CRAM: Coded Registers for Amplified Multiporting

Vignyan Reddy Kothinti Naresh, David J. Palframan, Mikko H. Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
Madison, Wisconsin

kothintinare@wisc.edu, palframan@wisc.edu, mikko@engr.wisc.edu

ABSTRACT

Modern out-of-order processors require a large number of register file access ports. However, adding more ports can drastically increase the delay, power and area of the register file. This relationship imposes constraints on existing superscalar designs while impeding implementation of faster and wider out-of-order processors. In this paper, we present a novel multi-ported register file using concepts from network coding. We split a true multi-ported register file into two interleaved banks, each having half the read and write ports. A third bank, storing the XOR of the write backs to the other two banks, is added to amplify the read and write bandwidth. When compared to a conventional register file, our 8R4W 128-entry coded CRAM register file reduces leakage power by 48%, area by 29% and delay by 9%. In addition, for SPEC2006 benchmarks, our implementation consumes 40% less register file dynamic energy on average with IPC degradation of 3%.

Categories and Subject Descriptors

C.1.0 [**Computer Systems Organization**]: Processor Architecture

Keywords

Register file, Banking, Network Coding, Microarchitecture, Low power, Microarchitecture

1. INTRODUCTION

Modern processors need large register files that are multi-ported to provide required operands for each issued instruction. A processor capable of issuing more instructions per clock needs more register access ports. Typically, an n-instruction issue requires 2n read ports and n write ports. This increase in access ports radically increases the access delay, power and area of the register file. An m-read, n-write register file is represented as mRnW in this paper.

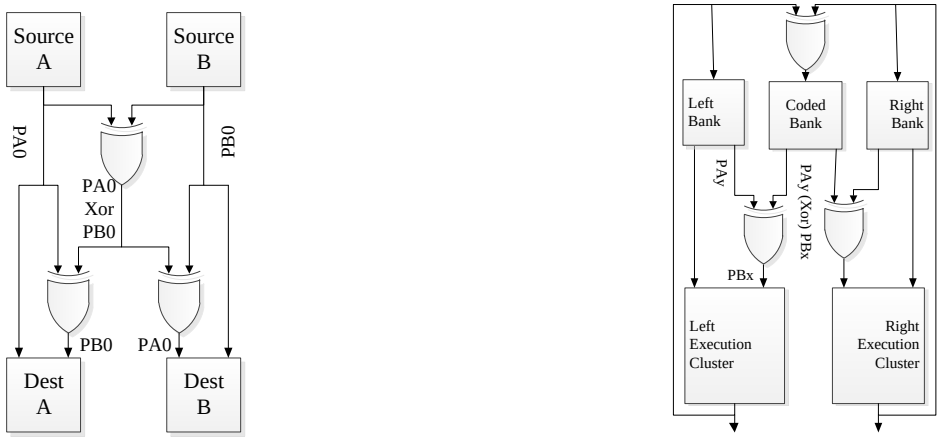
There are a number of factors that contribute the nonlinear relationship between the resources required and the number of register file access ports. Each additional access port adds more transistors per bit cell. The number of bit and word lines also increases with increased access ports and routing these wires has significant impact on area [1]. The load capacitance to be driven by the storage cell is also increased resulting in bigger storage cells and larger sense amplifiers. We analyzed the effect of increasing the access ports of a 128 entry 64-bit register file from 2R1W to 8R4W in the 32nm technology node. The results from CACTI6.5 [2] show an overhead of 50% in delay, 75% in independent read energy, 95% in leakage power and a 1000% in area for the configuration with more ports.

The performance of wider superscalar processors can be limited by the register file. Increased access delays can restrict the clock, add more pipeline stages or both. Additionally, increased power consumption of the register file reduces the power budget for other architectural enhancements. In this paper, we present a novel multi-ported register file using network coding concepts. In comparison to a true multiported register file, the proposed register file is smaller in area and operates faster while consuming less dynamic and static power. The microarchitectural changes required to use the proposed register file only slightly increase the average cycles per instruction.

Figure 1 illustrates the concept of network coding [19]. Two sources A and B share a common coded channel to send both packets PA0 and PB0 to both destinations A and B. Without coding, four channels would be needed to communicate both PA0 and PB0 to both destinations. With coding, the two values can share the coded channel, and the original values can be reconstructed at the destination nodes. We use this concept to increase the read and write bandwidth of our proposed register file. In the CRAM (Coded Registers for Amplified Multiporting) register file, the links in Figure 1(a) represent communication through the register file, where a new value is stored through a write port, and subsequently read via a read port. Read ports are amplified by replicating the register file for the left and right sides, but instead of requiring two write ports for each replica, a third coded register file is placed in the middle. This coded file holds the XOR of the values written back by each side, and can provide that value to either side whenever it is needed. Figure 1(b) shows an example 4R2W CRAM register file. This approach trades off partial replication and some additional control complexity in order to improve the power, area, and/or delay of the design in relation to a conventional multiported design. In summary, this paper makes the following novel contributions:

- Proposes the concept of storing register values in coded (XOR) form.
- Demonstrates how to amplify read and write register file bandwidth using a coded register bank in the CRAM register file design.

MICRO'11, December 3-7, 2011, Porto Alegre, Brazil



(a) **(b)**

Figure 1. (a) Network coding; (b) Coded 4R2W register bank using 3 2R1W register banks

- Describes, in detail, the microarchitectural changes required to incorporate the CRAM RF into an out-of-order superscalar processor.
- Carefully quantifies the power, performance, and area impacts of the proposed design.

The rest of this paper is organized as follows. Previous work in solving problems related to multiported register files is discussed in the next section. Section 3 presents some assumptions in the processor architecture and the rationale behind them. In Section 4, the CRAM register file architecture is introduced and the required changes to the processor pipeline are detailed. The test setup, details of analysis and the results are presented in Section 5. Section 6 concludes this study.

2. RELATED WORK

In the past, many techniques have been proposed to increase the efficiency of large multiported register files. These techniques can be broadly classified into the following categories based on their goals. These goals are either to reduce the number of register access ports or to reduce the number of registers. Our proposal aims to achieve the former.

