Effective Optimistic-Checker Tandem Core Design Through Architectural Pruning

Francisco J. Mesa-Martínez Jose Renau
Dept. of Computer Engineering, University of California Santa Cruz
http://masc.cse.ucsc.edu

Abstract

Design complexity is rapidly becoming a limiting factor in the design of modern, high-performance microprocessors. This paper introduces an optimization technique to improve the efficiency of complex processors. Using a new metric (μUtilization), the designer can identify infrequently-used functionality which contributes little to performance and then systematically "prune" it from the design. For cases in which architectural pruning may affect design correctness, previously proposed techniques can be applied to guarantee forward progress.

To explore the benefits of architectural pruning, we study a candidate Optimistic-Checker Tandem architecture, which combines a complex Alpha EV6-like out-of-order Optimistic core, with some of the underutilized functionality pruned from its design, with a non-pruned EV5-like in-order Checker core. Our results show that by removing 3% of infrequently used functionality from the optimistic core an increase in frequency of 25% can be realized. Taking into account the replay overhead triggered by the removed functionality, the Tandem system is still able to achieve a 12% overall speedup.

1 Introduction

Design cost is a limiting factor in the design of modern high performance architectures. The inherent complexity found in modern processors makes the optimization for area, power, and frequency a challenging task. It may be argued that excessive design complexity has terrible consequences as innovation may be hampered. It is therefore critical that designers are given new methods to meet aggressive design targets in the face of growing complexity.

This work proposes the systematic use of *Architectural Pruning* –the selective removal of infrequently used functionality– as a design optimization methodology. This new methodology and its associated metric, named *μUtilization*, allows designers to rank the Hardware Description Language (HDL) statements in a processor design based on their activity. Efficiency metrics that build on *μUtilization* correlate activity with contribution to performance or some other criteria. A set of heuristics determine removal since not every segregated element is equally valuable. Infrequently-used statements that contribute little to performance are then removed, and the rest of the design is re-optimized around them. Removing functionality may lead to the introduction of faults into a design. To handle possible faults and guarantee correct execution and forward progress in the event of the processor transitioning to a state, where removed functionality would otherwise be executed, any one of several previously proposed techniques can be used [2, 11, 19, 20, 21, 29]. In this work we use a simple in-order checker core.

Architectural pruning is motivated by the observation that significant hardware functionality is often required to handle extremely rare events. However, if forward progress and correctness can be ensured despite missing functionality, then the main design can be optimized.

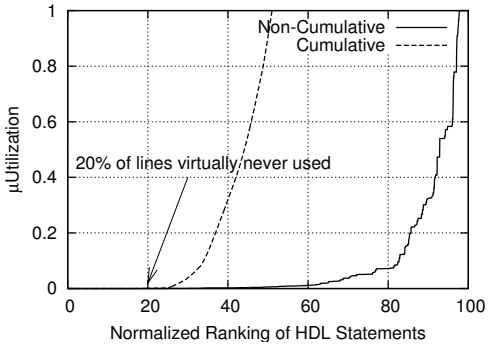

Figure 1. RiSC-16 [13] *μUtilization*.

To have some insights on the optimization opportunities, Figure 1 shows the *μUtilization* for a simple RiSC-16 [13] processor after executing a benchmark to solve Laplace equations. RiSC-16 is a simple, single-issue out-of-order processor. The non-cumulative graph in Figure 1 shows the normalized rankings of *μUtilization* for the HDL statements in the codebase. This is a listing of *μUtilization* data values in ascending order, normalizing to 100 statements. Using this plot, we can easily see that fully 80% of HDL statements (x-axis) are used less than 20% of the time (y-axis).

The cumulative graph in Figure 1 represents the integration of the *μUtilization* up to a particular statement. We

can easily see that 20% of the statements are virtually never used. Therefore, roughly 20% of the codebase can be removed without sacrificing any significant performance due to the replays needed to guarantee execution correctness and forward progress. This shows that for even simple hardware designs, significant portions of the codebase are dedicated to extremely rare events. By not having to explicitly handle these rare events, designers are afforded dramatic new opportunities for optimization. Once removed, previously complex structures are pruned-down, freeing valuable real-estate and reducing pressure on critical paths.

To demonstrate the effectiveness of architectural pruning as an optimization technique, we evaluate a *Tandem* processor organization combining a pruned out-of-order *Optimistic* core, to explore data and control behavior, with an in-order *Checker* core that guarantees forward progress. Under this *Optimistic Execution* approach, the Checker core combines possible future memory prefetching and prediction updates with verified past branch behavior to hide some of its associated latencies. The results obtained show that it is indeed possible to "prune" under performing structures from a complex candidate design. The pruned out-of-order core cycles 1.25 times faster. Despite the increased rate of replays needed to guarantee forward progress, the resulting system still exhibits a 12% performance increase with respect to the original complex processor [28] that serves as the basis for the pruned core in the *Tandem* configuration.

This paper makes several contributions; It proposes for the first time an architectural pruning methodology as a possible processor optimization technique. It quantitatively evaluates the effect of pruning on an HDL design for an out-of-order core –specifically, the Illinois Verilog Model (IVM) [28]. It also explores an architectural organization in which a pruned complex core is used to scout the memory and branch behavior for a simpler verified core.

The rest of the paper is organized as follows. Section 2 describes a systematic framework for architectural pruning. We begin by describing a new metric, μ*Utilization*, that measures the relative activity of HDL statements. We then describe how μ*Utilization* can be augmented to correlate activity with contribution to performance. Finally, we describe the architectural pruning process. Section 3 presents a *Tandem* architecture in which a pruned core is allowed to explore memory and branch behavior in an Optimistic fashion; Section 4 evaluates the setup for our evaluation; Section 5 displays and analyzes some of our results from simulation and implementation of the proposed *Tandem* architecture; Section 6 covers related work; and Section 7 concludes.

2 Architectural Pruning Methodology

The proposed optimization methodology requires an effective mechanism for the identification of complex structures and their relative utilization. To address this, we introduce μ*Utilization*, a new metric that couples complexity with utilization. For the remainder of this paper, we assume that processor designs can be represented as a collection of HDL statements. These statements provide a first-order approximation of design complexity [3]. Furthermore, the relative importance of a given statement can be approximated by how much that statement is exercised during normal processor operation.

