Compiler Controlled Speculation for Power Aware ILP Extraction in Dataflow Architectures

Muhammad Umar Farooq, Lizy John, and Margarida F. Jacome

Department of Electrical and Computer Engineering
The University of Texas at Austin
ufarooq@mail.utexas.edu, ljohn@ece.utexas.edu

Abstract. Traditional predicated execution uses two techniques: top predication – in which only the head of the dependence chain is predicated, and bottom predication – in which only the tail of the dependence chain is predicated. Top predication prevents speculative execution, thus delivering minimum performance at minimum energy cost, while bottom predication allows full speculation of the dependence chain, resulting in maximum performance at maximum energy cost. In this paper, we propose a novel *power-aware* ILP extraction technique, denoted the *'elastic-block'*, that combines these two extremes, exposing superior energy vs. performance trade-offs. Each instruction in the elastic-block is explicitly guarded by two predicates: the *speculative*, and the *final*. Instruction's final predicate is generated using traditional if-conversion technique, while the speculative predicate has its default value statically assigned by the compiler, enabling it to make power-performance trade-offs in the code. Several energy saving code optimizations are proposed for the elastic-block structure.

Keywords: Tiled dataflow architectures, predication, power-performance trade-offs.

1 Introduction

The formidable increases in raw transistor density projected for the next 10-15 years pose tremendous scalability challenges to future processor designs, as to how effectively use such devices. Tiled architectures, such as TRIPS, WaveScalar and RAW [1][2][3] exhibit very promising characteristics in that respect –namely, their decentralized organization eliminates several key scalability bottlenecks found in conventional superscalar processors, and reduces overall circuit complexity, effective wire delays and verification effort [4][5]. These favorable characteristics make tiled architectures highly relevant to the future of high performance computing. Large machines, exploiting the huge numbers of raw transistors, possible to integrate in future silicon technologies, can be built in a scalable way, by simply instantiating many such basic tiles on a processor's chip, and then hierarchically organizing them in a suitable way, see e.g. [1][2][3][6]. Aggressive instruction-level parallelism (ILP) extraction is key to the performance of tiled architectures, including WaveScalar, TRIPS, and RAW. Yet, performance/speed alone is not

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 324–338, 2009.

sufficient to quantify the effectiveness of such machines – achieving high energy efficiency is equally critical, so as to aggressively reduce the machine's energy consumption and power dissipation for a given performance point. In this paper, we propose a new *power-aware* ILP extraction technique, denoted the *'elastic-block'*, and show that it exposes superior energy vs. performance trade-offs for tiled architectures. We implemented the elastic-block on the WaveScalar ISA and computing model [2], so as to experimentally demonstrate its effectiveness on a concrete representative of the state-of-the-art in tiled dataflow architectures. Namely, we show that, by using the elastic-block structure, one can deliver almost the same performance of state-of-the-art aggressive ILP extraction techniques, while reducing the average number of instructions executed by 5.95%, and up to 9.95% for some benchmarks, and the average number of messages exchanged between instructions by 6.4%, and up to 23% for some benchmarks – which directly translates in enhanced energy efficiency.

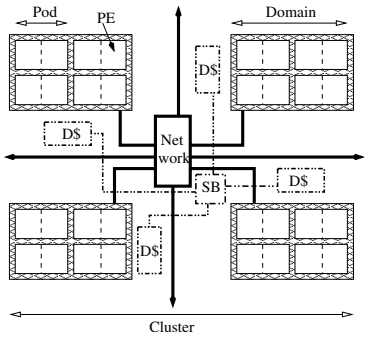

Fig. 1. WaveScalar Microarchitecture

In the next section we will give an overview of our target machine, WaveScalar. Section 3 will introduce elastic-block and its characteristics. Section 4 will explain implementation of elastic-block on WaveScalar. Our evaluation methodology and results are shown in section 5. Section 6 discusses related work in this area. Finally, in Section 7, we disscuss future work and conclude the paper.

2 Overview of Target Tiled Dataflow Machine: WaveScalar

WaveScalar is a dataflow architecture. As in other dataflow architectures, a program is represented as a dataflow graph and instruction dependencies are explicit [7][8]. There is no program counter, instructions are fetched and placed on the grid as they are required. There is no register file, the result produced by an instruction is directly communicated to all the consumers. In this architecture, instructions are grouped in blocks called waves. Waves can be defined as acyclic dataflow graphs for which each instruction executes at most once every time the

wave is executed, and to which control can enter at a single point. On exit and re-entry to this acyclic dataflow graph, the wave-number is increased.