Register file banking is a commonly used technique where the centralized register file is split into many interleaved banks having fewer access ports. This improves the delay, power and area of the register file, but bank conflicts can degrade performance significantly. There have been many proposals in the past to resolve, avoid and reduce bank conflicts. To alleviate bank conflicts, the required data has to be available from some other source. This redundancy can either come from bypass networks [6,7, 8, 9] or from other banks [10].

Wallace and Bagherzadeh [6] use the delayed register allocation technique proposed by Gonzalez et al. [4] to allocate physical registers in the write back stage. This enables the write registers to be mapped to different banks, thus solving the write conflicts problem. However, they do not provide a detailed solution for read conflict resolution. Late allocation schemes also require non-trivial hardware for correct functionality (e.g. to deal with potential deadlock).

Another technique proposed by Balasubramonian et al. [7] schedules groups of instructions that do not have any read conflicts. They extend their scheduling scheme to speculatively issue an instruction if its operands are predicted to be available from the bypass network. The speculative issue of instructions, whose operands are to be provided by the bypass network, requires special hardware of non-trivial complexity. Park et al. [8] suggest usage of wakeup tags to determine bypass ability. If this bypass hint fails, the read bandwidth is oversubscribed and the pipeline stalls until the read conflicts are resolved. Tseng and Asanovic [9] use a combination of register read port broadcasting and bypass networks to reduce the number of bank conflicts. However, the broadcast network adds significant complexity and cost to the implementation. Bank conflicts still occur in this scheme and require another pipe stage to resolve the associated problems.

Caching high activity registers in a smaller register file that consumes lower power reduces the average latency and the power consumed by the register file [11]. The register values to cache can be critical for performance as the data-copy penalty is considerable.

The Alpha 21264 [12] uses a clustered microarchitecture and replicates the register file with each register file supplying data for its own cluster. Each register file now has half the number of read ports but the number of write ports remains the same. The write-backs have to write to both mirrors of register file and hence the write energy is doubled.

Several clustered micro-architecture proposals [13, 14, 15, 16, 17] divide the register file space amongst different clusters. This division of register file into smaller banks of lower access ports improves the delay and power characteristics. The mapping of instructions to different clusters is of critical importance. These techniques either suffer from a large inter-cluster communication penalty and/or reduced performance compared to similarly-equipped unified resource architectures. Seznec et al. [17] propose binding the read ports and write ports of register banks to an execution cluster. This is an effective technique to reduce area, power and delay of the register file and we use this in our proposed architecture.

Our proposed microarchitecture and the CRAM register file rely heavily on the intellectual heritage of these prior designs. We use banking (or clustering) to improve read bandwidth, while amplifying write bandwidth with coding. We avoid bank conflicts with instruction steering, and by fully provisioning all execution lanes. These steering techniques are similar to prior work, and attempt to minimize the incidence and performance impact of copy operations. We rely on a form of late allocation for entries in the coded bank, and use techniques similar to prior work to ensure correct renaming and deadlock avoidance. Finally, our coded bank is conceptually similar to a register cache, though it is not tagged, and since it always stores a redundant copy, requires no write-throughs or write-backs.

3. ASSUMPTIONS & RESTRICTIONS

This section presents the assumptions of the processor architecture used with our proposed register file. These assumptions make the analysis clear and practical, but may not be essential.

3.1 Clustered Microarchitecture

Clusters are made from grouping execution lanes of different capabilities. Each execution lane can have a specific capability such as integer ALU or memory operations. All clusters in the system are identical. An example of this is our test system, which has two integer ALU execution lanes and two memory execution lanes. These execution lanes are grouped into two clusters, with each cluster having an integer ALU execution lane and a memory execution lane.

3.2 Execution Lane Binding

Instructions are bound to an execution lane at the dispatch stage before being inserted into a unified issue queue. This makes issuing multiple instructions easier. Lane binding is fixed on a set of rules for each individual instruction. In the baseline implementation using a true multiported register file, the instructions are bound to the lane having the least number of waiting instructions in the issue queue. This is similar to the lane binding in the Intel P6 microarchitecture [18].

3.3 Port Binding

Each register read port is bound to only one execution lane. Each write port is also bound to only one execution lane. This eliminates the crossbar and multiplexer networks between register file and execution units. Port binding is fixed in hardware and does not depend on individual instructions.

4. CRAM

In this section, we present the proposed architecture, followed by an instruction walk through the pipeline. Various issues associated with the proposed architecture and their solutions are provided in the subsequent section.

4.1 Basics

Figure 1(b) shows the proposed architecture of a 4R2W CRAM register file with N registers. This implementation uses three 2R1W register banks having N/2 registers each. The left and right banks are the primary banks and hold the physical register values. The coded bank stores the XOR value of simultaneous data writes to the primary banks. The registers written simultaneously into the primary banks and the coded bank register are stored in the register rename table as a three-field tuple. Figure 2 shows the format of an entry in the register rename table.

Primary Bank Reg.	Coded Bank Reg.	Secondary Bank Reg.

Figure 2. Format of an entry in the register rename table

Due to execution lane binding and port binding restrictions, instructions bound to execute on the left cluster can get their operands from the left bank and coded bank only. Similarly, the instructions bound to execute on the right cluster can get their operands from the right bank and the coded bank only. A primary bank register is the physical register that holds the actual value of the architectural register. If an instruction executing on a cluster can't read the primary bank register, it can use the secondary bank register and the coded bank register to obtain the operand value stored in the primary bank register. Since the left cluster never reads or writes into the right cluster and vice versa, there are no register bank conflicts. As shown in the Figure 1(b), only one read port of the primary bank is connected to the XOR gate. This will reduce the routing complexity and does not need de-multiplexers. However, an additional constraint in instruction steering logic is required for enabling this architecture. If an instruction has both its operands provided by the same register bank, it has to be executed on the corresponding cluster. Before we look at more details of the implementation, a detailed example can further illustrate the concept.

4.2 Example Instruction Flow

Let's consider an example instruction "ADD RD, RS1, RS2" and follow its flow through the processor pipeline using a simple algorithm for scheduling instructions. A 4R2W register file is assumed as shown in Figure 1(b) with the left bank (LB) having all even physical registers and right bank (RB) having all odd physical registers. Also, assume RS1 and RS2 are mapped as shown in Figure 3.

