Instruction Compounding for Embedded Microprocessors

Daniel U. Becker, David B. Sheffield, Vishal Parikh
{dub, dsheffie, vparikh1}@stanford.edu

Abstract

Microprocessors in contemporary embedded systems must provide high performance under tight energy and power constraints. Instruction compounding is a technique that improves performance in an energy-efficient way by merging pairs of dependent instructions. Previous studies have explored compounding in the context of complex out-of-order processors; however, such architectures are typically inadequate for use in embedded designs.

The present work introduces a compounding microarchitecture for in-order embedded processors. We show that our design can reduce the instruction stream size by up to 15% for representative embedded workloads, and that a 6% energy savings can be achieved by a straightforward RTL implementation.

1. Introduction

Embedded systems today must be able to handle increasingly demanding workloads, but are at the same time subject to strict power and energy constraints. Consequently, it is of prime importance that performance increases are achieved in an energy-efficient manner.

A simple technique for improving performance while simultaneously improving energy efficiency in a microprocessor is instruction compounding. Taking advantage of imbalances in logic complexity between different instructions and pipeline stages, this technique dynamically combines pairs of dependent simple instructions into a single more complex instruction. This has two benefits: On the one hand, fewer instructions must be executed, which leads to faster program execution. At the same time, switching activity in the pipeline is reduced due to fewer instructions traversing it, and sharing of operands between merged instructions reduces the number of required register file accesses.

Instruction compounding, also called instruction collapsing, has been the subject of prior research [3, 4, 5, 7]; however, earlier studies investigated its use in the context of superscalar and out-of-order processor architectures, and focused on the performance-related aspects.

The MIPS and ARM ISAs support a static variant of instruction compounding by allowing ALU operations to specify a constant shift amount to be applied to the result. More recently, a limited form of instruction compounding, with support for fusing instructions setting condition codes with subsequent branches, has also been implemented in the Intel Core microarchitecture [8]; future Intel processors are expected to extend this capability to support additional instruction pairs.

The traditional method of instruction compounding requires that instructions be reordered. In an out-of-order pipeline, much of the hardware required to support this (e.g. reservation stations, reorder buffer) is already in place. However, these structures are not present in scalar, in-order pipelines, and consequently, extending such pipelines to support instruction compounding is a more complex undertaking.

In the present work, we propose a compounding microarchitecture for in-order embedded microprocessors and evaluate its effectiveness using a set of common embedded workloads. We find that compounding can reduce the dynamic instruction count by up to 15% with only minor changes to the pipeline. We further present simulation results for a fully placed and routed RTL implementation of the proposed microarchitecture based on the LEON2 processor, and find that a 6% energy savings can be achieved even when supporting just a small subset of possible compound pairs.

The rest of the paper is organized as follows. Section 2 presents an overview of instruction compounding. Section 3 introduces a compounding microarchitecture for in-order processors. Section 4 investigates compounding opportunities in common embedded workloads. Section 5 presents simulation results for an RTL-level implementation of the proposed microarchitecture. Section 6 concludes the paper.

2. Instruction Compounding

Instruction compounding is the act of merging two separate instructions with a write-after-read (WAR) dependency into a single complex instruction, called a compound pair,

that traverses the pipeline as a unit. This is beneficial both from a performance and an energy efficiency point of view:

In exchange for performing more work in a given cycle, compounding effectively reduces the number of individual instructions that need to be executed, and thus program execution time. Effectively, compounding extracts parallelism from the instruction stream without incurring the overhead associated with a full superscalar implementation.

Additionally, each pipeline stage is traversed once by the compound instruction, rather than by each of the two original instructions individually. As a result, the average pipeline switching activity decreases. Furthermore, as operands are shared between both merged instructions, fewer register file accesses are required.