2.1 μ*Utilization* Metric Definition

μ*Utilization* extends traditional coverage metrics [31] by providing information about the exercise rate for a given HDL statement. Low μ*Utilization* for a statement suggest that it can be potentially removed. Nevertheless, some structures may have a low μ*Utilization* but can not be removed because they are required to guarantee forward progress. For example most of the reset logic, while used very infrequently, is critical for the correctness of processor operation. To provide for this improved identification strategy, μ*Utilization* can describe either the utilization frequency of a given statement or a value of 1. The latter value is used on statements that are determined by the designer to be essential for forward progress or processor operation (Equation 1). The overall μ*Utilization* for a given structure is equal to the arithmetic mean of the μ*Utilization* of its statements (Equation 2).

$$\mu Utilization = \begin{cases} \text{Required :} & 1 \\ \text{Not Required :} & \frac{\text{\# Cycles statement is used}}{\text{\# Total cycles}} \end{cases} \quad (1)$$

$$\mu Utilization_{structure} = \frac{1}{n}\sum_{i=0}^{n-1} \mu Utilization_i \quad (2)$$

2.2 Structural Efficiency

While the contribution of a particular HDL statement to overall performance is, in the general case, highly correlated with activity rate, this is not always the case. Structures that are seldom used may be critical for the correctness of the design, while certain structures that contribute little to performance may be very active. To decide which structures can be pruned, it is necessary to determine an optimization target and its associated evaluation function. Traditional optimization targets are overall performance, area, $E * D$, and $E * D^2$. The coupling of the μ*Utilization* metric with and optimization targets defines a new metric, *Structural Efficiency*.

Each HDL statement has a specific contribution to any of the many design metrics, such as Architectural Vulnerability Factor (AVF) [17], Δenergy, Δpower, Δarea, Δfrequency, and ΔIPC. Therefore, in order to compute

the *Structural Efficiency* for a pruned design it is necessary to know the individual contribution of the excised statements.

$$\textit{Structural Efficiency}_D = \text{Overall Speedup} =$$

$$= \frac{\text{IPC}_p}{\text{IPC}_o} \times \frac{\text{F}_p}{\text{F}_o} \times \frac{n}{n + \sum_{i=0}^{r-1} \mu Utilization_i \times RP_i \times AVF_i} \quad (3)$$

For example, Equation 3 represents the overall speedup due to pruning, which is equivalent to the *Structural Efficiency* when performance is optimization target. This equation assumes that r statements are selected for removal out of a codebase of size n. The first half of the equation is composed by the ratios of the original (o) and post-pruning (p) metrics. These ratios capture the overall improvement/degradation induced by the pruning of r statements. IPC_p assumes no overhead due pruning-induced faults, but does account for other forms of IPC degradations due to pruning, e.g., removing a section of the branch predictor. F_p captures the frequency for the design after pruning. The second component in the equation models the penalties associated with the faults induced by pruning.

Each statement has a baseline value of 1 as its overhead. Statements that induce a fault, after they are pruned from the design, add the replay penalty associated with their fault (RP_i). This overhead can be further refined by taking into account how frequently the output for the excised statement is used. This is captured by the architectural vulnerability factor [17, 28](AVF_i). Finally, the $\mu Utilization_i \times RP_i \times AVF_i$ product provides a first-order estimation of the performance impact induced by the pruning of a specific statement.

2.3 Architectural Pruning

Using architectural pruning, the designer manually removes isolated HDL statements. However, the actual effect of excising a given statement may be more dramatic. The most clear effect is the savings in area and power associated with the logic budget dedicated to implement the excised functionality. The removal of specific statements may also lead to the reduction of critical paths along the design. Furthermore, other structures may be pruned in a *transitive* fashion through optimizations performed automatically by the synthesis tools. If dependent functionality is no longer relevant once a given statement is removed, this functionality is automatically excised as well.

Transitive pruning induces the removal or resizing of an object that was not originally singled out for pruning, but that became resized or eliminated automatically due to the removal of a related statement.

As an example, Figure 2-(a) presents the mapping of a Verilog module into its equivalent gate representation.

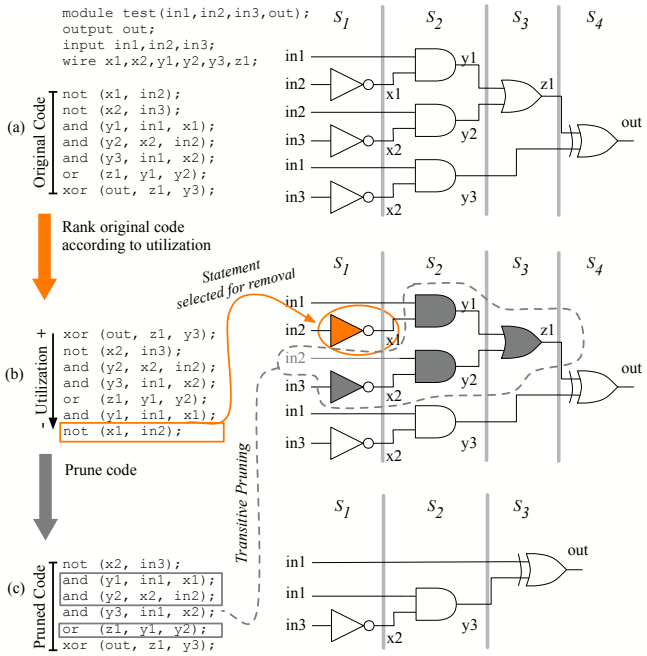

Figure 2. Architectural pruning example with the original structures (a), the $\mu Utilization$ ranked code (b), and the resulting pruned design (c).

$\mu Utilization$ metrics are obtained, and based on their *Structural Efficiency* the designer can excise a statement with little utilization. Assuming ground behavior for active high and floating logic, the elimination of certain HDL statements affects the original design as shown in Figure 2-(b). The elimination of the top *NOT* gate is due to a lack of activity for the *in2* input, this means that the gates producing *y1, y2* and *z1* are superfluous and can be safely eliminated through optimizations by the design tool. The resulting datapath in Figure 2-(c) not only has fewer gates due to direct pruning, but the number of logic propagation stages is also reduced by the elimination of the *OR* gate in level S_3

The ripple effect from the removal of a single statement may therefore lead to the removal of statements throughout the design. This transitive effect may also lead to the alteration of the $\mu Utilization$ for each of the affected statements. A detailed evaluation of these ideas is left to future work.

