Register Pointer Architecture for Efficient Embedded Processors

JongSoo Park, Sung-Boem Park, James D. Balfour, David Black-Schaffer
Christos Kozyrakis and William J. Dally

Computer Systems Laboratory
Stanford University
{jongsoo, sbpark84, jbalfour, davidbbs, kozyraki, dally}@stanford.edu

Abstract

Conventional register file architectures cannot optimally exploit temporal locality in data references due to their limited capacity and static encoding of register addresses in instructions. In conventional embedded architectures, the register file capacity cannot be increased without resorting to longer instruction words. Similarly, loop unrolling is often required to exploit locality in the register file accesses across iterations because naming registers statically is inflexible. Both optimizations lead to significant code size increases, which is undesirable in embedded systems.

In this paper, we introduce the Register Pointer Architecture (RPA), which allows registers to be accessed indirectly through register pointers. Indirection allows a larger register file to be used without increasing the length of instruction words. Additional register file capacity allows many loads and stores, such as those introduced by spill code, to be eliminated, which improves performance and reduces energy consumption. Moreover, indirection affords additional flexibility in naming registers, which reduces the need to apply loop unrolling in order to maximize reuse of register allocated variables.

1 Introduction

Embedded system designers must optimize three efficiency metrics: performance, energy consumption, and static code size. The processor register file helps improve the first two metrics. By storing frequently accessed data close to the functional units, the register file reduces the time and energy required to access data from caches or main memory. However, conventional register file architectures cannot fully exploit temporal locality because of their limited capacity and lack of support for indirection.

The *Register Pointer Architecture (RPA)* supports large register files to reduce the time and energy expended accesses data caches without increasing code size. The main idea of the RPA is to allow instructions to access registers indirectly through *register pointers*. This provides two key benefits:

- **Large Register File Capacity**: Indirect register access relaxes the correlation between register file capacity and instruction word length. Hence, large register files can be used without sacrificing code density.
- **Register Naming Flexibility**: By dynamically modifying register pointers, a small set of instructions can flexibly access data allocated in the register file in a way that maximizes data reuse.

We introduce extensions to the ARM instruction set to implement the RPA. In addition to the conventional register file, the modified architecture includes a *dereferencible register file (DRF)*. Existing arithmetic and load/store instructions can use a *register pointer (RP)* in any register operand position to access the contents of the DRF. We define efficient update policies for RPs to support common access patterns with minimal runtime overhead. At the microarchitecture level, we describe the interlocks and forwarding paths needed to minimize read-after-write hazards on RPs. We execute a set of embedded applications on a model of the modified processor to demonstrate that the RPA leads to a speedup of up to 2.8× and energy savings of up to 68%.

We compare the RPA to alternative techniques that provide register indexing flexibility or software controlled storage near to the processor. Loop unrolling can be used to follow arbitrary data patterns within the register file. We show that RPA leads to similar flexibility in register file accesses without the code size increases introduced by longer register names and replicated loop bodies. A software-controlled scratchpad memory could be used to capture temporal locality in embedded applications. Nevertheless, a scratchpad memory suffers from requiring explicit load and store instructions to make data available to arithmetic instructions.

In summary, the major contributions of this paper are: we introduce the RPA architecture, which supports indirect register file access through register pointers; we explore design options for RPA at the instruction set and microarchitecture level, including parameters such as the number of additional registers; and, we compare an embedded processor implementing the RPA to a conventional organization. We also compare to techniques such as loop unrolling and scratchpad memory.

The remainder of this paper is organized as follows. Section 2 provides a detailed description of RPA including instruction set and microarchitectural considerations. Sections 3 and 6 describes the experimental methodology and results. Sections 5 and 6 present related work and conclusions.

2 Register Pointer Architecture

This section describes the RPA architecture at the instruction set and microarchitecture levels.