Reg. Index	Primary Bank Reg.	Secondary Bank Reg.	Coded Bank Reg.
RS1	P4	P15	X7
RS2	P5	P12	X12
RD	-	-	-

Figure 3. Register rename entries for the illustrative example

Register Rename, Dispatch: Tuples corresponding to RS1 and RS2 are read out of the register rename table. Execution lane binding is influenced by the primary bank registers P4 and P5. According to our simple algorithm (details provided in Section 4.4), since these belong to separate banks and the instruction is commutative, the instruction is bound to an execution lane that has the least number of instructions waiting in the issue queue. In this case, assume the instruction is assigned to the right cluster. The output register is also allocated from the right bank-P19 in this case. The instruction is placed in the issue queue with the register file ids for operands as shown in the Figure 4.

OUT: P19	PE0:P5	PE1:P15	CE: X7

Figure 4. Register id part of the issue queue entry

Issue: Note that all physical registers whose tags are stored in the issue queue entry are accessible by the execution lane on which the instruction will execute. As in a conventional scheduler, the instruction is selected for issue when it is the oldest instruction and all its operands are available.

Register File Read: During the RF read stage, the right bank provides P5 as the right operand RS2, while reading both P15 (from the right bank) and X7 (from the coded bank) to reconstruct the left operand RS1. Note that no bank conflicts can occur here.

Write back: After execution is complete, the register P19 is written back into the right bank. The XOR of the write back values from the left and right execution clusters during the same cycle is written into the coded bank. If only one of the execution lanes writes back into its register bank in a given cycle, the other input to the XOR is forced to zero.

Let's assume that in the same cycle, the left bank is writing the result of another instruction into the physical register P36. The XOR of the values written into the registers P36 and P19 is stored in the coded bank register, say X20.

Tag broadcast and updates: The tag broadcast now consists of three values: P36, P19 and X20. This tag broadcast will be sent to the issue queue and rename table. The issue wake-up logic checks for P36 or P19 in any of the instruction's operands. If there is a match and the registers are not accessible by the execution lane the instructions are bound to, then the register field information in the issue queue is updated with the secondary bank and the coded bank information. The rename table updates the entry tuples of corresponding architectural registers with the broadcast tag information.

4.3 Register Renaming

As indicated in Figure 2, the rename table has three fields in each entry. One for the primary bank register, another for the secondary bank register and the third for the coded bank register. If the execution cluster does not have access to the primary bank's register due to port binding restrictions, the secondary bank register and coded bank register can be used to obtain and reconstruct the data.

On a physical register allocate, the rename table's first field is filled with the allocated physical register. Since the information for the next two fields is available only at the write-back stage, the rename table also gets the tag broadcast to update its entries. As the rename table is updated during different pipe stages, the rename table will either have to be split into two banks or be multiported to accommodate the updates. The tag broadcast is also modified to now include coded bank register's tag. This information is also required by the issue wake up logic to update entries in the issue queue.

4.4 Dispatch & Execution Lane Binding

Since we assume a clustered microarchitecture, instruction steering is a key factor for performance. In addition to the clustered architecture, we also have the following conditions to address:

1. Read register restriction: Our proposed architecture requires an instruction with both its operands coming from the same register bank to be scheduled on an execution cluster that can access this register bank. The instruction execution lane binding logic considers this constraint.
2. Commutability of monadic and dyadic instructions: Some monadic and dyadic instructions are commutable (ADD, AND, etc.) while others are not. For both execution clusters, the second operand (if present) can be provided by the coded bank. So, for non-commutable instructions, if the operands come from different register banks, the first operand register determines the execution cluster to which the instruction is to be bound. For the proposed architecture in Figure 1(b), assume the left bank has all even registers and the right bank has all odd registers. In that case, the left execution cluster can only execute instructions with even first operands when the instructions are non-commutative. Similarly, the right execution cluster can only execute instructions with odd first operand when the instructions are non-commutative (Note that even/odd here refer to the physical register names, which are under microarchitectural control, rather than architected/logical register names).

Once the instruction is bound to an appropriate execution lane, the issue queue entry for the instruction is updated with the register ids to access from the primary bank (PE0 and PE1) bound to the execution lane and to access the coded bank(CE). Figure 4 shows the register's fields for the issue queue entry. The PE0 and PE1 fields hold the register id of the first and the second operand. If the right operand is to be accessed from the coded bank, then the PE1 is replaced by the corresponding secondary bank register of the second operand and the CE is filled with the corresponding coded bank register id.

4.5 Coded bank

The primary and secondary bank registers are said to be "paired" in the coded bank register. We use this terminology to explain some concepts in this section. Some of the challenges that come with the coded bank are given below.

1. Register Deallocation and Referential Integrity: Partial free lists for all primary and coded bank registers are required when one of the paired registers in the coded bank is freed, while the other is not. The hardware complexity is non-trivial to implement the partial free lists and the required control logic.

   To ensure referential integrity, one physical register can't be paired with more than one physical register in the coded bank. Illustrating this, if a physical register P1 is paired with P2 in X1, the register X1 and the availability of P2 in coded bank has to be invalidated before P1 is paired with P3 in X4. In other words, the machine must not honor stale references to either coded or left/right bank registers that have been reclaimed and overwritten. Naively, all structures that contain rename tags must be searched and matching entries invalidated every time a register is reclaimed.

   Instead, we use a bit-vector, called the existence vector (EV), to guarantee referential integrity. The EV has a bit for each physical register specifying if the corresponding register's value can be obtained from the coded bank. When a physical register is written in the write-back stage, the bit corresponding to this physical register is set. The bit is cleared when the physical register is freed. To ensure referential integrity, the secondary bank register is also cleared from the EV when the primary bank register is freed. The EV design is much simpler and safer than attempting to explicitly invalidate all stale references in the machine.
2. Replacement Policy: When there are no free registers in the coded bank, the pipeline can be stalled until free registers are available. Alternatively, an existing valid register in the coded bank can be overwritten with a new coded value. We use a first-in, first-out (FIFO) replacement policy in our implementation. Of course, the EV must be updated to be in sync with the coded bank. In addition to setting bits in the EV when a coded register is written, the appropriate bits

must be cleared when a coded register is removed (overwritten). To clear EV bits in the latter case, we introduce an additional structure called the EV-FIFO. There is one entry in the EV-FIFO for each coded register. Each EV-FIFO entry consists of the two physical register ids paired to create the corresponding value in the coded register. This pair of ids is pushed into the EV-FIFO on each write-back. When a coded register is to be overwritten, the corresponding pair of ids are read from the EV-FIFO and the EV bits for these register ids are cleared.