Architectural pruning as an optimization is used in the following fashion; After gathering $\mu Utilization$ metrics for the source design and deciding on an optimization target, the designer can then rank and isolate sets of possible statements or structures to be pruned. The designer decides which statements to remove by examining the effect their removal has on the *Structural Efficiency* defined for the current optimization target.

3 A Pruning-Based Tandem Architecture

After pruning, the correctness of a design can not be guaranteed and thus it is assumed to be subject to hard errors. In order to illustrate the benefits derived from architectural pruning, we evaluate a *Tandem* architecture that combines a complex out-of-order core, architecturally-pruned with the goal of increasing its frequency, with a simpler in-order processor used to guarantee forward progress. Through the reminder of this paper the pruned core is labeled as the *Optimistic* core, with the non-pruned core which guarantees correctness as the *Checker* core.

Previous tandem designs like DIVA [2] and Slipstream [19] send complete pre-executed instructions to the Checker core which then performs parallel verification on those instructions and commits. Architectural pruning can certainly be applied to the Optimistic cores in such designs and is left for future work. The *Tandem* architecture we consider here is more similar to Paceline [12] and SRTR [27]. The Optimistic and Checker cores operate essentially in parallel with only the Checker core committing instruction results to architected state. The Optimistic and Checker cores compare the results –actually *signatures* of the results– of their committed instruction streams. On a mismatch, the checker core flushes the optimistic core, reloads it with correct register state, and restarts it. All exceptions and interrupts are handled in-order by the Checker's retirement stage

If both the Optimistic and Checker cores operate in lockstep, performance is limited by the retirement bandwidth of the Checker and the replay overhead. To address this, the Optimistic core is decoupled from the Checker– its "committed" instructions stored in a buffer – and allowed to run ahead. This organization allows the Optimistic core to accelerate the Checker by acting as an aggressive and accurate prefetch engine. It also allows the Optimistic core to improve the branch prediction accuracy for the Checker, using a new technique we describe in Section 3.2. Both prefetching and improved branch prediction are enhanced by the architectural pruning of the optimistic core which improves frequency.

Decoupled *Tandem* operation is illustrated in Figure 3. Under Optimistic Execution, the Optimistic core is allowed to run ahead by committing its Optimistic results to be verified by the Checker into a common buffer. The Checker is the only core allowed to commit results in the *Tandem* and it guarantees forward execution whenever an Optimistic result fails verification. To recover from a fault the Checker flushes both the shared buffer and the Optimistic pipeline. The Optimistic core is restarted after copying the architectural register file from the Checker, its rename table is also set accordingly. Copying this state is part of the replay overhead.

Figure 4-(a) presents a typical fully verified out-of-order core, with its major operational blocks; L1/L2 Cache, load store queue (LD-ST queue), out-of-order scheduler (Scheduler), register file (RF), branch prediction unit (BPU/BTB), execution units (EX), and reorder buffer (ROB). In contrast, Figure 4-(b) presents an *Optimistic-Checker Tandem* architecture with a buffer used to pass the Optimistic results to be verified by the Checker. The operation of this *Signature Buffer* is described in full detail in Section 3.1. Under the proposed *Tandem* configuration in this paper, the overall cache size does not change. Each core uses half the L1 size found in the original core, with the L2 remaining fixed in size and shared among both cores in the *Tandem*.

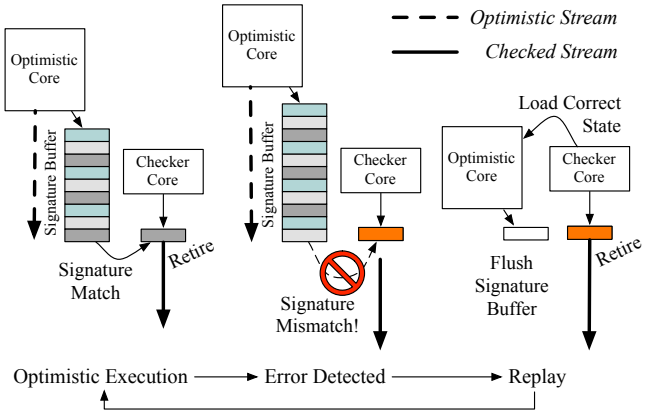

Figure 3. Decoupled *Tandem* operation.

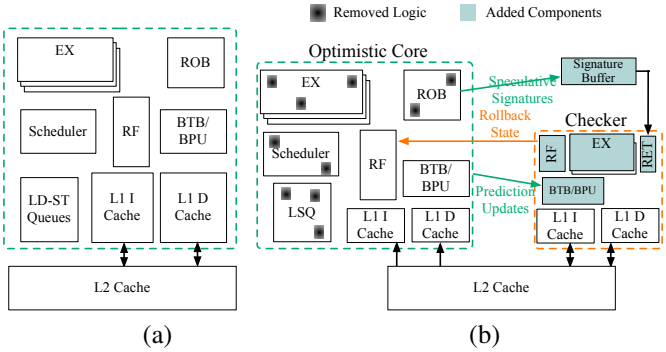

Figure 4. Traditional out-of-order processor core (a), and *Tandem* architecture with an *Optimistic* out-of-order core and in-order *Checker* (b).

3.1 Signature Buffer

Unlike previous approaches like Dual-Core Execution [32], Slipstream [19], and DIVA [2], our *Tandem* architecture does not pass full instruction state (PC, instruction bits, input/output values, reference address, control outcome, etc.) between cores. Instead the Optimistic core generates a *signature* for the state of the instruction to be verified. The *signature* for an instruction is a hash value that summarizes its state update [24].

Signatures capture state by taking into account the values from instructions retiring in the reorder buffer (ROB)

and load-store queue (LD-ST) of the Optimistic core. Only the result of an instruction needs to be hashed to represent its state. The hash is computed using a linear block code such as CRC-16, for our purposes 4 and 5-bit wide signatures provide good hash diversity. Signatures are computed with a single-instruction granularity, it is possible however to lump multiple instructions per signature since instructions are retired in-order. Information such as the program counter (PC) for the instruction is not required because the Optimistic core inserts signatures in order in a FIFO fashion. The Checker simply removes the oldest signature in the buffer and compares it with the most recent instruction it is considering for retirement. The depth of the signature buffer determines the maximum number of instructions that can be executed ahead of the stream for the Checker.