Each dynamic instruction is identified by a tag which is the aggregate of its wave-number and location on the grid. When an instruction has received all its input operands for a particular matching wave-number, it fires, provided there is room to store the result in the output queue, and an ALU is available. The output is temporary stored in the output queue before it is communicated to the consumers. Figure 1 shows the basic WaveScalar Microarchitecture. The substrate consists of replicated clusters connected through a dynamically routed packet network. Each cluster consists of four domains, communicating through a fixed-route network switch which has a 4 cycle latency. Additionally, each cluster has a 32KB 4-way set associative L1 data cache, and a store buffer. Each domain is composed of eight processing elements (PEs), grouped into pairs of two. Each pair is called a pod. Pods communicate through a fixed 1 cycle latency pipeline network. Within PE instructions communicate through a bypass network. Figure 2 shows the 5-stage in-order PE pipeline. Each PE has a small instruction cache capable of holding 64 static instructions. Each PE has a 16 entry input queue and an 8 entry output queue.

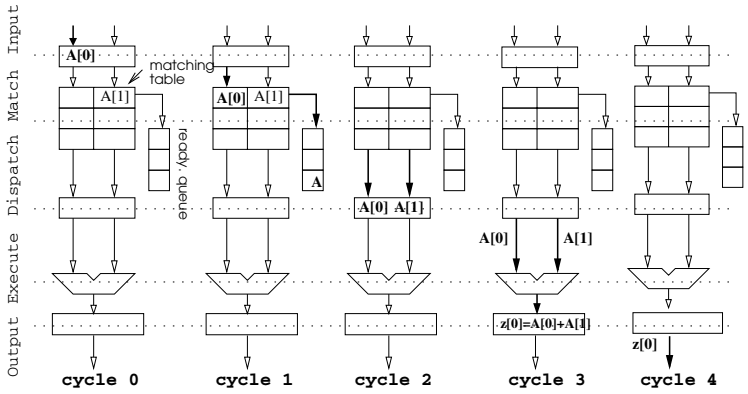

Fig. 2. Processing Element (PE) pipeline stages

2.1 Pipeline Stages of a PE

1. **INPUT:** Accepts input operand messages arriving from other PEs and from itself, and places the operands in the pipeline registers for the next stage.
2. **MATCH:** Operands are moved from the pipeline registers to the matching table at an index computed by XOR hash of the wave-number, thread-id, and destination instruction number of each operand. MATCH also determines which instructions have all their operands with the matching wave-number, thread-id and are ready to fire. It then issues all ready instructions to the DISPATCH stage by placing their matching table index in the ready queue.

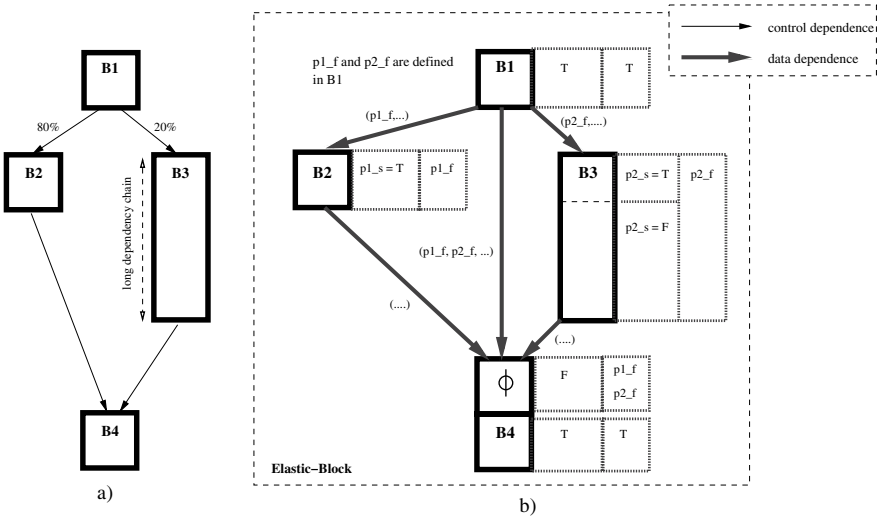

Fig. 3. Illustrating elastic-blocks. (a) Original control flow graph annotated with the dynamic frequency of execution of basic-blocks B2 and B3. B3 is depicted with a longer box relative to B2 in order to represent the fact that it contains much longer dependence chains. (b) Structure of resulting elastic-block code, with distinct default values assigned to the speculative predicate of instructions within B3, depending on their depth in the corresponding dependence chains.

3. **DISPATCH:** Removes the matching table indices from the ready queue, reads the corresponding operands from the matching table and forwards them to EXECUTE stage for execution.
4. **EXECUTE:** Instruction executes sending its results to the output queue.
5. **OUTPUT:** Removes the entries from the output queue sending them to the consumer instructions.

For better understanding of WaveScalar microarchitecture, we refer the readers to [9].

3 Power Aware ILP Extraction with Elastic-Blocks

In this section, we introduce the elastic-block and discuss its operation and power-aware features, in contrast to previous techniques.

3.1 The Elastic-Block Structure: Definition, Power-Aware ILP Extraction, and Energy Saving Code Optimizations

Guarding instructions with speculative and final predicates. The key innovation introduced in the elastic-block structure is the ability to explicitly guard instructions with two predicates – the *speculative*, and the *final* predicate.