3. Deadlock and Live-locks: All possible deadlocks are avoided by delayed allocation of coded registers and using it as a FIFO. Live-locks are very unlikely, but can be present given the right mix of issue latency, issue policy, latency of the special copy instruction, size of coded bank and maximum number of ready instructions in the issue queue that need to access the coded bank. We have observed this trend with very small coded banks with less than 3 entries with the special copy latency of 3 cycles. In this scenario, the copy instructions overwrite the previous values before they can be used by the instruction. However, with a coded bank with larger number of entries this problem does not exist.

Using coded bank as FIFO and maintaining the EV are workarounds for avoiding the issues mentioned above. A precise solution would to be to maintain partial free lists and track each free register and the coded bank it is tied to. Such a solution involves a large overhead with a trivial increase in performance.

4.6 Issue Stage

The information required to be put into the issue queue will not be provided from the register rename stage when the instruction is dependent on another instruction whose write back has not yet occurred. For this reason, the issue wake up logic will use the tag broadcast to update the values of these issue queue entries. The issue wake up logic requires additional CAM structures for operation. Precisely assessing the overhead in the issue stage would require physical implementation of the issue queue and is left to future work.

An instruction is issued to its bound lane only if all the required registers from primary and coded banks are available. If a register is not available in the coded bank (checked by using EV) and has already been written earlier, a special copy instruction for that physical register is issued. This special copy instruction operates on a physical register. The value of the physical register is read and the same physical register is written back. These copy instructions do not reside in the issue queue. Instead, they are issued directly by the issue logic when a required register is not available in the coded bank. When this register is written back, the coded bank also gets a copy. The latency of copy instruction is deterministic and in the evaluation presented in this paper, we assumed that the dependent instructions will be ready to issue after 3 cycles of issuing the special copy instruction. Issuing this copy instruction stalls ready instructions from being issued and can potentially hurt performance of applications. Copy instructions also increase reads and writes into the register banks contributing to the overall power consumption.

4.7 8-Read, 4-Write CRAM Register file

To support 4-wide issue across two clusters, we extend our proposed architecture to 8R4W N-entry register file using three 4R2W register banks, each having N/2 registers. Figure 5 shows the block diagram of this extension.

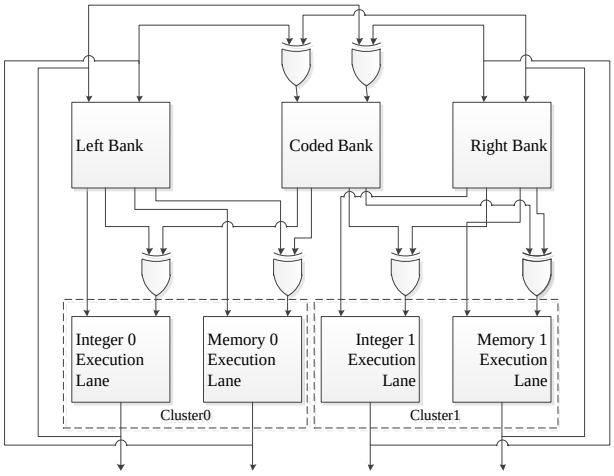


Figure 5. Implementation of 8R4W CRAM register file

4.8 Discussion

For the remainder of this paper, we will focus on the specific 8R4W design shown in Figure 5, integrated into an aggressive, out-of-order CPU core. However, the novel register file design principle proposed in this paper—use of reversible coding in a multibank register file to amplify read and write bandwidth—is applicable to a wide range of designs. Since parallelism is fundamentally more power-efficient than increased frequency, future designs, even with simple, in-order issue policies, are likely to support wide issue and require multiported register files. For example, wide-issue VLIW cores, which are popular in low-power DSP applications require large multiported register files for execution [29, 30]. Given the relative simplicity of the rest of the data path and the control path in such machines, the power-efficiency of the register file will be even more important than it is in an out-of-order superscalar processor. We leave detailed evaluation of a CRAM-RF design in these contexts to future work, but note that significant potential exists.

5. PERFORMANCE EVALUATION

The evaluation of the CRAM is split into two sections – static and dynamic analyses. The static analysis provides the delay, area and power estimates that do not depend on the program execution. This analysis is presented in Section 5.1. Dynamic analysis of CRAM is presented in Sections 5.2 through 5.5. Instructions per clock (IPC) and register energy are dependent on the runtime. These parameters are evaluated for a set of SPEC2006 benchmarks [20] and are presented in the dynamic analysis. All analyses have been performed on the integer register file alone. Although the floating point register file is not modified, floating point loads and stores requiring integer registers are still affected by the instruction steering mechanism. We did not apply the CRAM approach to the floating point register file, since it needs fewer ports in our machine configuration; we leave this to future work.