Since the signature hash is just a few bits wide, an incorrectly executed instruction from the Optimistic core may not trigger a replay. This is not a problem because the Optimistic core can execute and optimistically retire incorrect instructions. Eventually the Optimistic core will have a different hash value than the value generated by the Checker. Having the Optimistic core produce incorrect results affects only the performance of the *Tandem* processor, not its correctness.

3.2 Optimistic Branch Prediction

Optimistic *Tandem* architectures are not constrained to be just data prefetching tools. By having access to future execution information, control behavior for the instruction stream can also be explored.

We introduce an *Optimistic branch prediction* mechanism, under which both cores implement separate local hybrid branch predictors. The Optimistic core maintains its own predictor and behaves like a traditional branch predictor. However updates for the predictor tables found in the Checker come from the pruned core, only the Global History Register (GHR) in the Checker is updated by its own branch executions.

By updating the Checker branch prediction tables with the outcome of the branch predictions from the Optimistic core, we are providing near future execution information with the trend observed for "correct" past execution captured by the Checker GHR. As a result, the hybrid predictor found in the Checker core can be implemented using smaller history tables than traditional hybrid predictors without future and "correct" updates.

This approach is still considered a prediction for two reasons; first, the history tables have limited space and are subject to aliasing issues, and second the speculative nature of the results by the Optimistic core induces a certain degree of uncertainty to the future branch results recorded by the local history table.

This last point raises an important consideration regarding deep signature buffers. The more signatures in the buffer the longer it can potentially take the Checker to detect an error. This increases the probability that the Optimistic core generates erroneous branch results which are directly stored in the local table for the Checker, thus increasing the overall pollution in its predictor.

An alternative design could remove the branch predictor from the Checker core and just use branch outcomes from the Optimistic core. For such design, the pruned core needs to pass the predictions in-order to the Checker core. Therefore, it can pass the prediction at retire or speculatively at fetch. However, unless the Optimistic core is far enough ahead, passing its prediction at retirement may not be timely enough. Alternatively, passing the prediction speculatively leads to a worse prediction because of its speculative nature. As a result, we opted for the proposed Optimistic branch predictor model.

4 Evaluation Setup

To illustrate the principles behind our pruning methodology, we use an Alpha EV6-like processor to measure the μ*Utilization* of its structures and prune a few underutilized instances. The EV6-like processor is based on the Illinois Verilog Model (IVM) [28]. IVM implements a subset of the Alpha EV6 architecture, it was designed originally at the University of Illinois for fault-tolerance research. It contains over 30K HDL statements.

For the rest of this evaluation we refer to the original IVM EV6-like core as *Baseline* and the pruned IVM as the Optimistic core. We also consider two possible configurations for the Checker processor based on their superscalar width: 2-issue and 4-issue. The main simulation parameters for each core are captured in Table 1. Under the proposed *Tandem* configuration in this paper, the overall cache size does not change. Thus the Optimistic and Checker cores use half the L1 size found in the original *Baseline*. Other structures for the Optimistic and Checker cores such as the predictor tables, BTB, etc are also reduced in order to constrain storage requirements for the resulting *Tandem* with respect to *Baseline*. Finally, to copy the architectural state from the Checker core to the Optimistic core, we add a 40 cycle penalty for each replay (31 int regs + state overhead).

The *Baseline* core cycles at 3GHz, the Optimistic core frequency is 3.75GHz. The 25% frequency increase is equivalent to the frequency improvement observed on the pruned IVM (Table 2). Since the Checker core is a simple in-order processor, we assume that it can cycle as fast as the complex out-of-order Optimistic core. This seems a reasonable assumption backed by the fact that the out-ot-order EV6 Alpha, similar to *Baseline*, reaches 500Mhz at 0.35 μm while the in-order EV5 operates at 667Mhz using the same process technology. Recently, IBM has replaced the 90nm 2.2GHz out-of-order Power5+ with a much faster 65nm 4.7GHz in-order Power6. Memory access time for Optimistic and Checker is increased by 25% with respect

Structure	Baseline	Optimistic	Checker
Local History Table	16384 entries	8192 entries	4096 entries
Meta Table	16384 entries	8192 entries	4096 entries
GHR width	16 bit + XOR	8 bit + XOR	8 bit + XOR
Return Addr. Stack	64 entries	32 entries	16 entries
BTB	1024 entries 4-way assoc	512 entries 4-way assoc	256 entries 2-way assoc
L1 Data	32 KB, 2-way 32 Byte line 3 cycle hit	16 KB, 2-way 32 Byte line 3 cycle hit	16 KB, 2-way 32 Byte line 3 cycle hit
L1 Inst	32 KB, 2-way 64 Byte line 3 cycle hit	16 KB, 2-way 64 Byte line 3 cycle hit	16 KB, 2-way 64 Byte line 3 cycle hit
Register File	128 entries	128 entries	32 entries
Decode	4-issue	4-issue	4 or 2-issue
Issue	16-entry	16-entry	N/A
Memory	2 ports 70ns	1 port 70ns	1 port 70ns
Out-of-Order	Yes	Yes	No
Retire	64-entry ROB	64-entry ROB	N/A
Frequency	3GHz	3.75GHz	3.75GHz

Table 1. *Baseline*, *Optimistic* and *Checker* cores configuration assuming the same frequency.

to *Baseline* in order to account for the increased processor-memory speed differential due to the faster frequency.

In order to gather statement coverage metrics, the HDL code is instrumented using a set of tools built on top of a Verilog parser. The HDL code is instrumented at the parsed Abstract Syntax Tree (AST) by inserting Verilog PLI calls for each basic block.

4.1 Synthesis

Synthesis is the process of converting an HDL functional description into a gate-level netlist in the case of a standard-cell ASIC. Synopsys Design Compiler 2007.03 [26] is used to synthesize IVM with a 90nm (25C typical) technology library. We include a 25% clock skew and use the "ultra compile" option with the maximum optimization level.

In this work, memory blocks are treated as black-boxes during synthesis to avoid being synthesized with simple flops. The IVM multiplier relies on ASIC libraries. Since the default library does not have a high performance 64bit multiplier (less than 100MHz), the multiplier is also treated as a black-box assuming that it can be cycled faster than 500MHz. The input load capacitance for all the black boxes is modeled as 24 FO4s.