The instruction's *final* predicate is generated using traditional if-conversion technique [10][11]. Specifically, all control dependencies within the code region targeted for elastic-block formation are converted into data dependencies, similarly to what is done, e.g., in hyperblock [12], conditional branches are replaced with comparison instructions which set the *final* predicate of all instructions that are control dependent on such branches. Even if generated using well known techniques, the final predicate is a very unique operand type in our target dataflow ISA, in that an instruction may actually execute even if the value of its final predicate is still unknown. Speculative predicates, in turn, always have a default value statically assigned by the compiler – as it will be seen, different such assignments can implement distinct power-performance trade-offs in the code. Speculative predicates explicitly enable *control speculation* – that is, when the speculative predicate of an instruction is set to TRUE, if all 'regular' operands of that instruction become available, while the value of its final predicate is still *unknown*, the instruction becomes ready for execution.

We noted that within an elastic-block, the compiler can *selectively* and *individually* define which instructions should be speculatively executed, and which should not. Indeed, while the value of the final predicate of all instructions within a basic-block must necessarily be identical, this need not be the case for their corresponding speculative predicates – the fact that each instruction keeps its own copy of the speculative predicate in our target tiled dataflow architecture allows such a discrimination to be made in a very natural/simple way. Of course, the value of the speculative predicate, as the name suggests, is only relevant while the instruction's final predicate is not available – namely, if an instruction receives its final predicate while still waiting for other (regular) operands, the value of the final predicate alone determines if the instruction will be executed or squashed prior to execution.

In turn, if the speculative predicate of an instruction is set to FALSE, then control speculation is explicitly disabled, that is, the instruction will not be ready for execution until it actually receives its final predicate value. If such final predicate happens to be FALSE, the instruction is locally squashed *prior* to execution. Otherwise, it, of course, executes. Note that, the semantics of our final predicate is, thus, somewhat different form that of the standard predication model adopted, for example in [13], due to performance reasons, instructions always execute, and their predicates are only used to decide if their results should be committed or not. Reflecting that fact, traditional hyperblock selection approaches, such as [12], do not favour the inclusion of large basic-blocks in a hyperblock, since such blocks utilize many machine resources and may actually end up negatively impacting performance, as opposed to enhancing it. As will be seen below, the elastic-block's increased flexibility enables such aggressive performance-enhancing ILP extraction techniques to be enhanced with energy awareness and efficiency considerations.

Note finally that, whenever control speculation is explicitly enabled in the elastic-block, ϕ functions may need to be inserted in the corresponding code, so as to potentially reconcile multiple (speculative) definitions/updates of the

same variable, i.e., make sure that only the 'right' value is actually sent to the corresponding consumers. As discussed in more detail in Section 4, such ϕ functions are implemented by *move* instructions, each guarded by the same final predicate used on the actual basic-block where the value, being sent, was generated.[1]

Simple illustrative example of power-aware ILP extraction using the elastic-block. Consider the weighted control graph shown in Figure 3(a). In this simple example, a conditional branch instruction in basic-block B1 defines two control paths, one through basic-block B2 and another through basic-block B3. Although the control path through B3 is taken much less frequently than that through B2 (on average 1 out of 5 times, as indicated in the figure), B3 also contains much longer data dependence chains. So, even if executed infrequently, B3 takes much longer than B2 to complete, so much so that it does actually impact overall performance. Assume also, that control speculation would substantially improve performance for this code segment, i.e., performance can be enhanced by starting to execute B2 and/or B3's instructions, prior to knowing which control path will be taken. Figure 3(b) illustrates how such performance can be delivered, in an energy efficient way, using the elastic-block structure.

Note first that, as alluded to before, all control dependencies in the elastic-block region have been converted into data dependencies using standard if-conversion – as indicated in Figure 3(b), final predicates (denoted as p1_f and p2_f) guard the instructions originally in basic-blocks B2 and B3 respectively, and their corresponding predicate-define instructions have been 'inserted in B1'. Note further that, in the simple example of Figure 3, B1 and B4 represent simple straight line code that always executes, and thus, the compiler can directly assign the default value TRUE to the speculative predicate of the corresponding instructions, thereby, eliminating the need for final predicate.

The key idea in energy-aware ILP extraction is to enable the selective speculation of *only* those instructions that may actually payoff in terms of performance enhancement, thus avoiding wasteful energy spending. In the case of the illustrative code segment shown in Figure 3, for example, the compiler has detected that a performance gain can be achieved by speculating *all* of the instructions in the most commonly executed block, i.e., B2, and thus it did set the default value of their corresponding speculative predicates (denoted p1_s in Figure 3(b)) to TRUE. Accordingly, the instructions in B2 will become ready for execution as soon as they receive all of their 'regular' operands, but their final predicate. In addition, the compiler has detected that speculatively executing a *select subset* of the instructions in B3, namely, those located 'early' in B3's long dependence chains, would also give a relevant performance gain. Accordingly, it did set the speculative predicates of that select subset of B3's instructions to TRUE, and assigned FALSE to the remaining (see p2_s values in Figure 3(b)). As alluded to before, since each instruction keeps its own copy of the speculative predicate, such a discrimination can be made in a very simple way. Note finally that, as

[1] Naturally, the speculative predicate of such *move* instruction is always FALSE, see Figure 3(b).