2.1 Instruction Set Architecture

The RPA extends a base instruction set, such as ARM, to support indirect register file accesses. An instruction indirectly accesses a register by identifying in its encoding a *register pointer*, whose contents provide the address for the actual register file reference. While implementations of existing ISAs may use indirect register accesses to implement techniques such as register renaming [1], the RPA exposes the indirection to the software.

In addition to the conventional register file (RF), the RPA defines a *dereferencible register file* (DRF) and register pointers (RPs). The DRF contains all registers accessed indirectly, while the RPs contain indirection information (DRF address and other configuration fields). Data processing and transfer instructions can specify any of the following as an input or output operand: an RF entry, a DRF entry through an RP, or an RP. Note that only register operands are modified by the RPA; no additional opcodes or memory addressing modes are added to the instruction set.

The example instruction shown below illustrates important features of the RPA ISA. The instruction adds `r0`, a RF entry, to the DRF entry pointed by the RP `p0`. The sum is stored in the RP `p1`. The postfix operator "!" increments RP `p0` after it is dereferenced.

```
add  p1, r[p0]!, r0
```

Assembly instructions access the DRF by specifying an RP enclosed in square brackets after the symbol `r`. An RP is accessed as though it was a conventional general purpose register: by directly specifying the RP as an operand.

To reduce the overhead of RP updates, a register pointer may be incremented when named in an instruction. For further flexibility, we support circular addressing using two additional fields in each RP: a base address (`begin`), and a limit address (`end`). An attempt to increment the address beyond the `end` address causes the pointer to wrap around to the `begin` address. An overflow bit is set when the pointer wraps around to allow the software to detect the wrapping around if desired. Each RP stores the base address and limit address field in its higher bits. When an instruction dereferences an RP, it accesses only the least significant bits, whose contents are the actual address for DRF accesses.

The number of bits required to encode the RP and DRF addresses depends on the number of RPs, the binding of RPs, and the number of access modes. Note that the number of DRF entries does not influence the encoding of instructions. The specific modification on ARM ISA is described in Section 3.

2.2 Architectural Parameter Space

So far, we have described the ISA extensions for RPA in rather abstract terms. The exact parameters used to size pertinent resources introduce interesting tradeoffs amongst performance, software flexibility, energy consumption, and hardware complexity, as described below.

Number of DRF Entries: The relationship between the reduction of cache accesses and the number of DRF entries varies by application. For applications such as 1DFIR, cache access reduction is a linear function of the DRF size, while for matrix multiplication, it follows a square root function. For applications with table lookups, it is a step function: it saturates once the lookup table fits in the DRF.

There are two costs associated with larger DRF sizes. First, the energy consumption of a DRF access increases with the number of DRF entries. Second, the DRF access time increases because the larger row decoders incur longer delays while the discharge time of the longer bit-lines increases. The additional access latency should not adversely impact program execution times unless it requires increasing the clock cycle time. Given a small DRF (fewer than 256 entries), we expect the clock cycle time will be limited by the cache access latency rather than the DRF.

In Section 4, we examine how these factors influence the DRF size which best balances performance improvements with energy consumption.

Number of DRF Ports: We can limit the number of DRF ports to one read port and one write port, which reduces the DRF area and energy consumption. However, the reduced number of DRF ports penalizes the performance of applications which have multiple independent data streams, such as 1DFIR.

Number and Binding of Register Pointers: The simplest design, in terms of hardware complexity, would provide one dedicated RP for each operand position. Using the ARM ISA as an example, we would have `pd`, `pn` and `pm`, which correspond to the `d`, `n` and `m` operands, respectively. However, such a scheme lacks flexibility and may introduce overheads when the same stream is used as an input and an output or in different input operand positions. Providing more RPs with flexible bindings tends to increase the encoded instruction width and the complexity of the interlocks required in pipelined processor implementations.

We evaluate these parameters quantitatively in Section 4.1, focusing primarily on the number of DRF entries.