4.2 Applications

μ*Utilization* statistics are gathered by executing several SPECInt applications (bzip2, gap, gcc, gzip, mcf, perl) on the simulated Verilog design. SPECFP is not evaluated because IVM lacks support for floating point operations. Some integer benchmarks are not executed because IVM also lacks some of the functionality required.

Modeling at the register transfer level (RTL) is a slow process, so to speedup simulation we use SimPoint 3.2 [22]. Since RTL is several orders of magnitude slower than traditional architectural simulators, we use simulation intervals of only 20K instructions [1] with 30 intervals per benchmark. To avoid the problems associated with such small simulation intervals, we run the architectural simulator for longer periods and the RTL state is "loaded" with the architectural simulator state (caches, branch predictor, etc.). As a result, the RTL simulation has a correctly trained state before each simulation point begins.

5 Evaluation

The evaluation of the potential offered by the pruning of processor structures is presented in three sections: Section 5.1 shows pruning and μ*Utilization* results for the IVM processor; Section 5.2 shows the performance impact of pruning the Tandem architecture; Finally, Section 5.3 shows the area and power overhead considerations for the Checker.

5.1 Pruning Results

Figure 5 shows the μ*Utilization* for the IVM processor. We observe that close to 20% of the code is barely used (μ*Utilization* < 0.01). The cumulative μ*Utilization* starts to curve around 5%. This means that as long as a verification mechanism guarantees forward progress, close to 5% of the code can be removed with little performance impact. To simplify the pruning process, we do not remove any of the reset logic. As a result, we conservatively prune only 3% of the code.

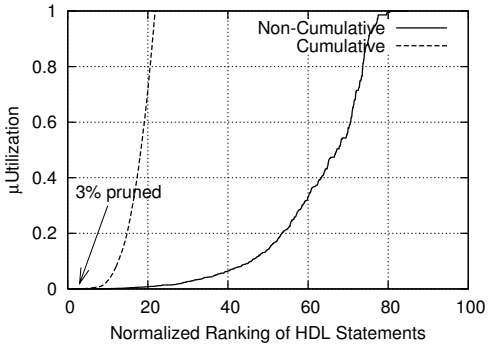

Figure 5. IVM μ*Utilization*.

As shown in Figure 2 (Section 2) removing some HDL statements has a ripple effect over the rest of the design. Once a piece of logic is pruned or removed, the synthesis tool (Synopsys) can further optimize other sections of the design. Unused functionality not pruned explicitly can still be removed transitively.

Table 2 shows the synthesis results. "Original" column has the synthesis results for the original HDL, "Pruned" has

[1] SimPoint recommends simulation intervals of 100M

the synthesis results when 3% of the code is excised. The second row reports the frequency (395MHz vs 500MHz). A 25% frequency improvement is achieved with just a 3% code pruning. This is a significant result that demonstrates the opportunity to prune processors for performance reasons. The third row shows the number of nets available in the design (25149 vs 25144). The small difference implies that the processor pruning performed does not have significant reduction in networks on a chip.

	Original	Pruned
% Pruned Code	0%	3%
Frequency (MHz)	395	500
# Nets	25149	25144
Combinational Area (mm^2)	2.15	1.81
No-Combinational Area (mm^2)	0.73	0.72
Total Area (mm^2)	2.89	2.50
Net Switch Power (mW)	1495.7	1486.6
Cell Internal Power (mW)	855.9	838.4
Total Dynamic Power (mW)	2351.7	2906.2
Leak Power (mW)	35.9	33.1

Table 2. IVM main synthesis results.

Rows four to six show area synthesis results with a breakdown between combinational and non-combinational area. The non-combinational area does not include the SRAM areas associated with the register file and caches. Those SRAM blocks should use a memory compiler or full custom macros and should not be affected by processor pruning. As expected, pruning has a much bigger impact on combinational area (15.9% reduction) than in non-combinational area reduction (1% reduction). This is because we do not target structure sizing, as a result the non-combinational logic remains relatively unchanged. Overall we observe a 13.5% reduction in total area for the pruned design.

Rows nine to eleven show the power reduction considering the 25% frequency improvement achieved when the IVM design is pruned. The result is a minor improvement in maximum cell switch power (2%) because the processor cycles faster [2]. The bigger reduction achieved in leakage power (35.9 vs 33.1 mW) is consistent with the area reduction. The 25% increase in frequency has a clear impact in the overall dynamic power which experiences a 23.5% increase for the pruned design.

IVM Pruning Insights: Up to now, we have provided a cumulative view of pruning, we now proceed to show further insights on the IVM pruned structures. A 3% code pruning has the following impact:

- We prune most of the logic that makes it possible to retire over 5 instructions per cycle.
- The ROB full detection signal is rarely asserted as the scheduler and other resources usually fill first. We remove this signal and associated logic.
- The LD and ST queues have to guarantee correctness but many situations are unlikely to happen. We also simplify the store set ids to detect dependences between load and stores. As a result, we prune several forwarding corner cases on the queues.
- The ST queue has the same size as the LD queue and thus is unlikely to fill. We remove the ST queue full signal and associated logic.
- In IVM, a conflict buffer handles write port conflicts from the shift and multiply units. Conflicts are very infrequent, so most of its logic is removed.
- We also remove most of the write port arbitration for several ALUs.
- Several infrequently used ALU operations like scaled longword add, special multiplications, and mask byte instructions are removed. They are handled only by the Checker.

IVM does not have a perfectly balanced pipeline. The scheduler is the most critical path, and to avoid bias towards it, this study does not prune it. Nevertheless, many of the inputs/outputs that interact with the scheduler in the same cycle are affected by pruning (ROB, LD-ST queues, port contention). Also in an indirect fashion, the smaller area results in shorter wires and better floorplaning. The pruned scheduler cycles over 25% faster than the unoptimized one. In order to achieve this frequency speedup, the second slowest pipeline stage (ROB/LD-ST queues interaction) is also optimized.

The insight gained in our evaluation points to the fact that the same pruning principles should also work for more balanced processors. The reason is that even balanced designs need logic for detecting infrequent situations (overflows, forwardings, port conflicts, etc.). Pruning optimizes these cases.

Even in the case that a critical pipeline stage can not be optimized, pruning still can be useful. The reason is that for the "faster pipeline stages", we can afford bigger structures and/or more energy efficient designs. For example, if the LD-ST queues can cycle faster after pruning, we can add entries and/or reduce the Vdd until we match its original cycle time. Obviously, this type of evaluation is outside the scope of this paper.