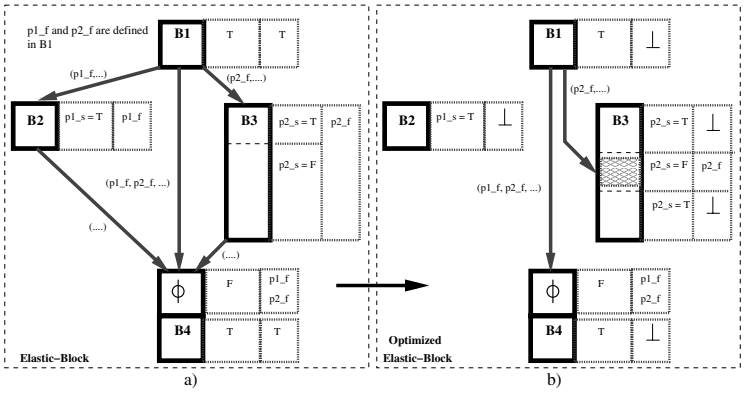

Fig. 4. Illustrating elastic-block code optimizations. (a) Non-optimized version. (b) Optimized version with elimination of final predicate messages. The edges in figure (b) represent only final predicate messages, for clarity. As it can be seen, the final predicate is no longer sent to the instructions originally in B2 – this is represented by placing the symbol bottom ($\perp$) in the corresponding field. In fact, the final predicate is now only sent to the first non-speculative instruction in the dependency chains of B3. Specifically, instructions that have data dependencies to these need not receive their final predicate as well, and in fact can be again made speculative, since they cannot execute unless their non-speculative predecessors send them their operands. So, we use transitivity effects to eliminate again final predicate messages.

indicated above, although the final predicates guarding B2 and B3's instructions are necessarily mutually exclusive (i.e., p2_f != p1_f), the speculative predicates guarding these blocks need not be, and in fact frequently will not be – this is why we have adopted naming conventions explicitly distinguishing among such predicates.

Energy Saving Code Optimizations. When an instruction is speculatively executed, yet its final predicate turns out to be FALSE, there is nothing to be done in the local context of the instruction – as alluded to above, the ϕ functions in the elastic-block code will make sure that only correct values are actually sent to the appropriate consumers. In fact, if the compiler can determine that the final predicate of an instruction will never arrive prior to its speculative execution, or will rarely do so, the message containing the final predicate should not even be sent to that particular instruction, thus reducing energy consumption as well as message traffic – this is one of the key energy saving optimizations that can be performed in elastic-block code, symbolically illustrated in block B2 and in the upper third of block B3, by placing the bottom or 'absence' symbol ($\perp$) in the corresponding final predicate fields, see Figure 4(b).

A second type of energy saving optimization can be done by directly relying on the very nature of the dataflow model and exploiting the transitivity of data dependencies inside a basic-block – this second type of optimization is symbolically illustrated in the bottom third of block B3 of Figure 4(b). Specifically,

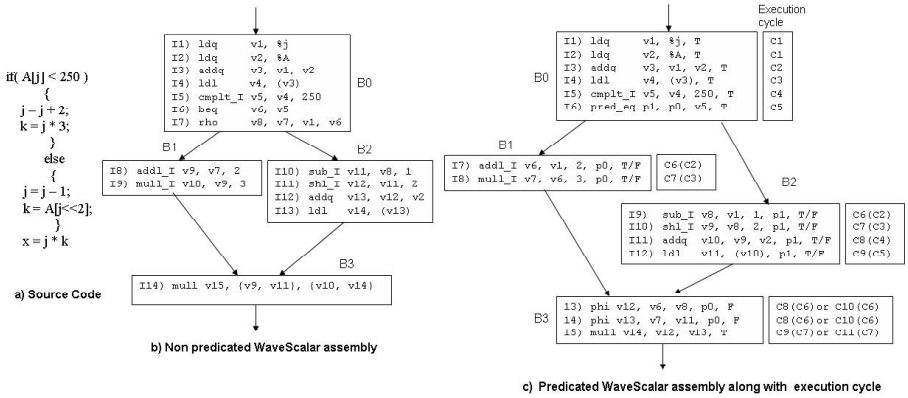

Fig. 5. Example of if-then-else predication/speculation. (a) source code, (b) non-predicated WaveScalar assembly code segment, (c) assembly code segment after predication along with default setting for speculative predicate. General format for the predicated WaveScalar assembly instruction is 'opcode destination(s), source(s), final predicate, speculative predicate'. Note that since basic-block B0 is a straight line code, its speculative predicate is set to T, eliminating the need for final predicate. In B1 and B2, depending on power-performance trade-off, amount of speculation can be adjusted from any where between full speculation to no speculation by setting the default value for speculative predicate appropriately.

any non-speculative instruction that consumes data from at least one other non-speculative instructions that is guarded by the same final predicate, can be immediately converted into a speculative instruction, since it cannot possibly execute unless the producer of that operand has already executed – so, by taking advantage of such transitivity within a basic-block, no final predicate needs to be explicitly sent to these 'dependent instructions', as indicated by placing the bottom symbol ($\perp$) in the corresponding field in Figure 4(b).