2.3 RPA Processor Organization

Figure 1 shows a five-stage processor pipeline modified to implement the RPA. The main additions are the DRF and RPs, which, as in a conventional scalar pipeline, are read in the decode stage. The post-increment of the RPs is also performed in the decode stage, in parallel with the DRF access, using dedicated incrementers. Writes of both the RPs and DRF are performed in the write-back stage, thereby avoiding output dependencies and anti-dependencies on these registers.

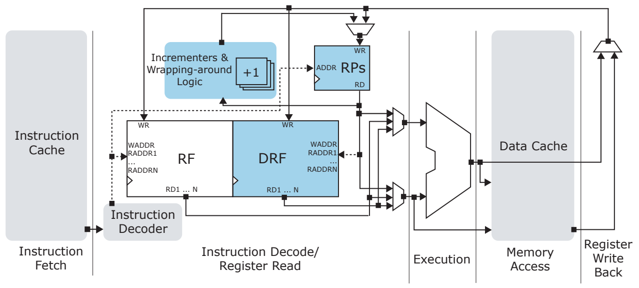


Figure 1: Register Pointer Architecture Pipeline Design

Table 1: Summary of Register Designator Patterns.

Register Designator	Uses
`r0`∼ `r15`	access `r0`∼`r15`
`r16`∼`r26`	reserved for future extension
`r27`	access DRF using the RP bound to the operand
`r28`	access DRF and auto-increment RP
`r29`	access `pd`
`r30`	access `pn`
`r31`	access `pm`

The modified pipeline must deal with true dependencies through RPs and the DRF. Dependencies through direct RP accesses are easily handled through forwarding in exactly the same manner that true dependencies are resolved for regular registers. Dependencies through DRF entries are slightly trickier, since their detection involves a comparison of the addresses stored in the corresponding RPs used for the DRF access, instead of using the name of the register. Facilitating these comparisons without increasing the clock cycle is a major motivation for keeping the RP design simple.

If an arithmetic or memory instruction directly accesses an RP as the destination operand, we can use the value of RP only after the execute or memory pipeline stage. Therefore, if the next instruction uses that RP to indirectly access a DRF entry, the RP value is not available in the decode pipeline stage, and we thus need to stall. However, in most performance critical loops, this does not occur and RP update patterns can be expressed by post-increments. In Section 4, we experimentally show that the pipeline bubbles introduced by RPA have a negligible effect on performance.

The architectural parameters described in the previous section may affect the clock frequency of the processor. The critical path in most embedded processors is the execution or the memory stage; thus it is important that all the added latencies do not make the decode stage the critical path. With the most basic setting, the addition of a serial read to a small value of RP should have negligible impact on the clock frequency.

Table 2: Benchmark Specification

1D FIR Filter	35 taps, 10000 integer samples
Insertion Sort	32 integers
Multi-way Merge	7 ways, 128 integers per way
MD5	16 KBytes input
Matrix Multiplication	140×140 matrices
2D FIR Filter	320×240 gray scale, 3×3 kernel
Stringsearch	1335 strings
GSM Decoding	large.au (52 seconds)
TiffDither	1520×1496, gray scale
PGP Verify	1024 bit RSA key

3 Methodology & Applications

We modified the ARM version of the SimpleScalar simulator [2] to implement the RPA. Specifically, each register operand was increased from four bits to five bits, where the highest five numbers indicate various DRF and RP access modes as shown in Table 1. The unused space between `r16` and `r26` can be used to implement other RPA variations with different architectural parameters. The extra three bits were added at the expense of conditional execution bits. In our experiment, the RPA configuration without conditional execution is compared to the baseline configuration with conditional execution. Stall cycles introduced by dependencies through RPs and the DRF are also included.

We used Sim-panalyzer [10] to estimate the energy consumption of the processor. We chose StrongARM-1110 as our processor parameter model, which has a 200MHz clock frequency, 32-way 16KB instruction cache, 32-way 8KB data cache, and a 128-entry bimodal branch predictor. We model a 0.18 μm technology with a 1.75V supply.