5.2 Performance Results

In order to study the performance implications of architectural pruning, we evaluate a *Tandem* architecture using the Optimistic and Checker cores described in Table 1. Our study focuses on two *Tandem* configurations, which use issue widths of 2 and 4, respectively.

[2]Synopsys reports power assuming a 50% activity rate for each input, power is lower when activity rate is used

Performance Impact of Pruning and Checker Width: For every structure pruned from IVM, the resulting design has potential for slowdowns due to the replay penalty induced every time a pruned structure is exercised. To evaluate the effect of replays, we execute a set of simulation points on top of the pruned HDL codebase for the IVM processor. In order to detect hard errors, the simulated Verilog is executed in parallel with an architectural IVM simulator executing the same program. Both simulators compare their results at retire, whenever a mismatch happens we add the offending instruction into an error trace. We generate a set of statistically significant simulation points as explained in Section 4.

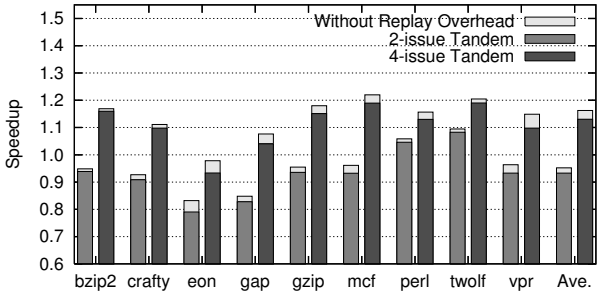

Figure 6. Speedup for *Tandem* configuration using 2 and 4-issue Checker with replay overhead breakdown. Results are normalized against *Baseline*.

Once the error traces have been generated, we proceed to re-execute every benchmark using a modified version of the *SimpleScalar* tools [5] that models *Tandem* operation. Every time an instruction in the error trace is encountered, the Optimistic pipeline is flushed and execution is rolled back to the current safe state for the Checker. A replay penalty is then added to the simulation statistics.

Figure 6 presents the simulation results for 2-issue and 4-issue *Tandem* using the Optimistic and Checker cores described in Table 1. Performance is normalized against *Baseline*. *Tandem* configurations using a 2-issue Checker suffer a slight 5.1% slowdown with respect to *Baseline*. This is due mostly to the limited retirement bandwidth for the narrow Checker. On the other hand, the wider 4-issue configuration displays an average speedup of roughly 14% over *Baseline*. After taking into account error penalties induced by pruning most benchmarks display a replay overhead of less than 1%, however vpr and eon suffer a replay overhead of 3% and 4% respectively. The wide *Tandem* configuration achieves an overall speedup of 12% over *Baseline* once the replay-induced overhead is taken into account. The overall penalty, required to service the faults induced by pruning 3% of the statements from the design of our target processor, is a mere 2% reduction in performance.

Performance Impact of Optimistic Execution *Optimistic Execution* has three main components; Optimistic frequency increase due to pruning, Optimistic data prefetching, and Optimistic branch prediction. In order to isolate the contribution to performance of each component, we analyze four different configurations for the 2 and 4-issue *Tandem* systems being considered in this study. Figure 7 presents the contribution to performance, normalized against *Baseline*, of each Optimistic component.

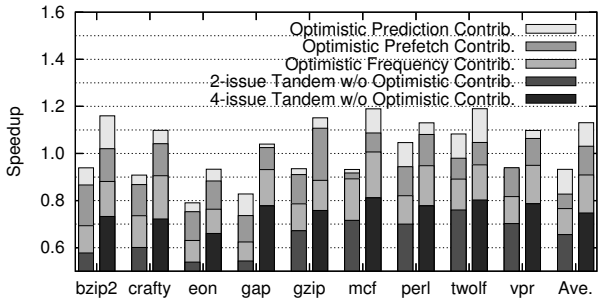

Figure 7. Speedup differential between *Tandem* elements that contribute to *Optimistic Execution*. Results are normalized against *Baseline*.

As expected, the worst performance corresponds to *Tandem* configurations that do not increase their operational frequency through pruning, and use Checker cores that do not access Optimistic prefetch and prediction data. A 2-issue system under these constraints running at the same frequency and memory access time as *Baseline* (2-issue *Tandem* w/o Optimistic contribution) achieves a net slowdown of 52%. Doubling the issue of the Checker (4-issue *Tandem* w/o Optimistic contribution) results in a 34% slowdown.

To isolate the effect of increases in frequency due to pruning, the operational frequency of the previous *Tandem* is increased by 25% following the Optimistic and Checker configurations from Table 1 (Optimistic Frequency contribution). Memory access time for the system is increased accordingly. A 2-issue simulated *Tandem* under this configuration obtains a slowdown of 22%, a wider 4-issue reduces the performance gap with *Baseline* to 10%.

Prefetching effects are isolated by allowing the Checker in the previous *Tandem* configuration to access prefetch information generated by the Optimistic core running ahead (*Optimistic Prefetch Contribution*). Allowing a 2-issue Checker to benefit from Optimistic frequency increases and prefetches reduces its overall slowdown to 12%. A similar configuration involving a wider 4-issue Checker is able to narrowly outperform the slower-clocked *Baseline* by 4% on average.

Finally, the effects of Optimistic branch prediction are exposed by allowing prediction updates from the pruned core to be used by the Checker in the evolving *Tandem* configuration in our study (*Optimistic Prediction Contri-*

bution). This yields a full *Tandem* configuration. A narrow 2-issue full *Tandem* performs 4% slower than *Baseline*. Doubling the issue of the Checker provides a 12% speedup.

Overall, for a *Tandem* configuration with a 4-issue Checker over 53% of its speedup over *Baseline* is due to 25% reduction in clock cycle through architectural pruning. 28% of the increase in performance is due to Optimistic data prefetching. Prediction improvements due to Optimistic branch prediction account for the remaining speedup.

Impact of Signature Buffer Sizing: Figure 8 presents a limit study for the effects of the signature buffer depth with respect to average branch prediction rates for a *Tandem* using the Optimistic and Checker cores found in Table 1. Both *Tandem* configurations obtain good prediction rates for distances of 125 to 500 instruction signatures between cores.