Implementation details on the WaveScalar architectures will be given in Section 4, and the impact of the above optimizations will be experimentally quantified for representative benchmarks, in Section 5.

4 Implementation on WaveScalar

This section will explain our implementation of elastic-block, and related optimizations, on a concrete representative of the state-of-the-art in tiled dataflow architectures, WaveScalar.

4.1 ISA Extensions

Figure 5(a) and 5(b) shows a simple if-then-else construct and its corresponding non-predicated WaveScalar assembly code. Figrue 5(c) shows our modified predicated WaveScalar assembly code. Changes are explained as below:

Adding speculative and final predicates: In the modified code, the execution of each instruction is guarded by two additional 1-bit operands – *final* predicate and the *speculative* predicate. Final predicate receives its value from *predicate-define* instruction (I6 in Figure 5(c)), which is also part of our ISA extension. Unlike final predicate, the value of speculative predicate is set by the compiler (either T or F) through program analysis, thus enabling the compiler to make power-performance trade-off in the code. Note that, basic-block B0 contains straight line code, its speculative predicate is set to T, eliminating the need for final predicate. Basic-blocks B1 and B2 have p0 and p1 respectively as their final predicate.

Addition of phi (ϕ) instruction: In Figure 5(c), instructions from basic-blocks B1 and B2 can execute speculatively (if speculative predicate is set to T). This requires addition of ϕ instructions (I13, I14 in Figure 5(c)) in the merge block B3. ϕ instruction takes two input values and a final predicate and, depending on the final predicate value, produce one of the inputs on its output. For correct execution, ϕ instruction can't execute speculatively and should wait for final predicate to arrive.

Removing rho (ρ) instruction: Figure 5(b) shows non-predicated, non-speculative WaveScalar assembly code. Instructions are executed only from the *'taken'* path. Instructions from *'not-taken'* path are prevented from execution by blocking their input operands using rho (ρ) instruction. The *rho (ρ)* instruction (I7 in Figure 5(b)), is a conditional split instruction. The ρ instruction takes an input value and a boolean output selector. It directs the input to one of two possible outputs depending on the selector value, effectively steering data values to the instructions in either basic-block B1 or B2. Speculative execution, however, allows execution from both basic-blocks i.e. B1 and B2. This is achieved by removing the ρ instruction and directly connecting its input operand with the input operands of its destination instructions.

4.2 Microarchitecture Support for Predicated and Speculative Execution

This section will explain microarchitecture modifications to the PE pipeline stages in order to support predication, and speculation.

Processing Element Modifications. Changes were made in the first two stages, namely INPUT and MATCH, of the PE pipeline described in Section 2.

1. Modified INPUT stage: Accepts input operands arriving from other PEs and from itself with the following additional logic: If the arriving operand is a '*final predicate*' operand with a FALSE value, the corresponding instruction is squashed by invalidating its entry in the matching table. However, this can lead to two special situations. Firstly, late arriving operands of an already squashed instruction will get a permanent entry in the matching table. Secondly, consumers of an squashed instruction keep waiting for the operand to arrive. To address the first situation each instruction has its 'current valid wave-number'

Table 1. Evaluating readiness of an instruction

Data operand	Final predicate	Spec. predicate	Action Taken
?	?	*	wait for data to arrive
?	TRUE	*	wait for data to arrive
?	FALSE	*	squash the instruction
data available	?	FALSE	wait for final predicate to arrive
data available	?	TRUE	execute instruction speculatively
data available	TRUE	FALSE	execute instructions normally
data available	FALSE	FALSE	squash the instruction

? = has not arrived, * = don't care

stored in the instruction cache. When an instruction is dispatched to the ready queue or squashed (if its final predicate is FALSE), its wave-number is stored as the 'current valid wave-number'. If the wave-number of an arriving operand is less or equal to 'current valid wave-number', it is not entered in the matching table. Second situation actually can never arise. If all the consumers of an squashed instruction and the squashed instruction itself are in the same basic-block, say B, they all will receive the same final predicate and eventually will be squashed. If however, consumers of an squashed instruction are in the merge block, they will receive their operands from the basic-block whose final predicate evaluates to TRUE i.e. sibling of B.

2. Modified MATCH stage: With non-speculative execution, an instruction only becomes ready to execute when all its operands have arrived. However, speculative execution requires modifying the logic that determines the readiness of an instruction. Table 1 lists all possible cases and the corresponding action taken.

4.3 Power-Performance Trade-Off Using Compiler Analysis

Traditionally the focus of compiler optimization has been on improving performance (see for example [14] and the references therein). However, performance alone is not sufficient to measure the effectiveness of machines. Other metrics such as energy efficiency and power dissipation are equally important. Unfortunately, there has been little effort to analyze the role of compiler in achieving high energy efficiency.