The comparison with scratchpad memory was performed by approximating it with an infinite cache. Such a cache has no capacity or conflict misses and provides an upper bound to the performance possible with a software managed scratchpad memory. For the comparison with loop-unrolling, all benchmarks were unrolled in assembly and hand-optimized to minimize the number of load/store instructions.

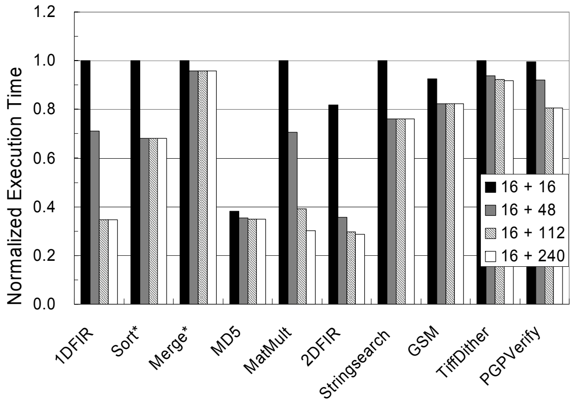
Applications with the following properties benefit most from the RPA extensions:

- The entire working set fits into the DRF or it can be decomposed into smaller blocks that will fit.
- Each datum in the working set has a fair amount of reuse in a short time span.

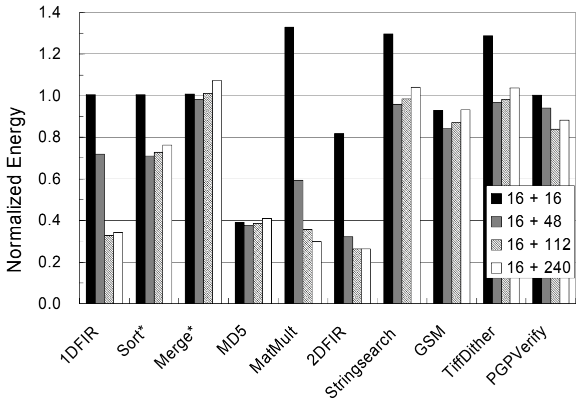
The six kernels and four applications [6] shown in Table 2 were selected for our evaluation because they exhibit these properties.

4 Experimental Results

In this section, we explore various design space parameters for the DRF and compare the RPA to a scratchpad memory and loop unrolling in terms of performance, code size, and energy consumption.



(a)



(b)

Figure 2: Sensitivity of Performance (a) and Energy Consumption (b) to the Number of DRF Entries.

4.1 DRF Design Space Exploration

This section evaluates design parameters, focusing mostly on the number of DRF entries. Since the RPA targets the embedded systems, we focus on minimizing the complexity and energy consumption of the additional hardware required to implement the RPA while preserving the RPA architecture's performance benefits.

4.1.1 Number of DRF Entries

Figure 2(a) shows application execution times as a function of the number of registers (lower is better). The execution times are normalized to the baseline StrongARM without a DRF (1.0). The number '16 + 48' means 16 RF entries and 48 DRF entries. Note that Sort and Merge are marked by stars (*) because it is hard to find representative input sizes for them, and we thus should not consider them when deciding the appropriate number of DRF entries. RPA with 16 DRF entries cannot exploit any performance critical working sets of 1DFIR, Stringsearch and PGPVerify. RPA with 16 DRF entries does not improve the performance of MatMult and TiffDither because the block size is too small to amortize the register pointer configuration overhead in the loop prologue code.

Applications that use blocking such as MatMult, 2DFIR and TiffDither show a relatively smooth execution time reduction with more DRF entries. For MD5, Stringsearch, and PGPVerify with 48 or 112 DRF entries, some lookup table entries are subword-packed. Since the ARM ISA does not support subword access to registers, shifting and bit masking operations have to be done for each subword-packed data access. This overhead negates some of the performance gain from the load and store instruction elimination. Overall, all applications benefit from 48 DRF entries, and improvements from more registers are typically smaller.