The first step in generating a compound pair is identifying pairs of suitable instructions, which involves decoding the instructions' types and operands, and subsequently checking for dependencies between them. Compound formation can be implemented in a variety of different ways. One alternative is to process the dynamic instruction stream as it flows through the pipeline, generating compound pairs on the fly. Alternatively, compounds can be formed whenever a cache line in the I-cache is refilled, and corresponding predecode information can be store alongside each cache line [7]. The same approach can also be applied to a trace cache [4], which has the added benefit of being able to compound instructions across basic blocks, and trivially avoids having to deal with the case where the second instruction in a pair represents a branch target. Finally, compounding can also be performed at compile-time; however, this requires modifications to the ISA, whereas the other two options are transparent to the architecture.

If compounding is performed on a pair of instructions that are not adjacent, they must effectively be reordered. This requires complex hardware to allow for precise exceptions, such as a reorder buffer [6]. While much of the required hardware is already present in out-of-order processors, supporting non-adjacent compound pairs in a scalar pipeline involves a significant cost in logic complexity.

Once a compound pair is generated, it progresses through the pipeline like a regular instruction. However, as the compound effectively executes two instructions at once, the execution stage must provide an ALU with the appropriate number of operand inputs. Alternatively, it would be possible to execute the compound instruction in a sequential fashion using a finite state machine; however, this approach would sacrifice any potential performance gain.

The number of register file ports limits compounding opportunities. In the general case, a pair of instructions can have up to four distinct source operands and up to two destination operands; however, the register files typically found in scalar processors only provide two read ports and a single write port. As adding ports to the register file represents a significant cost, it is reasonable to require that instructions in a compound pair have a read-after-write (RAW) dependency, and that they either also have a write-after-write (WAW) dependency, or that the second instruction not write a register at all. Similarly, one of the three remaining source operands can be required to be an immediate. Compound pairs satisfying these constraints can be supported even with simple register file configurations.

Technology	TSMC CL013G
Core frequency	200 MHz
Core area	2.10 mm^2
Pipeline depth	5 stages
Multiplier	16-bit w/ 40-bit accumulator
Instruction cache	16 kB, 2-way set associative
Data cache	16 kB, 2-way set associative
Floating point	not supported
Virtual memory	not supported

Table 1. Key characteristics of the LEON2

Compoundable instructions can be broken down into four main groups: Addition and subtraction, shifts, bitwise logic and memory operations. The constraints presented above imply that memory operations must be placed in the second slot of a compound pair: For loads, the result does not become available until the memory stage, preventing its use as an input to the second instruction's execute stage. Stores, on the other hand, do not generate results at all, and can thus never exhibit the required read-after-write dependency with a subsequent instruction. For the remaining three groups, the main factor in selecting eligible compound pairs is critical path length.

3. Compounding Microarchitecture

In the present work, we examine instruction compounding in the context of a basic in-order RISC pipeline representative of many embedded systems. Our discussion is based on the LEON2 embedded CPU, an open source implementation of the SPARC v8 ISA with a simple five-stage pipeline. However, the underlying concepts apply to other designs in a similar manner. Additional details about the LEON2 configuration used in our study can be found in Table 1.

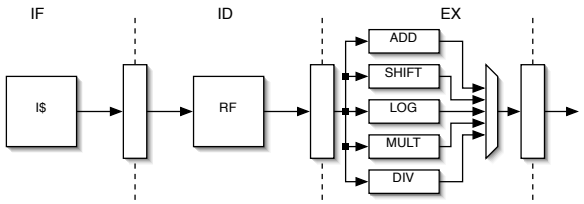
Coarse-grained pipelines can lead to an imbalance in logic depth between different pipeline stages. For example, the execution stage for basic ALU operations, such as bitwise logical operations or addition and subtraction, is significantly less complex than that of multiply instructions or than the instruction fetch stage, and thus typically does not lie on the critical path. As a result, the execution stages for a pair of simple ALU operations can be merged and still ex-

ecute in a single clock cycle without increasing the overall cycle time.