5.1 Static Analysis

Register file parameters are modeled using HP CACTI 6.5. The delay, access energy and the area numbers are cross-validated using the FabMem [21] tool. CACTI allows usage of three different cell types to use to model the memory. These cell types are ITRS High Performance (HP), ITRS Low Standby Power

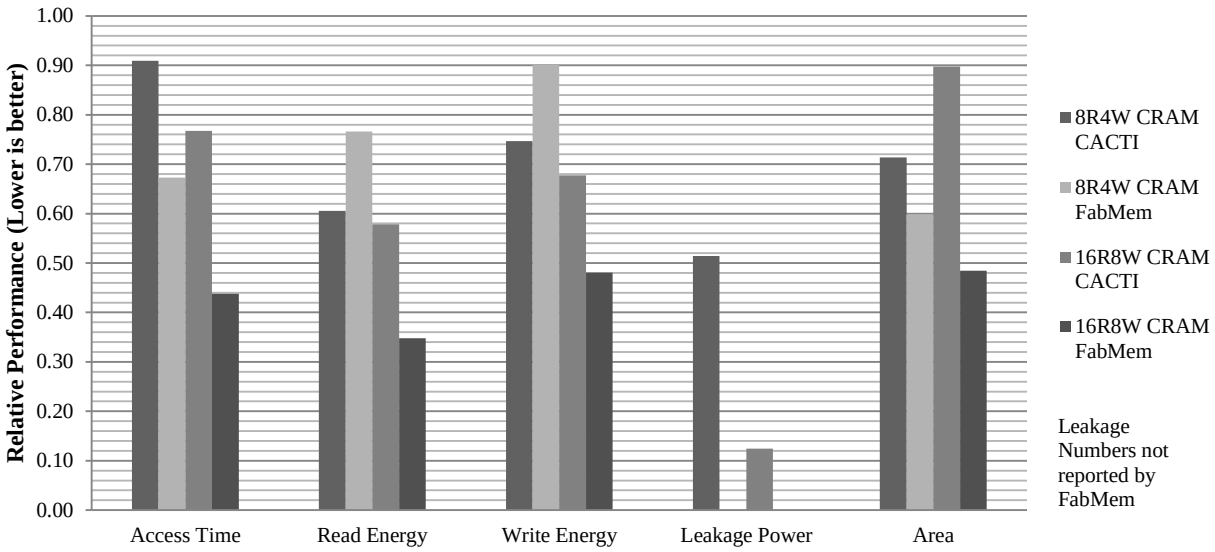


Figure 6. Performance of the CRAM register files normalized to their corresponding true-multiported versions. Results are obtained from static analysis of the register file implementation.

(LSTP), and ITRS Low Operating Power (LOP) and are detailed in [2]. The delay of these cell types increases progressively from HP, to LOP, to LSTP. LOP requires the lowest operating power and LSTP has the least leakage power. A combination of these cells for the data array and the peripherals is used to approximately match the delay of the proposed register file architecture to the true multi-ported register file. We attempt to hold delay roughly constant to simplify our performance and power evaluation. Clearly, CRAM principles could also be applied to instead minimize delay, at the expense of savings in area and/or power.

The leakage power and the area of the primary and coded banks are obtained separately and added. The delay, power and area of the XOR gates and the EV-FIFO contribute to the final value of the parameters used in this analysis. The operating energies for static analysis assumes that 50% of the reads and writes go to the coded bank. We show results obtained from both CACTI and FabMem.

Figure 6 shows the area, read and write energies, leakage power and area of the CRAM implementation of 128-entry 8R4W and 256-entry 16R8W register files relative to their corresponding true multiported versions. The static analysis using both tools shows a significant benefit using the CRAM architecture for multi-ported register files. The differences in the numbers reported by the individual tools have to do with the implementation and capabilities of the tools. The results from FabMem are intended to support the trends observed with CACTI.

5.2 TEST SETUP

To obtain performance and activity counts, we use an execution-driven x86 simulator derived from Bochs [23]. An internal RISC ISA is designed to break x86 instructions into uops that run in the timing part of the simulator. Activity counts are added to the simulator similar to Wattch [24]. In the CRAM implementation, the 8R4W register file is similar to the one shown in Figure 6.

Section 5.6 explores different CRAM configurations based on write port binding and their performance. These configurations are referred to as CRAM-S and CRAM-X. CRAM-X has higher performance than CRAM-S and the dynamic analysis presented in this paper is done on the CRAM-X configuration. See Section 5.6 for more details on these configurations.

Table 1. Configuration of the tested out-of-order machine

Attribute	Configuration
Branch Prediction	Combined bimodal (16k entry) / gshare (16k entry) with a selector (16k), 32-entry RAS, 4-way 2k-entry BTB
Out-of-order architectural features	128 Integer PRF, 64 FP-PRF; 4-wide fetch/commit, 6-wide issue (2 int, 2 mem and 2 FP), 128 ROB, 36 IQ, 48 LQ, 32 SQ; 11-stage pipeline, speculative scheduling with squashing recovery, aggressive memory reordering with store set predictor (4k ssit, 128 lfst) and flush recovery, runahead on L2 miss
Functional Units (latency)	2 integer ALU (4-cycle Mult/Div, 1-cycle for the rest), 2 Memory (1+2 cycles for load, 1 cycle for store address and data), 2 SIMD units (1 cycle), 2 FP add/mult (6 cycles), 1 FP div/square-root (12 cycles)
Memory	I-Cache: 32KB, 2-way, 64B lines 2 cycles; D-Cache: 32KB, 4-way, 64B lines 2 cycles; 2MB, 8-way unified, 128B lines (12), Off-chip memory: 168-cycles

5.3 Benchmark selection

The SPEC2006 benchmark suite and its inputs are intended to represent real world programs and applications. We chose a representative set [25] of SPEC2006 benchmarks for performance evaluation. The Pinpoint tool [26] is used to get simpoints [27,28] for each of these benchmarks. Each benchmark is fast-forwarded to the first simpoint so that the branch predictor, L1 I-Cache and L2 caches are warmed up. Timing analysis is performed on the 100 million instructions following the first simpoint. Throughout the paper the baseline with a configuration shown in Table 2 is used for any normalized results.

Table 2. Selected SPEC2006 Benchmarks and baseline performance

SPEC INT2006 Benchmarks	Baseline performance (IPC)
astar	1.08
bzip2	1.74
gcc	1.45
libquantum	3.00
mcf	1.31
omnetpp	1.91
perlbench	1.61
xalancbmk	2.14
SPEC FP2006 Benchmarks	**Baseline performance (IPC)**
cactusADM	1.35
calculix	1.56
dealII	1.08
GemsFDTD	1.63
leslie3d	1.72
lbm	1.02
povray	1.51
soplex	1.68

5.4 Energy Analysis

This section describes our approach for modeling the energy of the CRAM register file. In addition to the energy results obtained from the CACTI6.5 model for each of the register banks, we account for the additional logic and control structures required for correct operation. The XOR gates are connected to only one of the two ports of the primary bank bound to an execution lane. The operands that are read through the port connected to the execution lane via XOR gates take more energy than the operands that are read through the other. For the primary bank, these reads are classified as type A with energy REP_A (Read Energy Primary Bank Type A) and type B with energy REP_B. REP_B includes the energy consumed by the 64 2:1 XOR gates. Since the coded bank size can be different from the primary bank, this read energy is classified separately (denoted as REP_C). REP_C does not include the XOR gate energy as it is already considered in the REP_B energy.