Figure 2(b) shows energy consumption relative to the baseline ARM architecture for different numbers of registers. For MatMult and 2DFIR, the performance improvement dominates over the increase in register power, thus they do not show their minimum energy point up until 240 DRF entries. Except 1DFIR and PGPVerify, all the other applications achieve minimum energy consumption at 48 DRF entries.

Access time is another important metric. However, for a register file smaller than 16 + 240 entries, it is unlikely that the register file access time will be the determining factor for the clock frequency of the pipeline, given that SA-1110 has 32-way 8KB data cache, which has sufficiently longer access time.

Thus the optimal number of DRF entries balancing performance and energy-efficiency is between 48 and 112. If the processor will mostly run applications such as 1DFIR, MatMult, 2DFIR and PGPVerify, using 112 DRF entries would be more appropriate. In other cases, using 48 DRF entries is a better choice.

4.1.2 Other Parameters

We also studied the impact of other RPA design parameters. We summarize the major conclusions but do not present detailed results due to space limitations.

Number of DRF Read Ports: At most 3% energy reduction can be obtained by using a single read port over two read ports. The energy reductions were acquired at the expense of 13% and 10% performance loss from 1DFIR and Sort respectively. If energy efficiency is the most important metric of a processor or the target application does not have structured memory access patterns similar to 1DFIR, a single read port may be an appropriate design point.

Number of RPs: Most applications would not benefit from more RPs because they do not have more than three data streams. MD5 has five streams of pointer accesses, which interact with each other. However, MD5 using five RPs performs only 4% faster than the version with three RPs.

Binding of RPs with Operand Positions: Among the chosen applications, only Merge showed a benefit from more flexible RP binding.

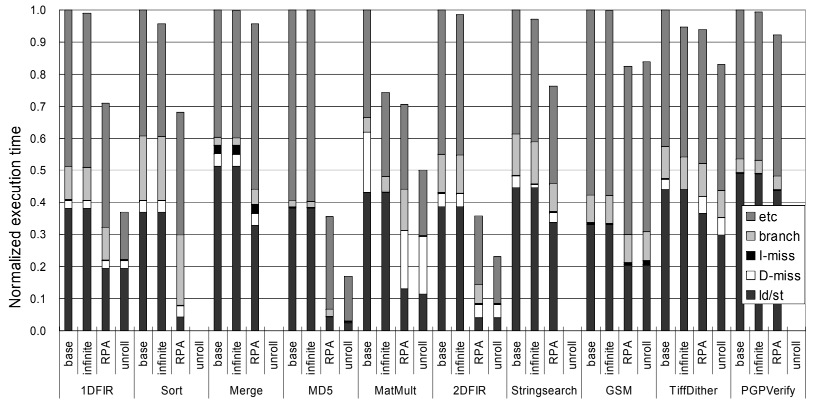


Figure 3: Performance Comparison

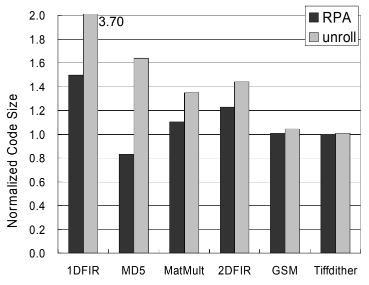


Figure 4: Relative Code Size Compare to the Baseline

RP Update Policies: Since memory operations already support indexed access, data can be properly aligned in the DRF to create a sequential register access pattern. For example, in the case of MatMult, one of the matrices can be transposed when being loaded. For this reason, stride or post-decrement access patterns are not particularly useful in RPA.

4.2 Comparison with Other Techniques

This section compares other techniques to RPA using a 48 entry DRF with two read ports, one write port, and three RPs, which corresponds to the optimal parameters identified in the previous section.