Elastic-block enables the compiler to make power-performance trade-offs in the code. Compared to 'hyperblock' [12], 'elastic-block' is capable of achieving more power-performance trade-off points. During hyperblock formation a basic-block is either fully included or fully excluded. With elastic-block, the compiler can *selectively* and *individually* define which instructions in the basic-block should be speculatively executed, and which should not. During the elastic-block formation, compiler profiles the execution frequeucy of individual basic-blocks, and partitions the instructions into 'levels' based on their dependence depth. Instructions that receive their operands from outside the elastic-block are at level-1. Instructions dependent on level-1 instructions are at level-2 and so on.

Table 2. Power-Performance Trade-off Points obtained using Elastic Block

Amount of Speculation	Cycles Taken	Instructions Executed
No speculation	9*60 + 11*40 = **980**	6*100 + 2*60 + 4*40 + 3*100 = **1180**
Only B1	7*60 + 11*40 = **860**	6*100 + 2*100 + 4*40 + 3*100 = **1260**
B1 + B2	7*60 + 7*40 = **700**	6*100 + 2*100 + 4*100 + 3*100 = **1500**
B1+ {I9} in B2	7*60 + 10*40 = **820**	6*100 + 2*100 + 100 + 3*40 + 3*100 = **1320**
B1 + {I9, I10} in B2	7*60 + 9*40 = **780**	6*100 + 2*100 + 2*100 + 2*40 + 3*100 = **1380**
B1 + {I9, I10, I11} in B2	7*60 + 8*40 = **740**	6*100 + 2*100 + 3*100 + 40 + 3*100 = **1440**

Note: Hyper blocks can only achieve first three points

Based on the execution frequency of the basic-block and the dependence level of the instruction, compiler decides whether to set the speculative predicate of that instruciton to TRUE or FALSE for a given power-performace point. Consider the same example shown in Figure 5(a) and its corresponding assembly in Figure 5(c). Assume that this code executes 100 times with basic-block B1 executing 60 times and basic-block B2 executing 40 times. Basic block B0 contains straight line code, its speculative predicate is set to TRUE, eliminating the need for final predicate. B1 and B2 are guarded by final predicate p0 and p1 respectively. Speculative predicate for instructions in B1 and B2 can be assigned either TRUE or FALSE. For this example, each instruciton is assumed to be single cycle. Instruction's execution cycle (both when executed speculatively and non-speculatively) is also shown in Figure 5(c). Table 2 shows various operating points that are achieved by selective speculation of instructions in a elastic-block structure. First row in Table 2 shows a low performance but most power efficient operating point where no instruction is speculatively executed. Third row shows the best performance but least power efficient point where both basic-blocks B1 and B2 are fully speculated. Rest of the rows shows several operating points between these two extremes.

5 Performance Analysis

5.1 Experimental Methodology

Elastic-block technique and related optimizations are implemented in WaveScalar compiler/binary-translator, and necessary microarchitectural support is provided in the WaveScalar simulator. Speculative execution is supported on all instructions but stores, phi and predicate-define instructions. Benchmarks from SPEC 2000, MediaBench, EEMBC benchmark suites are used for the evaluation. Our experimental setup was designed to evaluate the effectiveness and flexibility of elastic-blocks at exploiting the power-performance trade-off. Several configurations, each with varying depth of speculation, are computed by the compiler, by choosing different values for speculative predicate. For each configuration, the benchmarks are run till completion. IPC is not a meaningful metric in our case, since higher IPC does not necessarily mean higher performance because of 'unnecessary instructions' executed due to predication/speculation. So, we will measure: (1) number of cycles

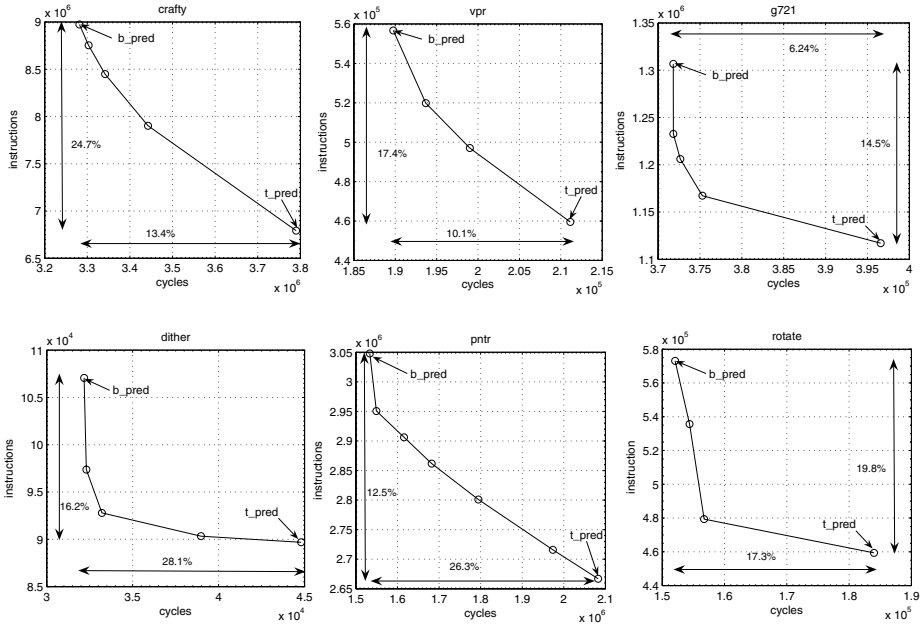

Fig. 6. Power-Performance trade-off points between top predication (t_pred) and bottom predication (b_pred)