The *Baseline* processor achieves an average branch prediction rate of 94.8%. We observe that *Tandem* configurations using buffers with low signature counts are not able to improve their Optimistic prediction rates over *Baseline*. On the other hand, having more than 500 signatures between cores worsens prediction results considerably. This is due mostly to the diminishing correlation between increasingly distant segments of code being executed by the Optimistic and Checker cores respectively. With the net effect of increasing considerably the amount of aliasing and false data induced by the updates from the Optimistic core. Pollution in the local history tables for the Checker affects more negatively than a lack of timely updates. Therefore, relying on the observed branching trends captured by the GHR in the Checker may be more beneficial than allowing too much pollution into its local predictor tables.

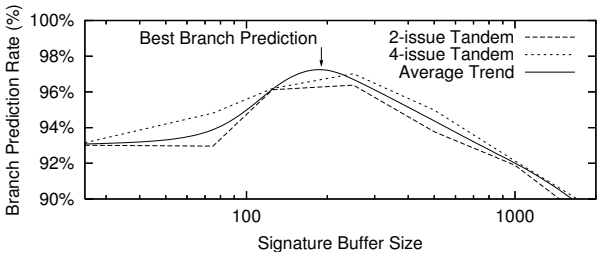

Figure 8. Effect of signature buffer sizing on Optimistic branch prediction.

As stated earlier, allowing the pruned core to run ahead of the Checker amounts to a prefetch engine. The average L2 miss rate for the *Baseline* is 9.8%. Both tandem configurations obtain better L2 hit behavior for signature buffer depths between 125 and 500 instructions. There is a correlation between memory access penalty (250 cycles) and the optimal number of entries in the signature buffer. However the raw miss rate for the L2 cache hides the fact that the pressure on the cache in increased by the parallel requests from the Optimistic and Checker cores. The overall number of cache accesses is increased by over 82% for a *Tandem* using a 4-issue Checker with respect to *Baseline*. The large number of accesses therefore reduces the nominal miss rate for the Optimistic core which has an average miss rate of 9.3%.

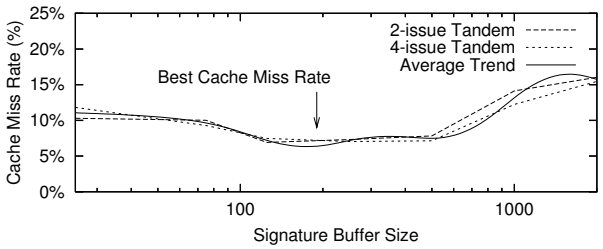

Figure 9. Effect of signature buffer sizing on L2 Cache miss rate.

Low signature counts between Optimistic and Checker cores do not provide enough slack to hide the misses induced by the requests already issued by the Optimistic core and the new memory accesses generated by the Checker. If both cores are separated by a large number of signatures between them, significant degrees of pollution are introduced into the L2 by the runaway pruned core. From the results in Figure 9 it is clear to see that signature buffers between 125 and 500 entries provide the best prefetching behavior.

5.3 Overall Power/Area Considerations

After the previous evaluation it is clear that a Checker that matches the width of the pruned core in the *Tandem* offers the best potential for performance. Both approaches, *Baseline* and *Tandem* with 4-issue Checker share the same L1 and L2 area budget. After pruning considerations, we are able to free over 13% of the combinatorial area from the original design. In order to study the power and area footprint of the Checker processor, we assume a decoupled tandem approach that combines two Alpha processors. The Optimistic core is based on the EV6, and the Checker is based on the in-order 4-issue EV5. After scaling both cores to the same 0.1μm feature size [15], the EV5 weighs in at 28% of a scaled 4-issue EV6. Taking into account the reduction in area achieved through architectural pruning, a pruned EV6 combined with an EV5 Checker shows a 15% area overhead with respect to the original EV6 core. Under the same scaling approach, an EV5 consumes 53% the peak power of an EV6. The tandem configuration would then display an overall increase in power of over 38%.

6 Related Work

To our knowledge, there has been no previous work published on quantitative approaches for improving processor performance through structural or architectural prun-

ing. The application of coverage metrics to account for the utilization of specific processor structures as used in the *μUtilization* metric is also novel. Previous alternative uses of coverage metrics, for purposes other than verification, have been focused on the simplification of system-level design and architectural exploration [6].

The pruning of processor structures amounts to the injection of faults in its design. Fault injection has been thoroughly studied and modeled [7, 28]. Accordingly, several proposals have aimed at offering fault tolerant designs. DIVA tolerates design faults in a complex processor by coupling it to a simpler, verified checker processor [2]. Other approaches such as AR-SMT [21], SRT [18, 20], and SRTR [27], offer recovery from faults through redundant thread schemes. Chip multiprocessors (CMPs) have been proposed for fault detection and recovery. CRTR [11] reduces the probability of thread corruption by having the leading and trailing threads execute on different processors. Reunion [25] correctd soft errors by having multiple cores executing identical threads, each core generates fingerprints which are shipped across cores for cross-verification.

Leader/follower approaches can also be used to improve performance. Future Execution [9] uses a leader core to generate value prediction for non-control instructions to prefetch data for the primary core. Strict leader/follower architectures use different forms of optimization to accelerate the leader and, by extension, the entire system. DIVA [2] and Paceline [12] use overclocking to increase performance, errors associated with a pipeline operating outside its design parameters are solved through the use of redundant cores in charge of verifying the overclocked results. Other approaches use different forms of speculation to accelerate the leader. In Dual-Core Execution (DCE) [32], the leader does not stall on long-latency loads. In Slipstream [19], the leader executes only a subset of the full instruction stream of the program. Architectural pruning could potentially be applied to the leader cores in these designs as well, providing further speedups.

Some predictor designs operate in a fashion similar to our Optimistic branch prediction by combining possible future execution with past trends. Falcon et al [8] proposes a two-tier approach to prediction which combines a prophet predictor that generates a set of future predictions, and a critic predictor that uses the observed history to question the prophet's prediction. Jimenez et al [14] propose a system in which a small, fast, and inaccurate predictor gets to issue pseudo future predictions, that will later be challenged by a more accurate and slow predictor. If the two predictions differ, all the work based on the fast prediction is flushed.