4.2.1 Performance Comparison

Figure 3 presents the execution times of various techniques relative to the baseline. The result of using an infinite cache on top of the baseline is shown as 'infinite'. Certain applications cannot be unrolled, hence the corresponding 'unroll' bar is not present. The time spent on load and store instructions is labeled with 'ld/st'. Branch execution and branch miss prediction penalties are labeled in 'branch'.

On average, RPA performs 46% better than the baseline with most of the gains coming from the removal of load and store instructions. The maximum gains are observed with MD5 and 2DFIR where RPA leads to a speedup of 2.8$\times$ relative to the baseline design.

The infinite cache simulations represent an optimistic model of a scratchpad memory. The results show that the RPA consistently outperforms scratchpad for these applications. Nevertheless, these conclusion does not generalize. A scratchpad memory typically has larger capacity and simpler hardware than multi-ported DRF. A processor can implement both a scratchpad memory and RPA, using them selectively depending on the application. For example, small and performance critical data can be allocated in the DRF to completely avoid loads and stores, while larger structures can be allocated in the scratchpad memory.

For the 'unroll' configuration, each application was unrolled using the same total number of registers as 'RPA'. Because Sort, Merge, Stringsearch and PGPVerify have data dependent control flow or table lookups, they were excluded from the unrolling experiment. RPA and unrolling lead to similar flexibility in register file usage and both allow for elimination of load and store operations. Nevertheless, as indicated by the graph, unrolling has an additional advantage of reducing branch overhead on loop iterations.

GSM is an interesting application in the unrolling experiment because the unrolled version's branch misprediction and instruction cache miss rates are higher. Because GSM's main loop has conditional statements in it, loop unrolling duplicates branches, which cause slower branch predictor warm-up time.

If unrolling does not contribute to a significant increase in code size, and performance is the most important metric, unrolling would be the best solution. RPA performs better when code cannot be unrolled or has an irregular loop structure, such as exhibited by GSM. Applications having a reasonable tradeoff between performance and code size, such as 1DFIR and MD5, also benefit from RPA.

4.2.2 Code Size Comparison

Figure 4 shows static code sizes for RPA and unrolled version, normalized to the baseline (ARM ISA). To factor out auxiliary code such as glibc, we present the aggregate sizes of the relocatable objects before the final linking step.

For the selected applications, RPA increases the static code size by an average of 5%, while unrolling increases it by 51%. Note that the static code size of an unrolled version does not account for the increase in instruction word size required to address a larger register file, making the actual difference larger.

4.2.3 Energy Comparison

Figure 5 shows the energy consumption relative to the baseline processor. On average, energy savings of 32% were achieved. In addition to the energy savings from the performance improvements, additional energy savings were obtained by reducing the number of cache accesses. Note that the energy estimates exclude ALU energy consumption. Using the energy breakdown for the StrongARM processor described in [4], we estimate that the energy reduc-

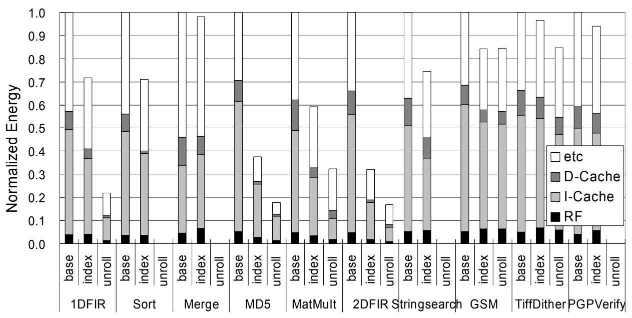


Figure 5: Energy Consumption (excluding ALU)

tion from RPA remains approximately 32% when the ALU energy is accounted for.