required to execute the application, which roughly relates to performance; and (2) corresponding total number of executed instructions; and (3) number of operand and predicate messages exchanged during that execution, which along with the instructions executed corresponds to the power consumed during execution.

5.2 Results

Adding predication and speculation improved WaveScalar average performance by 16.9% (for bottom predication), compared to no speculation (top predication), see Figure 6. However, this increase in performance comes at an steep cost of 17.51% extra instructions executed, and 14.35% additional messages sent, which is unwarrented for high performance, low power computing. Figure 6 and 7 shows that almost similar performance gain, 15.96%, can be achieved with an average 11.56% increase in instructions and 7.95% increase in operand messages, a reduction of 5.95% and 6.4% respectively. Another high performance point with 13.93% performance gain, can be achieved with an average 7.74% increase in instructions and 3.1% increase in operand messages, a reduction of 9.77% and 11.25% respectively. Operand messages, shown in Figure 7, scales with the number of instructions executed. Using the optimization explained earlier in Figure 4(b), predicate messages are independent of the number of instructions executed, see Figure 7.

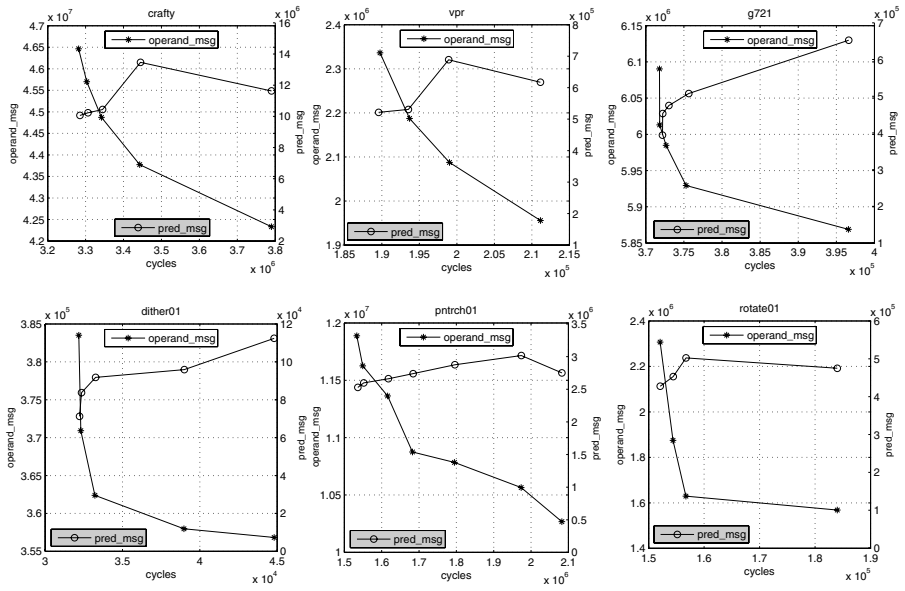

Fig. 7. Illustrating operand and predicate messages for the corresponding power-performance trade-off points shown in Figure 6

6 Related Work

DataFlow Predication for EDGE Architectures
Smith et al. has proposed dataflow predication for EDGE architectures [15]. Smith et al. uses top or bottom predication, in which either the first or the last instruction of a dependence chain is predicated. Top predication delivers low performance and low energy computation, as instructions are not executed speculatively, while bottom predication results in high performance and high energy computation as all the instructions in the dependence chain, except the bottom instruction, are fired speculatively. The focus of this work is to combine these two extremes, allowing the compiler to decide the optimal depth of speculation, for a given power-performance point. Second, in EDGE architecture, each instruction has a two-bit predicate field that specifies whether that instruction is predicated on a TRUE predicate, a FALSE predicate, or unpredicated. However, in our proposed work the speculative predicate is another operand, initially set by the compiler, but later can be modified through messages, thus allowing run-time adaptation, which is a subject of future work.