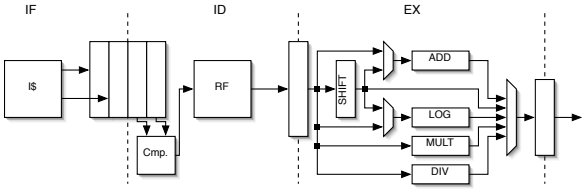
Instruction compounding can also be applied to more complex pipeline configurations: As pipeline depth increases, more energy can be saved by reducing the number of pipeline register traversals. However, as the pipeline becomes more fine-grained and logic on the critical path is spread across multiple stages, logic depth imbalance between individual pipeline stages naturally decreases, making it difficult to merge multiple different execution stages without incurring a cycle time penalty.

Section 2 outlined two possible approaches for compound formation: On the fly inside the pipeline, and upon I-cache refill. As caches make up a large portion of the power budget in typical embedded processors, it is undesirable from an energy-efficiency point of view to store additional predecode bits in the I-cache. Consequently, we perform compound formation at the beginning of the decode stage.

Figure 1 shows the changes required to implement compounding in the LEON2 pipeline. Up to two instructions per cycle are fetched from the I-cache and written into a buffer located between the fetch and decode stages. The buffer supplies pairs of candidate instructions to the compound formation logic, which performs the required dependency and type checking.



(a) Original LEON2 pipeline front-end



(b) Modified pipeline front-end with compounding support

Figure 1. Pipeline front-end

New instructions are only fetched into the buffer when its occupancy would otherwise drop below two instructions; all buffer entries are flushed whenever a branch is executed. The I-cache interface had to be widened from 32 to 64 bits in order to avoid stall cycles; the memory interface, however, remains 32 bits wide, and only a single instruction is sent to the fetch buffer in case of a cache miss. Unaligned cache accesses are not supported, so upon jumping to an odd address, the preceding instruction word is returned alongside the actual jump target, and only the latter is written into the fetch buffer.

Our basic implementation as depicted in Figure 1(b) can only compound the two frontmost instructions in the fetch buffer. At the cost of increased logic complexity, the architecture can be extended to enable compounding of the frontmost instruction with any other instruction in the buffer, subject to the following three constraints: Any intermediate instructions must not read the first instruction's result, they must not write any of the second instruction's operands, and they must not alter the control flow.

The basic type of compound pair supported in our architecture consists of two simple ALU operations, including bitwise logical operations, shifts, addition and subtraction (including tagged arithmetic) and the multiply step instruction, as well as the special SETHI instruction.

We furthermore allow compound pairs that consist of an ALU operation and a load or store instruction. Assuming that address generation for loads and stores takes place in the regular ALU, as is usually the case in embedded pipelines, we require that the read-after-write dependency be between the ALU operation and the memory address. By restricting loads and stores to be the second instruction, we also avoid having to deal with exceptions occurring within a pair.

Compounding of ALU operations with dependent branches, as implemented in the Intel Core microarchitecture [8], is not feasible in the LEON2 pipeline because branches are evaluated early on in the pipeline, and thus cannot depend on the result of an ALU instruction executing in the same cycle.

As shown in Figure 1(a), the execution stage in the LEON2 provides separate logic modules for addition and subtraction, shifts, bitwise logical operations, multiplication and division. As a result, compounding support can be added in a straightforward way by conditionally daisy-chaining existing logic blocks. The straightforward configuration depicted in Figure 1(b) restricts compounding to shift-add and shift-logical instruction pairs. While this restricts compounding opportunities, it only requires minimal hardware overhead in the execute stage. A full three-input ALU [7] can be used instead if support for additional compound types is desired.

4. Trace-Based Evaluation

As a first step, we developed a trace-based simulator for our proposed compounding microarchitecture. Given the dynamic instruction trace for a benchmark, the simulator generates the corresponding compounded instruction trace based on a flexible set of detailed compounding rules, which

specify eligible instruction types and combinations, dependency and locality requirements, as well as resource constraints (e.g. number of register file ports or ALU inputs).

This flexible simulation framework facilitates the design space exploration for our proposed microarchitecture, and allows us to rapidly evaluate its potential across a wide range of different workloads. A detailed evaluation at the RTL level can then be performed for the most promising points in the parameter space.