Similarly, writes into a primary bank will require a different energy (Write Energy Primary, denoted as WEP) than the writes into the coded bank (denoted as WEC). WEC adds the energy of 64 bit XOR gates and the EV-FIFO's read and write energy to the write energy of the coded bank. Table 3 summarizes the different read and write energies involved in this analysis.

Table 3. Summary of different energy consumptions used in this analysis

	Read Energy	Write Energy
Primary Bank	Primary Bank read energy, labeled as REP_A	Primary Bank write energy; labeled as WEP
	Primary Bank access energy + dynamic energy of 64 XOR gates; labeled as REP_B	
Coded Bank	Coded Bank read energy; labeled as REC	Coded Bank write energy + Operational energy of 64 XOR gates + EV-FIFO read energy +EV-FIFO write energy; labeled as WEC

Let there be nrPA reads that consume REP_A units per read, nrPB reads that consume REP_B units per read and nrC reads that consume REC units per read. The number nrPA includes the reads from the penalty instructions. The total read energy is given by the following formula.

$$Total\ Read\ Energy = REP\_A*nrPA + REP\_B*nrPB + REC*nrC$$

Let there be nwP writes that consume WEP units per write and nwC writes that consume WEC units per write. The number nwP includes all the writes required for penalty instructions as the penalty instructions write bank into the primary bank. The number nwC also includes all the writes required for penalty instructions as the penalty writes back into the coded bank too. The total write energy is given by

$$Total\ Write\ Energy = WEP*nwP + WEC*nwC$$

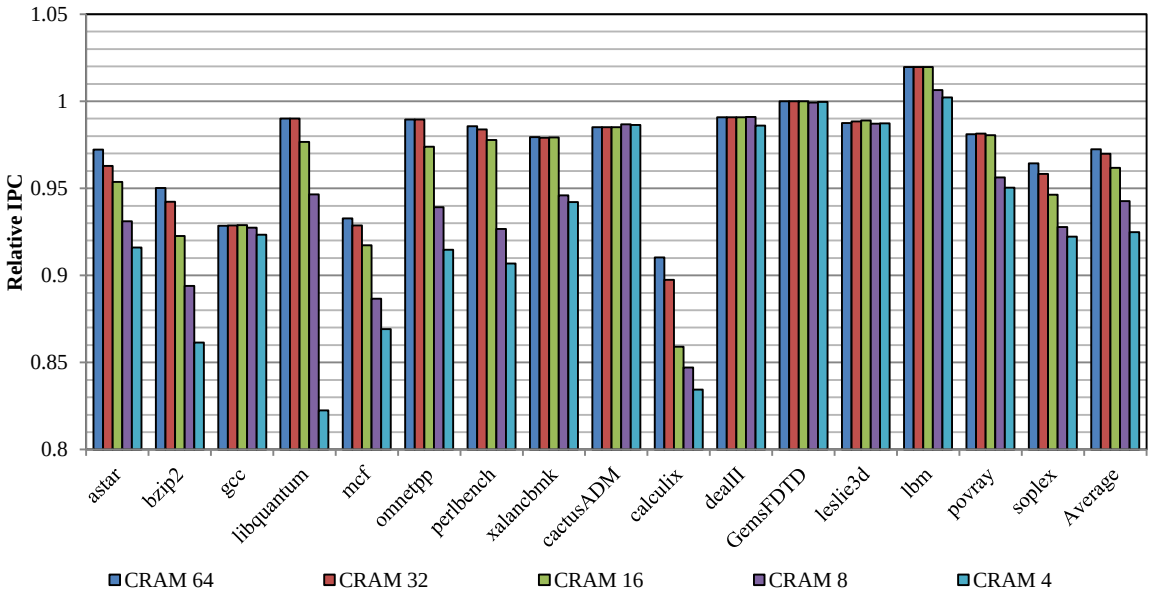


Figure 7. Relative IPC of different configurations of CRAM register file

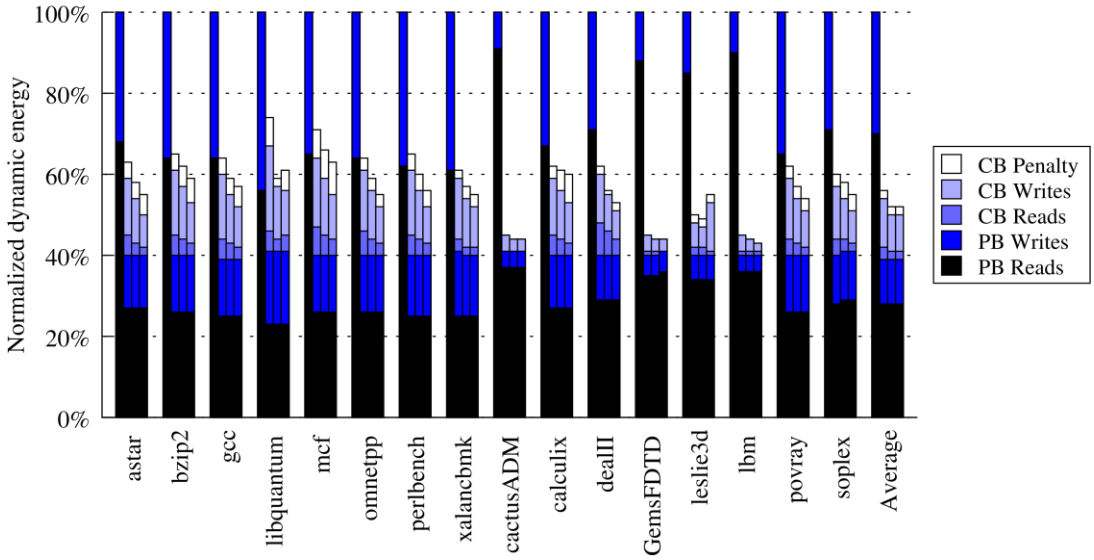


Figure 8. Relative dynamic energy consumed by the 8R4W CRAM register file. For each benchmark, relative energies of the baseline, CRAM 64, CRAM 32 and CRAM 16 are shown and in that order.

