The proposed HCP scheme and chain predicate predication addresses these problems. This paper also studied the effective approaches of constructing global history using path-based branch IDs and proposed a highly accurate predicate predictor design, which can be used in both conventional and distributed architectures.

6. Conclusions

Prediction of predicates is essential for good performance on out-of-order architectures running aggressively predicated code, in which many predicated paths can be fetched and in flight concurrently. Choosing which predicates to predict, and how to predict them well, is a challenge when code is a succession of predicates punctuated by branches, only some of which should be predicted. This challenge is particularly acute in EDGE architectures, in which aggressive predication is a crucial part of the execution model. Hierarchical control

prediction picks control points in the code (branches) and at instruction dispatch predicts a single path through each predicated region With this approach, multiple predicate prediction chains can be dispatched in parallel across distinct control regions. Essential to HCP is finding the right information to form good history vectors locally. In this paper, we evaluated the use of static tags in individual branches to support effective distributed predicate prediction with small (2KB) tables in each core.

The results show a 19% speedup (4% of which is due to compile-time path-based branch ID assignment) over predicated code with no prediction, which is half of the 38% performance increase, which is theoretically achievable with perfect prediction. More importantly, this prediction scheme operates with fully distributed control, and small numbers of lightweight control messages necessary to orchestrate correct global execution. Additionally, we show that throttling predictions based on confidence estimation turns out to be unimportant for a large-window (16-core) configuration. Since there are almost always branch mispredictions in flight anyway, the small additional reduction in mispredictions from throttling low-confidence predicate predictions does not improve performance. On smaller-window configurations, of course, confidence estimation remains important to determine which predicates should be predicted to maximize performance while reducing misprediction penalties.

In the long term, effective predicate prediction may enable more aggressive predication, in which any branch that is hard to predict (and which does not contain a function call down any frequently traversed paths), is predicated. With more aggressive conversion of hard-to-predict branches, confidence estimation should be important for even large-window configurations, and may enable a reduction in misprediction rates above and beyond what the best current branch predictors (i.e., L-TAGE, GEHL, and perceptrons) are able to achieve. Dataflow architectures that rely on complete predication are unlikely to compete with superscalar processors unless many of the control flow arcs are predicted. HCP is one approach for addressing that challenge.

References

[1] D.N. Pnevmatikatos and G.S. Sohi. Guarded execution and branch prediction in dynamic ILP processors. In *ISCA '94: Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 120–129, 1994.

[2] Scott A. Mahlke, Richard E. Hank, James E. McCormick, David I. August, and Wen-Mei W. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *ISCA '95: Proceedings of the 22nd Annual International Aymposium on Computer Architecture*, pages 138–150, 1995.

[3] Po-Yung Chang, Eric Hao, Yale N. Patt, and Pohua P. Chang. Using predicated execution to improve the performance of a dynamically scheduled machine with speculative execution. In *PACT '95: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 99–108, 1995.

[4] David I. August, Wen mei W. Hwu, and Scott A. Mahlke. A framework for balancing control flow and predication. In *MICRO 30: Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 92–103, 1997.

[5] Ralph M. Kling and Kalpana Ramakrishnan. Register renaming and scheduling for dynamic execution of predicated code. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 15, 2001.

[6] Weihaw Chuang and Brad Calder. Predicate prediction for efficient out-of-order execution. In *ICS '03: Proceedings of the 17th Annual International Conference on Supercomputing*, pages 183–192, 2003.

[7] Eduardo Quinones, Joan-Manuel Parcerisa, and Antonio Gonzailez. Selective predicate prediction for out-of-order processors. In *ICS '06: Proceedings of the 20th Annual International Conference on Supercomputing*, pages 46–54, 2006.

[8] Changkyu Kim et al. Composable lightweight processors. In *MICRO 40: Proceedings of the 40th Annual International Symposium on Microarchitecture*, 2007.

[9] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *MICRO 25: Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, 1992.

[10] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic. The multicluster architecture: reducing cycle time through partitioning. In *MICRO 30: Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 149–159, 1997.

[11] Ramon Canal, Joan-Manuel Parcerisa, and Antonio Gonzalez. A cost-effective clustered architecture. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, page 160, 1999.

[12] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *MICRO 36: Proceedings of the 36th Annual International Symposium on Microarchitecture*, page 291, 2003.

[13] Doug Burger et al. Scaling to the end of silicon with EDGE architectures. *Computer*, 37(7):44–55, 2004.

[14] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 186–197, 2007.

[15] C. Madriles et al. Mitosis: A speculative multithreaded processor based on precomputation slices. *IEEE Transactions on Parallel and Distributed Systems*, 19(7):914–925, July 2008.

[16] A. Seznec. The O-GEHL branch predictor. In *the 1st JILP Championship Branch Prediction Competition (CBP-1)*, 2004.

[17] Aaron Smith et al. Dataflow predication. In *MICRO 39: Proceedings of the 39th Annual International Symposium on Microarchitecture*, pages 89–102, 2006.

[18] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, 2001.

[19] Bert Maher et al. Merging head and tail duplication for convergent hyperblock formation. In *MICRO 39: Proceedings of the 39th Annual International Symposium on Microarchitecture*, pages 65–76, 2006.

[20] Hyesoon Kim, O. Mutlu, J. Stark, and Y.N. Patt. Wish branches: combining conditional branching and predication for adaptive predicated execution. In *MICRO 38: Proceedings of the 38th Annual International Symposium on Microarchitecture*, pages 43–54, 2005.