We selected the MiBench [2] benchmark suite as the basis for our evaluation. MiBench comprises a set of commercially representative embedded workloads from a variety of application areas. Dynamic instruction traces for the individual benchmarks were generated using Shade [1]. Due to incompatibilities with our tool flow, we were unable to generate traces for the *ispell*, *sphinx*, *pgp* and *gsm* benchmarks.

Figure 2 shows the reduction in dynamic instruction stream size that can be achieved by our proposed microarchitecture, broken down into the two basic compound types described in Section 3. The corresponding simulation runs were conducted under the following constraints: We first assumed a basic configuration that can only compound adjacent instructions in the frontmost two entries of the fetch buffer. A full three-input ALU was modeled, allowing for combinations of any type of ALU operations excluding multiplication and division. Finally, the register file was modeled as having two read ports and a single write port.

The benchmarks in the *consumer* group achieve the highest average compounding rate of 9%, with individual benchmarks ranging from 5% for *lame* to over 15% for the *jpeg*. Compoundability in *lame* is limited by the fact that 20% of the instructions are floating point operations; similarly, compounding in *tiff2rgba* is limited by the very high fraction of loads and stores. Benchmarks in the *office* and *telecomm* groups averaged 4% and 2% compounding rate, respectively. The result for the latter group is dominated by the fact that *crc32* exhibits virtually no compoundability at all. Finally, compounding shows little benefit for the benchmarks in the *network* and *security* groups, with average compounding rates of less than 1% in both cases.

In general, we found that the instruction scheduling applied at higher optimization levels severely degrades the compounding rate. Common optimizations attempt to maximize instruction-level parallelism (ILP) by spreading out dependent instructions. While this allows superscalar processors to issue multiple independent instructions in the same cycle, it is counterproductive for compounding, which requires dependencies to be present. We observe that all the benchmarks in Figure 2 that exhibit minimal compounding potential are compiled at -O3 by default; similarly, increasing the optimization level for the *jpeg* benchmarks from -O1 to -O3 leads to a significant reduction in compounding rate.

In an additional set of simulations, we investigated the performance benefits achievable by allowing non-adjacent instructions to be compounded. In particular, we studied configurations that support compounding of any two instructions in the fetch buffer, subject to the constraints on intermediate instructions that were outlined in Section 3. Simulations with four and six buffer entries did not exhibit significant increases in compounding rate for the vast majority of benchmarks. While we found that a significant number of non-consecutive eligible instructions did exist, the constraints on intermediate instructions prevented compounding in most cases. Even larger buffer sizes are unlikely to offer further benefits, as they would exceed the average basic block length for the majority of benchmarks [2].

We also ran a set of simulations to explore the impact of adding a third read port, a second write port, or both to the register file. The addition of a third read port eliminates the constraint that one of the input operands must be an immediate value, whereas adding a second write port eliminates the requirement for a write-after-write dependency between the two paired instructions. Our results show that, with few exceptions, relaxing these constraints yields virtually no increase in compounding opportunities across the board.

Overall, our results suggest that configurations which only support compounding of adjacent instructions and do not provide additional register file ports represent the best tradeoff between implementation complexity and compounding potential.

5. RTL-Level Simulation

To evaluate the energy savings achievable through instruction compounding, we synthesized the RTL implementation of the basic back-to-back compounding configuration using Synopsys Design Compiler 2005.09-SP2 and performed place and route using Cadence Encounter v06.10. We then used Mentor Graphics ModelSim 6.0SE to extract net switching statistics for a set of kernels from the final netlist.

The addition of the fetch buffer, compound formation logic, and the required changes to the ALU and the cache interface increased the gate count of the LEON2 core by 21% (from 19k to 23k gates). However, it should be noted that we used a minimal configuration of the LEON2 core that does not support floating point operations or virtual memory; the overhead would be significantly smaller for more complex configurations. Our modifications did not negatively impact the cycle time.

Because our RTL implementation provides only limited memory resources, no operating system and no C runtime library support, we conducted our evaluation using the set of kernel benchmarks described in Table 2, rather than the MiBench benchmarks. All kernels execute directly on the hardware.