It can be seen that for applications that do not satisfy the conditions listed in Section 3, the RPA extensions incur no performance penalty and increase energy consumption by at most 3%: the register file consumes 5% of the total processor power and register file with 64 entries dissipate 47% more power than register file with 16 entries.

5 Related Work

The Cydra-5 VLIW supercomputer provided a Rotating Register File (RRF) [3] to address the unrolling incurred by modulo variable expansion [8]. Register Connection (RC) [7] was proposed to incorporate large number of registers without enlarging the instruction word size. Although RC resolves the capacity problem of register file, it does not address the naming flexibility problem described in Section 1. The Register Queue (RQ) [11] concept combines the above two techniques to resolve loop unrolling without increasing the instruction word size. However, similar to RRF, RQ mostly focuses on software pipelining, and thus it cannot be efficiently utilized in many embedded applications, such as the benchmarks we examined. For example, because only write operations rotate a register queue, we cannot load data to a register queue and then read them multiple times, as was done for FIR and MatMult to exploit locality. Moreover, because the queue sizes are fixed, we pay the overhead of traversing multiple queues if the data is larger than a single queue.

The windowed register file architecture for low power processor [5] has also been introduced to address the limited number of bits available to encode operand specifiers which incurs power consuming memory accesses. The windowed register file addresses the capacity problem in a manner largely orthogonal to RPA. The SIMdD architecture [9] has Vector Pointer Registers (VPR) which is similar to RPs. However, SIMdD architecture focuses on the naming flexibility, more specifically data alignment and reuse problem in SIMD DSP architecture.

6 Conclusions

This paper introduces the RPA, a register file architecture that is based on indirect access through register pointers. RPA addresses both capacity and naming flexibility limitations inherent in conventional register files. It allows the number of registers to be increased significantly without increasing the instruction word length, and supports flexible indirect accessing of the register file. We presented simulation results for an augmented StrongARM processor which show that RPA leads to 46% average performance improvement and 32% average reduction in energy consumption without significant increase in the code size.

We compared RPA to other techniques addressing limitations of a conventional register file. A scratchpad memory addresses the capacity problem of register file, but still requires load/store instructions and lacks multiple ports. Unrolling resolves the naming flexibility problem of the register file, but cannot be applied to data dependent access pattern and leads to large code size increase.

References

[1] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. The IBM System/360 model 91: machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 11(1):8–24, 2000.

[2] D. Burger and T. M. Austin. The SimpleScalar Tool Set, version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, 1997.

[3] J. C. Dehnert, P. Y. Hsu, and J. P. Bratt. Overlapped loop support in the Cydra 5. In *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 26–38, 1989.

[4] J. M. et. al. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *Digital Tech. J.*, 9(1):49–62, 1997.

[5] R. A. R. et. al. Increasing the number of effective registers in a low-power processor using a windowed register file. In *CASES '03: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 125–136, 2003.

[6] M. Guthaus, R. J.S., D. A. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC-4 2001: IEEE International Workshop on Workload Characterization*, pages 3–14, 2001.

[7] T. Kiyohara, S. Mahlke, W. Chen, R. Bringmann, R. Hank, S. Anik, and W. Hwu. Register connection: a new approach to adding registers into instruction set architectures. In *ISCA '93: Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 247–256, 1993.

[8] M. D. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language Design and Implementation*, pages 318–328, 1988.

[9] D. Naishlos, M. Biberstein, S. Ben-David, and A. Zaks. Vectorizing for a SIMdD DSP architecture. In *CASES '03: Proceedings of the 2003 International Conference Compilers, Architecture and Synthesis for Embedded Systems*, pages 2–11, 2003.

[10] The SimpleScalar-Arm Power Modeling Project. Web Page: http://www.eecs.umich.edu/∼panalyzer.

[11] G. S. Tyson, M. Smelyanskiy, and E. S. Davidson. Evaluating the use of register queues in software pipelined loops. *IEEE Trans. Comput.*, 50(8):769–783, 2001.