The energy values of REP_A, REP_B, REC, WEP and WEC are obtained from CACTI. Our simulator provides the numbers for reads and writes into primary and coded banks. We use this information to calculate the relative power savings of CRAM over the true multi-ported register file.

5.5 Performance Analysis

Figure 7 shows the relative IPC of CRAM for different coded bank sizes compared to the baseline implementation for the selected SPEC2006 benchmarks. We are using an 8R4W CRAM register with 128 physical registers in this analysis. In the figure, CRAM 64 means the coded bank size is 64 registers in the CRAM architecture. A geometric mean of the IPC losses for all the benchmarks shows an average IPC loss of 3% for the CRAM 64 design point. As the coded bank size decreases, we see a fall in the average IPC which becomes significant when the coded bank has less than 16 registers. Note that the primary banks remain fixed at 64 registers each.

Since the IPC impact is too severe below a coded bank size of 16, we consider CRAM 64, CRAM 32 and CRAM 16 for the dynamic energy comparison. Figure 8 shows the relative energy consumed by baseline implementation, CRAM 64, CRAM 32 and CRAM 16 for different benchmarks. Each graph bar also shows the relative energy consumed by different components. The baseline will only have primary bank read and write energies (PB read and PB write). CRAM implementations have the additional energy consumed by the coded bank reads (CB reads), coded bank writes (CB writes) and the penalties involved (CB penalties). Each penalty adds a read and a write to the primary bank and a write into the coded bank. The penalties increase as the coded bank size decreases, but the energy required to access the coded bank also decreases. Based on the number of penalty instructions, the energy consumed can be higher for CRAM 16 than CRAM 32 for some benchmarks. CRAM 64 consumes 45% lesser register file energy while CRAM 32 and CRAM 16 consume 48% less register file energy than the baseline implementation.

Figure 9 shows the dynamic energy delay products, ED and ED^2, for different CRAM configurations assuming that the register file uses 20% of the processor's energy. The energy delay product for all the benchmarks is obtained by formulas suggested by Sazeides et al. [31]. CRAM 32 has the lowest energy delay product of the three configurations which makes it the best solution for our implementation.

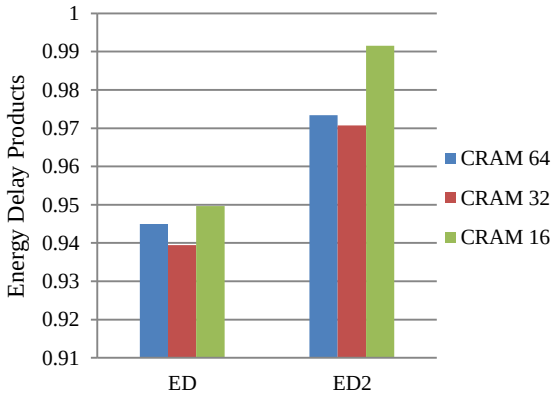


Figure 9. Average energy-delay products for different CRAM configurations assuming 20% of the system power is consumed by the register file.

5.6 Effects of Write Port Bindings

Read port binding and write port binding of a cluster are independent of each other. The write port can go to a different register bank than the one supplying the register read values. We simulated both configurations and studied the workload distribution amongst the clusters. The symmetric CRAM register configuration where the cluster writes back to the same primary bank that it reads from is referred to as CRAM-S. CRAM-X refers to a cross-coupled configuration where the cluster writes to the primary register bank that it does not read from. CRAM-X avoids work imbalance between the clusters due to the register read restriction as mentioned in Section 4.4.

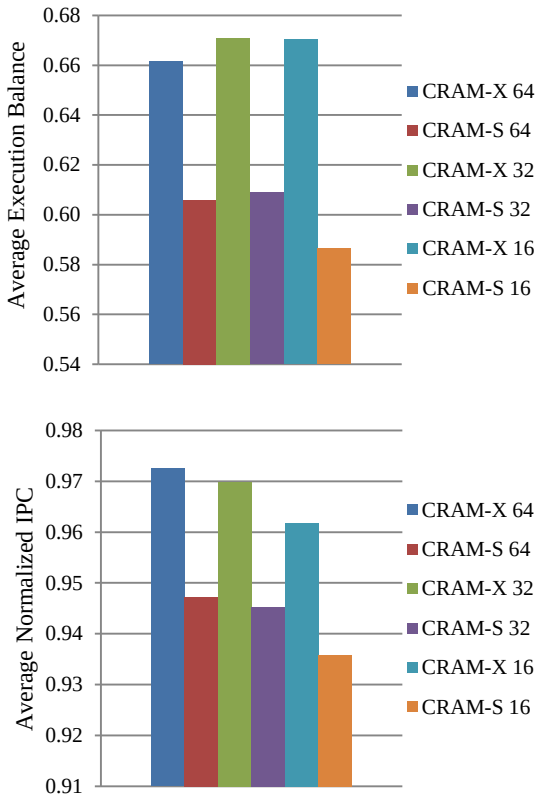


Figure 10: Average Execution balance and Average relative IPC of different CRAM configurations. For Execution balance, a value closer to 1 implies better balanced work amongst the clusters.

The CRAM-X implementation might present wiring challenges, but innovative layout techniques can be employed to ease routing. Laying out the register banks in opposing directions is one such solution. However, the complexity involved is determined by detailed layouts and is left for future work. In the presented work, we assume that CRAM-X has the same energy, delay and area parameters as CRAM-S.