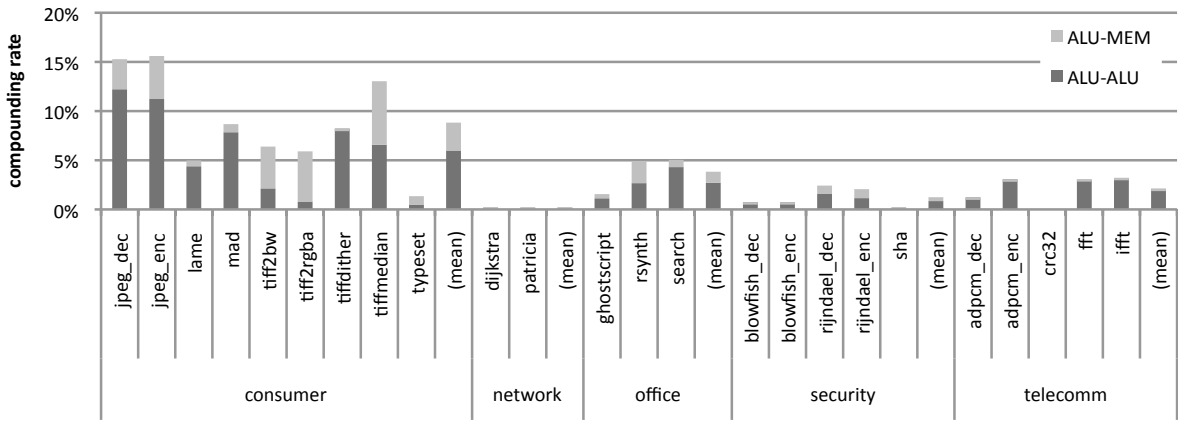


Figure 2. Trace-based simulation results

Kernel	Description
aes	Advanced Encryption Standard with 128-bit key.
crc	Cyclic redundancy check used in IEEE 802.3. Computes 32-bit checksums over two blocks of 64 words.
dct	2D Discrete Cosine Transform of an 8x8 region used in JPEG encoding.
huffman	Huffman encoder used to encode DC coefficients in the JPEG standard. Encodes 44 differential DC coefficient values.
viterbi	Viterbi decoder used in the GSM standard for voice data. A 1/2 rate convolution coder with a constrain length of 5 and a 189-bit frame. Input frames are represented as 3-bit soft-decision sample values.

Table 2. Kernels for RTL evaluation

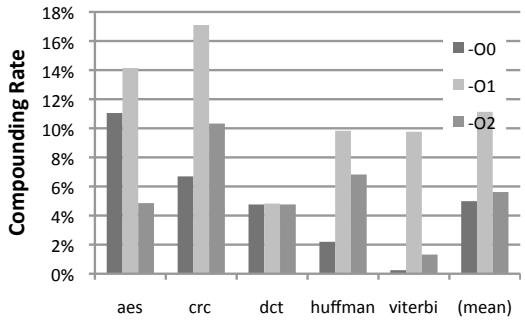


Figure 3. Impact of optimization level

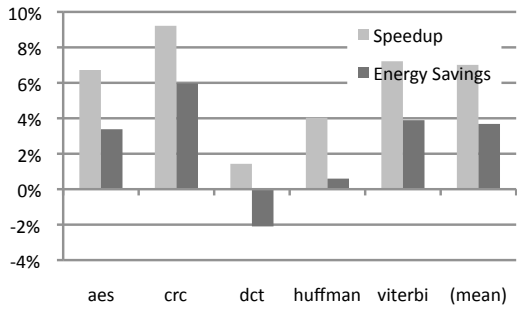


Figure 4. RTL simulation results

As in our MiBench simulations, we found that optimal compounding is achieved at optimization level -O1, and that higher levels have a detrimental effect due to counterproductive instruction scheduling. A breakdown of compounding opportunities as a function of the optimization level is presented in Figure 3.