Predication for Superscalar Architectures
Mahlke et al. proposed a compiler structure, hyperblock, that groups together most frequently executed basic-blocks from different control paths, allowing effective scheduling for these basic-blocks [12]. In case of an hard-to-predict branch (say 60/40), basic-blocks from *'both'* control-flow paths are included in the

hyperblock , and all instructions in these basic-blocks are executed all the time. In our proposed *'elastic-block'* structure, basic-blocks from *'both'* control-flow paths will be included, but speculative execution of instructions in these basic-blocks will be proportional to their execution frequency. Kim et al. combined the use of conditional branches, for easy-to-predict branches, with predicated execution, for hard-to-predict branches [16]. Their motivation for not converting every conditional branch into predicated code is twofold: First, the processor needs to fetch useless instruction, thus wasting the fetch bandwidth. Second, compared to branch prediction in which instructions are executed before the branch is resolved, predicated instructions add extra delay, as they have to wait for the predicate value to be ready. In our proposed work, we transformed all branches to predicated code, as we don't have the aforementioned overheads: First, instructions are stored on the execution grid, once they are fetched from the memory, and second, predicated instructions can execute speculatively before the predicate value is ready (by setting p_s = TRUE).

7 Conclusion and Future Work

A novel *power-aware* ILP extraction technique, that combines predication with speculation, is introduced for tiled dataflow architectures. Each instruction in this flexible structure, denoted the *elastic-block*, is guarded explicitly by two predicate operands: the final predicate, and the speculative predicate. By assigning the default value of speculative predicate to TRUE, compiler can *selectively* and *individually* enable the speculation of *only* those instructions that may actually payoff in terms of performance improvement, thus avoiding wasteful energy spending. This is in contrast to the existing techniques for predicated execution, namely top predication and bottom predication, in which either the head or the tail of the dependence chain is predicated. Results showed that by merging top and bottom predication, and allowing the compiler to determine the depth of speculation, performance close to traditional predication can be delivered while improving the energy efficiency. The key advantage of elastic-block structure will be its inherent potential for run-time adaptivity, and is a subject of future work.

References

1. Burger, D., Keckler, S.W., McKinley, K.S., Dahlin, M., John, L.K., Lin, C., Moore, C.R., Burrill, J., McDonald, R.G., Yoder, W.: The TRIPS Team: Scaling to the End of Silicon with EDGE Architectures. Computer 37(7), 44–55 (2004)
2. Swanson, S., Michelson, K., Schwerin, A., Oskin, M.: WaveScalar. In: MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, p. 291. IEEE Computer Society, Los Alamitos (2003)
3. Waingold, E., Taylor, M., Sarkar, V., Lee, V., Lee, W., Kim, J., Frank, M., Finch, P., Devabhaktumi, S., Barua, R., Babb, J., Amarsinghe, S., Agarwal, A.: Baring It All to Software: The Raw Machine. Technical report, Cambridge, MA, USA (1997)

4. Hrishikesh, M.S., Keckler, S.W., Burger, D., Agarwal, V.: Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. ISCA 00, 248 (2000)
5. Hunt, W.: Introduction: Special Issue on Microprocessor Verification. Formal Methods in System Design, 135–137 (2002)
6. Mai, K., Paaske, T., Jayasena, N., Ho, R., Dally, W.J., Horowitz, M.: Smart Memories: A Modular Reconfigurable Architecture. In: ISCA 2000: Proceedings of the 27th Annual International Symposium on Computer Architecture, pp. 161–171. ACM Press, New York (2000)
7. Dennis, J.B., Misunas, D.P.: A Preliminary Architecture For a Basic Data-flow Processor. SIGARCH Comput. Archit. News 3(4), 126–132 (1974)
8. Papadopoulos, G.M., Culler, D.E.: Monsoon: An Explicit Token-Store Architecture. In: ISCA 1998: 25 years of the International Symposia on Computer Architecture (selected papers), pp. 398–407. ACM Press, New York (1998)
9. Putnam, A., Swanson, S., Mercaldi, M., Petersen, K.M.A., Schwerin, A., Oskin, M., Eggers, S.: The Microarchitecture of a Pipelined WaveScalar Processor: An RTL-based Study. Technical report, Washington, DC, USA (2004)
10. Allen, J.R., Kennedy, K., Porterfield, C., Warren, J.: Conversion of Control Dependence to Data Dependence. In: POPL 1983: Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 177–189. ACM Press, New York (1983)
11. Park, J.C.H., Schlansker, M.S.: On Predicated Execution. Technical report, Palo Alto, CA, USA (May 1991)
12. Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E., Bringmann, R.A.: Effective Compiler Support for Predicated Execution Using the Hyperblock. In: 25th Annual International Symposium on Microarchitecture (1992)
13. Chang, P.P., Mahlke, S.A., Chen, W.Y., Warter, N.J., Hwu, W.m.W.: IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors. In: ISCA 1991: Proceedings of the 18th Annual International Symposium on Computer Architecture, pp. 266–275. ACM Press, New York (1991)
14. Wolfe, M.: High Performance Compilers for Parallel Computing. Pearson Education POD (1995)
15. Smith, A., Nagarajan, R., Sankaralingam, K., McDonald, R., Burger, D., Keckler, S.W., McKinley, K.S.: Dataflow Predication. In: MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, pp. 89–102. IEEE Computer Society, Los Alamitos (2006)
16. Kim, H., Mutlu, O., Stark, J., Patt, Y.N.: Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution. In: MICRO 38: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, pp. 43–54. IEEE Computer Society, Los Alamitos (2005)