Figure 10 shows the average execution balance and relative IPC for different CRAM-X and CRAM-S configurations. The execution balance is used to measure the work executed on each of the clusters. It's a ratio of instructions executed on each cluster. For comparison purposes, we invert ratios greater than 1 to get the execution balance. Execution balance closer to 1 implies better balance of work amongst clusters. As seen from the Figure 10, the average IPC correlates with the execution balance.

6. CONCLUSION & FUTURE WORK

In this paper, we propose a novel architecture for a multiported register file and the required changes to the pipeline for its implementing. The proposed CRAM register file amplifies register file read and write bandwidth by employing a coded register bank for communicating across execution clusters. This architecture is shown to be faster and more energy-efficient while occupying less area than a conventional multiported register file. A CRAM 32 register file has a reduction of 9% delay, 48% dynamic energy, 57% leakage power and 40% area, and incurs only a small IPC penalty.

Compared to a simple banking technique, we significantly reduce the stalls for read conflicts and remove any write stalls completely. This enables significant performance improvement over the simple banking techniques. In doing so, we still improve the access latency of the register file by not having a crossbar to access the register banks.

The coded bank used in the proposed architecture can also be viewed as an enhanced bypass network between two clusters. While this analysis uses the issue width of a typical processor, we believe that savings are proportional to the number of access ports to the register that might be required for specialized processors (e.g. VLIW) . The lower access time of the CRAM register file also enables higher clock frequencies for the wider issue processors that may otherwise be limited by the cycle time of the register file. Alternatively, the improved access time could be used to reduce the number of pipeline stages dedicated to register file access.

This architecture could also be used to enable horizontal fusion of two cores. This can speed up execution of single threaded programs, since the coded bank will act as a communication channel between the two cores. However, that study requires substantial changes to the processor architecture and is deferred to future work.

7. ACKNOWLEDGEMENTS

This work was supported in part by National Science Foundation award CCF-1116450, and benefited from initial discussions and analysis with Kewal Saluja and Eric Weglarz.

8. REFERENCES

[1] V. Zyuban and P. Kogge. The energy complexity of register files. ISLPED'98.

[2] S. Thoziyoor, J.H. Ahn, M. Monchiero, J.B. Brockman and N.P. Jouppi. A Comprehensive Memory Modeling Tool and Its Application to the Design and Analysis of Future Memory Hierarchies. ISCA-32, 2008.

[3] Farkas, K.I.; Jouppi, N.P.; Chow, P. Register file design considerations in dynamically scheduled processors. HPCA-2, 1996.

[4] A. Gonzalez, J. Gonzalez and M. Valero. Virtual-Physical Registers, in the proceedings of HPCA, 1998.

[5] T. Monreal, A. Gonzalez, M. Valero, J. Gonzalez and V. Vinals. Delaying Register Allocation Through Virtual-Physical Registers. in the proceedings of MICRO, 1999.

[6] S. Wallace and N. Bagherzadeh. A scalable register file architecture for dynamically scheduled processors. In Proc. PACT, October 1996.

[7] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In MICRO-34, December 2001

[8] I. Park, M. D. Powell, and T. N. Vijaykumar. Reducing register ports for higher speed and lower energy. In MICRO-35, Istanbul, Turkey, November 2000

[9] J.H. Tseng, K. Asanovic. Banked multiported register files for high-frequency superscalar microprocessors. ISCA-30, 2003.

[10] Nam Duong; R. Kumar. Register Multimapping: A technique for reducing register bank conflicts in processors with large register files. Application Specific Processors, 2009. SASP '09.

[11] J.-L. Cruz, A. Gonzalez, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In ISCA-27.

[12] R. E. Kessler. The Alpha 21264 microprocessor, IEEE Micro, March/April 1999.

[13] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In ISCA-22, 1995.

[14] V. V. Zyuban and P. M. Kogge. Inherently lower power high-performance superscalar architectures. IEEE Trans. on Computers, March 2001.

[15] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko G. Vranesic. The Multicluster architecture: Reducing cycle time through partitioning. In MICRO-30, 1997.

[16] S. Palacharla, N. Jouppi, and J. E. Smith. Complexity effective superscalar processors. In ISCA-24, June 1997.

[17] A. Seznec, E Toullec and O.Rochecouste. Register write specialization register read specialization: A path to complexity-effective wide-issue superscalar processors. In MICRO-35, Istanbul, Turkey, November 2002.

[18] J. P. Shen; M. H. Lipasti. Modern processor design: fundamentals of superscalar processors. McGraw Hill, 2005.

[19] Christina Fragouli, Jean-Yves Le Boudec, Jörg Widmer. Network coding: an instant primer. ACM SIGCOMM Computer Communication Review, Volume 36 , Issue 1 (January 2006)

[20] Standard Performance Evaluation Corporation. http://www.spec.org/cpu2006

[21] N. K. Choudhary, S. V. Wadhavkar, T. A. Shah, H. Mayukh, J. Gandhi, B. H. Dwiel, S. Navada, H. H. Najaf-abadi, and E. Rotenberg. FabScalar: Composing Synthesizable RTL Designs of Arbitrary Cores within a Canonical Superscalar Template. ISCA-38, June 2011

[22] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In Proceedings of the 10th international conference on Architectural support for programming languages and operating systems, 2002.

[23] Bochs: The Cross Platform IA-32 Emulator, http://bochs.sourceforge.net

[24] David Brooks, Vivek Tiwari, and Margaret Martonosi. 2000. Wattch: a framework for architectural-level power analysis and optimizations. ISCA-27, 2000.

[25] A.Phansalkar, A. Joshi, and L.K. John. Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite. In ISCA-34, 2007.

[26] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In MICRO-37, 2004.

[27] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In PACT, 2001.

[28] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In ASPLOS-10, 2002.

[29] J. Glossner, E. Hokenek, and M. Moudgill. The Sandbridge Sandblaster Communications Processor. In 3rd Workshop on Application Specific Processors, Sept. 2004.

[30] Wayne Wolf. High-Performance Embedded Computing: Architectures, Applications, and Methodologies. Morgan Kaufmann, 2006

[31] Y. Sazeides, R. Kumar, D. M. Tullsen and T. Constantinou. The Danger of Interval-Based Power Efficiency Metrics: When Worst Is Best. In Computer Architecture Letters, Vol 4, 2005.