In all cases, code compiled at -O1 running on our modified microarchitecture outperformed higher optimization levels running on the baseline processor. Figure 4 shows the speedup over the fastest non-compounding optimization level for each kernel and the associated pipeline energy savings.

The addition of the compound formation logic and the fetch buffer increases the average power dissipated by the integer unit by about 3.6% (from 16.8 mW to 17.4 mW). However, as Figure 4 shows, this power increase is accompanied by performance increases as high as 9% for the different kernels. This results in net energy savings ranging from 1% for *huffman* to 6% for *crc*. In the case of *dct*, compounding only achieves marginal speedup because this kernel has a tight inner multiply-accumulate loop that is largely not compoundable. As a result, this kernel actually runs

Operation	Energy
32-bit I-cache read	207 pJ
64-bit I-cache read	276 pJ
Fetch buffer read and write	9.5 pJ
Register file read	21 pJ
Addition	4 pJ
Multiplication	20 pJ

Table 3. Energy cost of various operations

slightly less efficiently on our implementation than it does on the baseline LEON2.

Overall, our implementation achieves a mean compounding rate of 7% across all kernels, and decreases energy consumption by 4% on average. A breakdown of the energy cost of various important operations is provided in Table 3.

Across all kernels, there is a relatively constant 3% delta between the achieved speedup and the reduction in core energy, which is a result of the overhead associated with the fetch buffer and the compounding logic. Noting that, for simplicity, we only implemented support for a limited subset (shift-add/sub and shift-logical) of all possible compound pairs in our RTL, we expect that a more complete implementation should yield higher compounding rates and further reduce energy consumption.

Having an instruction buffer between the fetch and decode stages provides an important side benefit: The LEON2's I-cache requires its enable signal to be asserted one cycle in advance. As stall information is not yet available at that point, the baseline LEON2 must access the I-cache in every cycle. However, in our modified implementation, if the fetch buffer is full in the current cycle, at least two instruction will remain in the next cycle even if a compound pair is executed, and thus the I-cache can safely be turned off. As cache accesses represent a large fraction of the LEON2's overall energy consumption, this results in significant energy savings. However, these savings are not directly related to compounding, and could be achieved by just adding a simple fetch buffer to the baseline LEON2 configuration. Consequently, we have not included the I-cache energy in our comparison.

6. Conclusions

In the present work, we proposed a microarchitecture for supporting instruction compounding in in-order embedded microprocessors.

Compounding improves both performance and energy efficiency by dynamically merging pairs of dependent instructions into more complex compound instructions. This effectively leads to fewer instructions being executed, less pipeline switching activity, and fewer register file accesses.

Using a trace-based evaluation of the MiBench benchmark suite, we showed that significant compounding opportunities of up to 15% exist in representative embedded workloads.

To validate our approach, we integrated the proposed architecture into the LEON2 processor core. Simulations running a set of embedded kernels on a placed and routed netlist show that compounding can reduce pipeline energy consumption by up to 6%.

Our results show that the availability of compounding opportunities in the instruction stream is highly dependent on the compiler's code generation; thus, we believe that modifying the compiler's instruction scheduler to optimize for compounding represents a promising opportunity for further research.

References

[1] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 128–137, 1994.

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

[3] S. Hu and J. E. Smith. Using dynamic binary translation to fuse dependent instructions. *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, pages 213–224, 2004.

[4] Q. Jacobson and J. E. Smith. Instruction pre-processing in trace processors. *HPCA '99: Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pages 125–129, 1999.

[5] Y. Sazeides, S. Vassiliadis, and J. E. Smith. The performance potential of data dependence speculation and collapsing. *MICRO 29: Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 238–247, 1996.

[6] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, 1988.

[7] S. Vassiliadis, B. Blaner, and R. J. Eickemeyer. Scism: A scalable compound instruction set machine. *IBM Journal of Research and Development*, 38(1):59–78, 1994.

[8] O. Wechsler. Inside intel core microarchitecture: Setting new standards for energy-efficient performance. *Technology@Intel Magazine*, 2006.