7 Conclusions

Due to the increasing sophistication of critical structures in complex microprocessors, designers have a more difficult time meeting aggressive frequency, power and area budgets. This paper introduces a new mechanism to gain further insight that may alleviate some of the difficulties associated with the task of optimizing complex processor cores. By using the proposed *μUtilization* metric, designers can identify processor structures ideally suited for optimization. Consequently, the designer can selectively prune infrequently-exercised HDL statements within those structures using a quantitative approach. Although a fully automated pruning tool could be highly interesting, it is beyond the scope of this paper.

Using this new optimization methodology, a detailed analysis is performed for an Alpha EV6-like processor [28]. By removing 3% of HDL statements, the resulting processor exhibits a 25% frequency improvement. To guarantee forward progress we combine the pruned core with a simpler in-order Checker. The resulting Tandem architecture can leverage the prefetching and early branch exploration qualities of the Optimistic (pruned) core with the verified execution of the Checker. A *Tandem* configuration yields a 12% performance increase with respect to the original out-of-order core that served as basis for the Optimistic core. Our proposed pruning approach can be used to reduce the area and power requirements associated with leader/follower architectures.

The work presented in this paper provides insights into the design of existing processors. The *μUtilization* ranking and *Structural Efficiency* metrics provide opportunities for further synthesis optimizations. These metrics may also provide valuable insight in the development of new architectures. Designers can now access a new metric to evaluate the efficiency of competing proposals and decide which structures must be prioritized in their design and verification efforts. *Tandem* processors offer opportunities for further optimizations by using finer degrees of architectural pruning. We expect further work exploring these opportunities.

Acknowledgments

We like to thank the reviewers for their feedback on the paper. Special thanks to Amir Roth for his invaluable help and feedback. The authors gratefully acknowledge Michael Huang, Luigi Capodieci, Andrea Di Blas, and Matthew Guthaus for their feedback on the pruning infrastructure. This work was supported in part by the National Science Foundation under grants 0546819 and 720913; Special Research Grant from the University of California, Santa Cruz; Sun OpenSPARC Center of Excellence at UCSC; gifts from SUN, Altera, Xilinx, and ChipEDA. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *the 36th International Symposium on Microarchitecture*, Nov 2003.

[2] T. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *the 32th International Symposium on Microarchitecture*, pages 196–207, 1999.

[3] C. Bazeghi, F. Mesa-Martinez, and J. Renau. μComplexity: Estimating Processor Design Effort. In *the 38th International Symposium on Microarchitecture*, Nov 2005.

[4] F. A. Bower, D. J. Sorin, and S. Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *the 38th International Symposium on Microarchitecture*, pages 197–208, Washington, DC, USA, 2005. IEEE Computer Society.

[5] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, 1996.

[6] D. V. Campenhout, H. Al-Asaad, J. P. Hayes, T. Mudge, and R. B. Brown. High-level design verification of microprocessors via error modeling. *ACM Trans. Des. Autom. Electron. Syst.*, 3(4):581–599, 1998.

[7] K. Cheng, S. Huang, and W. Dai. Fault emulation: a new approach to fault grading. In *the 1995 international conference on Computer-aided design*, pages 681–686, Washington, DC, USA, 1995. IEEE Computer Society.

[8] A. Falcon, J. Stark, A. Ramirez, K. Lai, and M. Valero. Prophet/critic hybrid branch prediction. *SIGARCH Comput. Archit. News*, 32(2):250, 2004.

[9] I. Ganusov and M. Burtscher. Future execution: A prefetching mechanism that uses multiple cores to speed up single threads. *ACM Trans. Archit. Code Optim.*, 3(4):424–449, 2006.

[10] D. Gil, R. Martinez, J. V. Busquets, J. C. Baraza, and P. J. Gil. Fault injection into VHDL models: Experimental validation of a fault tolerant microcomputer system. In *European Dependable Computing Conference*, pages 191–208, 1999.

[11] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *the 30th International Symposium on Computer Architecture*, pages 98–109, 2003.

[12] B. Greskamp and J. Torrellas. Paceline: Improving single-thread perfomance in nanoscale cmps through core overclocking. In *the 16th International Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, USA, 2007. IEEE Computer Society.

[13] B. Jacob. ENEE 446: Digital Computer Design An Out-of-Order RiSC-16. Technical Report UMD-SCA-2000-02, 2000.

[14] D. A. Jimenez and C. Lin. Neural methods for dynamic branch prediction. *ACM Trans. Comput. Syst.*, 20(4):369–397, 2002.

[15] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, 2005.

[16] J. Martínez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors. In *International Symposium on Microarchitecture*, Nov 2002.

[17] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin. Measuring architectural vulnerability factors. *IEEE Micro*, 23(6):70–75, 2003.

[18] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *the 29th International Symposium on Computer Architecture*, pages 99–110, Washington, DC, USA, 2002. IEEE Computer Society.

[19] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A study of slipstream processors. In *the 33th International Symposium on Microarchitecture*, pages 269–280, 2000.

[20] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *the 27th International Symposium on Computer Architecture*, pages 25–36, 2000.

[21] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *the 1999 Symposium on Fault-Tolerant Computing*, pages 84–91, 1999.

[22] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 2002.

[23] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, 1993.

[24] J. Smolens, B. Gold, J. Kim, B. Falsafi, J. Hoe, and A. Nowatzyk. Fingerprinting: Bounding soft-error-detection latency and bandwidth. *IEEE Micro*, 24(6):22 – 29, November 2004.

[25] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-effective multicore redundancy. In *the 39th International Symposium on Microarchitecture*, pages 223–234, Washington, DC, USA, 2006. IEEE Computer Society.

[26] Synopsys. Design Compiler Product Information, 2005. http://www.synopsys.com.

[27] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *the 29th international symposium on Computer architecture*, pages 87–98, Washington, DC, USA, 2002. IEEE Computer Society.

[28] N. Wang, J. Quek, T. Rafacz, and S. Patel. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. In *International Conference on Dependable Systems and Networks*. IEEE Computer Society, Jun 2004.

[29] C. Weaver and T. Austin. A fault tolerant approach to microprocessor design. In *the 2001 International Conference on Dependable Systems and Networks*, pages 411–420, Washington, DC, USA, 2001. IEEE Computer Society.

[30] S. Weiss and J. E. Smith. *IBM Power and PowerPC*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[31] B. Wile, J. Goss, and W. Roesner. *Comprehensive Functional Verification*. Morgan Kaufman, 2005.

[32] H. Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 231–242, Washington, DC, USA, 2005. IEEE Computer Society.